Lists

5

Up until this point, every variable that we have created has held one value. The value could be a integer, a Boolean, a string, or a value of some other type. While using one variable for each value is practical for small problems it quickly becomes untenable when working with larger amounts of data. Lists help us overcome this problem by allowing several, even many, values to be stored in one variable.

A variable that holds a list is created with an assignment statement, much like the variables that we have created previously. Lists are enclosed in square brackets, and commas are used to separate adjacent values within the list. For example, the following assignment statement creates a list that contains 4 floating-point numbers and stores it in a variable named `data`. Then the values are displayed by calling the `print` function. All 4 values are displayed when the `print` function executes because `data` is the entire list of values.

```
data = [2.71, 3.14, 1.41, 1.62]
print(data)
```

A list can hold zero or more values. The empty list, which has no values in it, is denoted by `[]` (an open square bracket immediately followed by a close square bracket). Much like an integer can be initialized to 0 and then have value added to it at a later point in the program, a list can be initialized to the empty list and then have items added it to it as the program executes.

5.1 Accessing Individual Elements

Each value in a list is referred to as an *element*. The elements in a list are numbered sequentially with integers, starting from 0. Each integer identifies a specific element in the list, and is referred to as the *index* for that element. In the previous code segment the element at index 0 in `data` is 2.71 while the element at index 3 is 1.62.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_5

An individual list element is accessed by using the list's name, immediately followed by the element's index enclosed in square brackets. For example, the following statements use this notation to display `3.14`. Notice that printing the element at index 1 displays the second element in the list because the first element in the list has index 0.

```
data = [2.71, 3.14, 1.41, 1.62]
print(data[1])
```

An individual list element can be updated using an assignment statement. The name of the list, followed by the element's index enclosed in square brackets, appears to the left of the assignment operator. The new value that will be stored at that index appears to the assignment operator's right. When the assignment statement executes, the element previously stored at the indicated index is overwritten with the new value. The other elements in the list are not impacted by this change.

Consider the following example. It creates a list that contains four elements, and then it replaces the element at index 2 with 2.30. When the `print` statement executes it will display all of the values in the list. Those values are 2.71, 3.14, 2.30 and 1.62.

```
data = [2.71, 3.14, 1.41, 1.62]
data[2] = 2.30
print(data)
```

5.2 Loops and Lists

A `for` loop executes once for each item in a collection. The collection can be a range of integers constructed by calling the `range` function. It can also be a list. The following example uses a `for` loop to total the values in `data`.

```
# Initialize data and total
data = [2.71, 3.14, 1.41, 1.62]
total = 0

# Total the values in data
for value in data:
  total = total + value

# Display the total
print("The total is", total)
```

This program begins by initializing `data` and `total` to the values shown. Then the `for` loop begins to execute. The first value in `data` is copied into `value` and then the body of the loop runs. It adds `value` to the total.

Once the body of the loop has executed for the first time control returns to the top of the loop. The second element in `data` is copied into `value`, and the loop body executes again which adds this new value to the total. This process continues until the

loop has executed once for each element in the list and the total of all of the elements has been computed. Then the result is displayed and the program terminates.

Sometimes loops are constructed which iterate over a list's indices instead of its values. To construct such a loop we need to be able to determine how many elements are in a list. This can be accomplished using the `len` function. It takes one argument, which is a list, and it returns the number of elements in the list.[1]

The `len` function can be used with the `range` function to construct a collection of integers that includes all of the indices for a list. This is accomplished by passing the length of the list as the only argument to `range`. A subset of the indices can be constructed by providing a second argument to `range`. The following program demonstrates this by using a `for` loop to iterate through all of `data`'s indices, except the first, to identify the position of the largest element in `data`.

```
# Initialize data and largest_pos
data = [1.62, 1.41, 3.14, 2.71]
largest_pos = 0

# Find the position of the largest element
for i in range(1, len(data)):
  if data[i] > data[largest_pos]:
    largest_pos = i

# Display the result
print("The largest value is", data[largest_pos], \
      "which is at index", largest_pos)
```

This program begins by initializing the `data` and `largest_pos` variables. Then the collection of values that will be used by the `for` loop is constructed using the `range` function. It's first argument is 1, and it second argument is the length of `data`, which is 4. As a result, `range` returns a collection of sequential integers from 1 up to and including 3, which is also the indices for all of the elements in `data`, except the first.

The `for` loop begins to execute by storing 1 into `i`. Then the loop body runs for the first time. It compares the value in `data` at index `i` to the value in `data` at index `largest_pos`. Since the element at index `i` is smaller, the `if` statement's condition evaluates to `False` and the body of the if statement is skipped.

