10 | Basics of Data Visualisation in Python Using Matplotlib

More and more applications of programming using Python involve large amounts of data. Visualising those data is an essential part of understanding what the data say, as every scientist, data scientist, and anyone who works with data will confirm. Displaying data visually is important for those studying the data and also for those to whom the data is presented. In this Chapter, you'll learn about the basics of data visualisation in Python.

There are several third-party modules in Python that you can use to visualise data. One of the most important of these is Matplotlib. There are also newer modules that are very popular in specific applications. However, Matplotlib remains the most widely-used data visualisation module across Python in general. Even if you'll eventually move to other visualisation libraries, a good knowledge of Matplotlib is essential. You can also translate many of the concepts you'll learn about in this Chapter to other libraries that are used for data visualisation in Python.

In this Chapter, you'll learn:

- the **fundamentals** of plotting figures
- when and how to use the **two interfaces** in Matplotlib
- how to plot **2D figures**, including using **subplots**
- how to **display images**
- how to plot **3D figures**
- how to **create animations**

What this Chapter will **not** do is teach you about every function available in Matplotlib and how to plot every type of graph you'll ever need. Matplotlib is a vast library that can be used in many versatile ways. However, once you understand the fundamentals, you'll be able to find solutions to plot more advanced figures, too. The excellent Matplotlib documentation will help you along your journey.

A video course mirroring the content of this chapter is coming soon at ***The Python Coding Place***

Getting Started With Matplotlib

Matplotlib is a third-party library that you'll need to install first. You can refer to the section *Installing Third-Party Modules* in the Chapter about using NumPy, which has detailed instructions on the options available to install modules. You can either use your IDE's in-built tools or type the following in the Terminal:

```
pip install matplotlib
```

or

```
python -m pip install matplotlib
```

You've already installed NumPy when working on a previous Chapter that introduced this module. However, if you hadn't already done so, installing Matplotlib will also install NumPy.

Plotting your first figure

Later in this Chapter, you'll read about the two interfaces you can use in Matplotlib to plot figures. For now, you'll use the simpler option:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 68

plt.plot(steps_walked)
```

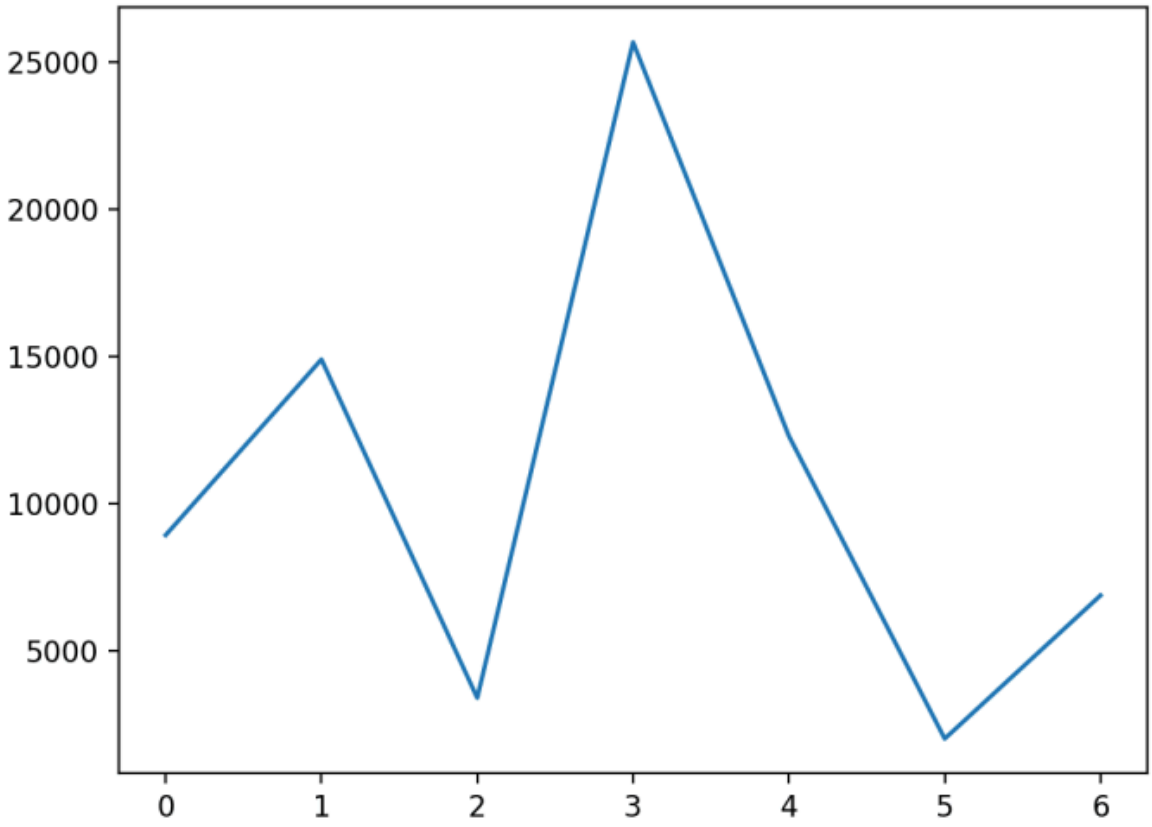
```
plt.show()
```

You start by importing `matplotlib.pyplot` and use the alias `plt`, which is the alias used by convention for this submodule. Matplotlib is a library that contains several submodules such as `pyplot`.

After defining the two lists `days` and `steps_walked`, you use two of the functions from `matplotlib.pyplot`. The `plot()` function plots your data using the data in its argument. In this case, the data are the numbers in `steps_walked`.

When writing code in a script, as in the example above, `plot()` by itself is not sufficient to display the graph on the screen. `show()` tells the program that you want the plot to be displayed. When using an interactive environment such as the Console, the call to `show()` is not required. In the Snippets section at the end of this Chapter you can also read about Jupyter, another interactive coding environment used extensively for data exploration and presentation.

When you run the code above, the program will display a new window showing the following graph:

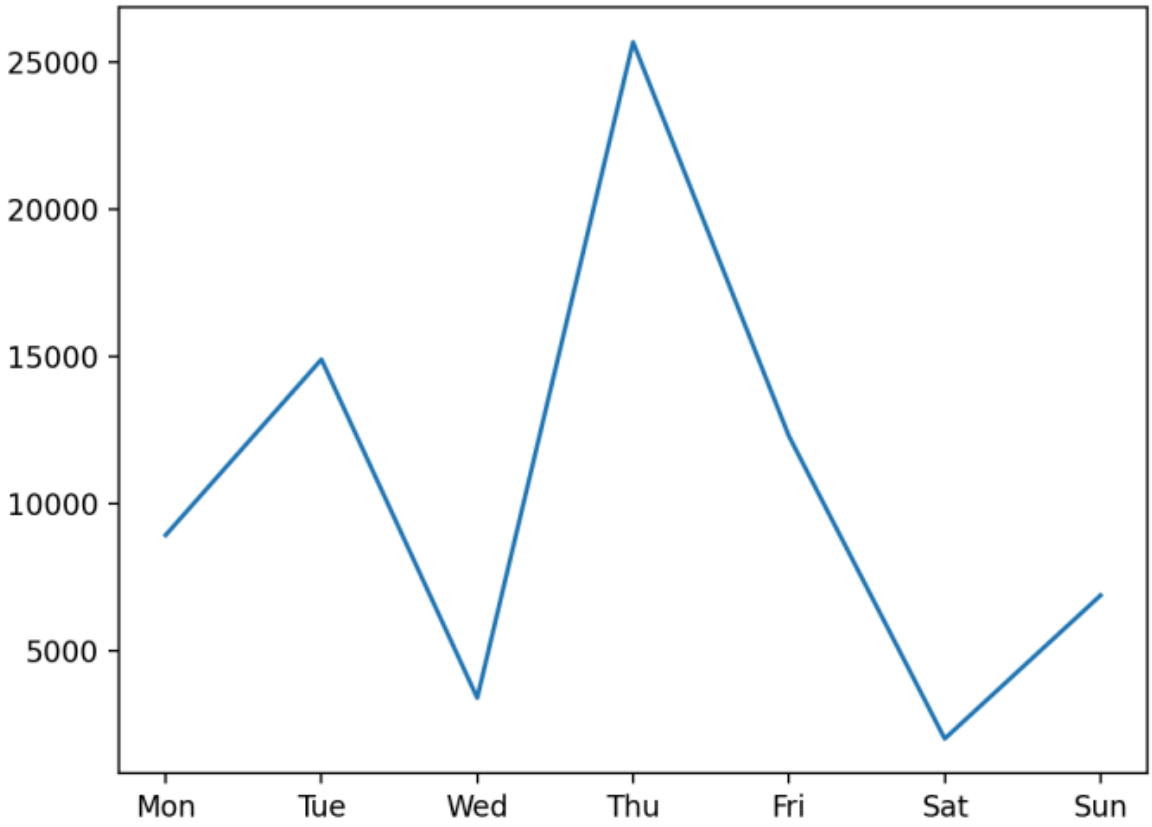


The plot shows a line graph connecting the values for the number of steps walked each day. The labels on the *y*-axis show the number of steps. However, the *x*-axis shows the numbers between `0` and `6`. These numbers are the index positions of the values within the list.

You can call `plot()` with two input arguments instead of one to determine what data to use for both the *x*– and *y*-axes:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6

plt.plot(days, steps_walked)
plt.show()
```

