

# Programming Exercises

The exercises in this section are optional and do not report to the performance dashboard.

Instructors can decide whether to assign these exercises and students can check the correctness of their programs using the Check Exercise tool.

## Note

If the program prompts the user to enter a list of values, enter the values on one line separated by spaces.

## Sections 8.2–8.3

- \*8.1** (*Sum elements column by column*) Write a function that returns the sum of all the elements in a specified column in a matrix using the following header:

```
def sumColumn(m, columnIndex):
```

Write a test program that reads a  $3 \times 4$  matrix and displays the sum of each column.

- \*8.2** (*Sum the major diagonal in a matrix*) Write a function that sums all the numbers of the major diagonal in an  $n \times n$  matrix of integers using the following header:

```
def sumMajorDiagonal(m):
```

The major diagonal is the diagonal that runs from the top left corner to the bottom right corner in the square matrix. Write a test program that reads a  $4 \times 4$  matrix and displays the sum of all its elements on the major diagonal.

**\*8.3** (*Sort students by grades*) Rewrite [LiveExample 8.2](#), GradeExam.py, to display the students in increasing order of the number of correct answers.

**\*\*8.4** (*Compute the weekly hours for each employee*) Suppose the weekly hours for all employees are stored in a table. Each row records an employee's seven-day work hours with seven columns. For example, the following table stores the work hours for eight employees. Write a program that displays employees and their total hours in decreasing order of the total hours.

	<i>Su</i>	<i>M</i>	<i>T</i>	<i>W</i>	<i>Th</i>	<i>F</i>	<i>Sa</i>
Employee 0	2	4	3	4	5	8	8
Employee 1	7	3	4	3	3	4	4
Employee 2	3	3	4	3	3	2	2
Employee 3	9	3	4	7	3	4	1
Employee 4	3	5	4	3	6	3	8
Employee 5	3	4	4	6	3	4	4
Employee 6	3	7	4	8	3	8	4
Employee 7	6	3	5	9	2	7	9

- 8.5** (*Algebra: add two matrices*) Write a function to add two matrices. The header of the function is:

```
def addMatrix(a, b):
```

In order to be added, the two matrices must have the same dimensions and the same or compatible types of elements. Let **c** be the resulting matrix. Each element  $c_{ij}$  is  $a_{ij} + b_{ij}$ . For example, for two  $3 \times 3$  matrices **a** and **b**, **c** is:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13} + b_{13} \\ a_{21} + b_{21} & a_{22} + b_{22} & a_{23} + b_{23} \\ a_{31} + b_{31} & a_{32} + b_{32} & a_{33} + b_{33} \end{pmatrix}$$

Write a test program that prompts the user to enter two  $3 \times 3$  matrices and displays their sum.

The numbers in each matrix are entered in one line.

- \*\*8.6** (*Algebra: multiply two matrices*) Write a function to multiply two matrices. The header of the function is:

```
def multiplyMatrix(a, b)
```

To multiply matrix **a** by matrix **b**, the number of columns in **a** must be the same as the number of rows in **b**, and the two matrices must have elements of the same or compatible types. Let **c** be the result of the multiplication. Assume the column size of matrix **a** is **n**. Each element  $c_{ij}$  is  $a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \dots + a_{in} \times b_{nj}$ . For example, for two  $3 \times 3$  matrices **a** and **b**, **c** is:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

where  $c_{ij} = a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + a_{i3} \times b_{3j}$ .

Write a test program that prompts the user to enter two  $3 \times 3$  matrices and displays their product.

- \*8.7** (*Points nearest to each other*) The program in [Listing 8.3](#) finds the two points in a two-dimensional space nearest to each other. Revise the program so that it finds the two points in a three-dimensional space nearest to each other. Use a two-dimensional list to represent the points. Test the program using the following points:

```
points = [[-1, 0, 3], [-1, -1, -1], [4, 1, 1],
          [2, 0.5, 9], [3.5, 2, -1], [3, 1.5, 3], [-1.5, 4, 2],
          [5.5, 4, -0.5]]
```

The formula for computing the distance between two points  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$  in a three-dimensional space is  $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$ .