Now control returns to the top of the loop. The next value in the range, which is 2, is stored into `i`, and the body of the loop executes for a second time. The value at index `i` is compared with the value at index `largest_pos`. Since the value at index `i` is larger, the body of the if statement executes, and `largest_pos` is set equal to `i`, which is 2.

The loop runs one more time with `i` equal to 3. The element at index `i` is less than the element at index `largest_pos` so the body of the if statement is skipped. Then the loop terminates and the program reports that the largest value is 3.14, which is at index 2.

[1] The `len` function returns 0 if the list passed to it is empty.

While loops can also be used when working with lists. For example, the following code segment uses a `while` loop to identify the index of the first positive value in a list. The loop uses a variable, `i`, which holds the indices of the elements in the list, starting from 0. The value in `i` increases as the program runs until either the end of the list is reached or a positive element is found.

```
# Initialize data
data = [0, -1, 4, 1, 0]

# Loop while i is a valid index and the value at index i is not a positive value
i = 0
while i < len(data) and data[i] <= 0:
  i = i + 1

# If i is less than the length of data then the loop terminated because a positive number was
# found.  Otherwise i will be equal to the length of data, indicating that a positive number
# was not found.
if i < len(data):
  print("The first positive number is at index", i)
else:
  print("The list does not contain a positive number")
```

When this program executes it begins by initializing `data` and `i`. Then the `while` loop's condition is evaluated. The value of `i`, which is 0, is less than the length of `data`, and the element at position `i` is 0, which is less than or equal to 0. As a result, the condition evaluates to `True`, the body of the loop executes, and the value of `i` increases from 0 to 1.

Control returns to the top of the `while` loop and its condition is evaluated again. The value stored in `i` is still less than the length of `data` and the value at position `i` in the list is still less than or equal to 0. As a result, the loop condition still evaluates to `True`. This causes the body of the loop to execute again, which increases the value of `i` from 1 to 2.

When `i` is 2 the loop condition evaluates to `False` because the element at position `i` is greater than or equal to 0. The loop body is skipped and execution continues with the `if` statement. It's condition evaluates to `True` because `i` is less than the length of `data`. As a result, the body of the `if` part executes and the index of the first positive number in `data`, which is 2, is displayed.

5.3 Additional List Operations

Lists can grow and shrink as a program runs. A new element can be inserted at any location in the list, and an element can be deleted based on its value or its index. Python also provides mechanisms for determining whether or not an element is present in a list, finding the index of the first occurrence of an element in a list, rearranging the elements in a list, and many other useful tasks.

Tasks like inserting a new element into a list and removing an element from a list are performed by applying a method to a list. Much like a function, a *method* is a collection of statements that can be called upon to perform a task. However, the syntax used to apply a method to a list is slightly different from the syntax used to call a function.

A method is applied to a list by using a statement that consists of a variable containing a list,[2] followed by a period, followed by the method's name. Like a function call, the name of the method is followed by parentheses that surround a comma separated collection of arguments. Some methods return a result. This result can be stored in a variable using an assignment statement, passed as an argument to another method or function call, or used as part of a calculation, just like the result returned by a function.

5.3.1 Adding Elements to a List

Elements can be added to the end of an existing list by calling the `append` method. It takes one argument, which is the element that will be added to the list. For example, consider the following program:

```
data = [2.71, 3.14, 1.41, 1.62]
data.append(2.30)
print(data)
```

The first line creates a new list of 4 elements and stores it in `data`. Then the `append` method is applied to `data` which increases its length from 4 to 5 by adding 2.30 to the end of the list. Finally, the list, which now contains 2.71, 3.14, 1.41, 1.62, and 2.30, is printed.

Elements can be inserted at any location in a list using the `insert` method. It requires two arguments, which are the index at which the element will be inserted and its value. When an element is inserted any elements to the right of the insertion point have their index increased by 1 so that there is an index available for the new element. For example, the following code segment inserts 2.30 in the middle of `data` instead of appending it to the end of the list. When this code segment executes it will display `[2.71, 3.14, 2.30, 1.41, 1.62]`.

```
data = [2.71, 3.14, 1.41, 1.62]
data.insert(2, 2.30)
print(data)
```

[2] Methods can also be applied to a list literal enclosed in square brackets using the same syntax, but there is rarely a need to do so.

