TASK

Introduction to Python — Data Structures - 2D Lists

Visit our website

Introduction

WELCOME TO THE DATA STRUCTURES TASK!

This task introduces you to a fundamental data structure in Python: the 2D list, also known as a grid or nested list. Essentially a 2D list is a list of lists. Every element of the list is another list. In this task, we will explore the creation and usage of 2D lists.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

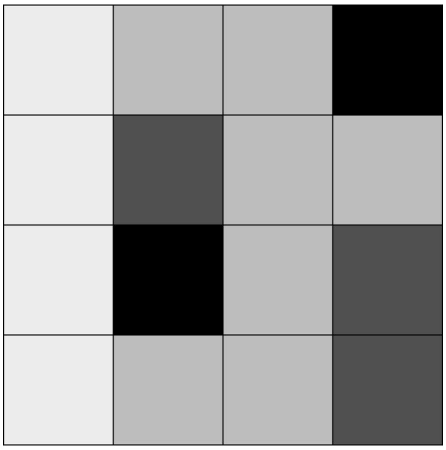
Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

USE CASES OF 2D LISTS

The typical Python list that you have used in previous tasks is basically used to store multiple pieces of information in a linear order. You could think of this linear order as one single dimension or you could visualise this as a row.

In a 2D list, the same is true. The lists still contain pieces of information in a linear order, except a second dimension is added. You could visualise this as a column. There are many use cases where two dimensions of data are useful. Some common uses are images, where pixels are arranged in a row and column format, or a board game, where the spaces on the board are arranged in rows and columns.

The following grayscale image for example could be represented in a grid format where the numbers represent the shade of grey:



236	189	189	0
236	80	189	189
236	0	189	80
236	189	189	80

If we were to have a representation of this image in a 2D list in Python, it would look like the following:

```
grayscale_image = [[236, 189, 189, 0], [236, 80, 189, 189], [236,
0, 189, 80], [236, 189, 189, 80]]
```

The list of lists represents the whole image, each list element within the list is a row and the elements of the nested lists are the columns. It then becomes clear how using 2D lists to represent images can be used to manipulate the pixel colours of the image. If you are interested in image processing you can look into the Pillow library for image processing at the following link:
https://pillow.readthedocs.io/en/stable/

DECLARING AND CREATING TWO-DIMENSIONAL LISTS

Creating 2D lists in Python where you already have the values for the lists is relatively easy. The example for the grayscale image is an example of how you would declare that list. This declaration is called static declaration as each element in the grid is specifically declared. To make it easier to read and to more closely represent a table or matrix, the list could also be declared in the following manner:

```
grayscale_image = [[236, 189, 189,   0],
                   [236,  80, 189, 189],
                   [236,   0, 189,  80],
                   [236, 189, 189,  80]]
```

We can also dynamically declare a grid. To create an empty grid i.e. a grid filled with None values, we can employ the following code where we can specify the number of rows and the number of columns (note that we can initialise this grid with any default values for elements by replacing the None value with another value. It can be any datatype i.e. string, integer etc.):

```
# we intialise variables for the specific size of the grid.
# In this case we have a 3 by 2 grid.
number_of_rows = 3
number_of_columns = 2

# We create the None value twice in a list for the columns, and we
employ a loop to do it three times for the number of rows.
empty_grid = [[None] * number_of_columns for _ in range(number_of_rows)]

print(empty_grid)
# printing this grid will give the following output
#[[None, None], [None, None], [None, None]]
```


ASSIGNING VALUES TO ELEMENTS IN A TWO-DIMENSIONAL LIST

Like a single-dimensional list, we use the list indices to access elements in a two-dimensional list. However, unlike a single-dimensional list, a two-dimensional list contains two sets of indices.

