Chapter 3 Container Classes

Objectives

- To understand the list ADT as a general container class for manipulating sequential collections.
- To understand how lists are implemented in Python and the implications this has for the efficiency of various list operations.
- To develop intuition about collection algorithms such as selection sort and use Python operator overloading to make new sortable classes.
- To learn about Python dictionaries as an implementation of a general mapping and understand the efficiency of various dictionary operations.

3.1 Overview

Program design gets more interesting when we start considering programs that manipulate large data sets. Typically, we need more efficient algorithms to operate on large collections. Oftentimes the key to an efficient algorithm lies in how the data is organized, that is, the so-called *data structures* on which the algorithms operate. Object-oriented programs often use *container classes* to manage collections of objects. An instance of a container class manages a single collection. Objects can be inserted into and retrieved from the container object at run-time. Python includes a number of container classes as built-in types. You are probably familiar with lists and dictionaries, which are the two main container classes in Python.

In this chapter, we review the basics of Python lists and dictionaries and also take a look at how these containers are implemented in Python. Knowing how

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

a collection is implemented is often crucial to understanding the efficiency of the supported operations.

3.2 Python Lists

Lists are one of the main workhorse data structures in the Python language. Just about every program makes use of lists in some form. A thorough understanding of lists is essential for anyone writing in Python. Given their usefulness, it is not surprising that containers similar to Python lists are provided by virtually every high-level programming language.

Informally, a list is a collection of objects that is stored in sequential order. For example, a list might be used to represent the students in a class or a deck of cards. Because a list has an ordering, it is meaningful to talk about things such as the first object in a list or the next object in a list.

Using our new terminology from last chapter, we can think of a Python list as implementing an ADT for a sequential collection. Python provides quite a number of operations on lists. Some operations are supported by built-in functions and operators, whereas others are list methods. Here is a specification for some of the operations provided:

Concatenation (*list1* + *list2*) Returns a new list that contains the elements of *list1* followed by the elements of *list2*.

Repetition (*list1* * *int1* or *int1* * *list1*) Returns a new list corresponding to the list of elements obtained by concatenating *list1* with itself *int1* times.

Length (`len(`*list1*`)`) Returns the number of items in *list1*.

Index (*list1*`[`*int1*`]`) Returns the item at position *int1* in *list1*. The first item in the list is at index 0 and the last item is at index `len(`*list1*`)-1`.

Slice (*list1*`[`*int1*`:`*int2*`]`) Returns a new list containing the items in *list1* starting at position *int1* up to, but not including, *int2*. If $int2 \leq int1$ the resulting list is empty (assuming *int1* and *int2* are non-negative).

Check membership (*item* `in` *list1*) Returns `True` if *item* occurs in *list1* and `False` otherwise.

Add at end (*list1*`.append(`*obj1*`)`) Modifies *list1* by adding *obj1* to the end.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

Add anywhere (*list1*.insert(*int1*,*obj1*)) Modifies *list1* by adding *obj1* at position *int1*. The original items from position *int1* on are "shifted" to make room.

Delete index (*list1*.pop(*int1*)) Returns the item at *list1*[*int1*] and modifies *list1* by deleting this item from the list. Items in position *int1*+1 on are shifted down one index to "fill the gap." If *int1* is not supplied, the last item in the sequence is the one deleted.

Remove object (*list1*.remove(*obj1*)) Deletes the first occurrence of *obj1* in *list1*.

You probably used descriptions similar to this when you first learned how to use Python lists. Notice that the description says nothing about how a Python list is actually implemented in the computer; that's the hallmark of an ADT. A little later on, we'll take a look under the hood to see how lists can be implemented. Right now, we're taking a client's point of view and looking only at how lists are used.

3.3 A Sequential Collection: A Deck of Cards

Since Python provides an implementation of lists, it is common to make use of this built-in type to implement various collection abstractions. Continuing our card-game example from last chapter, let's try implementing a collection to represent a deck of cards. As a starting point, we need to determine the set of operations that will be useful for a deck of cards. Obviously, we will need a way to create a new (full) deck of cards. Usually, the deck is shuffled and used to deal cards into hands. If we are modeling the ADT using a Python class, we might try something along these lines:

```
class Deck(object):

    def __init__(self):
        """post: Create a 52-card deck in standard order"""

    def shuffle(self):
        """Shuffle the deck
        post: randomizes the order of cards in self"""

    def deal(self):
        """Deal a single card
        pre:  self is not empty
        post: Returns the next card in self, and removes it from self."""
```

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

A quick inspection of this specification shows a shortcoming of our design so far. Notice that the `deal` method contains a precondition, since we can't deal any card from an empty deck. For completeness, we should add a way for client code to check this precondition. We could add something like an `isEmpty` method that tells when the deck is exhausted. More generally, we might have a `size` method that gives the number of cards left in the deck. In many card games, it's important to know how many cards are left, so the latter approach seems a bit more useful. Let's add it to the specification.

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

Adding this operation to the ADT also allows us to state the precondition for the `deal` method more precisely. Here's the improved specification:

```
    def deal(self):
        """Deal a single card
        pre:  self.size() > 0
        post: Returns the next card in self, and removes it from self."""
```

Having thought out the interface for our ADT, we're now ready to start implementing. Obviously, a deck is a sequence of cards, so a natural choice of representation is to use a Python list to hold the cards in the deck. Here's a constructor for our `Deck`.

```
# Deck.py
from random import randrange
from Card import Card

class Deck(object):

    def __init__(self):
        cards = []
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                cards.append(Card(rank,suit))
        self.cards = cards
```

Notice how this code uses nested loops to produce every possible combination of rank and suit. Each subsequent card is appended to the list of cards, and the resulting list is stored away as an instance variable of the `Deck` object.

Once we have created a `Deck` object, checking its size and dealing cards from the deck can be accomplished with simple list operations.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
    def size(self):
        return len(self.cards)

    def deal(self):
        return self.cards.pop()
```

The `deal` method returns cards in order from the end of the list. Using this approach the ordering imposed by the Python list data structure determines the order in which the cards are dealt.

Now all we need is a method to shuffle a deck (i.e., put it into a random order). This gives us a chance to exercise our algorithm development skills. You probably know some ways of shuffling a deck of cards, but the usual methods don't transfer very well into code. One way to think about the problem is to consider the task of putting the cards into a specific arrangement. The shuffle operation should ensure that any of the 52! possible arrangements of the deck is equally likely. That means that every card in the deck has to have an equal chance of being the first card, and each of the remaining cards has an equal chance of being the second card, etc.

We can implement a shuffle algorithm by building a new list out of the cards in the original list. We start with an empty list and repeatedly transfer a card chosen at random from the old list to the new list. Here's how the algorithm looks in code:

```
    def shuffle(self):
        cards0 = self.cards
        cards1 = []
        while cards0 != []:
            # delete a card at random from those in original list
            pos = randrange(len(cards0))
            card = cards0.pop(pos)

            # transfer the card to the new list
            cards1.append(card)

        # replace old list with the new
        self.cards = cards1
```

We can improve this algorithm slightly by doing the shuffle in place. Rather than going to the trouble of building a second list, we could choose a card at random and move it to the front of the existing list. Then we ccould pick a card from locations 1 through n and move it to position 1, etc. There is one subtlety in this approach; when we place a random card into a given position, we have to be careful not to clobber the card that is currently in that position. That is, we need to save the card that is being replaced somewhere so that it is still part of the pool for subsequent

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

placement. The easiest way to do this is to simply swap the positions of the two cards. Here's the in-place version of our shuffle algorithm:

```
    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
```

Notice that in this code it is not necessary to do `self.cards = cards` at the end of the method. The assignment statement immediately before the loop sets `cards` to be a reference to the same list as `self.cards`. Therefore, the changes made to this list (swapping cards) are changing `self.cards`. The local variable `cards` is used for convenience (so we don't have to keep typing `self.cards`) and efficiency (retrieval of local variable values is more efficient than retrieval of instance variables).

We now have a complete `Deck` class. Let's take it for an interactive test drive.