5.3.2 Removing Elements from a List

The `pop` method is used to remove an element at a particular index from a list. The index of the element to remove is provided as an optional argument to `pop`. If the argument is omitted then `pop` removes the last element from the list. The `pop` method returns the value that was removed from the list as its only result. When this value is needed for a subsequent calculation it can be stored into a variable by calling `pop` on the right side of an assignment statement. Applying `pop` to an empty list is an error, as is attempting to remove an element from an index that is beyond the end of the list.

A value can also be removed from a list by calling the `remove` method. It's only argument is the value to remove (rather than the index of the value to remove). When the `remove` method executes it removes the first occurrence of its argument from the list. An error will be reported if the value passed to `remove` is not present in the list.

Consider the following example. It creates a list, and then removes two elements from it. When the first print statement executes it displays `[2.71, 3.14]` because 1.62 and 1.41 were removed from the list. The second print statement displays `1.41` because 1.41 was the last element in the list when the `pop` method was applied to it.

```
data = [2.71, 3.14, 1.41, 1.62]

data.remove(1.62)    # Remove 1.62 from the list
last = data.pop()    # Remove the last element from the list

print(data)
print(last)
```

5.3.3 Rearranging the Elements in a List

Sometimes a list has all of the correct elements in it, but they aren't in the order needed to solve a particular problem. Two elements in a list can be swapped using a series of assignment statements that read from and write to individual elements in the list, as shown in the following code segment.

```
# Create a list
data = [2.71, 3.14, 1.41, 1.62]

# Swap the element at index 1 with the element at index 3
temp = data[1]
data[1] = data[3]
data[3] = temp

# Display the modified list
print(data)
```

When these statements execute `data` is initialized to `[2.71, 3.14, 1.41, 1.62]`. Then the value at index 1, which is 3.14, is copied into `temp`. This is

followed by a line which copies the value at index 3 to index 1. Finally, the value in `temp` is copied into the list at index 3. When the print statement executes it displays `[2.71, 1.62, 1.41, 3.14]`.

There are two methods that rearrange the elements in a list. The `reverse` method reverses the order of the elements in the list, and the `sort` method sorts the elements into ascending order. Both `reverse` and `sort` can be applied to a list without providing any arguments.[3]

The following example reads a collection of numbers from the user and stores them in a list. Then it displays all of the values in sorted order.

```
# Create a new, empty list
values = []

# Read values from the user and store them in a list until a blank line is entered
line = input("Enter a number (blank line to quit): ")
while line != "":
  num = float(line)
  values.append(num)

  line = input("Enter a number (blank line to quit): ")

# Sort the values into ascending order
values.sort()

# Display the values
for v in values:
  print(v)
```

5.3.4 Searching a List

Sometimes we need to determine whether or not a particular value is present in a list. In other situations, we might want to determine the index of a value that is already known to be present in a list. Python's `in` operator and `index` method allow us to perform these tasks.

The `in` operator is used to determine whether or not a value is present in a list. The value that is being searched for is placed to the left of the operator. The list that is being searched is placed to the operator's right. Such an expression evaluates to `True` if the value is present in the list. Otherwise it evaluates to `False`.

The `index` method is used to identify the position of a particular value within a list. This value is passed to `index` as its only argument. The index of the first occurrence of the value in the list is returned as the method's result. It is an error to call the `index` method with an argument that is not present in the list. As a result,

[3] A list can only be sorted if all of the elements in it can be compared to one another with the less than operator. The less than operator is defined for many Python types include integers, floating-point numbers, strings, and lists, among others.

programmers sometimes use the `in` operator to determine whether or not a value is present in a list and then use the `index` method to determine it's location.

The following example demonstrates several of the methods and operators introduced in this section. It begins by reading integers from the user and storing them in a list. Then one additional integer is read from the user. The position of the first occurrence of this additional integer in the list of values is reported (if it is present). An appropriate message is displayed if the additional integer is not present in the list of values entered by the user.

```
# Read integers from the user until a blank line is entered and store them all in data
data = []
line = input("Enter an integer (blank line to finish): ")
while line != "":
  n = int(line)
  data.append(n)

  line = input("Enter an integer (blank line to finish): ")

# Read an additional integer from the user
x = int(input("Enter one additional integer: "))

# Display the index of the first occurrence of x (if it is present in the list)
if x in data:
  print("The first", x, "is at index", data.index(x))
else:
  print(x, "is not in the list")
```

5.4 Lists as Return Values and Arguments

Lists can be returned from functions. Like values of other types, a list is returned from a function using the `return` keyword. When the return statement executes, the function terminates and the list is returned to the location where the function was called. Then the list can be stored in a variable or used in a calculation.