- \*\*8.8** (*All closest pairs of points*) Revise [LiveExample 8.4](#), `FindNearestPoints.py`, to find all the nearest pairs of points that have the same minimum distance.

- \*\*\*8.9** (*Game: play a tic-tac-toe game*) In a game of tic-tac-toe, two players take turns marking an available cell in a  $3 \times 3$  grid with their respective tokens (either X or O). When one player has placed three tokens in a horizontal, vertical, or diagonal row on the grid, the game is over and that player has won. A draw (no winner) occurs when all the cells in the grid have been filled with tokens and neither player has achieved a win. Create a program for playing tic-tac-toe.

The program prompts two players to alternately enter an X token and an O token.

Whenever a token is entered, the program redisplay the board on the console and

determines the status of the game (win, draw, or continue).

**\*8.10** (*Largest rows and columns*) Write a program that randomly fills in 0 s and 1 s into a  $4 \times 4$  matrix, prints the matrix, and finds the rows and columns with the most 1 s.

**\*\*8.11** (*Game: nine heads and tails*) Nine coins are placed in a  $3 \times 3$  matrix with some face up and some face down. You can represent the state of the coins with the values 0 (heads) and 1 (tails). Here are some examples:


0 0 0	1 0 1	1 1 0	1 0 1	1 0 0
0 1 0	0 0 1	1 0 0	1 1 0	1 1 1
0 0 0	1 0 0	0 0 1	1 0 0	1 1 0

Each state can also be represented using a binary number. For example, the preceding matrices correspond to the numbers:

000010000 101001100 110100001 101110100 100111110

There are a total of 512 possibilities. So, you can use the decimal numbers 0 , 1 , 2 , 3 , ..., and 511 to represent all states of the matrix. Write a program that prompts the user to enter a number between 0 and 511 and displays the corresponding  $3 \times 3$  matrix with the characters H and T .

The user entered 7 , which corresponds to 000000111 . Since 0 stands for H and 1 for T , the output is correct.

**\*\*8.12** (*Financial application: compute tax*) Rewrite Listing 3.6 , ComputeTax.py, using lists. For each filing status, there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, from the taxable income of \$400,000 for a single filer, \$8,350 is taxed at 10%,  
(33,950 – 8,350) at 15%,  
(82,250 – 33,950) at 25%,  
(171,550 – 82,250) at 28%,

(372,950 – 171,550) at 33%, and  
 (400,000 – 372,950) at 35%. The six rates are the same for all filing statuses, which can be represented in the following list:

```
rates = [0.10, 0.15, 0.25, 0.28, 0.33, 0.35]
```

The brackets for each rate for all the filing statuses can be represented in a two-dimensional list as follows:

```
brackets = [
    [8350, 33950, 82250, 171550, 372950], # Single filer
    [16700, 67900, 137050, 208850, 372950], # Married
    jointly/qualified widow(er)
    [8350, 33950, 68525, 104425, 186475], # Married separately
    [11950, 45500, 117450, 190200, 372950] # Head of household
]
```

Suppose the taxable income is \$400,000 for single filers. The tax can be computed as follows:

```
tax = brackets[0][0] * rates[0] +
      (brackets[0][1] - brackets[0][0]) * rates[1] +
      (brackets[0][2] - brackets[0][1]) * rates[2] +
      (brackets[0][3] - brackets[0][2]) * rates[3] +
      (brackets[0][4] - brackets[0][3]) * rates[4] +
      (400000 - brackets[0][4]) * rates[5]
```

**\*8.13** (*Locate the largest element*) Write the following function that returns the location of the largest element in a two-dimensional list:

```
def locateLargest(a):
```

The return value is a one-dimensional list that contains two elements. These two elements indicate the row and column indexes of the largest element in the two-dimensional list. Write a test program that prompts the user to enter a two-dimensional list and displays the location of the largest element in the list.