The first argument in `plot()` corresponds to data you want to use for the *x*-axis, and the second argument represents the *y*-axis values. The code now gives the following output:



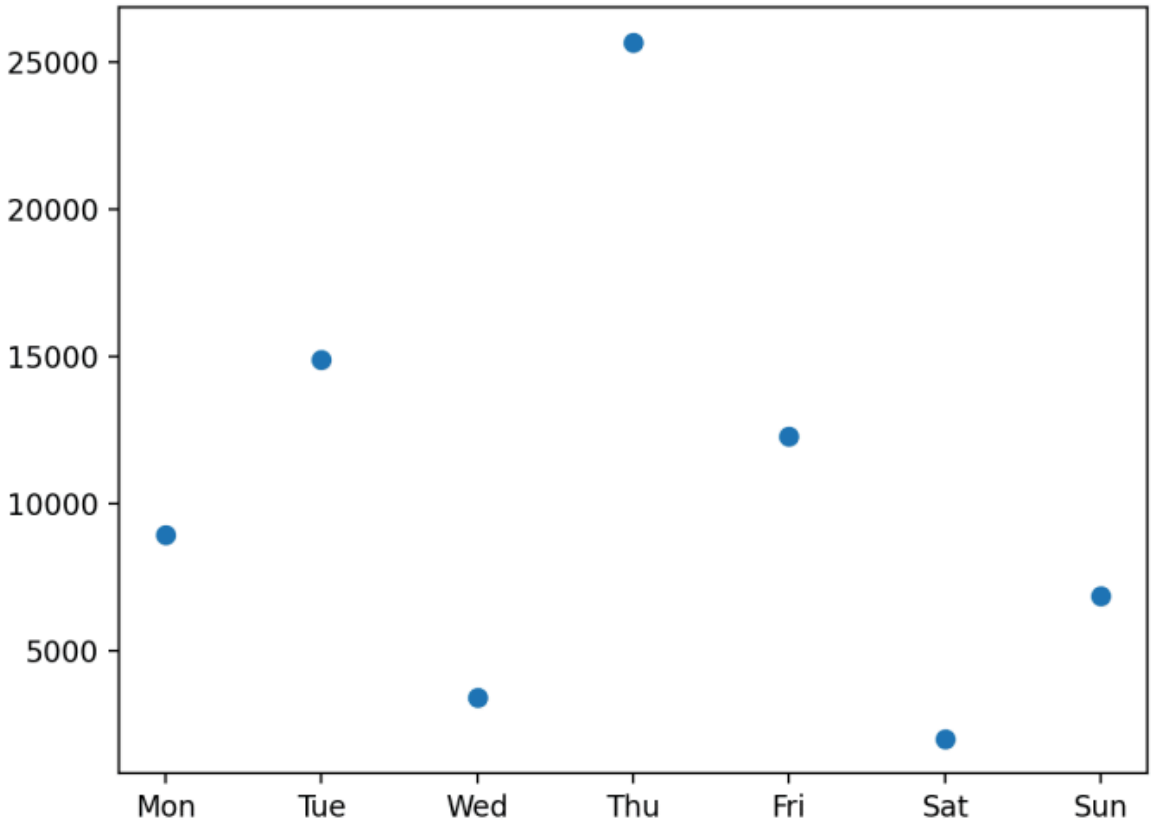
The labels on the *x*-axis now show the days of the week since these are the values in the list `days`.

Customising the plots

You can customise the plot further. First, you can add a marker to show where each data point is:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6

plt.plot(days, steps_walked, "o")
plt.show()
```

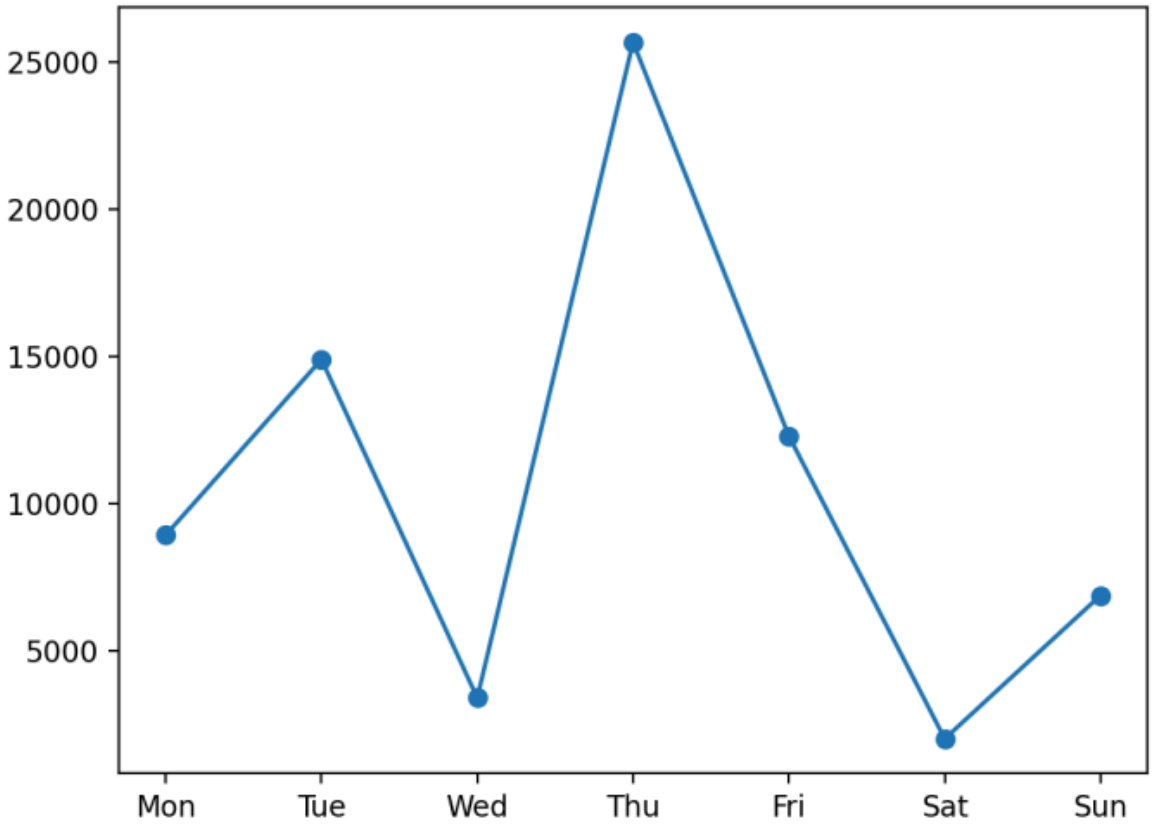
The third argument in `plot()` now indicates what marker you'd like to use. The string `"o"` represents filled circles. The output now looks like this:



There's now a dot to show each data point. However, the line is no longer there. If you'd like to plot markers but keep the line connecting the data points, you can use `"o-"` as the format string:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6

plt.plot(days, steps_walked, "o-")
plt.show()
```

Your plot now shows the marker and the line:



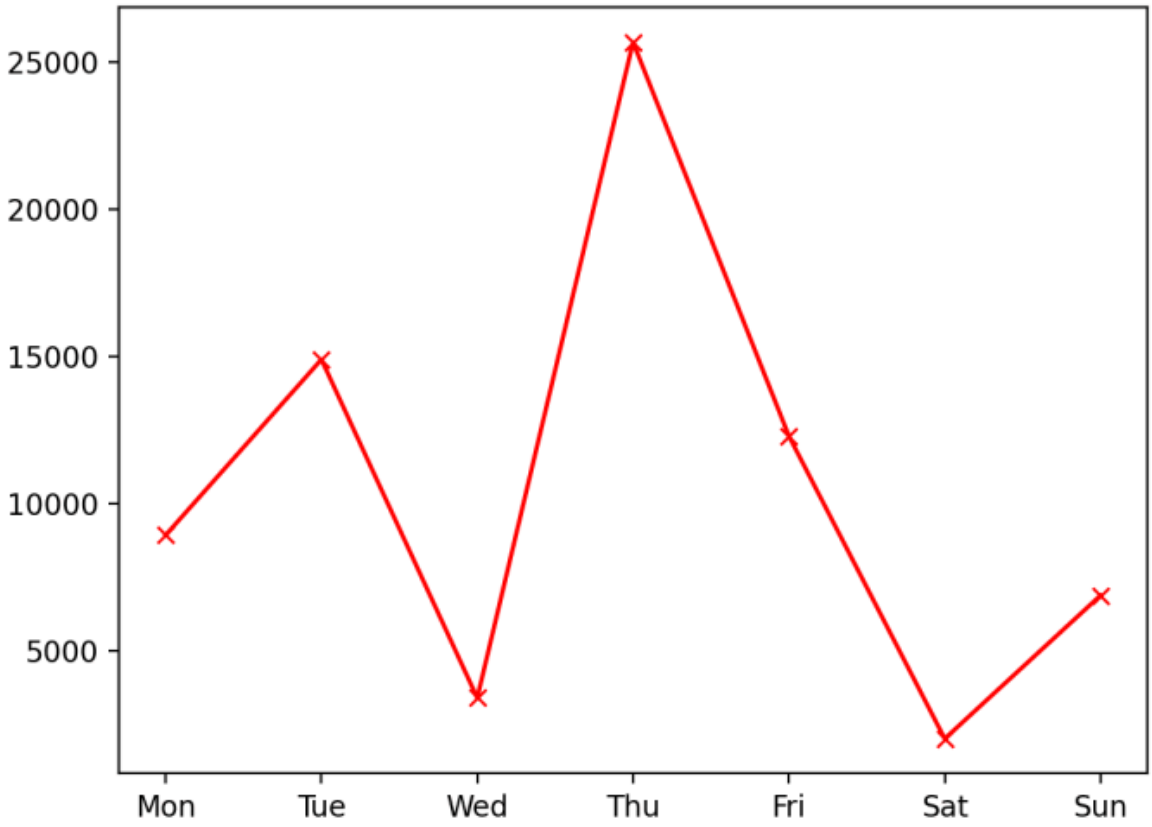
And you can add colour to the format string, too. In the example below, you also change the marker to an x marker:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6

plt.plot(days, steps_walked, "x-r")
plt.show()
```

The format string now includes three characters:

- x shows that the marker should be the x symbol
- - draws the line
- r indicates the colour should be red

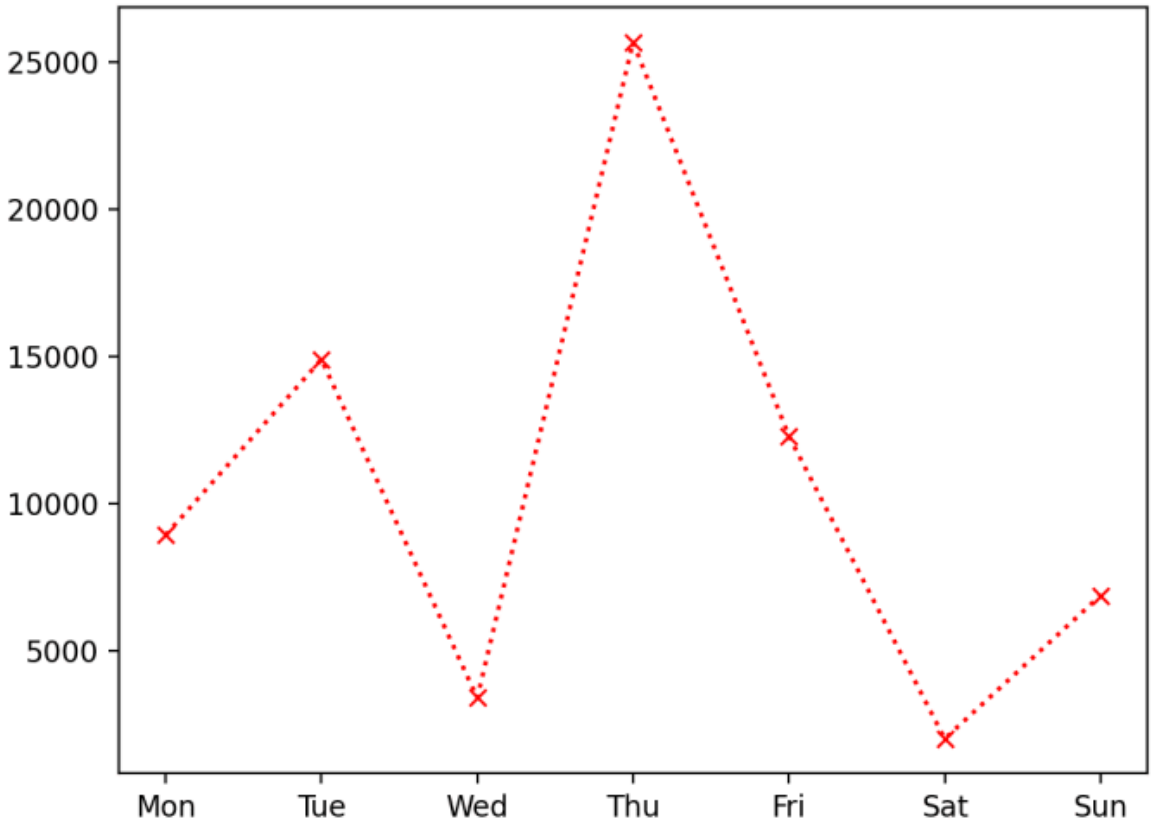
This code gives the following output:



You can finish this section with one final example using the format string `"x:r"`. The colon indicates that the line drawn should be a dotted line:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6

plt.plot(days, steps_walked, "x:r")
plt.show()
```

The plot now has `x` as a marker and a dotted line connecting the data points:



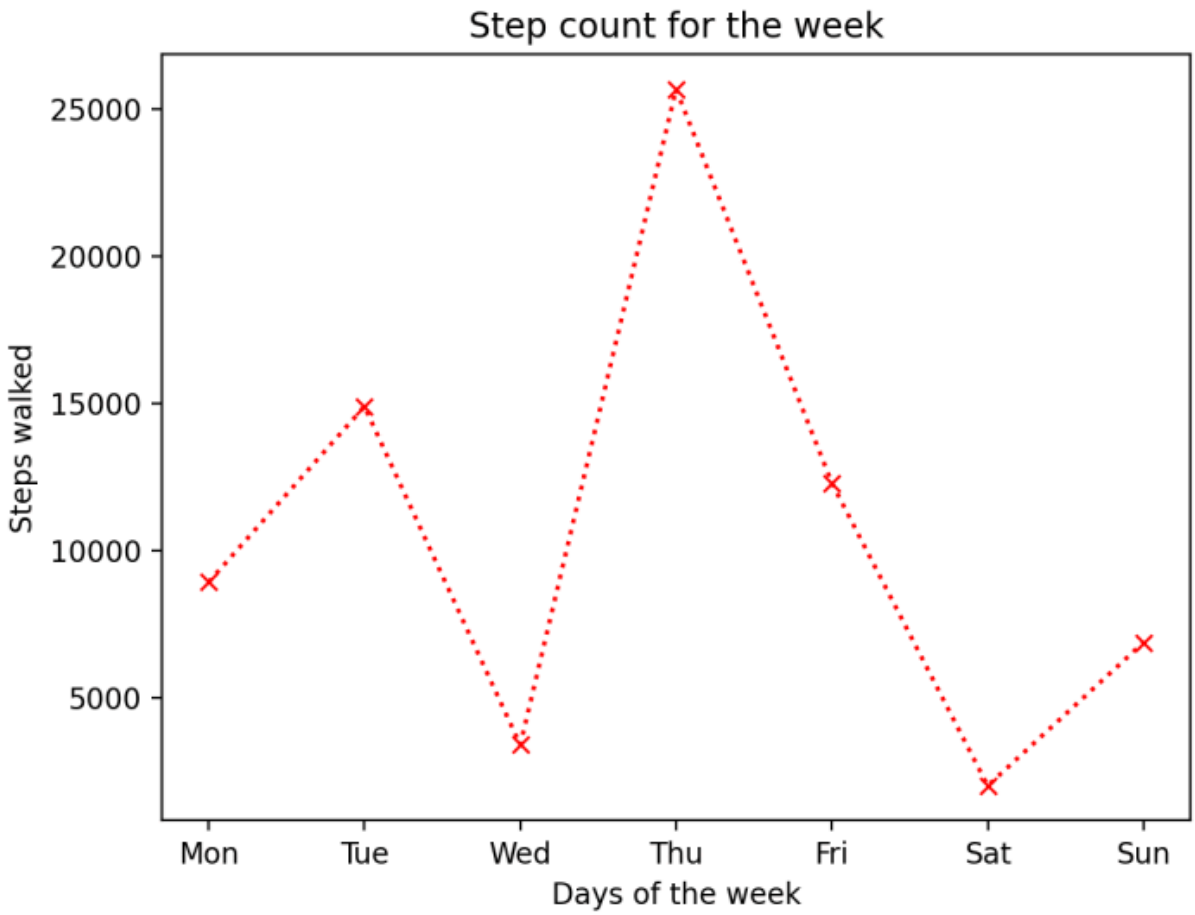
You can see a list of all the markers, colours, and line styles you can use in the section labelled *Notes* on the documentation page for `plot()`.