```
>>> d = Deck()
>>> print d.deal()
King of Spades
>>> print d.deal()
Queen of Spades
>>> print d.deal()
Jack of Spades
>>> d.shuffle()
>>> d.size()
49
>>> print d.deal()
Seven of Hearts
>>> print d.deal()
Nine of Diamonds
```

Notice how the initial deck deals cards out from the standard ordering. After shuffling, the cards come out randomly, just as we expect.

3.4 A Sorted Collection: Hand

In the previous section, we used a Python list as a container class to implement a deck of cards. A deck has an implicit ordering of cards, namely the order in which the cards are dealt, and so it made sense to use a list to store the cards. Of course, the particular order that the deck is in is supposed to be random; that's why we

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

shuffle a deck. Sometimes we want the objects in a container to be in a specific order according to the value of each item. The process of putting a collection in order by value is called *sorting*. In this section, we'll look at an example of a sorted collection.

3.4.1 Creating a Bridge Hand

Let's put our `Deck` class to work in an actual application. Suppose we are writing a program to play the popular card game bridge. Building such a program incrementally, our first task might be to deal four 13-card hands from a shuffled deck. We'd also like to display the hands nicely so that we can analyze them. For example, newspapers that carry bridge columns often show hands arranged by suit (in the order spades, hearts, diamonds, clubs) with cards in each suit arranged by decreasing rank (ace, king, queen, ..., 2). Note that aces are considered higher than kings in bridge.

Our task is to deal cards into hands and then to arrange those hands into the specified order. This suggests the invention of a new kind of collection, a `Hand` class. A `Hand` is initially empty, and cards are added to it one by one as they are dealt. Considering our `Hand` as an ADT, we need operations to create a hand, add a card, put the hand in order (sort it), and display the cards in the hand. An initial specification of the class looks like this:

```
# Hand.py
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 card to the hand """

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

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

We have added to our initial description the ability to give each hand a name or label to identify it. Traditionally, bridge hands are identified with the compass points north, east, south and west. Notice that we have also added a `dump` method to display the contents of the hand. This is useful for testing and debugging purposes.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

Since hands are ordered, a Python list is again the container of choice for implementing the new collection. Most of the operations are trivial to implement. The constructor must store away the label and create an empty collection. Let's store it in an instance variable called `cards`:

```
# Hand.py
class Hand(object):

    def __init__(self, label=""):
        self.label = label
        self.cards = []
```

The `add` operation takes a card as a parameter and puts it into the collection. A simple `append` suffices:

```
    def add(self, card):
        self.cards.append(card)
```

To dump the contents of the hand, we just need to print out a heading and then loop through the list to print each card.

```
    def dump(self):
        print self.label + "'s Cards:"
        for c in self.cards:
            print "   ", c
```

Let's try out what we've got so far.

```
>>> from Hand import Hand
>>> from Card import Card
>>> h = Hand("North")
>>> h.add(Card(5,"c"))
>>> h.add(Card(10,"d"))
>>> h.add(Card(13,"s"))
>>> h.dump()
North's Cards:
    Five of Clubs
    Ten of Diamonds
    King of Spades
>>>
```

That looks good. Notice how the listing of the cards is indented under the hand heading.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

3.4.2 Comparing Cards

That leaves us with the problem of putting the hand in order. The sorting problem is an important and well-studied one in computer science. We'll take a quick look at it here and revisit it again in later chapters. If we want to put some things in a particular order, the first problem we have to solve is what exactly the ordering should be.

In the case of our bridge program, we want to order our `Card` objects, grouping them first by suit and then ordering by rank within suit. Usually orderings are determined by a relation such as "less than." For example, if we say that a list of numbers is in increasing order, that means that for any two numbers x and y in the list, if $x < y$ then x must precede y in the list. Similarly, we need a way of comparing cards so that we can order them in our `Hand`. In Chapter 2, we saw how Python operator overloading allows us to build new classes that "act like" existing classes. Here, we would like our cards to behave like numbers so that we can compare them using the standard Python operators such as $<$, $==$, $>$, and so on.

We can do this by defining methods for these operations in the `Card` class. Here are the definitions of the "hook" functions for these operators.

```
    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
```

Notice that we've given "primitive" definitions for `__eq__` and `__lt__`; the rest of the necessary operators can easily be defined in terms of these two. We have not bothered to write definitions for `__gt__` and `__ge__` because Python gives us these for free. In an expression such as `x > y`, when the `>` operator is not implemented

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

for `x`, Python will try the symmetric operation `y < x`. Similarly, `x >= y` invokes `y <= x`.

Now that our `Card` objects are comparable, there's one last detail to clean up. When we originally created the `Card` class, we used a rank of 1 to represent an ace, but in bridge aces are the highest card, coming right after the king. Right now, our comparison method will put aces at the low end, since the rank is 1.

We can handle this issue in a couple of ways. One approach would be to code a special case for aces into the comparison methods. Another solution is to simply modify the `Card` class to use ranks that run from 2 to 14 with 14 representing the ace. Taking the latter approach, the start of our modified `Card` class would now look like this:

```
class Card(object):
    """A simple playing card. A Card is characterized by two
    components:
    rank: an integer value in the range 2-14, inclusive (Two-Ace)
    suit: a character in "cdhs" for clubs, diamonds, hearts, and
    spades."""

    SUITS = "cdhs"
    SUIT_NAMES = ["Clubs", "Diamonds", "Hearts", "Spades"]

    RANKS = range(2,15)
    RANK_NAMES = ["Two", "Three", "Four", "Five", "Six",
                  "Seven", "Eight", "Nine", "Ten",
                  "Jack", "Queen", "King", "Ace"]
...
```

Recall that our `Deck` class actually has to generate every possible card to create the initial deck. As such, the `Deck` class depends on the `Card` class, and changing the interface to the `Card` class might break the `Deck` class, since it might not know that 14 is now a legal rank but 1 isn't. Fortunately, when we originally coded up `Deck` we used `Card.RANKS` to generate all the possible ranks rather than using a hard-coded range such as `range(1,14)`. By changing this constant in the `Card` class, we still are playing with a full deck. This illustrates the design advantage of using named constants rather than filling your code with "magic values." In this case, use of the constant helps us maintain the abstraction barrier between `Card` and `Deck`.

