

Lists help programmers manage larger amounts of data by allowing several (or even many) values to be stored in one variable. This makes it practical to solve larger problems that involve many data values. To solve the exercises in this chapter you should expect to:

- Create a variable that holds a list of values
- Modify a list by appending, inserting, updating and deleting elements
- Search a list for a value
- Display some or all of the values in a list
- Write a function that takes a list as a parameter
- Write a function that returns a list as its result

Exercise 104: Sorted Order

(Solved—21 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 order from smallest to largest, with one value appearing on each line. Use either the `sort` method or the `sorted` function to sort the list.

Exercise 105: 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 106: Remove Outliers

(Solved—43 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. Your function should read a list of numbers from the user and remove the two largest and two smallest values from it. 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 107: 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 entered. For example, if the user enters:

```
first
second
first
third
second
```

then your program should display:

```
first
second
third
```

Exercise 108: Negatives, Zeros and Positives

(Solved—38 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 109: 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 110: 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 109 when completing this task.

Exercise 111: 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 of text 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 remove include commas, periods, question marks, hyphens, apostrophes, exclamation points, colons, and semicolons. Do not remove punctuation marks that appear in the middle of a words, such as the apostrophes used to form a contraction. For example, if your function is provided with the string "Examples of contractions include: don't, isn't, and wouldn't." then your function should return the list ["Examples", "of", "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 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 Exercise 158. As a result, you should ensure that your main program only runs when your file has not been imported into another program.

Exercise 112: 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 113: Formatting a List

(Solved—43 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 114: 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 115: 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 116: Pig Latin Improved

(51 Lines)

Extend your solution to Exercise 115 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 117: 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 x part of the first coordinate on its own line, followed by the y part of the first coordinate on its own line. Allow the user to continue entering coordinates, with the x and y parts 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 you calculated using 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 118: Shuffling a Deck of Cards

(Solved—48 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 take 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.

Exercise 119: 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 118 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 120: 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 121: Count the Elements

(Solved—49 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` which determines and returns 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 122: Tokenizing a String

(Solved—64 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. Some tokens, such as `*`, `/`, `^`, `(` and `)` are easy to identify because the token is a single character, and the character is never part of another token. The `+` and `-` symbols are a little bit more challenging to handle because they might represent the addition or subtraction operator, or they might be part of a number token.

Hint: A `+` or `-` is an operator if the non-whitespace character immediately before it is part of a number, or if the non-whitespace character immediately before it is a close parenthesis. Otherwise it is part of a number.

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 with an optional leading `+` or `-` (for simplicity we will only work with integers in this problem). Return the list of tokens as the function's 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 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 123: Infix to Postfix

(62 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 both 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**

 Add the token to the end of *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 add it to *postfix*

 Add token to the end of *operators*

If the token is an open parenthesis **then**

 Add token to the end of *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 add it to *postfix*

 Remove the open parenthesis from *operators*

While *operators* is not the empty list **do**

 Remove the last item from *operators* and add it to *postfix*

Return *postfix* as the result of the algorithm

Use your solution to Exercise 122 to tokenize a mathematical expression. 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 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 124. You may find your solutions to Exercises 90 and 91 helpful when completing this problem.

The algorithms provided in Exercises 123 and 124 do not perform any error checking. As a result, you may crash your program or receive incorrect results if you provide them with invalid input. These algorithms 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 124: Evaluate Postfix

(58 Lines)

Evaluating a postfix expression is easier than evaluating an infix expression because it does not contain any brackets 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 add it to the end of *values*

Else

 Remove an item from the end of *values* and call it *right*

 Remove an item from the end of *values* and call it *left*

 Apply the operator to *left* and *right*

 Append the result to the end of *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, evaluates it, and displays its value. Uses your solutions to Exercises 122 and 123 along with the algorithm shown above to solve this problem.

Exercise 125: Does a List contain a Sublist?

(44 Lines)

A sublist is a list that makes up 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 126: Generate All Sublists of a List

(Solved—40 Lines)

Using the definition of a sublist from Exercise 125, 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 127: 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.

Dictionaries are another data structure that Python programmers can use to manage larger amounts of data. While many of the exercises in this chapter can be solved with lists or if statements, most (or even all) of them have solutions that are well suited to dictionaries. As a result, you should use dictionaries to solve all of these exercises instead of (or in addition to) using the constructs that you have been introduced to in the previous chapters. Completing the exercises in this chapter will help you learn to:

- Create a new variable that holds a dictionary
- Add a key-value pair to a dictionary
- Update the value associated with a key in a dictionary
- Iterate over all of the keys and/or values in a dictionary
- Write functions that take dictionaries as parameters

Exercise 128: Reverse Lookup

(Solved—40 Lines)

Write a function named `reverseLookup` that finds all of the keys in a dictionary that map to a specific value. The function will take the dictionary and the value to search for as its only parameters. It will return a (possibly empty) list of keys from the dictionary that map to the provided value.