Adding titles and labels

You can explore a few more functions in `matplotlib.pyplot` to add titles and labels to the plot:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6

plt.plot(days, steps_walked, "x:r")
plt.title("Step count for the week")
plt.xlabel("Days of the week")
plt.ylabel("Steps walked")
plt.show()
```

The `title()` function does what it says! And `xlabel()` and `ylabel()` add labels to the two axes:

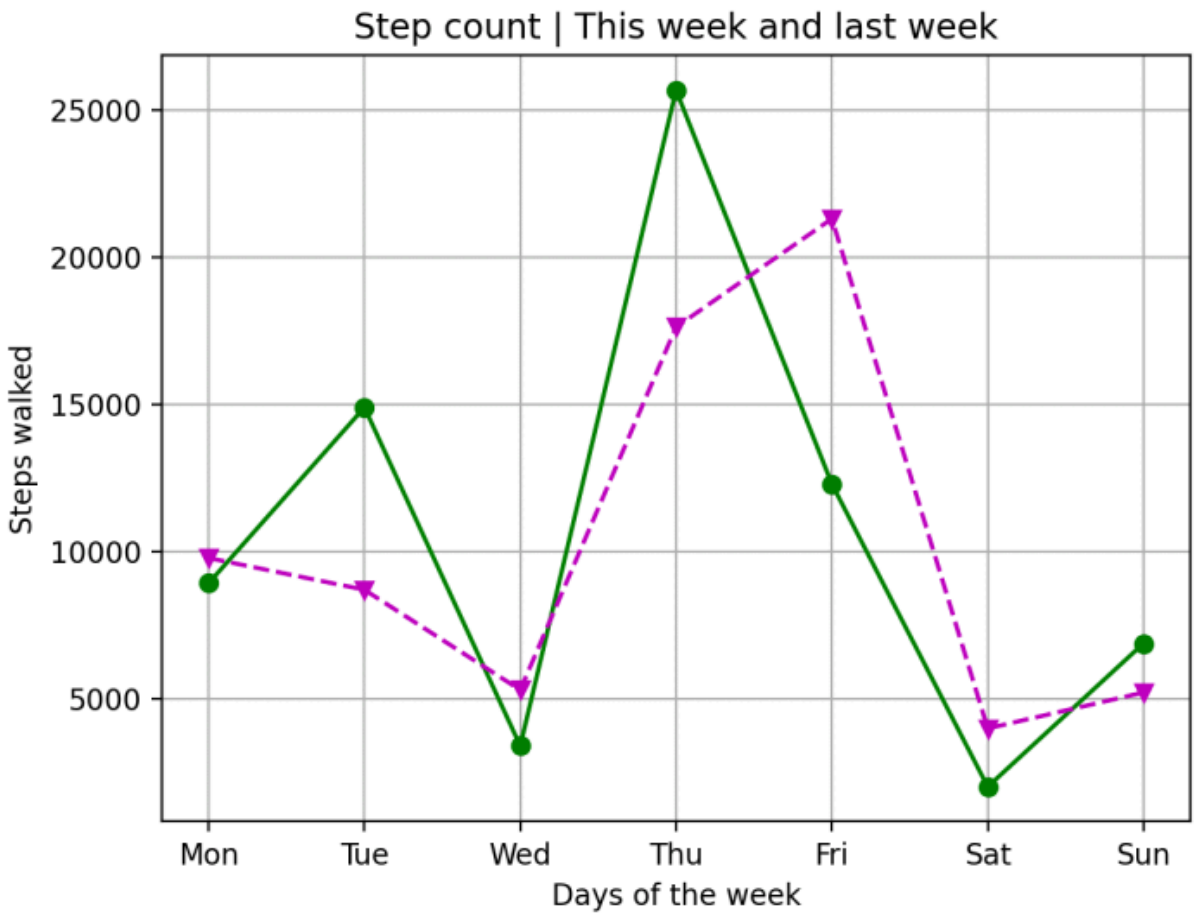


You can now add a second list with the number of steps walked the previous week so that you can compare this week's step count with the previous week's:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

plt.plot(days, steps_walked, "o-g")
plt.plot(days, steps_last_week, "v--m")
plt.title("Step count | This week and last week")
plt.xlabel("Days of the week")
plt.ylabel("Steps walked")
plt.grid(True)
plt.show()
```

You call `plot()` twice in the code above. One call plots the data in `steps_walked`. The format string you use is `"o-g"` which represents green circle markers and a solid line. The second call to `plot()` has `steps_last_week` as its second argument and the format string `"v--m"` which represents magenta triangle markers connected with a dashed line.

You also include a call to the `grid()` function, which allows you to toggle a grid displayed on the graph. The code above gives the following output:

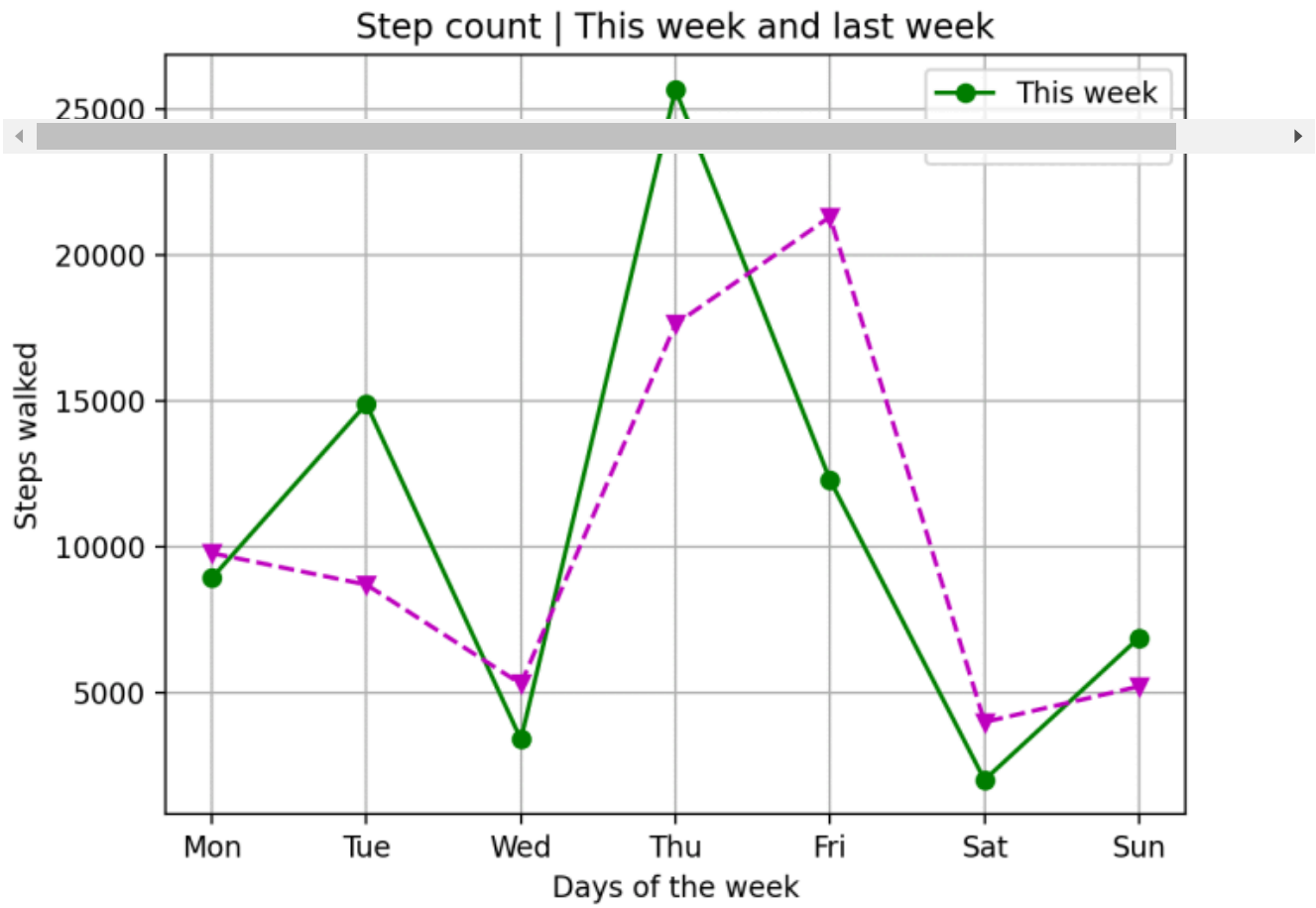


To finish off this graph, you need to identify which plot is which. You can add a legend to your plot:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