Given these modifications to our `Card` class, we can now compare cards just as if they were numbers using the relational operators:

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
>>> Card(14,"c") < Card(2,"d")
True
>>> Card(8,"s") > Card(10,"s")
False
>>> Card(6,"c") == Card(6,"c")
True
>>>
```

Notice how the ace of clubs is “less than” the two of diamonds, since we have said that all clubs proceed any diamond.

3.4.3 Sorting Cards

Now that we can compare cards, we just need to come up with an algorithm to put them in order. Perhaps surprisingly, we can use an algorithm very similar to the one we used to shuffle the deck. Instead of choosing a card at random to become the first card in the hand, we choose the biggest card. Then we choose the biggest of the remaining cards to be the next one, and so on. This algorithm is known as a selection sort. As we'll see later, it's not the most efficient way of sorting a list, but it's an easy algorithm to develop and analyze.

In Python, a particularly simple way to implement the selection sort algorithm is to use two lists. The “old” list is the original hand, and the “new” list will be the ordered hand, which starts out empty. As long as there are cards in the old list, we simply find the largest one, remove it from the old list, and place it at the back of the new list. When the old list is empty, the new list contains the cards in descending order. Here's an implementation:

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

Notice how the step of finding the largest card in the old list (`cards0`) is accomplished using the Python built-in function `max`. This is a nice side effect of implementing the comparison operators. Now that `Card` objects can be compared, any existing Python sequence operations that rely on comparing elements can be used on collections of `Card` objects. That certainly simplifies things, doesn't it?

Notice that we have developed a general sorting algorithm. It should work for sorting lists of any type of object. Right now, it sorts by creating a brand new list

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

that is in sorted order. However, just like the shuffling algorithm we did earlier, the selection sort can easily be converted to sort a list in place. Performing this conversion is left as an exercise.

Actually, we have done more work than necessary to sort our hands. Since our `Card` objects are now comparable, we can let Python do the sorting of the cards for us by using the `sort` method that is built into the Python list type. Of course, the built-in sort will put the cards into ascending order. To get the cards into descending order, we'll need to reverse them after sorting. Here's a version of the `sort` method using this approach.

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

That's certainly easiest, but if we had jumped to this solution right away, we would have missed out on the excitement of developing our own sorting algorithm.

How efficient is the selection sort algorithm that we developed? Obviously the main work of the function is being done inside the `while` loop. Notice that the loop continues until the `cards0` list is empty. Each time through the loop, exactly one item is removed from `cards0`, so it's clear that this loop will execute n times, where n is the number of items in the original list. Each time through the loop, we need to find the largest card in `cards0`. In order to find the largest card, the Python `max` function must look at each card in the list in turn and keep track of which is the largest. That's a $\Theta(c)$ operation, where c is the number of items in the list being analyzed. The first time through the `while` loop, `max` examines n cards. The next time through, it only has $n-1$ cards to consider, then $n-2$, etc. So the total work done in all the iterations of the `while` loop is $n + (n-1) + (n-2) + \ldots + 1$. As we discussed in subsection 1.3.4, this sum is given by the formula $\frac{n(n+1)}{2}$. That makes our selection sort at least an n^2 algorithm. While it can be no better than $\Theta(n^2)$, it could even be worse, depending on the efficiencies of the `remove` and `insert` methods, which are also executed in the body of the `while` loop. We'll consider those operations in section 3.5.

By contrast, the built-in `sort` method in Python is a $\Theta(n \log n)$ algorithm, which is much more efficient. For our simple hands of 13 cards, that doesn't make much difference, but for a large list, it can mean the difference between sorting the collection in seconds vs. hours or days. We'll see how to design more efficient sorting algorithms in section 6.5.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

3.5 Python List Implementation

When we analyzed the selection sort above, we concentrated on the `max` operation, which turned out to be $\Theta(n)$, but we ignored the `insert` and `remove` methods for lists. It turns out that both of these methods have the same time complexity as `max`. How do we know that? Just as the choice of using a Python list to implement our collection classes `Deck` and `Hand` determines the relative efficiency of the methods in these classes, the choice of data structures in the implementation of Python lists determines the efficiency of various list operations. Therefore, understanding the true efficiency of various operations requires some understanding of Python's underlying data structures.

3.5.1 Array-based Lists

So how can we efficiently store and access a collection of objects in computer memory? Recall that computer memory is simply a sequence of storage locations. Each storage location has a number associated with it (much like an index) called its *address*. A single data item may be stored across a number of contiguous memory locations. To retrieve an item from memory, we need a way to either look up or compute the starting address of the object. If we want to store a collection of objects, we need to have some systematic method for figuring out where each object in the collection is located.

Consider the case when all of the objects in a collection are the same size, that is they all require the same number of bytes to be stored. This would be the case with a homogeneous (all the same type) collection. A simple method for storing the collection would be to allocate a single contiguous area of memory sufficient to hold the entire collection. The objects could then be stored one after the next. For example, suppose an integer value requires 4 bytes (32 bits) of memory to store. A collection of 100 integers could be stored sequentially into 400 bytes of memory. Let's say the collection of integers starts at the memory location with the address 1024. This means the number at index 0 in the list starts at address 1024, index 1 is at 1028, index 2 is at 1032, etc. The location of the ith item can be computed simply using the formula $address\_of\_ith = 1024 + 4 * i$.

What we have just described is a data structure known as an *array*. Arrays are a common data structure used for storing collections, and many programming languages use arrays as a basic container type. Arrays are very memory efficient and support quick random access (meaning we can "jump" directly to the item we want) via the address calculation we just discussed. By themselves, however, they are somewhat restrictive. One issue is the fact that arrays must generally be

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

homogeneous. For example, it's usually not possible to have an array that contains both integers and strings. In order for the address calculations to work, all elements must be the same size.

Another shortcoming of arrays is that the size of the array is determined when memory is allocated for it. In programming language terminology, arrays are said to be *static.* When we allocate an array for 100 items, the underlying operating system grants us an area of memory sufficient to hold that collection. The memory around the array will be allocated to other objects (or even other running programs). There is no way for the array to grow, should more elements be added later. Programmers can work around this limitation to some extent by creating an array large enough to hold some theoretical maximum collection size. By keeping track of how many slots of the array are actually in use, the programmer can allow the collection to grow and shrink up to that maximum size. However, this negates the memory efficiency of arrays, since it forces the programmer to request more memory than might actually be needed. And, of course, we're still out of luck if the size of the collection needs to grow beyond the anticipated maximum.

In contrast to arrays, Python lists are heterogeneous (they can mix objects of different types) and dynamic (they grow and shrink). Underneath, Python lists are actually implemented using arrays. Remember that Python variables store references to the actual data objects. Don't worry too much if you are not familiar with or do not fully understand the concept of references; we will discuss them in detail in the next chapter. The point here is that what is stored in the consecutive memory locations of the Python list array are the *addresses* of actual data objects. Each address is the same length (typically 32 or 64 bits on modern CPUs). To retrieve a value from a list, the Python interpreter first uses the indexing formula to find the location of the reference (address) to the object and then uses the reference to retrieve the object. So an array with fixed-sized elements can be used to store the addresses that are then used to retrieve arbitrarily sized objects.

Of course, Python lists can also grow by calling methods such as `insert` and `append`. Internally, Python allocates a fixed-sized array for a list and keeps track of this maximum fixed size and the current size of the list. When an attempt is made to add elements beyond the current maximum size, a new contiguous section of memory large enough to store all the elements must be allocated. The references stored in the old array are then copied to the new larger array, and finally the memory for storing the old list is deallocated (given back to the operating system). Using this trick of dynamic array allocation, Python lists can continue to grow as long as enough system memory is available to hold the new list.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

3.5.2 Efficiency Analysis

Knowing that Python lists are implemented as dynamically resizing arrays, we are now in a position to analyze the run-time efficiency of various list operations.

Allocating a new larger array is a relatively expensive operation, so the new array that is allocated is typically significantly larger. Allocating a much larger array prevents the resize operation from being necessary until quite a number of additional items have been added to the array. This means appending onto the end of a Python list will occasionally require $\Theta(n)$ computation (to allocate a new array and copy the existing items over), but most of the time it is a $\Theta(1)$ operation. If the size of the array is doubled each time it needs to be made larger, then the $\Theta(n)$ resize operation only needs to be executed every n appends. Amortizing the cost of creating the new larger array over the n appends that can be performed without the resize operation results in the average cost of an append being $\Theta(1)$.

The situation for arbitrary insertion operations anywhere in the list is a little different. Because the elements of an array are in contiguous memory locations, to insert into the middle of an array we have to first create a "hole" by shifting all of the following items one place to the right. When the insertion is at the very front of the list, the Python interpreter has to move all n elements currently in the array. So the insertion operation is still $\Theta(n)$ even if the size doubling trick is used when the array is full.

Python lists also support a method to delete elements from an existing list. The analysis for deletion is the same as for insertion. If we delete the element in position four, all the elements in positions five and above must be shifted down one location. So deletion, like insertion, is a $\Theta(n)$ operation. When deleting elements, we do not need to change the maximum size of the list; however, if a list grows very large for a short time period and then shrinks and stays much smaller for the rest of the program, the memory allocated to store the largest size will always be in use.

3.6 Python Dictionaries (Optional)

Python lists are an example of a sequential data structure. There is an inherent ordering of the data. Even in our implementation of the randomly shuffled deck, the items in the underlying list are still indexed by the natural numbers (0, 1, 2, ...), which gives the collection a natural ordering. In fact, one can view lists abstractly as just a kind of *mapping* from indexes to items in the list. That is, each valid index is associated with (maps to) a particular list item.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

The idea of mapping is very general and need not be restricted to using numbers as the indexes. If you think about it a bit, you can probably come up with all sorts of useful collections that involve other sorts of mappings. For example, a phone book is a mapping from names to phone numbers. Mappings pop up everywhere in programming, and that is why Python provides an efficient built-in data structure for managing them, namely a dictionary.

3.6.1 A Dictionary ADT

You have probably run across Python dictionaries before, but perhaps not given them much thought. A dictionary is a data structure that allows us to associate keys with values, that is, it implements a mapping. Abstractly, we can think of a dictionary as just a set of ordered `(key, value)` pairs. Viewed as an ADT, we just need a few operations in order to have a useful container type.

`Create`
  post: Returns an empty dictionary.

`put(key,value)`
  post: The value `value` is associated with `key` in the dictionary. `(key,value)` is now the one and only pair in the dictionary having the given key.

`get(key)`
  pre: There is an `X` such that `(key,X)` is in the dictionary.
  post: Returns `X`.

`delete(key)`
  pre: There is an `X` such that `(key,X)` is in the dictionary.
  post: `(key,X)` is removed from the dictionary.

There are many programming situations that call for dictionary-like structures. Some programming languages such as Python and Perl provide built-in implementations of this important ADT. Other languages such as C++ and Java provide them as part of a standard collection library.

3.6.2 Python Dictionaries

A Python dictionary provides a particular implementation of the dictionary ADT. Let's start with a simple example. Remember in our `Card` example we needed to be able to turn characters representing suits into full suit names. That's a perfect job for a dictionary. We could define a suitable Python dictionary like this:

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
suits = { "c":"Clubs", "d":"Diamonds", "h":"Hearts", "s":"Spades" }
```