- \*8.14** (*Explore matrix*) Write a program that prompts the user to enter the length of a square matrix, randomly fills in 0 s and 1 s into the matrix, prints the matrix, and finds the rows, columns, and major diagonal with all 0 s or all 1 s.

## Sections 8.4–8.7

- \*8.15** (*Geometry: same line?*) **Programming Exercise 6.19** gives a function for testing whether three points are on the same line. Write the following function to test whether all the points in the `points` list are on the same line:

```
def sameLine(points):
```

Write a program that prompts the user to enter five points in one line and displays whether they are on the same line.

- \*8.16** (*Sort a list of points on y-coordinates*) Write the following function to sort a list of points on their y-coordinates. Each point is a list of two values for x- and y-coordinates.

```
# Returns a new list of points sorted on the y-coordinates
def sort(points):
```

For example, the points `[[ 4 , 2 ], [ 1 , 7 ], [ 4 , 5 ], [ 1 , 2 ], [ 1 , 1 ], [ 4 , 1 ]]` will be sorted to `[[ 1 , 1 ], [ 4 , 1 ], [ 1 , 2 ], [ 4 , 2 ], [ 4 , 5 ], [ 1 , 7 ]]`. Write a test program that displays the sorted result for points `[[ 4 , 34 ], [ 1 , 7.5 ], [ 4 , 8.5 ], [ 1 , -4.5 ], [ 1 , 4.5 ], [ 4 , 6.6 ]]` using `print(list)`.

**\*\*\*8.17** (*Financial tsunami*) Banks lend money to each other. In tough economic times, if a bank goes bankrupt, it may not be able to pay back the loan. A bank's total assets are its current balance plus its loans to other banks. The diagram in [Figure 8.7](#) shows five banks. The banks' current balances are `25 , 125 , 175 , 75 , and 181` million dollars. The directed edge from node 1 to node 2 indicates that bank 1 lends `40` million dollars to bank 2.

---

### Figure 8.7

---

Banks lend money to each other.

---

If a bank's total assets are under a certain limit, the bank is unsafe. The money it borrowed cannot be returned to the lender, and the lender cannot count the loan in its total assets. Consequently, the lender may also be unsafe if its total assets are under the limit. Write a program to find all unsafe banks. Your program should read the input as follows. It first reads two integers `n` and `limit`, where `n` indicates the number of banks and `limit` is the minimum total assets for keeping a bank safe. It then reads `n` lines that describe the information for `n` banks with ids from `0` to `n-1`. The first number in the line is the bank's balance, the second number indicates the number of banks that borrowed money from the bank, and the rest are pairs of two numbers. Each pair describes a borrower. The first number in the pair is the borrower's id and the second is the amount borrowed. For example, the input for the five banks in [Figure 8.7](#) is as follows (note that the limit is `201`):



```

5 201
25 2 1 100.5 4 320.5
125 2 2 40 3 85
175 2 0 125 3 75
75 1 0 125
181 1 2 125

```

The total assets of bank 3 are (  $75 + 125$  ), which is under  $201$  , so bank 3 is unsafe. After bank 3 becomes unsafe, the total assets of bank 1 fall below the limit (  $125 + 40$  ), so bank 1 also becomes unsafe. The output of the program should be:

```
Unsafe banks are 3 1
```

(Hint: Use a two-dimensional list `borrowers` to represent loans. `borrowers[i][j]` indicates the loan that bank  $i$  loans to bank  $j$ . Once bank  $j$  becomes unsafe, `borrowers[i][j]` should be set to `0` .)

