Overview

When using the `grid` geometry manager, every widget is placed into a **cell** that is defined by a row and a column. Understanding how rows, columns, and cells work together is essential for building clean and well-aligned user interfaces.

This lesson explains how grid positions widgets, how row and column numbering works, and how grid automatically sizes rows and columns based on their contents.

Important Information

Rows and Columns

In `grid`, each widget is given a row number and a column number. These two values determine where the widget appears relative to other widgets in the layout.

Widgets that share the same **row** are placed next to each other from left to right. Widgets that share the same **column** are stacked above and below one another.

```
label = ttk.Label(root, text="Name:")
label.grid(row=0, column=0)

entry = ttk.Entry(root)
entry.grid(row=0, column=1)
```

In this example, both widgets are placed in row `0`. Since they are in different columns, the label appears on the left and the entry appears on the right.

The Cell Concept

A **cell** is the space created where a row and a column intersect. By default, each widget occupies exactly one cell in the grid.

Cells are created automatically as widgets are added. You do not need to define rows or columns ahead of time. Thinking in terms of cells helps when planning a layout, especially when sketching an interface before writing code.

Row and Column Numbering

Row and column numbers must be **non-negative integers**, which means they start at `0` and count upward *(0, 1, 2, 3, ...)*.

You are not required to start at row `0` or column `0`, and the numbers do not need to be consecutive. It is perfectly valid to leave gaps in numbering.

```
title = ttk.Label(root, text="Settings")
title.grid(row=0, column=0)

save_button = ttk.Button(root, text="Save")
save_button.grid(row=10, column=1)
```

Leaving gaps in row or column numbers can make it easier to insert new widgets later without having to rewrite existing layout code.

Relative Positioning

Grid placement is **relative**, not based on exact pixel positions. A widget in column `2` is simply placed to the right of column `1`. A widget in row `5` appears below row `4`.

Grid handles the exact spacing and alignment automatically. This makes layouts easier to adjust and maintain, especially as the interface grows or changes.

Automatic Sizing of Rows and Columns

Rows and columns do not have fixed sizes by default. Instead, grid sizes them based on the widgets they contain.

The width of a column is determined by the widest widget in that column. The height of a row is determined by the tallest widget in that row.

```
short_label = ttk.Label(root, text="ID")
short_label.grid(row=0, column=0)

long_label = ttk.Label(root, text="Full Legal Name")
long_label.grid(row=1, column=0)
```

In this example, column `0` automatically becomes wide enough to fit `"Full Legal Name"`. The two rows can also end up with different heights. Rows and columns do not need to be equal in size, which allows the layout to adapt naturally to its content.

Row and Column Weights

Row and column **weights** control how extra space is distributed when a container is larger than the widgets inside it. By default, all rows and columns have a weight of `0`, which means they **do not expand**.

A weight is assigned using `grid_rowconfigure()` and `grid_columnconfigure()` on the **container**, not on the widget.

```
root.grid_rowconfigure(0, weight=1)
root.grid_columnconfigure(0, weight=1)
```

A row or column with a higher weight gets a larger share of extra space. If multiple rows or columns have weights, the space is divided proportionally.

```
root.grid_columnconfigure(0, weight=1)
root.grid_columnconfigure(1, weight=2)
```

In this example, column `1` grows twice as fast as column `0`.

Weights work together with the `sticky` option. A widget will only stretch to fill its cell if:

- the row and/or column has a positive weight
- the widget uses `sticky` (`"n"`, `"s"`, `"e"`, `"w"`)

```
frame.grid_rowconfigure(0, weight=1)
frame.grid_columnconfigure(0, weight=1)

label.grid(row=0, column=0, sticky="nsew")
```

Without row and column weights, `sticky` has no effect because the cell itself never grows.

Planning Layouts with Grid

When planning a layout, it helps to think in rows and columns rather than pixels. Related widgets should be grouped into shared rows or columns, and unused row or column numbers can be left open for future expansion.

This approach leads to layouts that are easier to read, easier to modify, and more adaptable as your program grows.