Lists can also be passed as arguments to functions. Like values of other types, any lists being passed to a function are included inside the parentheses that follow the function's name when it is called. Each argument, whether it is a list or a value of another type, appears in the corresponding parameter variable inside the function.

Parameter variables that contain lists can be used in the body of a function just like parameter variables that contain values of other types. However, unlike an integer, floating-point number, string or Boolean value, changes made to a list parameter variable can impact the argument passed to the function, in addition to the value stored in the parameter variable. In particular, a change made to a list using a method (such as `append`, `pop` or `sort`) will change the value of both the parameter variable and the argument that was provided when the function was called.

Updates performed on individual list elements (where the name of the list, followed by an index enclosed in square brackets, appears on the left side of an assignment operator) also modify both the parameter variable and the argument that was provided when the function was called. However, assignments to the entire list (where only the name of the list appears to the left of the assignment operator) only impact the parameter variable. Such assignments do not impact the argument provided when the function was called.

The differences in behavior between list parameters and parameters of other types may seem arbitrary, as might the choice to have some changes apply to both the parameter variable and the argument while others only change the parameter variable. However, this is not the case. There are important technical reasons for these differences, but those details are beyond the scope of a brief introduction to Python.

5.5 Exercises

All of the exercises in this chapter should be solved using lists. The programs that you write will need to create lists, modify them, and locate values in them. Some of the exercises also require you to write functions that return lists or that take them as arguments.

Exercise 110: Sorted Order

(*Solved, 22 Lines*)

Write a program that reads integers from the user and stores them in a list. Your program should continue reading values until the user enters 0. Then it should display all of the values entered by the user (except for the 0) in ascending order, with one value appearing on each line. Use either the `sort` method or the `sorted` function to sort the list.

Exercise 111: Reverse Order

(*20 Lines*)

Write a program that reads integers from the user and stores them in a list. Use 0 as a sentinel value to mark the end of the input. Once all of the values have been read your program should display them (except for the 0) in reverse order, with one value appearing on each line.

Exercise 112: Remove Outliers

(*Solved, 44 Lines*)

When analysing data collected as part of a science experiment it may be desirable to remove the most extreme values before performing other calculations. Write a function that takes a list of values and an non-negative integer, `n`, as its parameters. The function should create a new copy of the list with the `n` largest elements and the `n` smallest elements removed. Then it should return the new copy of the list as the function's only result. The order of the elements in the returned list does not have to match the order of the elements in the original list.

Write a main program that demonstrates your function. It should read a list of numbers from the user and remove the two largest and two smallest values from it by calling the function described previously. Display the list with the outliers removed, followed by the original list. Your program should generate an appropriate error message if the user enters less than 4 values.

Exercise 113: Avoiding Duplicates

(*Solved, 21 Lines*)

In this exercise, you will create a program that reads words from the user until the user enters a blank line. After the user enters a blank line your program should display each word entered by the user exactly once. The words should be displayed in the same order that they were first entered. For example, if the user enters:

```
first
second
first
third
second
```

then your program should display:

```
first
second
third
```

Exercise 114: Negatives, Zeros and Positives

(*Solved, 36 Lines*)

Create a program that reads integers from the user until a blank line is entered. Once all of the integers have been read your program should display all of the negative numbers, followed by all of the zeros, followed by all of the positive numbers. Within each group the numbers should be displayed in the same order that they were entered by the user. For example, if the user enters the values 3, −4, 1, 0, −1, 0, and −2 then

your program should output the values −4, −1, −2, 0, 0, 3, and 1. Your program should display each value on its own line.

Exercise 115: List of Proper Divisors

(*36 Lines*)

A proper divisor of a positive integer, n, is a positive integer less than n which divides evenly into n. Write a function that computes all of the proper divisors of a positive integer. The integer will be passed to the function as its only parameter. The function will return a list containing all of the proper divisors as its only result. Complete this exercise by writing a main program that demonstrates the function by reading a value from the user and displaying the list of its proper divisors. Ensure that your main program only runs when your solution has not been imported into another file.

Exercise 116: Perfect Numbers

(*Solved, 35 Lines*)

An integer, n, is said to be *perfect* when the sum of all of the proper divisors of n is equal to n. For example, 28 is a perfect number because its proper divisors are 1, 2, 4, 7 and 14, and $1 + 2 + 4 + 7 + 14 = 28$.