plt.plot(days, steps_walked, "o-g")
plt.plot(days, steps_last_week, "v--m")
```

```
plt.title("Step count | This week and last week")
plt.xlabel("Days of the week")
plt.ylabel("Steps walked")
plt.grid(True)
plt.legend(["This week", "Last week"])
plt.show()
```

You use a list of strings as an argument for `legend()` which gives the following figure:



Now that you know the basics of plotting using Matplotlib, you can dive a bit deeper into the various components that make up a figure.

What's a Matplotlib Figure Made Of?

When working with data visualisation in Python, you'll want to have control over all aspects of your figure. In this section, you'll learn about the main components that make up a figure in Matplotlib.

Everything in Python is an object, and therefore, so is a Matplotlib figure. In fact, a Matplotlib figure is made up of several objects of different data types.

There are three main parts to a Matplotlib figure:

- **`Figure`**: This is the whole region of space that's created when you create any figure. The `Figure` object is the overall object that contains everything else.
- **`Axes`**: An `Axes` object is the object that contains the *x*-axis and *y*-axis for a 2D plot. Each `Axes` object corresponds to a plot or a graph. You can have more than one `Axes` object in a `Figure`, as you'll see later on in this Chapter.
- **`Axis`**: An `Axis` object contains one of the axes, the *x*-axis or the *y*-axis for a 2D plot.

Therefore, a Matplotlib figure is a `Figure` object which has one or more `Axes` objects. Each `Axes` object has two or three `Axis` objects.

You can see the relationship between these three parts of a figure in the diagram below:

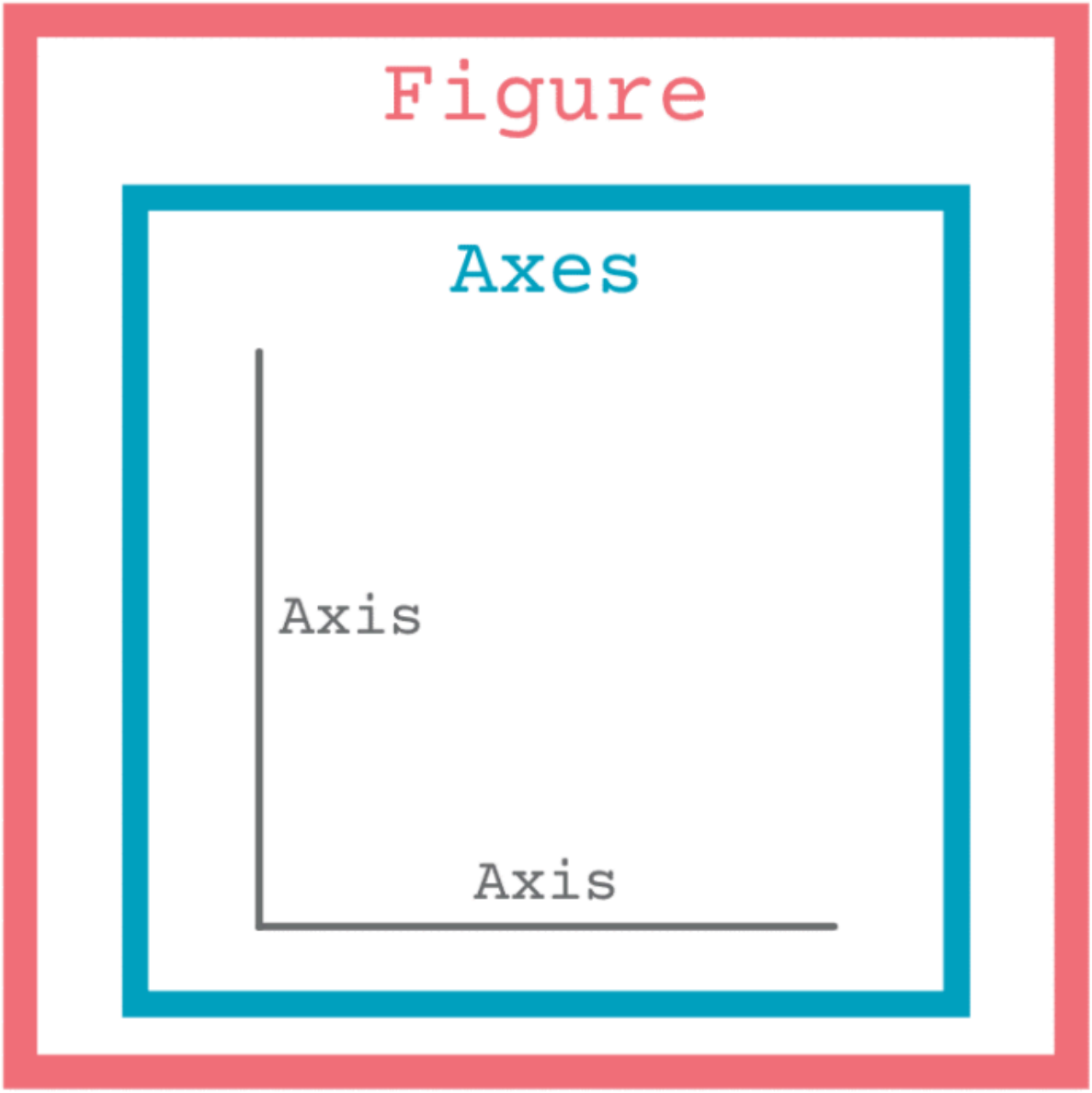
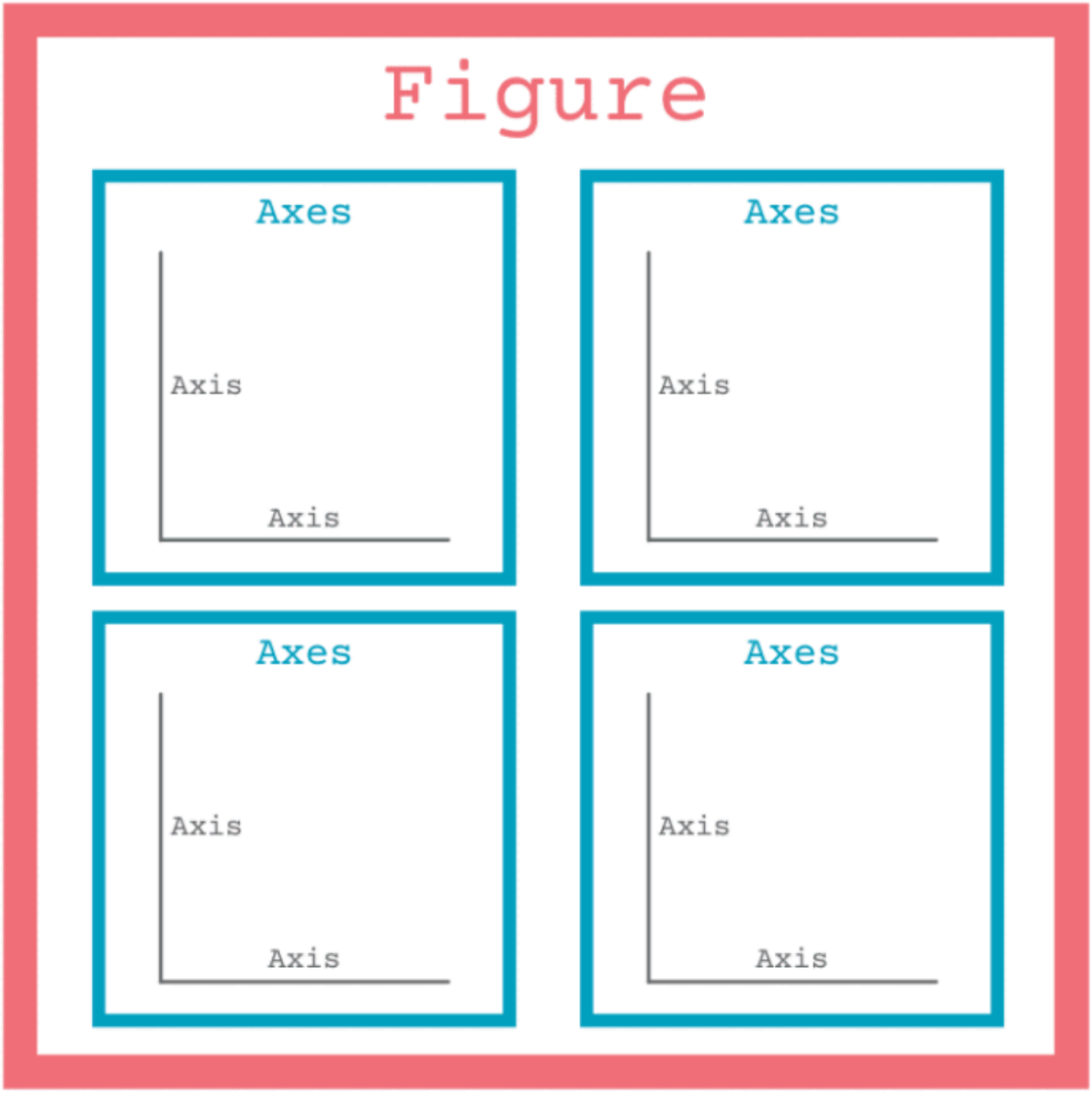


Figure
Axes
Axis
Axis



The first diagram shows the simplest figure you can create in Matplotlib in which the `Figure` object contains one `Axes` object. The `Axes` object contains two `Axis` objects. The diagram on the right shows four `Axes` objects within the same `Figure` object. These are called subplots, and you'll read more about them shortly.

There are other objects present in a figure, too. The general data type for objects in a figure is the **`Artist`** type. These include components of a figure such as the markers, lines connecting the data points, titles, legends, labels, and more.

The Alternative Way of Creating a Figure in Matplotlib

Matplotlib offers two ways of creating a figure. You've already seen how to use one of the interfaces earlier in this Chapter. In this section, you'll learn about the second option. You may wonder why you need to have two ways to do the same thing. You'll find that each interface has some advantages and disadvantages. The short answer is that one option is simpler to use, and the other option gives you more flexibility to customise.

The two methods are:

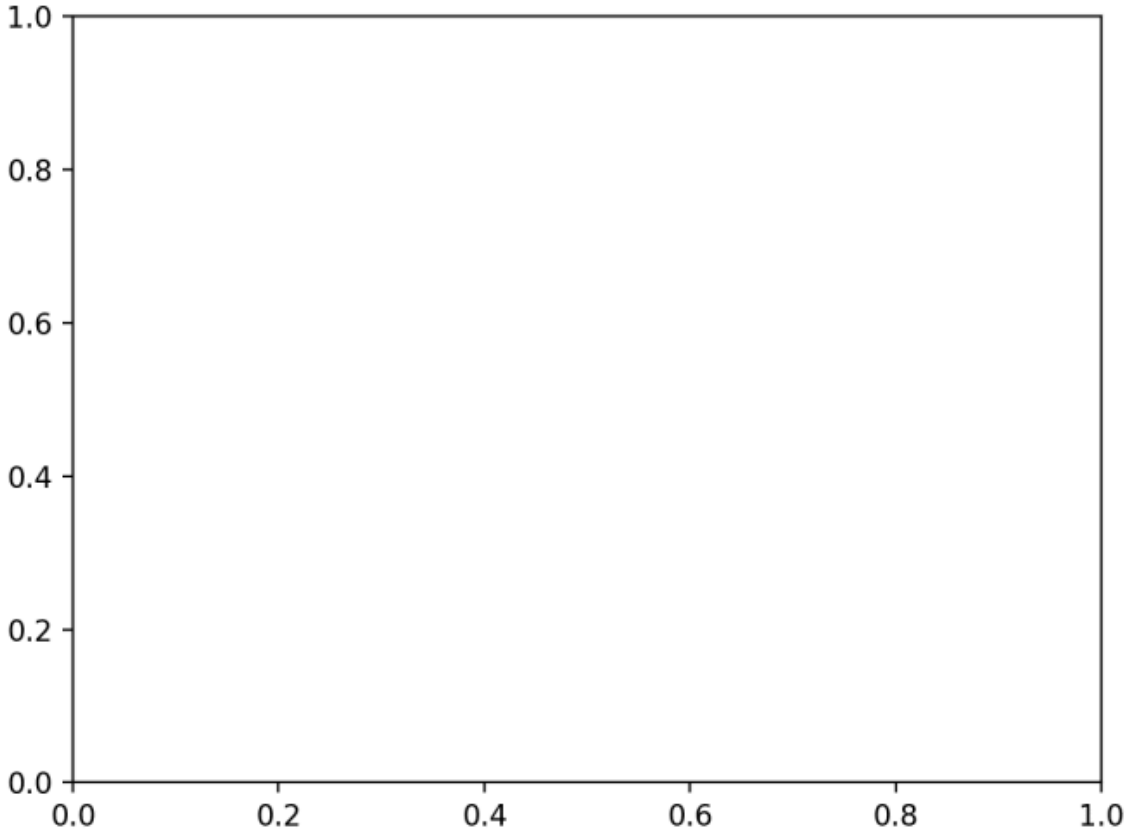
- Use `pyplot` functions directly. These functions will automatically create `Figure`, `Axes`, and other objects and manage them for you. This is the method you used earlier in this Chapter.
- Create `Figure` and `Axes` objects and call each object's methods. This is the object-oriented programming approach.

You can now recreate the last figure you plotted earlier using the object-oriented method. To create a figure, you can use the function `subplots()`, which returns a tuple containing a `Figure` object and an `Axes` object when it's called without arguments:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

plt.show()
```