As you can see, the syntax for a dictionary literal resembles our abstract description of a dictionary being a set of pairs. In Python, the key-value pairs are joined with a colon. In this case, we are saying that the string `"c"` maps to the string `"Clubs"`, `"d"` maps to `"Diamonds"`, etc.

Values can be retrieved from a Python dictionary via a `get` method, but Python also allows dictionaries to be indexed in a manner similar to lists. Here are some interactive examples:

```
>>> suits
{"h": "Hearts", "c": "Clubs", "s": "Spades", "d": "Diamonds"}
>>> suits.get("c")
'Clubs'
>>> suits["c"]
'Clubs'
>>> suits["s"]
'Spades'
>>> suits["j"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: "j"
>>> suits.get("j")
>>> suits.get("x", "Not There")
'Not There'
```

Notice that when `suits` was evaluated, the key-value pairs did not print out in the same order as when the dictionary was created. Dictionaries do not preserve the ordering of items, only the mapping. The last interactions show a subtle difference between indexing and the `get` operation. Trying to index into a dictionary using a nonexistent key raises a `KeyError` exception. However, the `get` method simply returns `None` as a default value in this case. As illustrated in the last interaction, `get` also allows an optional second parameter to provide an alternative default value, should the key lookup fail.

The abstract `put` operation for changing entries in a dictionary or extending it with new entries is implemented via assignment in Python. Again, this makes the syntax for working with dictionaries very similar to that of lists. Here are a few examples:

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
>>> suits["j"] = "Joker"
>>> suits
{'h': 'Hearts', 'c': 'Clubs', 'j': 'Joker', 's': 'Spades', 'd': 'Diamonds'}
>>> suits["j"]
'Joker'
>>> suits["c"] = "Clovers"
>>> suits["s"] = "Shovels"
>>> suits
{'h': 'Hearts', 'c': 'Clovers', 'j': 'Joker', 's': 'Shovels', 'd': 'Diamonds'}
```

To remove items, Python dictionaries understand the `del` function, just as Python lists do. You can also remove all the entries from a dictionary using the `clear` method.

```
>>> suits
{'h': 'Hearts', 'c': 'Clovers', 'j': 'Joker', 's': 'Shovels', 'd':
'Diamonds'}
>>> del suits['j']
>>> suits
{'h': 'Hearts', 'c': 'Clovers', 's': 'Shovels', 'd': 'Diamonds'}
>>> suits.clear()
>>> suits
{}
```

In addition to these basic operations, Python provides a number of conveniences for working with dictionaries. For example, we often want to do something to every item in a dictionary. For that, it's useful to deal with the dictionary in a sequential fashion. Python dictionaries support three methods for producing list representations of dictionary components: `keys` returns a list of keys, `values` returns a list of values, and `items` returns a list of `(key,value)` pairs.[1] You can also directly iterate through the keys of a dictionary using a `for` loop and check whether a given key is in the dictionary using the `in` operator.

[1] In Python 3.0, these methods return iterator objects (see Chapter 4). They can easily be converted to lists, for example `list(myDictionary.items())`.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
>>> suits.keys()
['h', 'c', 's', 'd']
>>> suits.values()
['Hearts', 'Clovers', 'Shovels', 'Diamonds']
>>> suits.items()
[('h', 'Hearts'), ('c', 'Clovers'), ('s', 'Shovels'), ('d', 'Diamonds')]
>>> for key in suits:
...    print key, suits[key]

h Hearts
c Clovers
s Shovels
d Diamonds
>>> 'c' in suits
True
>>> 'x' in suits
False
```

3.6.3 Dictionary Implementation

As with virtually any ADT, there are numerous ways one could go about implementing dictionaries. The choice of implementation will determine how efficient the various operations will be. One simple representation would be to store the dictionary entries as a list of key-value pairs. A `get` operation would involve some form of lookup on the list to find the pair with the specified key. Other operations could also be performed using simple list manipulation. Unfortunately, this approach will not be very efficient, as some of the operations will require $\Theta(n)$ effort. (An exact analysis of the situation is left as an exercise.)