Write a function that determines whether or not a positive integer is perfect. Your function will take one parameter. If that parameter is a perfect number then your function will return `True`. Otherwise it will return `False`. In addition, write a main program that uses your function to identify and display all of the perfect numbers between 1 and 10,000. Import your solution to Exercise 115 when completing this task.

Exercise 117: Only the Words

(*38 Lines*)

In this exercise you will create a program that identifies all of the words in a string entered by the user. Begin by writing a function that takes a string as its only parameter. Your function should return a list of the words in the string with the punctuation marks at the edges of the words removed. The punctuation marks that you must consider include commas, periods, question marks, hyphens, apostrophes, exclamation points, colons, and semicolons. Do not remove punctuation marks that appear in the middle of a word, such as the apostrophes used to form a contraction. For example, if your function is provided with the string `"Contractions include: don't, isn't, and wouldn't."` then your function should return the list `["Contractions", "include", "don't", "isn't", "and", "wouldn't"]`.

Write a main program that demonstrates your function. It should read a string from the user and then display all of the words in the string with the punctuation marks removed. You will need to import your solution to this exercise when completing Exercises 118 and 167. As a result, you should ensure that your main program only runs when your file has not been imported into another program.

Exercise 118: Word by Word Palindromes

(*34 Lines*)

Exercises 75 and 76 previously introduced the notion of a palindrome. Such palindromes examined the characters in a string, possibly ignoring spacing and punctuation marks, to see if the string was the same forwards and backwards. While palindromes are most commonly considered character by character, the notion of a palindrome can be extended to larger units. For example, while the sentence "Is it crazy how saying sentences backwards creates backwards sentences saying how crazy it is?" isn't a character by character palindrome, it is a palindrome when examined a word at a time (and when capitalization and punctuation are ignored). Other examples of word by word palindromes include "Herb the sage eats sage, the herb" and "Information school graduate seeks graduate school information".

Create a program that reads a string from the user. Your program should report whether or not the entered string is a word by word palindrome. Ignore spacing and punctuation when determining the result.

Exercise 119: Below and Above Average

(*44 Lines*)

Write a program that reads numbers from the user until a blank line is entered. Your program should display the average of all of the values entered by the user. Then the program should display all of the below average values, followed by all of the average values (if any), followed by all of the above average values. An appropriate label should be displayed before each list of values.

Exercise 120: Formatting a List

(*Solved, 41 Lines*)

When writing out a list of items in English, one normally separates the items with commas. In addition, the word "and" is normally included before the last item, unless the list only contains one item. Consider the following four lists:

```
apples
apples and oranges
apples, oranges and bananas
apples, oranges, bananas and lemons
```

Write a function that takes a list of strings as its only parameter. Your function should return a string that contains all of the items in the list, formatted in the manner described previously, as its only result. While the examples shown previously only include lists containing four elements or less, your function should behave correctly for lists of any length. Include a main program that reads several items from the user, formats them by calling your function, and then displays the result returned by the function.

Exercise 121: Random Lottery Numbers

(*Solved, 28 Lines*)

In order to win the top prize in a particular lottery, one must match all 6 numbers on his or her ticket to the 6 numbers between 1 and 49 that are drawn by the lottery organizer. Write a program that generates a random selection of 6 numbers for a lottery ticket. Ensure that the 6 numbers selected do not contain any duplicates. Display the numbers in ascending order.

Exercise 122: Pig Latin

(*32 Lines*)

Pig Latin is a language constructed by transforming English words. While the origins of the language are unknown, it is mentioned in at least two documents from the nineteenth century, suggesting that it has existed for more than 100 years. The following rules are used to translate English into Pig Latin:

- If the word begins with a consonant (including `y`), then all letters at the beginning of the word, up to the first vowel (excluding `y`), are removed and then added to the end of the word, followed by `ay`. For example, `computer` becomes `omputercay` and `think` becomes `inkthay`.
- If the word begins with a vowel (not including `y`), then `way` is added to the end of the word. For example, `algorithm` becomes `algorithmway` and `office` becomes `officeway`.

Write a program that reads a line of text from the user. Then your program should translate the line into Pig Latin and display the result. You may assume that the string entered by the user only contains lowercase letters and spaces.

Exercise 123: Pig Latin Improved

(*51 Lines*)

Extend your solution to Exercise 122 so that it correctly handles uppercase letters and punctuation marks such as commas, periods, question marks and exclamation marks. If an English word begins with an uppercase letter then its Pig Latin representation should also begin with an uppercase letter and the uppercase letter moved to the end of

the word should be changed to lowercase. For example, `Computer` should become `Omputercay`. If a word ends in a punctuation mark then the punctuation mark should remain at the end of the word after the transformation has been performed. For example, `Science!` should become `Iencescay!`.