Include a main program that demonstrates the `reverseLookup` function as part of your solution to this exercise. Your program should create a dictionary and then show that the `reverseLookup` function works correctly when it returns multiple keys, a single key, and no keys. Ensure that your main program only runs when the file containing your solution to this exercise has not been imported into another program.

Exercise 129: Two Dice Simulation

(Solved—42 Lines)

In this exercise you will simulate 1,000 rolls of two dice. Begin by writing a function that simulates rolling a pair of six-sided dice. Your function will not take any parameters. It will return the total that was rolled on two dice as its only result.

Write a main program that uses your function to simulate rolling two six-sided dice 1,000 times. As your program runs, it should count the number of times that each total occurs. Then it should display a table that summarizes this data. Express the frequency for each total as a percentage of the total number of rolls. Your program should also display the percentage expected by probability theory for each total. Sample output is shown below.

Total	Simulated Percent	Expected Percent
2	2.90	2.78
3	6.90	5.56
4	9.40	8.33
5	11.90	11.11
6	14.20	13.89
7	14.20	16.67
8	15.00	13.89
9	10.50	11.11
10	7.90	8.33
11	4.50	5.56
12	2.60	2.78

Exercise 130: Text Messaging

(21 Lines)

On some basic cell phones, text messages can be sent using the numeric keypad. Because each key has multiple letters associated with it, multiple key presses are needed for most letters. Pressing the number once generates the first letter on the key. Pressing the number 2, 3, 4 or 5 times generates the second, third, fourth or fifth character listed for that key.

Key	Symbols
1	. , ? ! :
2	A B C
3	D E F
4	G H I

5	J K L
6	M N O
7	P Q R S
8	T U V
9	W X Y Z
0	space

Write a program that displays the key presses that must be made to enter a text message read from the user. Construct a dictionary that maps from each letter or symbol to the key presses. Then use the dictionary to generate and display the presses for the user’s message. For example, if the user enters `Hello, World!` then your program should output `4433555555666110966677755531111`. Ensure that your program handles both uppercase and lowercase letters. Ignore any characters that aren’t listed in the table above such as semicolons and brackets.

Exercise 131: Morse Code

(15 Lines)

Morse code is an encoding scheme that uses dashes and dots to represent numbers and letters. In this exercise, you will write a program that uses a dictionary to store the mapping from letters and numbers to Morse code. Use a period to represent a dot, and a hyphen to represent a dash. The mapping from letters and numbers to dashes and dots is shown in Table 6.1.

Your program should read a message from the user. Then it should translate each letter and number in the message to Morse code, leaving a space between each sequence of dashes and dots. Your program should ignore any characters that are not letters or numbers. The Morse code for `Hello, World!` is shown below:

. -.-. -.-. --- .- - - - .- . . . -..

Table 6.1 Morse Code Letters and Numbers

Letter	Code	Letter	Code	Letter	Code	Number	Code
A	.-	J	.- - -	S	...	1	.- - - -
B	- . . .	K	- . -	T	-	2	.. - - -
C	- . - .	L	.- . .	U	.. -	3	... - -
D	- . .	M	- -	V	... -	4 -
E	.	N	- .	W	.- -	5
F	.. - .	O	- - -	X	- . . -	6	-
G	- - .	P	.- - .	Y	- . - -	7	- - . . .
H	Q	- - . -	Z	- - . .	8	- - - . .
I	..	R	.- .	0	- - - - -	9	- - - - .

Morse code was originally developed in the nineteenth century for use over telegraph wires. It is still used today, over 160 years after it was first created.

Exercise 132: Postal Codes

(24 Lines)

In a Canadian postal code, the first, third and fifth characters are letters while the second, fourth and sixth characters are numbers. The province can be determined from the first character of a postal code, as shown in the following table. No valid postal codes currently begin with D, F, I, O, Q, U, W, or Z.

Province	First character(s)
Newfoundland	A
Nova Scotia	B
Prince Edward Island	C
New Brunswick	E
Quebec	G, H and J
Ontario	K, L, M, N and P
Manitoba	R
Saskatchewan	S
Alberta	T
British Columbia	V
Nunavut	X
Northwest Territories	X
Yukon	Y

The second character in a postal code identifies whether the address is rural or urban. If that character is a 0 then the address is rural. Otherwise it is urban.

Create a program that reads a postal code from the user and displays the province associated with it, along with whether the address is urban or rural. For example, if the user enters T2N 1N4 then your program should indicate that the postal code is for an urban address in Alberta. If the user enters X0A 1B2 then your program should indicate that the postal code is for a rural address in Nunavut or Northwest Territories. Use a dictionary to map from the first character of the postal code to the province name. Display a meaningful error message if the postal code begins with an invalid character.