- \*8.18** (*Shuffle rows*) Write a function that shuffles the rows in a two-dimensional list using the following header:

```
def shuffle(m):
```

Write a test program that shuffles the following matrix:

```
m = [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]]
```

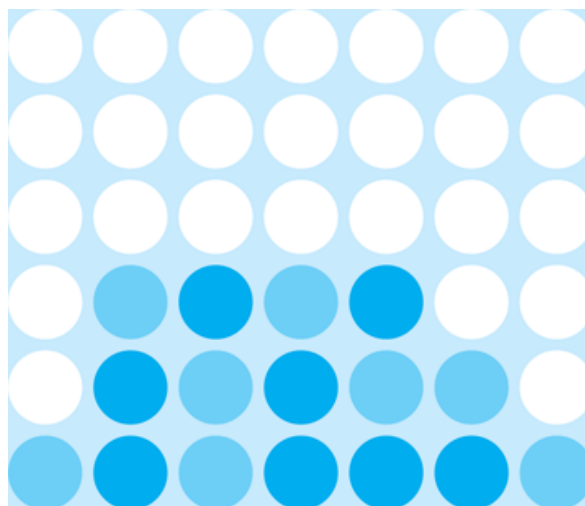
- \*\*8.19** (*Pattern recognition: four consecutive equal numbers*) Write the following function that tests whether a two-dimensional list has four consecutive numbers of the same value, either horizontally, vertically, or diagonally:

```
def isConsecutiveFour(values):
```



Write a test program that prompts the user to enter the number of rows and columns of a two-dimensional list and then the values in the list. The program displays **True** if the list contains four consecutive numbers with the same value; otherwise, it displays **False**. Here are some examples of the **True** cases:

0 1 0 3 1 6 1	0 1 0 3 1 6 1	0 1 0 3 1 6 1	0 1 0 3 1 6 1
0 1 6 8 6 0 1	0 1 6 8 6 0 1	0 1 6 8 6 0 1	0 1 6 8 6 0 1
5 6 2 1 8 2 9	5 5 2 1 8 2 9	5 6 2 1 6 2 9	9 6 2 1 8 2 9
6 5 6 1 1 9 1	6 5 6 1 1 9 1	6 5 6 6 1 9 1	6 9 6 1 1 9 1
1 3 6 1 4 0 7	1 5 6 1 4 0 7	1 3 6 1 4 0 7	1 3 9 1 4 0 7
3 3 3 3 4 0 7	3 5 3 3 4 0 7	3 6 3 3 4 0 7	3 3 3 9 4 0 7

- \*\*\*8.20** (*Game: Connect Four*) Connect Four is a two-player board game in which the players alternately drop colored disks into a seven-column, six-row vertically suspended grid, as shown below:



The objective of the game is to connect four same-colored disks in a row, column, or diagonal before your opponent does. The program prompts two players to drop a red or yellow disk alternately. Whenever a disk is dropped, the program redisplay the board on the console and determines the status of the game (win, draw, or continue).

- \*\*\*8.21** (*Game: multiple Sudoku solutions*) The complete solution for the Sudoku problem is given in [Supplement III.A](#). A Sudoku problem may have multiple solutions. Modify `Sudoku.py` in [Supplement III.A](#) to display the total number of the solutions. Display two solutions if multiple solutions exist.
- \*\*8.22** (*Even number of 1s*) Write a program that generates a  $6 \times 6$  two-dimensional matrix filled with 0s and 1s, displays the matrix, and checks to see if every row and every column have the even number of 1s.
- \*8.23** (*Game: find the flipped cell*) Suppose you are given a  $6 \times 6$  matrix filled with 0s and 1s. All rows and all columns have the even number of 1s. Let the user flip one cell (i.e., flip from 1 to 0 or from 0 to 1) and write a program to find which cell was flipped. Your program should prompt the user to enter a  $6 \times 6$  two-dimensional list with 0s and 1s and find the first row `r` and first column `c` where the even number of 1s property is violated. The flipped cell is at (`r`, `c`).
- \*8.24** (*Check Sudoku solution*) [Listing 8.6](#)  checks whether a solution is valid by checking whether every number is valid in the grid. Rewrite the program by checking whether every row, column, and box has the numbers 1 to 9. A sample run of the program is same as for [Listing 8.6](#) .
- \*8.25** (*Markov matrix*) An  $n \times n$  matrix is called a *positive Markov matrix* if each element is positive and the sum of the elements in each column is 1. Write the following function to check whether a matrix is a Markov matrix:

```
def isMarkovMatrix(m):
```