Python uses a more efficient data structure called a *hash table*. Hash tables are covered in-depth in section 13.5. Here we just want to give you some intuition so that you can understand the efficiency of various dictionary operations. That will enable you to judge the efficiency of algorithms that use Python dictionaries.

The heart of a hash table is a *hashing function*. A hashing function takes a key as a parameter and performs some simple calculations on it to produce a number. Since all data on the computer is ultimately stored as bits (binary numbers), it's pretty easy to come up with hashing functions. Python actually has a built-in function `hash` that does this. You can try it out interactively.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
>>> hash(2)
2
>>> hash(3.4)
-751553844
>>> hash("c")
-212863774
>>> hash("hello")
-1267296259
>>> hash(None)
135367456
>>> hash((1,"spam",4,"U"))
40436063
>>> hash([1,"spam",4,"U"])
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Feeding anything that is “hashable” to `hash` produces an `int` result. Take a close look at the last two interactions. A tuple is hashable, but a list is not. One requirement of a hash function is that whenever it is called on a particular object, it must always produce the exact same result. Since the hash function relies on the underlying representation of the object to produce a hash value, the value is guaranteed to be valid only for objects whose underlying representations are not subject to change. In other words, we can only hash immutable objects. Numbers, strings, and tuples are all immutable and, hence, hashable. Lists can be changed, so Python does not allow them to be hashed.

With a suitable hash function in hand, it's easy to create a hash table to implement a dictionary. A hash table is really just a large list that stores `(key,value)` pairs. However, the pairs are not just stored sequentially one right after another. Instead they are stored in the list at an index determined by hashing the key. For example, suppose we allocate a list of size 1000 (this is our “table”). To store the pair `("c","Clubs")` we compute `hash("c") % 1000` = 226. Thus, the item will be stored in location 226. Notice that the remainder operation guarantees we get a result in `range(1000)`, which will be a valid index for our table. With a good hashing function, items will be distributed across the table in a relatively uniform way.

As long as no two keys in the dictionary hash to the exact same location, this implementation will be very efficient. Inserting a new item takes constant time, since we just apply the hash function and assign the item to a location in the list. Lookup has similar complexity; we just compute the hash and then we know where to go grab the item. To delete an item we can just put a special marker (e.g., `None`)

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

into the appropriate slot. So all basic dictionary operations can be accomplished in constant ($\Theta(1)$) time.

But what happens when two keys hash to the same spot? This is called a *collision*. Dealing with collisions is an important issue that is covered in section 13.5. For now it suffices to note that there are good techniques for dealing with this problem. Using these techniques and ensuring a table of adequate size yields data structures that, in practice, allow for constant time operations. Python dictionaries are very efficient and can easily handle thousands, even millions of entries, provided you have enough memory available. The Python interpreter itself relies heavily on the use of dictionaries to maintain namespaces, so the dictionary implementation has been highly optimized.

3.6.4 An Extended Example: A Markov Chain

Let's put our new knowledge of dictionaries to use in a program that combines several Python container classes to build Markov models. A Markov model is a statistical technique for modelling systems that change over time. One application of Markov models is in the area of systems for natural language understanding. For example, a speech recognition system can use predictions about what word is likely to come next in a sentence in order to decide among homonyms such as "their," "they're," and "there."

Our task is to develop a Markov class that could be used in such applications. We will demonstrate our class by using it to construct a program that can generate "random" language of a particular style. For example, if we train the program by feeding it mystery novels, it will generate gibberish that sounds like it came from a (really) bad mystery novel.

The basic idea behind a Markov model of language is that one can make predictions about the next word of an utterance by looking at some small sequence of preceding words. For example, a trigram model looks at the preceding two words to predict the next (third) word in a sequence. More or fewer words could be used as a "window" depending on the application. For example, a bigram model would predict the probabilities for the next word based only on the immediately preceding word. Our initial design will be for a trigram model; extending the program to arbitrary length prefixes is left as an exercise.

Here is a quick specification of our Markov class.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
class Markov(object):

    """A simple trigram Markov model.  The current state is a sequence
       of the two words seen most recently. Initially, the state is
       (None, None), since no words have been seen. Scanning the
       sentence "The man ate the pasta" would cause the
       model to go through the sequence of states: [(None,None),
       (None, 'The'), ('The', 'man'), ('man','ate'), ('ate','the'),
       ('the','pasta')]"""

    def __init__(self):

        """post: creates an empty Markov model with initial state
                 (None, None)."""

    def add(self, word):

        """post: Adds word as a possible following word for current
                 state of the Markov model and sets state to
                 incorporate word as most recently seen.

           ex: If state was ("the", "man") and word is "ate" then
               "ate" is added as a word that can follow "... the man" and
               the state is now ("man", "ate")"""

    def randomNext(self):

        """post: Returns a random choice from among the possible choices
                 of next words, given the current state, and updates the
                 state to reflect the word produced.

           ex: If the current state is ("the", "man"), and the known
               next words are ["ate", "ran", "hit", "ran"], one of
               these is selected at random. Suppose "ran" is selected,
               then the new state will be: ("man", "ran"). Note the
               list of next words can contain duplicates so the
               relative frequency of a word in the list represents its
               probability of being the next word."""

    def reset(self):

        """post: The model state is reset to its initial
                 (None, None) state.

           note: This does not change the transition information that
                 has been learned so far (via add()), it
                 just resets the state so we can start adding
                 transitions or making predictions for a "fresh"
                 sequence."""
```

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

Reading this specification closely reveals a number of container structures that we must weave together to produce a working class. An instance of the `Markov` class must always know its current state, which is a sequence of the last two words encountered. We could represent this sequence as either a list or a tuple. Given the current state, we need some sort of model that allows us to retrieve a collection of possible next words. That's just a mapping, so we can use a dictionary to implement the model. The keys for the dictionary will be pairs of words, and the values will be lists of possible next words. Note we must use a tuple to represent the word pair, since Python lists are not hashable.

We are now in a position to write the code for this class.

```
import random

class Markov(object):

    def __init__(self):
        self.model = {}  # maps states to lists of words
        self.state = (None, None)  # last two words processed

    def add(self, word):
        if self.state in self.model:
            # we have an existing list of words for this state
            # just add this new one (word).
            self.model[self.state].append(word)
        else:
            # first occurrence of this state, create a new list
            self.model[self.state] = [word]
        # transition to the next state given next word
        self._transition(word)

    def reset(self):
        self.state = (None, None)

    def randomNext(self):
        # get list of next words for this state
        lst = self.model[self.state]
        # choose one at random
        choice = random.choice(lst)
        # transition to next state, given the word choice
        self._transition(choice)
        return choice

    def _transition(self, next):
        # help function to construct next state
        self.state = (self.state[1], next)
```

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

You should read this code carefully to make sure that you understand how the class makes use of Python dictionaries, lists, and tuples.

All that remains to complete our gibberish-generating program is to write some code to "train" a model on a large sample of input text and then use the resulting model to generate a stream of output. Here are a couple of functions that fit the bill.