By convention, the names `fig` and `ax` are used for `Figure` and `Axes` objects, although you can, of course, use other variable names if you have a reason to do so.

The visual output from this code is a figure containing a pair of axes:



Within your code, you have access to the `Figure` object and the `Axes` object separately through the variables `fig` and `ax`. Note that when using the simpler method earlier, you didn't need to call any function to create the `Figure` or `Axes` objects as this was done automatically when you first call `plt.plot()`.

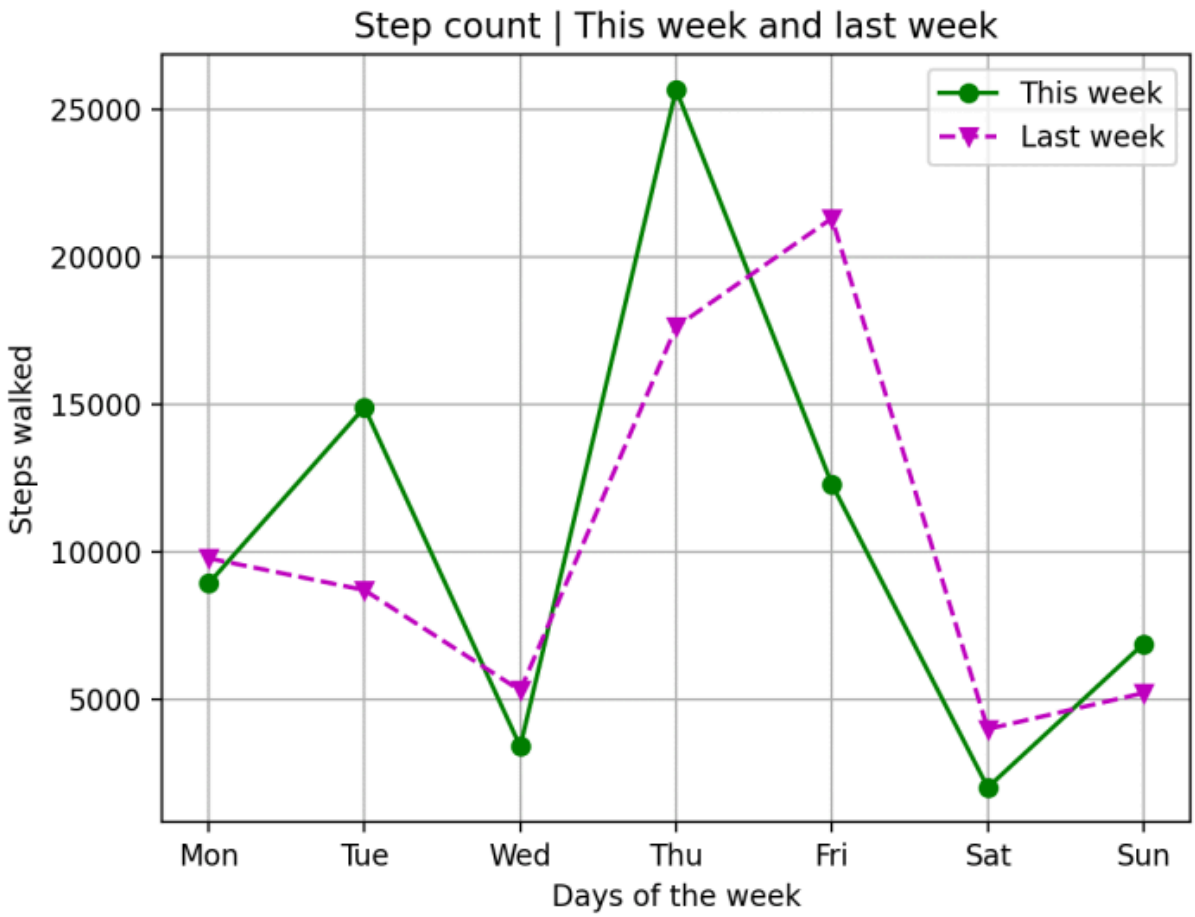
You can now plot the two sets of data and add the other components you had earlier:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 68
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, ax = plt.subplots()

ax.plot(days, steps_walked, "o-g")
ax.plot(days, steps_last_week, "v--m")
ax.set_title("Step count | This week and last week")
ax.set_xlabel("Days of the week")
ax.set_ylabel("Steps walked")
```