Write a test program that prompts the user to enter a  $3 \times 3$  matrix of numbers and tests whether it is a Markov matrix.

- \*8.26** (*Row sorting*) Implement the following function to sort the rows in a two-dimensional list. A new list is returned and the original list is intact.

```
def sortRows(m):
```

Write a test program that prompts the user to enter a  $3 \times 3$  matrix of numbers and displays a new row-sorted matrix.

- \*8.27** (*Column sorting*) Implement the following function to sort the columns in a two-dimensional list. A new list is returned and the original list is intact.

```
def sortColumns(m):
```

Write a test program that prompts the user to enter a  $3 \times 3$  matrix of numbers and displays a new column-sorted matrix.

- 8.28** (*Strictly identical lists*) The two-dimensional lists `m1` and `m2` are strictly identical if their corresponding elements are equal. Write a function that returns `True` if `m1` and `m2` are strictly identical, using the following header:

```
def equals(m1, m2):
```

Write a test program that prompts the user to enter two  $3 \times 3$  lists of integers and displays whether the two are strictly identical.

- 8.29** (Identical lists) The two-dimensional lists `m1` and `m2` are identical if they have the same contents. Write a function that returns `True` if `m1` and `m2` are identical, using the following header:

```
def equals(m1, m2):
```

Write a test program that prompts the user to enter two  $3 \times 3$  lists of integers and displays whether the two are identical.

- \*8.30** (*Algebra: solve linear equations*) Write a function that solves the following  $2 \times 2$  system of linear equations:

$$\begin{array}{l} a_{00}x + a_{01}y = b_0 \\ a_{10}x + a_{11}y = b_1 \end{array} \quad x = \frac{b_0a_{11} - b_1a_{01}}{a_{00}a_{11} - a_{01}a_{10}} \quad y = \frac{b_1a_{00} - b_0a_{10}}{a_{00}a_{11} - a_{01}a_{10}}$$

The function header is:

```
def linearEquation(a, b):
```

The function returns `None` if  $a_{00}a_{11} - a_{01}a_{10}$  is `0`; otherwise, it returns the solution for  $x$  and  $y$  in a list. Write a test program that prompts the user to enter  $a_{00}$ ,  $a_{01}$ ,  $a_{10}$ ,  $a_{11}$ ,  $b_0$ , and  $b_1$  and displays the result. If  $a_{00}a_{11} - a_{01}a_{10}$  is `0`, report that `The equation has no solution.`

- \*8.31** (*Geometry: intersecting point*) Write a function that returns the intersecting point of two lines. The intersecting point of the two lines can be found by using the formula shown in [Programming Exercise 3.25](#). Assume that  $(x_1, y_1)$  and  $(x_2, y_2)$  are the two points on line 1 and  $(x_3, y_3)$  and  $(x_4, y_4)$  are the two points on line 2. The function header is:

```
def getIntersectingPoint(points):
```

The points are stored in the  $4 \times 2$  two-dimensional list `points`, with  $(points[0][0], points[0][1])$  for  $(x_1, y_1)$ . The function returns the intersecting point  $(x, y)$  in a list, and `None` if the two lines are parallel. Write a program that prompts the user to enter four points and displays the intersecting point.