```
# test_Markov.py
def makeWordModel(filename):
    # creates a Markov model from words in filename
    infile = open(filename)
    model = Markov()
    for line in infile:
        words = line.split()
        for w in words:
            model.add(w)
    infile.close()
    # Add a sentinel at the end of the text
    model.add(None)
    model.reset()
    return model

def generateWordChain(markov, n):
    # generates up to n words of output from a model
    words = []
    for i in range(n):
        next = markov.randomNext()
        if next is None: break  # got to a final state
        words.append(next)
    return " ".join(words)
```

Here is an example of the output obtained by training a model on Lewis Carroll's *Alice's Adventures in Wonderland*:

> Alice was silent. The King looked anxiously at the mushroom for a rabbit! 'I suppose I ought to have it explained,' said the Caterpillar angrily, rearing itself upright as it was written to nobody, which isn't usual, 'Oh, don't talk about cats or dogs either,' if you want to go nearer till she got up and down in an encouraging opening for a minute or two. 'They couldn't have wanted it much,' said Alice, swallowing down her anger as well as she did not get dry again: they had a little before she made it out to sea. So they began solemnly dancing round and round Alice, every now and then treading on her face brightened up at the Caterpillar's making such a curious appearance in the middle of one!

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

As you can see, parts of this output are tantalizingly close to coherent sentences. In contrast, here is sample output from a program that just chooses words at random from the text:

> of cup,' sort!' forehead you However, house the went to me unhappy up impossible settled We had help the always in see, forgot tree of you 'for night? because hadn't her ear. all confused sit took the care went quite do up, 'How three An Turtle, the was soldiers, solemnly, so went of the sharply. to Rabbit 'Tis there last, a with that o'clock and below, he Writhing, don't to wig, she into three, But said there,' offended. turning This some (she together."' such be because and what the had to hatters better This Mouse new said the pool whiting. with could from bank-the mile said I she all! turning when 'Begin By how as head them, little, and Latitude he

Clearly, the trigram model is capturing some important regularities in language. That's what makes it useful in many language processing tasks such as generating annoying email solicitations to defeat spam filters. Knowledge is power; please don't abuse your new skills!

3.7 Chapter Summary

This chapter has introduced the idea of container classes as a mechanism for dealing with collections of objects. Here is a summary of some of the key concepts.

- Container objects are used to manage collections. Items can be added to and removed from containers at run-time.
- The built-in Python list is an example of a container class.
- Lists define a sequential collection where there is a first item and each item (except the last) has a natural successor.
- Lists can be used to store both sorted and unsorted sequences. Selection sort is a $\Theta(n^2)$ algorithm for sorting a sequence.
- Python lists are implemented using arrays of references. When a list grows too large for the current array, Python automatically allocates a new larger one. This technique allows `append` operations to be done in $\Theta(1)$ (amortized) time, but operations that insert or delete items in the midst of the list require $\Theta(n)$ time.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

- A Python dictionary is a container object that implements a general mapping.
- Dictionaries are implemented with hash tables. Hash tables allow for very efficient lookup, insertion, and deletion of new mappings, but do not preserve ordering (sequence) of the items.
- A Markov chain is a mathematical model that predicts the next item in a sequence based on a fixed window of immediately preceding items. It is sometimes used as a simple model of natural language for language processing applications.

3.8 Exercises

True/False Questions

1. Python is the only high-level language that has a built-in container type for sequential collections.
2. The indexing operation on lists returns a sublist of the original.
3. The constructor for the `Deck` class presented in the chapter creates a deck of cards that is randomly ordered.
4. Instances of Python classes that implement the necessary hook methods can be compared using the standard relational operators (such as $<$, $==$, and $>$).
5. Python lists are implemented using contiguous arrays.
6. A Python list is a homogeneous container.
7. Arrays do not allow efficient random access.
8. On average, appending to the end of a Python list is a $\Theta(n)$ operation.
9. Inserting into the middle of Python list is a $\Theta(n)$ operation.
10. `Card(6, "c") < Card(3, "s")`
11. Python is unique in that it has a built-in container type that implements a general mapping (dictionaries).
12. Python dictionary keys must be immutable objects.
13. Looking up an item in a Python dictionary is a $\Theta(n)$ operation.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

Multiple Choice Questions

1. Which of the following is not true of Python lists?
   a) They are implemented underneath as contiguous arrays.
   b) All of the items in a list must be of the same type.
   c) They can grow and shrink dynamically.
   d) They allow for efficient random access.

2. Which of the following is a $\Theta(n)$ operation?
   a) Appending to the end of a Python list.
   b) Sorting a list with selection sort.
   c) Deleting an item from the middle of a Python list.
   d) Finding the ith item in a Python list.

3. Which of the following is not a method of the `Deck` class presented in the chapter?
   a) `size`
   b) `shuffle`
   c) `deal`
   d) All of the above are methods of the class.

4. Which of the following is not a method of the `Hand` class presented in the chapter?
   a) `add`
   b) `sort`
   c) `deal`
   d) All of the above are methods of the class.

5. What is the time efficiency of the selection sort algorithm?
   a) $\Theta(\log n)$ b) $\Theta(n \log n)$ c) $\Theta(n)$ d) $\Theta(n^2)$

6. What is the time efficiency of the Python built-in list method `sort`?
   a) $\Theta(\log n)$ b) $\Theta(n \log n)$ c) $\Theta(n)$ d) $\Theta(n^2)$

7. What is the time efficiency of the operation `max(myList)`?
   a) $\Theta(\log n)$ b) $\Theta(n \log n)$ c) $\Theta(n)$ d) $\Theta(n^2)$

8. What operation is not supported for Python dictionaries?
   a) Item insertion
   b) Item deletion
   c) Item lookup
   d) Item ordering (sorting)

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

9. Which of the following is not true of Python dictionaries?
   a) They are implemented as hash tables.
   b) Values must be immutable.
   c) Lookup is very efficient.
   d) All of the above are true.

10. A trigram model of natural language
    a) uses a prefix of three words to predict the next word.
    b) uses a prefix of two words to predict the next word.
    c) is more useful than a Markov model.
    d) is used to send money overseas.

Short-Answer Questions

1. Using the `Deck` and `Hand` classes from this chapter, write snippets of code to do each of the following:

   a) Print out the names of all 52 cards.

   b) Print out the names of 13 random cards.

   c) Choose 13 cards at random from a 52-card deck and show the cards in value order (Bridge hand order).

   d) Deal and display four 13-card hands dealt from a shuffled deck.

2. What is the run-time efficiency (Θ) of the two shuffling algorithms discussed in the chapter (using two lists vs. in place). The discussion suggested that the latter is more efficient. Is this consistent with your Θ analysis? Explain.

3. Suppose you are involved in designing a system that must maintain information about a large number of individuals (for example, customer records or health records). Each person will be represented with an object that contains all of their critical information. Your job is to design a container class to hold all of these records. The following operations must be supported:

   `add(person)` – adds `person` object to the collection

   `remove(name)` – removes the person named `name` from the collection.

   `lookup(name)` – returns the record for the person named `name`.

   `list_all` – returns a list of all the records in the collection in order by name.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

For each of the following ways of organizing the data, give an analysis of the efficiency of the above operations. You should justify each of your analyses with a sentence or two explaining the algorithm that would achieve that efficiency. Try to come up with the best approach for each organizational strategy.

(a) The objects are stored in a Python list in the order that they are added.

(b) The objects are stored in a Python list in order by name.

(c) The objects are stored in a Python dictionary indexed by name.

4. Python has a `set` type that efficiently implements mathematical sets. You can get information on this container class by consulting reference documents or typing `help(set)` at a Python prompt. Suppose you are implementing your own `Set` class that includes `add`, `remove`, `clear`, `__contains__`, `intersection`, `union`, and `difference` operations. Utilizing each of the following concrete data structures, explain how you would implement the required operations and provide an analysis of the run-time efficiency of each operation.

   (a) an unordered Python list.

   (b) a sorted Python list.

   (c) a Python dictionary. (Note: the elements of the set will be the keys, you can just use `None` or `True` as the value.)

5. Suppose you were using a language that had dictionaries, but not lists/arrays. How would you implement a sequential collection? Analyze the efficiency of operations in the basic list ADT using your approach.

Programming Exercises

1. Modify the `Deck` class to keep track of the current size of the deck using an instance variable. Does this change the run-time efficiency of the `size` operation? Do a bit of research to answer this question.

2. Look into the functions provided by the Python `random` module to simplify the shuffling code in the `Deck` class.

3. Suppose we want to be able to place cards back into a deck. Modify the `Deck` class to include the operations `addTop`, `addBottom`, and `addRandom` (the last one inserts the card at a random location in the deck).

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

4. Instead of shuffling a deck of cards, another way to get a random distribution is to deal cards from random locations of an ordered deck. Implement a `Deck` class that uses this approach. Analyze the efficiency of the operations you provide.

5. It can be inconvenient to test programs involving decks of cards if cards are always dealt in random order. One solution is to allow the deck to be "stacked" in a particular order. Design a `Deck` class that allows the contents of the deck to be read from a file.

6. Modify the `sort` method in the `Hand` class so that it sorts the hand "in place." Hint: look at the in-place shuffling algorithm.

7. Another way to put a hand in order is to place each card into its proper location as it is added to the hand. This algorithm is called an *insertion sort*. Implement a version of `Hand` that uses this method to keep the hand in order.

8. Implement an extended `Deck` class with operations suitable for playing the card game war. You will need to be able to create an empty deck and place cards into it.

9. Write a program to play the following simple solitaire game. N cards are dealt face up onto the table. If two cards have a matching rank, new cards are dealt face up on top of them. Dealing continues until the deck is empty or no two stacks have matching ranks. The player wins if all the cards are dealt. Run simulations to find the probability of winning with various values of N.

10. Write a program that deals and evaluates poker hands.

11. Write a program to simulate the game of blackjack.

12. Write a program to deal and evaluate bridge hands to determine if they have an opening bid.

13. Modify the Markov gibberish generator so that it works at the level of characters rather than words. Note: you should not need to modify the class to do this, only how it is used.

14. Extend the Markov gibberish generator to allow the size of the prefix to be determined when the model is created. The constructor will take a parameter specifying the length of the prefix. Experiment with different prefix lengths on texts of various size to see what happens. Combining this with the previous project, you can produce a very versatile and entertaining gibberish generator.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

15. Write your own dictionary class that implements the various operations of the mapping ADT. Use a list of pairs as your concrete representation. Write suitable tests for your class and also provide a theta analysis of each operation.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

Chapter 4 Linked Structures and Iterators

Objectives

- To understand Python's memory model and the concepts of names and references.
- To examine different designs for lists, evaluate when each one is appropriate, and analyze the efficiency of the methods for each implementation.
- To learn how to write linked structures in Python.
- To understand the iterator design pattern and learn how to write iterators for container classes in Python.

4.1 Overview

When you first began learning Python, you may not have concerned yourself with the details of exactly how variables and their values are stored internally by the Python interpreter. For many simple programs, all you need to know is that variables are used to store values; however, as you write larger programs and begin to use more advanced features, it's important to understand exactly what the Python interpreter is doing when you assign a variable name to a value (an object). Understanding these details will help you avoid certain kinds of mistakes, allow you to better understand the efficiency of your code, and open the door to new ways of implementing data structures. It will also make it easier for you to learn other programming languages that support a similar memory model and understand the trade-offs when you learn languages with differing models.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

After we cover the details of Python's memory model, we will use that information to implement lists in a new way, using a so-called *linked* structure. The linked implementation makes some operations more efficient and other operations less efficient than they are for the built-in Python list. Understanding these trade-offs will allow you to choose the appropriate implementation techniques depending on what operations are needed by your application. Along the way, we will also discuss the iterator pattern, a technique that allows client programs to access items in a collection without making any assumptions about how the collection is implemented.

If you already understand Python references and Python's memory model, you may be tempted to skip the next section; however, we suggest you read through it, as these concepts are crucial for understanding many of the topics covered later. Unless you are a Python expert, you will likely learn something new in this material.

4.2 The Python Memory Model

In traditional programming languages, variables are often thought of as being named memory locations. Applying that idea to Python, you might think of a variable in Python as a place, a sort of cubbyhole, corresponding to a location in the computer's memory where you can store an object. This way of thinking will work pretty well for simple programs, but it's not a very accurate picture for how Python actually manages things. In order to avoid confusion with other languages, some people prefer to talk about *names* in Python rather than using the traditional term *variables.*

In Python, a name always refers to some object that is stored in memory. When you assign a Python name to an object, internally the Python interpreter uses a dictionary to map that name to the actual memory location where the object is stored. This dictionary that maintains the mapping from names into objects is called a *namespace.* If you later assign the same name to a different object, the namespace dictionary is modified so that it maps the name to the new memory location. We are going to walk through an interactive example that demonstrates what is happening "under the hood." The details of this are a bit tedious, but if you fully understand them, you will have a much easier time understanding many of the topics discussed later.

Let's start with a couple simple assignment statements.