The example below shows how to assign the number 4 to the element in the second row and first column of a grid named table :

```
table[1][0] = 4 #table[row 2][column 1]
```

Likewise, we can take a value from a specific element in a grid and assign it to a variable. In the below example the value of last_pixel will be 80.

```
grayscale_image = [[236, 189, 189,   0],
                    [236,  80, 189, 189],
                    [236,   0, 189,  80],
                    [236, 189, 189,  80]]

last_pixel = grayscale_image[3][3]
```

To loop through grids, we need to make use of nested loops. In the following example, rows represent a school term and columns represent one of 5 test scores for that term We can use nested loops to print out the scores with percentages for each specific term as follows:

```
student_scores = [ [72, 85, 87, 90, 69],
                   [80, 87, 65, 89, 85],
                   [96, 91, 70, 78, 97],
                   [90, 93, 91, 90, 94] ]
# Use a for loop to print the entire elements of the two dimensional array.
row_index = 0
for row in student_scores:  # outer loop for rows
    print(f'Term {row_index + 1}: ') #row index used for the term number
    row_index +=1  #increment row index
    for col in row:  # inner loop for columns
        print(col, end = "% ") # print each column value with percentage
symbol
    print()
```

Running this code will produce the following output:

```
Term 1:
72% 85% 87% 90% 69%
Term 2:
80% 87% 65% 89% 85%
Term 3:
96% 91% 70% 78% 97%
Term 4:
90% 93% 91% 90% 94%
```

A note from the
Hyperion Team

RAGGED 2D LISTS AKA NON-RECTANGULAR LISTS

In a two-dimensional list, each row is itself a list. Therefore, since Python does not enforce the lists to be of the same length, the rows can have different lengths. lists with rows of varying lengths are known as ragged lists.

Here is an example of a ragged list:

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

Iterating through a ragged list is a bit trickier than iterating through a grid with lists of equal length. We would need to determine the length of the current list in the loop before running the nested loop and for each iteration of the nested loop, the length needs to be updated for the current row.
The following is an example of printing every element in the above-ragged list:

```
rows = len(ragged_list)
for row in range(rows):
    cols = len(ragged_list[row]) # now the number of cols depends on each row’s 
length
    print("Row", row, "has", cols, "columns: ", end="")
    for col in range(cols):
        print(ragged_list[row][col], " ", end="")
    print()
```

The following is the output for the code example above:

```
Row 0 has 3 columns: 1  2  3
Row 1 has 2 columns: 4  5
Row 2 has 1 columns: 6
Row 3 has 4 columns: 7  8  9  10
```

Compulsory Task 1

Create a file named **minesweeper.py**

Create a function that takes a grid of # and -, where each hash (#) represents a mine and each dash (-) represents a mine-free spot.
Return a grid, where each dash is replaced by a digit, indicating the number of mines immediately adjacent to the spot i.e. (horizontally, vertically, and diagonally).

Example of an input:

```
[ ["-", "-", "-", "#", "#"],
  ["-", "#", "-", "-", "-"],
  ["-", "-", "#", "-", "-"],
  ["-", "#", "#", "-", "-"],
  ["-", "-", "-", "-", "-"] ]
```

Example of the expected output:

```
[ ["1", "1", "2", "#", "#"],
  ["1", "#", "3", "3", "2"],
  ["2", "4", "#", "2", "0"],
  ["1", "#", "#", "2", "0"],
  ["1", "2", "2", "1", "0"] ]
```

Here is a tip. When checking adjacent positions to a specific position in the grid, the following table might assist you in determining adjacent indexes:

NW position = current_row - 1 current_col - 1	N position = current_row - 1 current_col	NE position = current_row - 1 current_col + 1
W position = current_row current_col - 1	Current position = current_row current_col	E position = current_row current_col + 1
SW position = Current_row + 1 current_col - 1	S position = current_row +1 current_col	SE position = current_row + 1 current_col + 1

Also ensure that when checking adjacent positions in the grid that you take into account that on the edges of the grid, you may go out of bounds.

Lastly, you could make use of the enumerate function in Python to keep track of the index points and values without having to create a count variable and explicitly iterate the count variable to keep track of the current row or column index.

Below is an example of how the enumerate function works.

```
#list to be iterated through
values = ["a", "b", "c"]

# "count" here is used to keep track of the index point
# "value" is the value of the current element in the loop
# The enumerate method takes 2 arguments the iterable and the starting
# value for “count” which we set at 0 to represent the index of the first
# index in the list.
for count, value in enumerate(values, start = 0):
    print(f'Index {count} contains the value {value}')

Below is the output generated:
Index 0 contains the value a
Index 1 contains the value b
Index 2 contains the value c
```


If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.