- \*8.32** (*Geometry: area of a triangle*) Write a function that returns the area of a triangle using the following header:

```
def getTriangleArea(points):
```

The points are stored in the  $3 \times 2$  two-dimensional list `points`, with  $(points[0][0], points[0][1])$  for  $(x_1, y_1)$ . The triangle area can be computed using the formula in [Programming Exercise 2.14](#). The function returns `None` if the three points are on the same line. Write a program that prompts the user to enter three points and displays the area of the triangle.

- \*8.33** (*Geometry: polygon subareas*) A convex four-vertex polygon is divided into four triangles, as shown in [Figure 8.8](#).

---

**Figure 8.8**

---

A four-vertex polygon is defined by four vertices.



---

Write a program that prompts the user to enter the coordinates of four vertices and displays the areas of the four triangles in increasing order.

- \*8.34** (*Geometry: rightmost lowest point*) In computational geometry, often you need to find the rightmost lowest point in a set of points. Write the following function that returns the rightmost lowest point in a set of points:

```
# Return a list of two values for a point
def getRightmostLowestPoint(points):
```

Write a test program that prompts the user to enter the coordinates of six points and displays the rightmost lowest point.

- \*8.35** (*Central city*) Given a set of cities, the central point is the city that has the shortest total distance to all other cities. Write a program that prompts the user to enter the coordinates of the cities (i.e., their coordinates) and finds the central city.
- \*\*8.36** (*Simulation using Turtle: self-avoiding random walk*) A self-avoiding walk in a lattice is a path from one point to another that does not visit the same point twice. Self-avoiding walks have applications in physics, chemistry, and mathematics. They can be used to model chainlike entities such as solvents and polymers. Write a Turtle program that displays a random path that starts from the center and ends at a point on the boundary, as shown in [Figure 8.9a](#) , or ends at a dead-end point (i.e., surrounded by four points that have already been visited), as shown in [Figure 8.9b](#) . Assume the size of the lattice is  $16 \times 16$ .

**Figure 8.9**

(a) A path ends at a boundary point. (b) A path ends at dead-end point.

(Screenshots courtesy of Apple.)

**\*\*8.37** (*Simulation: self-avoiding random walk*) Write a simulation program to show that the chance of getting dead-end paths increases as the grid size increases. Your program simulates lattices with sizes from 10 to 80 with increment 5. For each lattice size, simulate a self-avoiding random walk 10,000 times and display the probability of the dead-end paths, as shown in the following sample output:

**\*\*8.38** (*Turtle: draw a polygon/polyline*) Write the following functions that draw a polygon/polyline to connect all points in the list. Each element in the list is a list of two coordinates.

```
# Draw a polyline to connect all the points in the list
def drawPolyline(points):

# Draw a polygon to connect all the points in the list and
# close the polygon by connecting the first point with the last
point
def drawPolygon(points):

# Fill a polygon by connecting all the points in the list
def fillPolygon(points):
```

**\*\*8.39** (*Guess the capitals*) Write a program that repeatedly prompts the user to enter a capital for a state. Upon receiving the user input, the program reports whether the answer is correct. Assume that 50 states and their capitals are stored in a two-dimensional list, as shown in [Figure 8.10](#). The program prompts the user to answer all the states' capitals and displays the total correct count. The user's answer is not case sensitive. Implement the program using a list to represent the data in the following table.



**Figure 8.10**

Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
...	...
...	...

A two-dimensional list stores states and their capitals.

- \*\*8.40** (*Latin square*) A Latin square is an `n` by `n` list filled with `n` different Latin letters, each occurring exactly once in each row and once in each column. Write a program that prompts the user to enter the number `n` and the list of characters, as shown in the sample output and check if the input list is a Latin square. The characters are the first `n` characters starting from `A`.
- \*\*8.41** (*Sort students*) Write a program that prompts the user to enter the students' names and their scores on one line and prints student names in increasing order of their scores. (Hint: Create a list. Each element in the list is a sublist with two elements: score and name. Apply the `sort` method to sort the list. This will sort the list on scores.)