Exercise 124: Line of Best Fit

(*41 Lines*)

A line of best fit is a straight line that best approximates a collection of n data points. In this exercise, we will assume that each point in the collection has an x coordinate and a y coordinate. The symbols $\bar{x}$ and $\bar{y}$ are used to represent the average x value in the collection and the average y value in the collection respectively. The line of best fit is represented by the equation $y = mx + b$ where m and b are calculated using the following formulas:

$$m = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{\sum x^2 - \frac{(\sum x)^2}{n}}$$

$$b = \bar{y} - m\bar{x}$$

Write a program that reads a collection of points from the user. The user will enter the first x coordinate on its own line, followed by the first y coordinate on its own line. Allow the user to continue entering coordinates, with the x and y values each entered on their own line, until your program reads a blank line for the x coordinate. Display the formula for the line of best fit in the form $y = mx + b$ by replacing m and b with the values calculated by the preceding formulas. For example, if the user inputs the coordinates (1, 1), (2, 2.1) and (3, 2.9) then your program should display $y = 0.95x + 0.1$.

Exercise 125: Shuffling a Deck of Cards

(*Solved, 49 Lines*)

A standard deck of playing cards contains 52 cards. Each card has one of four suits along with a value. The suits are normally spades, hearts, diamonds and clubs while the values are 2 through 10, Jack, Queen, King and Ace.

Each playing card can be represented using two characters. The first character is the value of the card, with the values 2 through 9 being represented directly. The characters "T", "J", "Q", "K" and "A" are used to represent the values 10, Jack, Queen, King and Ace respectively. The second character is used to represent the suit of the card. It is normally a lowercase letter: "s" for spades, "h" for hearts, "d" for diamonds and "c" for clubs. The following table provides several examples of cards and their two-character representations.

Card	Abbreviation
Jack of spades	Js
Two of clubs	2c
Ten of diamonds	Td
Ace of hearts	Ah
Nine of spades	9s

Begin by writing a function named `createDeck`. It will use loops to create a complete deck of cards by storing the two-character abbreviations for all 52 cards into a list. Return the list of cards as the function's only result. Your function will not require any parameters.

Write a second function named `shuffle` that randomizes the order of the cards in a list. One technique that can be used to shuffle the cards is to visit each element in the list and swap it with another random element in the list. You must write your own loop for shuffling the cards. You cannot make use of Python's built-in shuffle function.

Use both of the functions described in the previous paragraphs to create a main program that displays a deck of cards before and after it has been shuffled. Ensure that your main program only runs when your functions have not been imported into another file.

A good shuffling algorithm is unbiased, meaning that every different arrangement of the elements is equally probable when the algorithm completes. While the approach described earlier in this problem suggested visiting each element in sequence and swapping it with an element at a random index, such an algorithm is biased. In particular, elements that appear later in the original list are more likely to end up later in the shuffled list. Counterintuitively, an unbiased shuffle can be achieved by visiting each element in sequence and swapping it to a random index between the position of the current element and the end of the list instead of randomly selecting any index.

Exercise 126: Dealing Hands of Cards

(*44 Lines*)

In many card games each player is dealt a specific number of cards after the deck has been shuffled. Write a function, `deal`, which takes the number of hands, the number of cards per hand, and a deck of cards as its three parameters. Your function should return a list containing all of the hands that were dealt. Each hand will be represented as a list of cards.

When dealing the hands, your function should modify the deck of cards passed to it as a parameter, removing each card from the deck as it is added to a player's hand. When cards are dealt, it is customary to give each player a card before any

player receives an additional card. Your function should follow this custom when constructing the hands for the players.

Use your solution to Exercise 125 to help you construct a main program that creates and shuffles a deck of cards, and then deals out four hands of five cards each. Display all of the hands of cards, along with the cards remaining in the deck after the hands have been dealt.

Exercise 127: Is a List already in Sorted Order?

(*41 Lines*)

Write a function that determines whether or not a list of values is in sorted order (either ascending or descending). The function should return `True` if the list is already sorted. Otherwise it should return `False`. Write a main program that reads a list of numbers from the user and then uses your function to report whether or not the list is sorted.

> Make sure you consider these questions when completing this exercise: Is a list that is empty in sorted order? What about a list containing one element?