```
>>> d = 'Dave'
>>> j = d
```

When the statement `d = 'Dave'` is executed, Python allocates a string object containing `Dave`. The assignment statement `j = d` causes the name `j` to refer to the

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

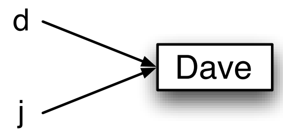

Figure 4.1: Two variables assigned to an object

same object as the name `d`; it does not create a new string object. A good analogy is to think of assignment as placing a sticky note with the name written on it onto the object. At this point, the data object `Dave` has two sticky notes on it: one with the name `d` and one with the name `j`. Figure 4.1 should help clarify what is happening. In diagrams such as this, we use an arrow as an intuitive way to to show the "value" of a reference; the computer actually stores a number that is the address of what our arrow is pointing to.

Of course, the Python interpreter can't use sticky notes, it keeps track of these associations internally using the namespace dictionary. We can actually access that dictionary with the built-in function called `locals()`.

```
>>> print locals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__',
'j': 'Dave', '__doc__': None, 'd': 'Dave'}
```

In this example, you can see that the local dictionary includes some Python special names (`__builtins__`, `__name__`, and `__doc__`) some of which you may recognize. We're not really concerned about those here. The point is that our assignment statements added the two names `d` and `j` to the dictionary. Notice, when the dictionary is printed, Python shows us the names as keys and a representation of the actual data objects to which they map as values. Keep in mind that the namespace dictionary actually stores the address of the object (also called a *reference* to the object). Since we usually care about the data, not locations, the Python interpreter automatically shows us a representation of what is stored at the address, not the address itself.

If, out of curiosity, we should want to find the actual address of an object, we can do that. The Python `id` function returns a unique identifier for each object; in most versions of Python, the `id` function returns the memory address where the object is stored.