```
ax.grid(True)
ax.legend(["This week", "Last week"])
```

You can compare this code with the example you wrote earlier using the simpler method. You'll notice a few differences. Firstly, you're now calling methods of the `Axes` class by using the variable name `ax`. In the previous code, you used `plt`, which is the alias for the submodule `matplotlib.pyplot`.

Secondly, although some method names mirror the function names you used earlier, others are different. You've used the methods `set_title()`, `set_xlabel()`, and `set_ylabel()` which have different names to the `plt` functions you used earlier.



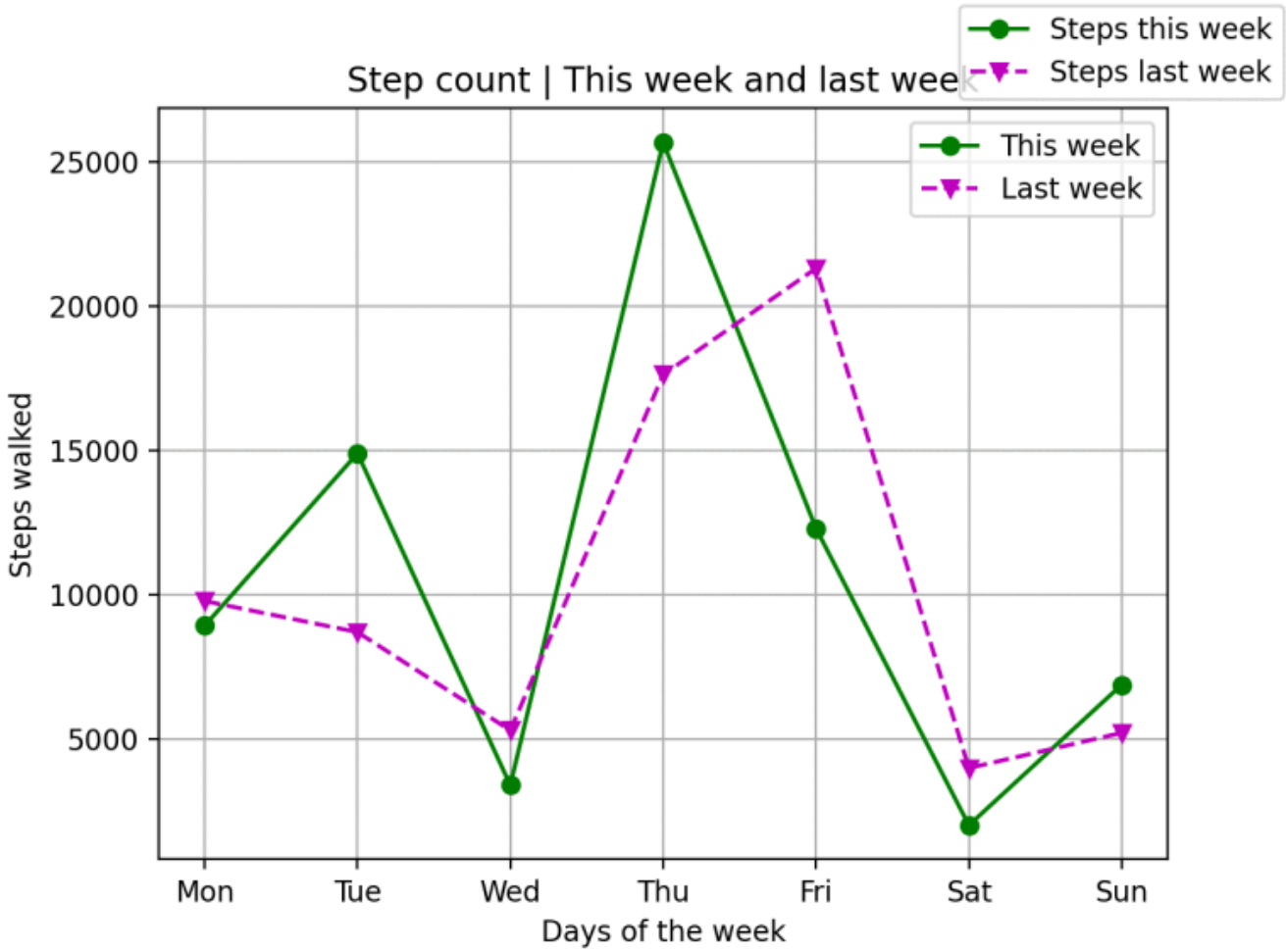
This figure is identical to the one you plotted earlier. So, why do we need two methods? Let's start exploring the additional flexibility you get from the object-oriented interface.

Adding `Figure`-specific components

The methods `plot()` and `grid()` have to be methods associated with `Axes` as that's where the plot goes. However, the legend and title of a figure could be either linked to the `Axes` or the `Figure`. You can start by creating a second legend but this time linked to the `Figure` object:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, ax = plt.subplots()

ax.plot(days, steps_walked, "o-g")
ax.plot(days, steps_last_week, "v--m")
ax.set_title("Step count | This week and last week")
ax.set_xlabel("Days of the week")
ax.set_ylabel("Steps walked")
ax.grid(True)
ax.legend(["This week", "Last week"])

fig.legend(["Steps this week", "Steps last week"])

plt.show()
```

You're now calling two methods called `legend()`. However, `ax.legend()` is a method of the `Axes` class, whereas `fig.legend()` is a method of the `Figure` class. The text in the figure-wide legend is different in this example to make it easier to identify which legend is which:



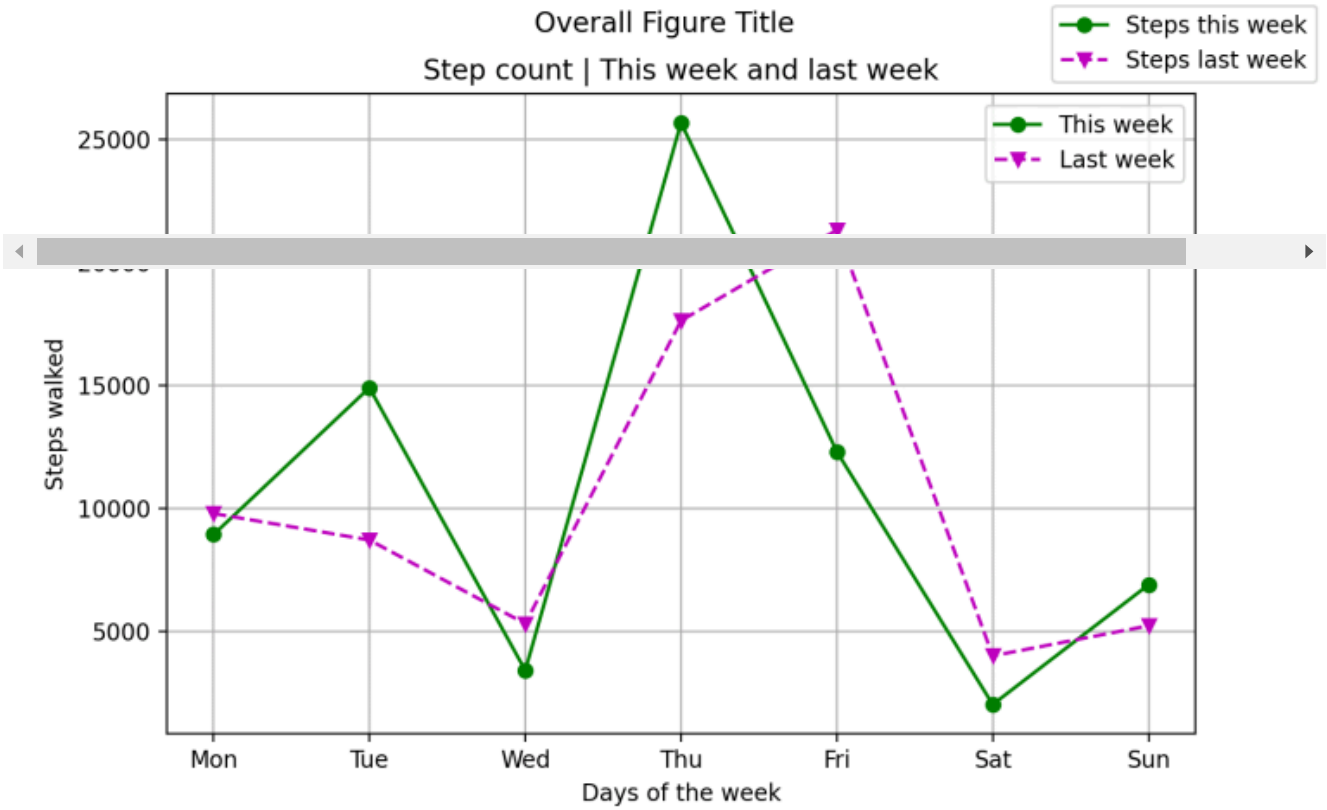
There are now two legends in the output. One is linked to the `Axes` object and the other to the `Figure` object. The reason why you may need access to both versions will become clearer when you learn about subplots later in this Chapter.

