HyperionDev

# Data Structures – 2D Lists

## Task

Visit our website

# Introduction

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

# Exploring 2D list use cases

In Python, a typical list stores multiple pieces of information in a linear order, which you can think of as a single row.

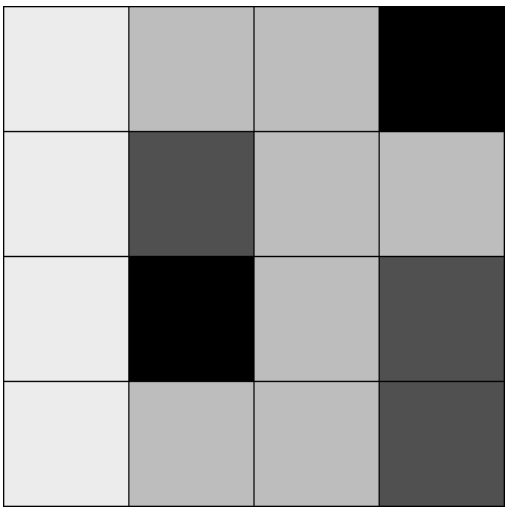
A 2D list, or a list of lists, expands this concept by adding another dimension. You can think of this as having rows and columns. Here are some common uses for 2D lists:

- **Images:** Pixels are arranged in rows and columns.
- **Board games:** Spaces on the board are arranged in rows and columns.
- **Data representation:** Data stored in a tabular format like a spreadsheet.

Images are made up of tiny dots called pixels. Each pixel can have a colour value that defines what colour it should be. For example, in a grayscale image, each pixel is represented by a single number that indicates its shade of grey. The number usually ranges from 0 to 255:

- `0` represents black.
- `255` represents white.
- Values between `0` and `255` represent varying shades of grey.

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



Each number corresponds to the intensity of grey for a pixel in the image.

The higher the number, the lighter the grey. For instance:

- `236` is a light grey.
- `189` is a medium grey.
- `80` is a darker grey.
- `0` is black.

The following is the grid format of numbers used to represent the greyscale image above.

| | | | |
|---|---|---|---|
| 236 | 189 | 189 | 0 |
| 236 | 80 | 189 | 189 |
| 236 | 0 | 189 | 80 |
| 236 | 189 | 189 | 80 |

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

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

In this 2D list:

- Each inner list represents a row.
- Each element within these inner lists represents a column.

This structure allows you to manipulate the pixel colours of an image easily.

If you're interested in learning more about image processing, you can explore the **Pillow library** for Python.

A `4x4` grid is a simplified example to illustrate the concept of representing images with a 2D list. It is easy for beginners to visualise and understand. In reality, images can be much larger. For example:

- A typical HD image might be `1920` pixels wide and `1080` pixels tall, resulting in a `1920x1080` grid.
- A `4K` image is `3840x2160` pixels.

# Declaring and creating 2D lists

Creating 2D lists in Python when you already have the values for the lists is relatively easy. 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; values can be any data type, e.g., string, integer, etc.

```
# Initialise 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

# Create the None values twice in a list for the columns
# then 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 2D list

As with a single-dimensional list, we use the list indices to access elements in a 2D list. However, unlike a single-dimensional list, a 2D 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` (remember that the rows and columns are numbered from zero):

```
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 example below, 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 five test scores for that term. There is a widely used convention of having the first index represent the row and the second index represent the column.

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 all 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 % 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%
```

# Handling ragged 2D lists

In a 2D list, each row is itself a list. Python does not enforce that lists be of the same length, and so they 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.
- Update the length to be the length of the current row for each iteration of the nested loop.

The following is an example of printing every element in the ragged list shown earlier:

```
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
```

If you are having difficulty following what happens in the above example, try copying and running the code. Also, try creating a trace table to keep track of what the values of the variables are in each iteration of the nested loops. Remember that for each iteration of the outer loop (the row loop), the inner loop (the column loop) will run as many times

as the number of columns in that row. For example, for row zero, the outer loop will run once and the inner loop will run three times as there are three columns in row zero.

Are you ready to apply what you've learned to a more complex scenario like a game? Consider this animated **tutorial** showing how to use 2D lists to create a Connect Four game. What's great about it is that it not only visualises the progress through the code, but explains exactly why the code is written the way it is in order to achieve the final outcome.

## Take note

The task below is **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select "Request Review", the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you've done that, feel free to progress to the next task.

---

## Auto-graded task

Now it's time to see whether you're ready to apply what you've learned to some code of your own! This is a challenging task, but worth persisting through as you'll gain valuable experience with 2D lists and nested loops.

1. Create a file named **minesweeper.py**.

2. Create a function that takes a grid of `#` and `-`, where each hash (`#`) represents a mine and each dash (`-`) represents a mine-free spot.
3. 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] ]
```

**Below are some hints to get you started.**

When checking adjacent positions to a specific position in the grid, the following table might assist you in determining adjacent indices:

| | | |
|---|---|---|
| NW position =<br>`current_row - 1`<br>`current_col - 1` | N position =<br>`current_row - 1`<br>`current_col` | NE position =<br>`current_row - 1`<br>`current_col + 1` |
| W position =<br>`current_row`<br>`current_col - 1` | Current position =<br>`current_row`<br>`current_col` | E position =<br>`current_row`<br>`current_col + 1` |
| SW position =<br>`current_row + 1`<br>`current_col - 1` | S position =<br>`current_row + 1`<br>`current_col` | SE position =<br>`current_row + 1`<br>`current_col + 1` |

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

There may be quite a lot of repetition in this task to do things like checking whether a particular row and column combination (i.e., cell) is a valid position in the grid (within bounds), and to increment the counts of the number of adjacent `#` signs. It makes sense to create functions to handle the repetitive aspects.

You could (but **do not have to** – the problem can be solved in a variety of ways without it) 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 position of the first
index in the list.
print("Below is the output generated:")
for count, value in enumerate(values, start = 0):
    print(f'Index {count} contains the value {value}')
```

The code provided above will produce the following output:

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

Be sure to place files for submission inside your **task folder** and click "**Request review**" on your dashboard.

**Important:** Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

---

## Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

---