Exercise 128: Count the Elements

(*Solved, 48 Lines*)

Python's standard library includes a method named `count` that determines how many times a specific value occurs in a list. In this exercise, you will create a new function named `countRange`. It will determine and return the number of elements within a list that are greater than or equal to some minimum value and less than some maximum value. Your function will take three parameters: the list, the minimum value and the maximum value. It will return an integer result greater than or equal to 0. Include a main program that demonstrates your function for several different lists, minimum values and maximum values. Ensure that your program works correctly for both lists of integers and lists of floating-point numbers.

Exercise 129: Tokenizing a String

(*Solved, 47 Lines*)

Tokenizing is the process of converting a string into a list of substrings, known as tokens. In many circumstances, a list of tokens is far easier to work with than the original string because the original string may have irregular spacing. In some cases substantial work is also required to determine where one token ends and the next one begins.

In a mathematical expression, tokens are items such as operators, numbers and parentheses. The operator symbols that we will consider in this (and subsequent) problems are `*`, `/`, `^`, – and +. Operators and parentheses are easy to identify because

the token is always a single character, and the character is never part of another token. Numbers are slightly more challenging to identify because the token may consist of multiple characters. Any sequence of consecutive digits should be treated as one number token.

Write a function that takes a string containing a mathematical expression as its only parameter and breaks it into a list of tokens. Each token should be a parenthesis, an operator, or a number. (For simplicity we will only work with integers in this problem). Return the list of tokens as the function's only result.

You may assume that the string passed to your function always contains a valid mathematical expression consisting of parentheses, operators and integers. However, your function must handle variable amounts of whitespace (including no whitespace) between these elements. Include a main program that demonstrates your tokenizing function by reading an expression from the user and printing the list of tokens. Ensure that the main program will not run when the file containing your solution is imported into another program.

Exercise 130: Unary and Binary Operators

(*Solved, 45 Lines*)

Some mathematical operators are unary while others are binary. Unary operators act on one value while binary operators act on two. For example, in the expression `2 * -3` the `*` is a binary operator because it acts on both the 2 and the −3 while the `-` is a unary operator because it only acts on the 3.

An operator's symbol is not always sufficient to determine whether it is unary or binary. For example, while the `-` operator was unary in the previous expression, the same character is used to represent the binary `-` operator in an expression such as `2 - 3`. This ambiguity, which is also present for the + operator, must be removed before other interesting operations can be performed on a list of tokens representing a mathematical expression.

Create a function that identifies unary + and - operators in a list of tokens and replaces them with `u+` and `u-` respectively. Your function will take a list of tokens for a mathematical expression as its only parameter and return a new list of tokens where the unary + and - operators have been substituted as its only result. A + or - operator is unary if it is the first token in the list, or if the token that immediately precedes it is an operator or open parenthesis. Otherwise the operator is binary.

Write a main program that demonstrates that your function works correctly by reading, tokenizing, and marking the unary operators in an expression entered by the user. Your main program should not execute when your function is imported into another program.

Exercise 131: Infix to Postfix

(*63 Lines*)

Mathematical expressions are often written in infix form, where operators appear between the operands on which they act. While this is a common form, it is also possible to express mathematical expressions in postfix form, where the operator appears after all of its operands. For example, the infix expression `3 + 4` is written as `3 4 +` in postfix form. One can convert an infix expression to postfix form using the following algorithm:

```
Create a new empty list, operators
Create a new empty list, postfix

For each token in the infix expression
    If the token is an integer then
        Append the token to postfix
    If the token is an operator then
        While operators is not empty and
                the last item in operators is not an open parenthesis and
                precedence(token) < precedence(last item in operators) do
            Remove the last item from operators and append it to postfix
        Append the token to operators
    If the token is an open parenthesis then
        Append the token to operators
    If the token is a close parenthesis then
        While the last item in operators is not an open parenthesis do
            Remove the last item from operators and append it to postfix
        Remove the open parenthesis from operators

While operators is not the empty list do
    Remove the last item from operators and append it to postfix

Return postfix as the result of the algorithm
```

Use your solutions to Exercises 129 and 130 to tokenize a mathematical expression and identify any unary operators in it. Then use the algorithm above to transform the expression from infix form to postfix form. Your code that implements the preceding algorithm should reside in a function that takes a list of tokens representing an infix expression (with the unary operators marked) as its only parameter. It should return a list of tokens representing the equivalent postfix expression as its only result. Include a main program that demonstrates your infix to postfix function by reading an expression from the user in infix form and displaying it in postfix form.