Exercise 133: Write Out Numbers in English

(65 Lines)

While the popularity of cheques as a payment method has diminished in recent years, some companies still issue them to pay employees or vendors. The amount being paid normally appears on a cheque twice, with one occurrence written using digits, and the other occurrence written using English words. Repeating the amount in two different forms makes it much more difficult for an unscrupulous employee or vendor to modify the amount on the cheque before depositing it.

In this exercise, your task is to create a function that takes an integer between 0 and 999 as its only parameter, and returns a string containing the English words for that number. For example, if the parameter to the function is 142 then your function should return “one hundred forty two”. Use one or more dictionaries to implement your solution rather than large if/elif/else constructs. Include a main program that reads an integer from the user and displays its value in English words.

Exercise 134: Unique Characters

(Solved—14 Lines)

Create a program that determines and displays the number of unique characters in a string entered by the user. For example, `Hello, World!` has 10 unique characters while `zzz` has only one unique character. Use a dictionary or set to solve this problem.

Exercise 135: Anagrams

(Solved—39 Lines)

Two words are anagrams if they contain all of the same letters, but in a different order. For example, “evil” and “live” are anagrams because each contains one e, one i, one l, and one v. Create a program that reads two strings from the user, determines whether or not they are anagrams, and reports the result.

Exercise 136: Anagrams Again

(48 Lines)

The notion of anagrams can be extended to multiple words. For example, “William Shakespeare” and “I am a weakish speller” are anagrams when capitalization and spacing are ignored.

Extend your program from Exercise 135 so that it is able to check if two phrases are anagrams. Your program should ignore capitalization, punctuation marks and spacing when making the determination.

Exercise 137: Scrabble™ Score

(Solved—18 Lines)

In the game of Scrabble™, each letter has points associated with it. The total score of a word is the sum of the scores of its letters. More common letters are worth fewer points while less common letters are worth more points. The points associated with each letter are shown below:

One point	A, E, I, L, N, O, R, S, T and U
Two points	D and G
Three points	B, C, M and P
Four points	F, H, V, W and Y
Five points	K
Eight points	J and X
Ten points	Q and Z

Write a program that computes and displays the Scrabble™ score for a word. Create a dictionary that maps from letters to point values. Then use the dictionary to compute the score.

A Scrabble™ board includes some squares that multiply the value of a letter or the value of an entire word. We will ignore these squares in this exercise.

Exercise 138: Create a Bingo Card

(Solved—58 Lines)

A Bingo card consists of 5 columns of 5 numbers. The columns are labeled with the letters B, I, N, G and O. There are 15 numbers that can appear under each letter. In particular, the numbers that can appear under the B range from 1 to 15, the numbers that can appear under the I range from 16 to 30, the numbers that can appear under the N range from 31 to 45, and so on.

Write a function that creates a random Bingo card and stores it in a dictionary. The keys will be the letters B, I, N, G and O. The values will be the lists of five numbers that appear under each letter. Write a second function that displays the Bingo card with the columns labeled appropriately. Use these functions to write a program that

displays a random Bingo card. Ensure that the main program only runs when the file containing your solution has not been imported into another program.

You may be aware that Bingo cards often have a “free” space in the middle of the card. We won’t consider the free space in this exercise.

Exercise 139: Checking for a Winning Card

(102 Lines)

A winning Bingo card contains a line of 5 numbers that have all been called. Players normally mark the numbers that have been called by crossing them out or marking them with a Bingo dauber. In our implementation we will mark that a number has been called by replacing it with a 0 in the Bingo card dictionary.

Write a function that takes a dictionary representing a Bingo card as its only parameter. If the card contains a line of five zeros (vertical, horizontal or diagonal) then your function should return `True`, indicating that the card has won. Otherwise the function should return `False`.

Create a main program that demonstrates your function by creating several Bingo cards, displaying them, and indicating whether or not they contain a winning line. You should demonstrate your function with at least one card with a horizontal line, at least one card with a vertical line, at least one card with a diagonal line, and at least one card that has some numbers crossed out but does not contain a winning line. You will probably want to import your solution to Exercise 138 when completing this exercise.

Hint: Because there are no negative numbers on a Bingo card, finding a line of 5 zeros is the same problem as finding a line of 5 entries that sum to zero. You may find the summation problem easier to solve.

Exercise 140: Play Bingo

(88 Lines)

In this exercise you will write a program that simulates a game of Bingo for a single card. Begin by generating a list of all of the valid Bingo calls (B1 through O75). Once the list has been created you can randomize the order of its elements by calling the `shuffle` function in the `random` module. Then your program should consume calls out of the list, crossing out numbers on the card, until the card contains a crossed out line (horizontal, vertical or diagonal). Simulate 1,000 games and report the minimum, maximum and average number of calls that must be made before the card wins. Import your solutions to Exercises 138 and 139 when completing this exercise.