```
>>> print id(d), id(j)
432128 432128
```

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

As you can see by the output of the id function, after the assignment statement j = d, both the names j and d refer to the same data object. Internally, the Python interpreter keeps track of the fact that there are two references to the string object containing "Dave"; this is referred to as the *reference count* for the object.

Continuing with the example, let's do a couple more assignments.

```
>>> j = 'John'
>>> print id(d), id(j)
432128 432256
>>> d = 'Smith'
>>> print id(d), id(j)
432224 432256
```

When we assign j = 'John', a new string object containing "John" is created. Using our sticky note analogy, we have moved the sticky note j to the newly created data object containing the string "John". The output of the id function following the statement j = 'John' shows that the name d still refers to the same object as before, but the name j now refers to an object at a different memory location. The reference count for each of the two string objects is now one.

The statement d = 'Smith' makes the name d refer to a new string object containing "Smith". Note that the address for the string object "Smith" is different from the string object "Dave". Again, the address that the name maps to changes when the name is assigned to a different object. This is an important point to note: *Assignment changes what object a variable refers to, it does not have any effect on the object itself.* In this case, the string "Dave" does not change into the string "Smith", but rather a new string object is created that contains "Smith".

At this point, nothing refers to the string "Dave" so its reference count is now zero. The Python interpreter automatically deallocates the memory for the string object containing "Dave", since there is no longer a way to access it. By deallocating objects that can no longer be accessed (when their reference count changes to zero), the Python interpreter is able to reuse the same memory locations for new objects later on. This process is known as *garbage collection*. Garbage collection adds some overhead to the Python interpreter that slows down execution. The gain is that it relieves the programmer from the burden of having to worry about memory allocation and deallocation, a process that is notoriously knotty and error prone in languages that do not have automatic memory management.

It is also possible for the programmer to explicitly remove the mapping for a given name.

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

```
>>> del d
>>> print locals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__',
'j': 'John', '__doc__': None}
```

The statement `del d` removes the name `d` from the namespace dictionary so it can no longer be accessed. Attempting to execute the statement `print d` now would cause a `NameError` exception to be raised just as if we had never assigned an object to `d`. Removing that name reduces the reference count for the string `"Smith"` from one to zero so it will now also be garbage collected.

There are a number of benefits to Python's memory model. Since a variable just contains a reference to an object, all variables are the same size (the standard address size of the computer, usually four or eight bytes). The data type information is stored with the object. The technical term for this is *dynamic typing*. That means the same name can refer to different types as a program executes and the name gets reassigned. This also makes it very easy for containers such as lists, tuples, and dictionaries to be heterogeneous (contain multiple types), since they also simply maintain references to (addresses of) the contained objects.

The Python memory model also makes assignment a very efficient operation. An expression in Python always evaluates to a reference to some object. Assigning the result to a name simply requires that the name be added to the namespace dictionary (if it's not already present) along with the four- or eight-byte reference. In a simple assignment like `j = d` the effect is to just copy `d`'s reference over to `j`'s namespace entry.

It should be clear by now that Python's memory model makes it trivial (usual, in fact) for multiple names to refer to the exact same object. This is known as *aliasing*, and it can lead to some interesting situations. When multiple names refer to the same object, changes to the object through one of the names will change the data that all the names refer to. Thus, changes to the data using one name will be visible via accesses through other names. Here's a simple illustration using lists.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = lst1
>>> lst2.append(4)
>>> lst1
[1, 2, 3, 4]
```

Since `lst1` and `lst2` refer to the same object, appending `4` to `lst2` also affects `lst1`. Unless you understand the underlying semantics it seems like `lst1` has changed "magically," since there are no intervening uses of `lst1` between the first and last lines of the interaction. Of course these potentially surprising results of

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

aliasing crop up only when the shared object happens to be mutable. Things like strings, ints, and floats simply can't change, so aliasing is not an issue for these types.

When we want to avoid the side effects of aliasing, we need to make separate copies of an object so that changes to one copy won't affect the others. Of course a complex object such as a list might itself contain references to other objects, and we have to decide how to handle those references in the copying process. There are two different types of copies known as *shallow copies* and *deep copies*. A shallow copy has its own top-level references, but those references refer to the same objects as the original. A deep copy is a completely separate copy that creates both new references and, where necessary, new data objects at all levels. The Python copy module contains useful functions for copying arbitrary Python objects. Here's an interactive example using lists to demonstrate.

```
>>> import copy
>>> b = [1, 2, [3, 4], 6]
>>> c = b
>>> d = copy.copy(b)     # creates a shallow copy
>>> e = copy.deepcopy(b) # creates a deep copy
>>> print b is c, b == c
True True
>>> print b is d, b == d
False True
>>> print b is e, b == e
False True
```

In this code, c is the same list as b, d is a shallow copy, and e is a deep copy. By the way, there are numerous ways to get a shallow copy of a Python list. We could also have used slicing (d = b[:]) or list construction (d = list(b)) to create a shallow copy.

So what's up with the output? The Python is operator tests whether two expressions refer to the exact same object, whereas the Python == operator tests to see if two expressions yield equivalent data. That means a is b implies a == b but not vice versa. In this example, you can see that assignment does not create a new object since b is c holds after the initial assignment. However both the shallow copy d created by slicing and the deep copy e are distinct new objects that contain equivalent data to b. While these copies contain equivalent data, their internal structures are not identical. As depicted in Figure 4.2, the shallow copy simply contains a copy of the references at the top level of the list, while the deep copy contains a copy of the mutable parts of the structure at all levels. Notice that the

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++

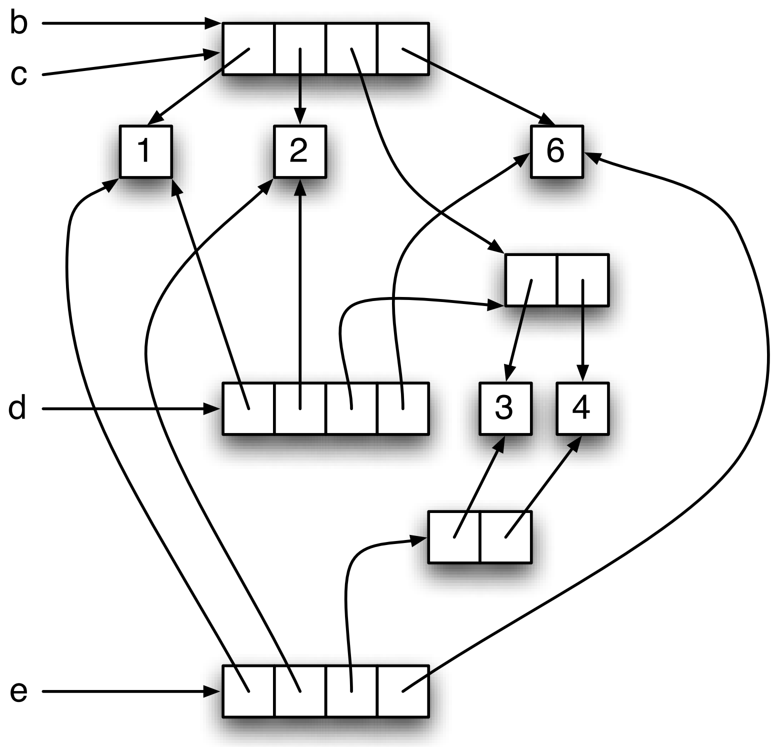

Figure 4.2: Pictorial representation of shallow and deep copies

02/10/2018 - RS0000000000000000000000381359 - Data Structures and Algorithms Using Python and C++