The purpose of converting from infix form to postfix form will become apparent when you read Exercise 132. You may find your solutions to Exercises 96 and 97 helpful when completing this problem. While you should be able to use your solution to Exercise 96 without any modifications, your solution to Exercise 97 will need to be extended so that it returns the correct precedence for the unary operators. The unary operators should have higher precedence than multiplication and division, but lower precedence than exponentiation.

Exercise 132: Evaluate Postfix

(*63 Lines*)

Evaluating a postfix expression is easier than evaluating an infix expression because it does not contain any parentheses and there are no operator precedence rules to consider. A postfix expression can be evaluated using the following algorithm:

Create a new empty list, *values*

For each token in the postfix expression
  If the token is a number **then**
    Convert it to an integer and append it to *values*
  Else if the token is a unary minus **then**
    Remove an item from the end of *values*
    Negate the item and append the result of the negation to *values*
  Else if the token is a binary operator **then**
    Remove an item from the end of *values* and call it *right*
    Remove an item from the end of *values* and call it *left*
    Compute the result of applying the operator to *left* and *right*
    Append the result to *values*

Return the first item in *values* as the value of the expression

Write a program that reads a mathematical expression in infix form from the user, converts it to postfix form, evaluates it, and displays its value. Use your solutions to Exercises 129, 130 and 131, along with the algorithm shown above, to solve this problem.

The algorithms provided in Exercises 131 and 132 do not perform any error checking. As a result, your programs may crash or generate incorrect results if you provide them with invalid input. The algorithms presented in these exercises can be extended to detect invalid input and respond to it in a reasonable manner. Doing so is left as an independent study exercise for the interested student.

Exercise 133: Does a List Contain a Sublist?

(*44 Lines*)

A sublist is a list that is part of a larger list. A sublist may be a list containing a single element, multiple elements, or even no elements at all. For example, `[1]`, `[2]`, `[3]` and `[4]` are all sublists of `[1, 2, 3, 4]`. The list `[2, 3]` is also a sublist of `[1, 2, 3, 4]`, but `[2, 4]` is not a sublist `[1, 2, 3, 4]` because the elements `2` and `4` are not adjacent in the longer list. The empty list is a sublist of

any list. As a result, `[]` is a sublist of `[1, 2, 3, 4]`. A list is a sublist of itself, meaning that `[1, 2, 3, 4]` is also a sublist of `[1, 2, 3, 4]`.

In this exercise you will create a function, `isSublist`, that determines whether or not one list is a sublist of another. Your function should take two lists, `larger` and `smaller`, as its only parameters. It should return `True` if and only if `smaller` is a sublist of `larger`. Write a main program that demonstrates your function.

Exercise 134: Generate All Sublists of a List

(*Solved, 41 Lines*)

Using the definition of a sublist from Exercise 133, write a function that returns a list containing every possible sublist of a list. For example, the sublists of `[1, 2, 3]` are `[]`, `[1]`, `[2]`, `[3]`, `[1, 2]`, `[2, 3]` and `[1, 2, 3]`. Note that your function will always return a list containing at least the empty list because the empty list is a sublist of every list. Include a main program that demonstrate your function by displaying all of the sublists of several different lists.

Exercise 135: The Sieve of Eratosthenes

(*Solved, 33 Lines*)

The Sieve of Eratosthenes is a technique that was developed more than 2,000 years ago to easily find all of the prime numbers between 2 and some limit, say 100. A description of the algorithm follows:

Write down all of the numbers from 0 to the limit
Cross out 0 and 1 because they are not prime

Set *p* equal to 2
While *p* is less than the limit **do**
  Cross out all multiples of *p* (but not *p* itself)
  Set *p* equal to the next number in the list that is not crossed out

Report all of the numbers that have not been crossed out as prime

The key to this algorithm is that it is relatively easy to cross out every *n*th number on a piece of paper. This is also an easy task for a computer—a for loop can simulate this behavior when a third parameter is provided to the `range` function. When a number is crossed out, we know that it is no longer prime, but it still occupies space on the piece of paper, and must still be considered when computing later prime numbers. As a result, you should **not** simulate crossing out a number by removing it from the list. Instead, you should simulate crossing out a number by replacing it with 0. Then, once the algorithm completes, all of the non-zero values in the list are prime.

Create a Python program that uses this algorithm to display all of the prime numbers between 2 and a limit entered by the user. If you implement the algorithm correctly you should be able to display all of the prime numbers less than 1,000,000 in a few seconds.

This algorithm for finding prime numbers is not Eratosthenes' only claim to fame. His other noteworthy accomplishments include calculating the circumference of the Earth and the tilt of the Earth's axis. He also served as the Chief Librarian at the Library of Alexandria.