You may have noticed that the figure-wide legend is partially obscuring the title. You can customise your plot in any way you wish to resolve issues with the default sizes and positions. In this case, you can choose a wider size for your figure by setting the `figsize` parameter when you create the figure. You can also add a figure-wide title using the `suptitle()` method of the `Figure` class:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, ax = plt.subplots(figsize=(8, 5))

ax.plot(days, steps_walked, "o-g")
ax.plot(days, steps_last_week, "v--m")
ax.set_title("Step count | This week and last week")
```

```
    ax.set_xlabel("Days of the week")
    ax.set_ylabel("Steps walked")
    ax.grid(True)
    ax.legend(["This week", "Last week"])

    fig.legend(["Steps this week", "Steps last week"])
    fig.suptitle("Overall Figure Title")
```

The default size unit in Matplotlib is inches. The image displayed is now wider and the figure-wide legend no longer obscures the title. There are also two titles. One is linked to the `Axes` object and the other to the `Figure` object:



The distinction between figure-wide components and those specific to a set of axes will become clearer in the next section when you learn about subplots.

Creating Subplots

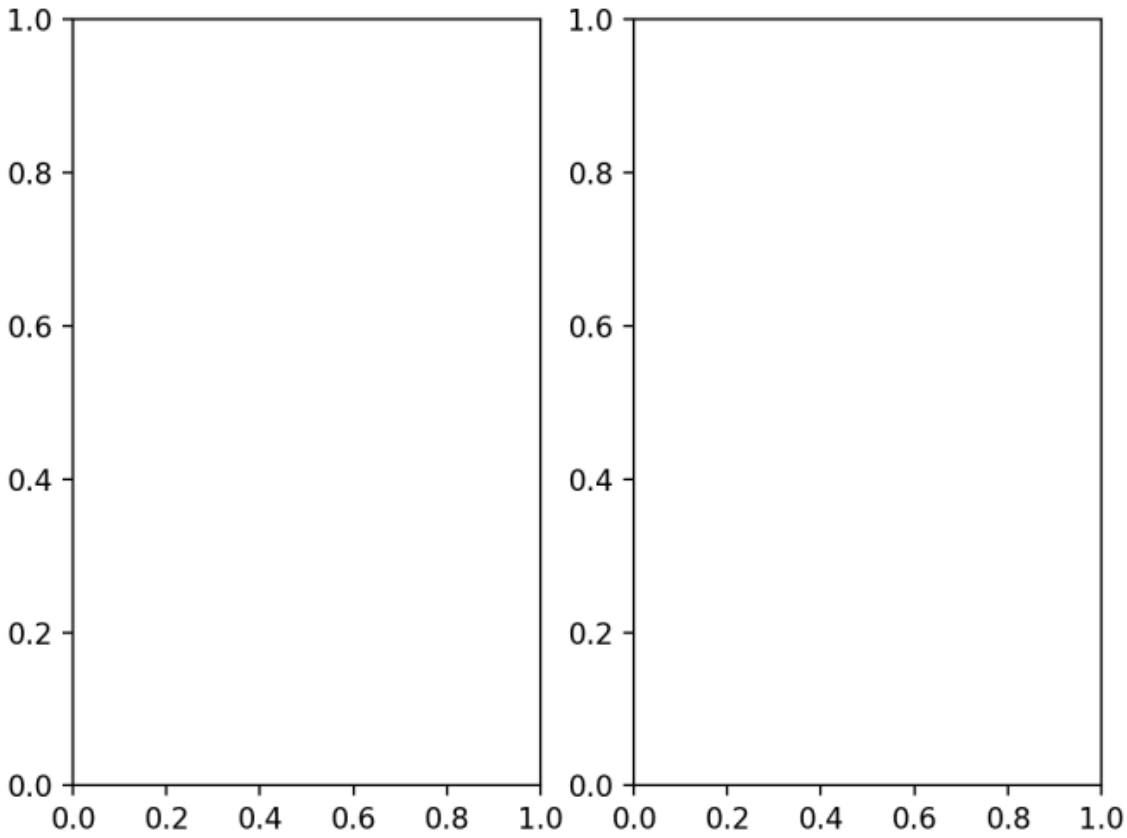
In the previous example, you used `plt.subplots()` either with no input arguments or with the keyword argument `figsize`. The function also has two positional arguments `nrows` and `ncols`, which both have a default value of `1`.

You can create a `Figure` object that contains more than one `Axes` object by using different values for `nrows` and `ncols`:

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 2)

plt.show()
```

The arguments `1, 2` define a grid of axes consisting of one row and two columns. Therefore, this call to `subplots()` creates two `Axes` objects. When you create multiple `Axes` in this way, by convention, you can use the variable name `axs`, which is a `numpy.ndarray` containing the `Axes` objects.

The output from this code shows the two sets of axes displayed in the same figure:



Therefore, `plt.subplots()` returns one of the following return values:

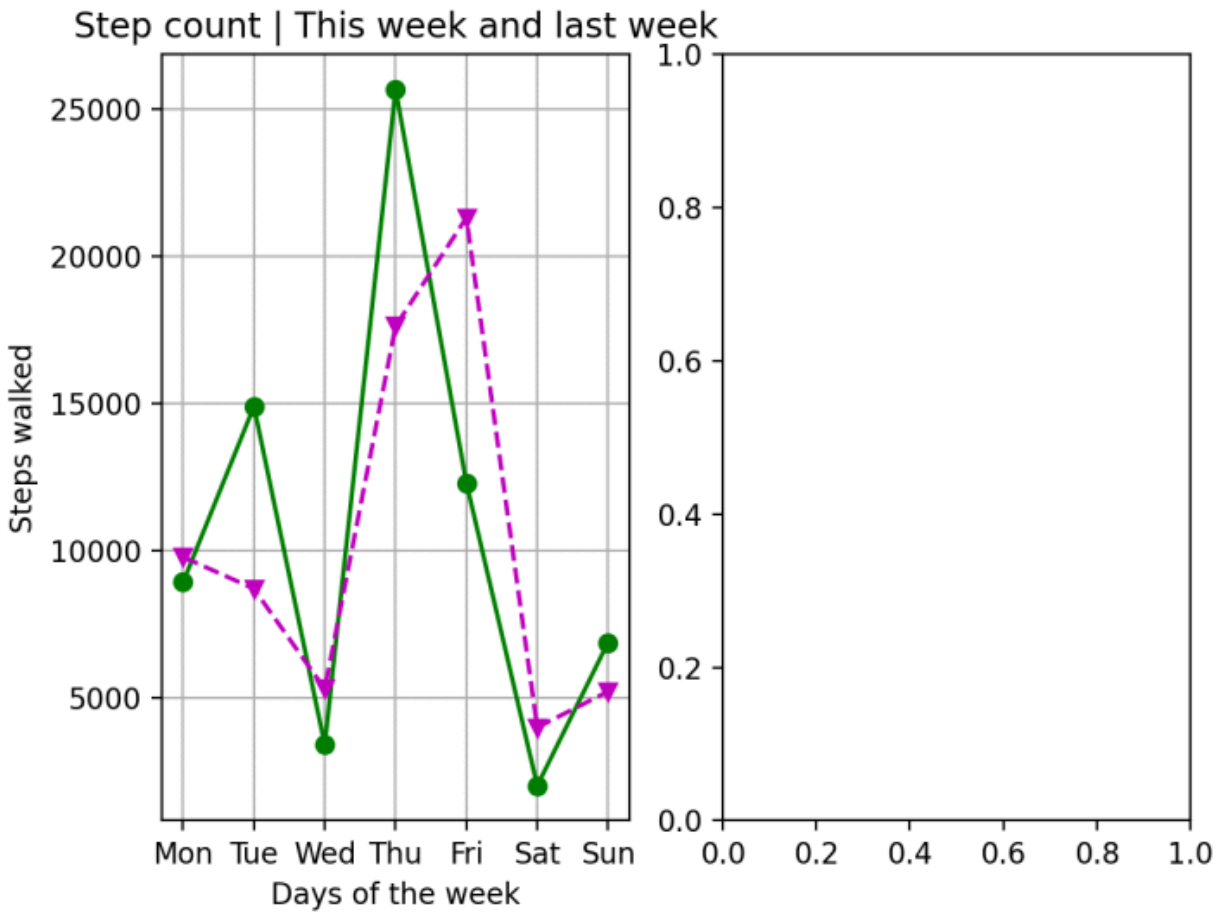
- a tuple containing a `Figure` object and an `Axes` object if `nrows` and `ncols` are both `1`. In this case, only one `Axes` object is created

- a tuple containing a `Figure` object and a `numpy.ndarray` if `nrows` and `ncols` aren't both `1`. The array contains `Axes` objects. The shape of the array is the same as the shape of the grid containing the subplots. Therefore, the call `plt.subplots(2, 2)`, for example, will return a `2x2` `numpy.ndarray` as its second return value.

Now, you can recreate the plot you worked on earlier in the first of these subplots:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 68
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, axs = plt.subplots(1, 2)

axs[0].plot(days, steps_walked, "o-g")
axs[0].plot(days, steps_last_week, "v--m")
axs[0].set_title("Step count | This week and last week")
axs[0].set_xlabel("Days of the week")
axs[0].set_ylabel("Steps walked")
axs[0].grid(True)

plt.show()
```

Since you're creating a `1x2` grid of subplots, the array `axs` is also a `1x2` array. Therefore, `axs[0]` is the `Axes` object representing the first set of axes in the figure. This code gives the following output:

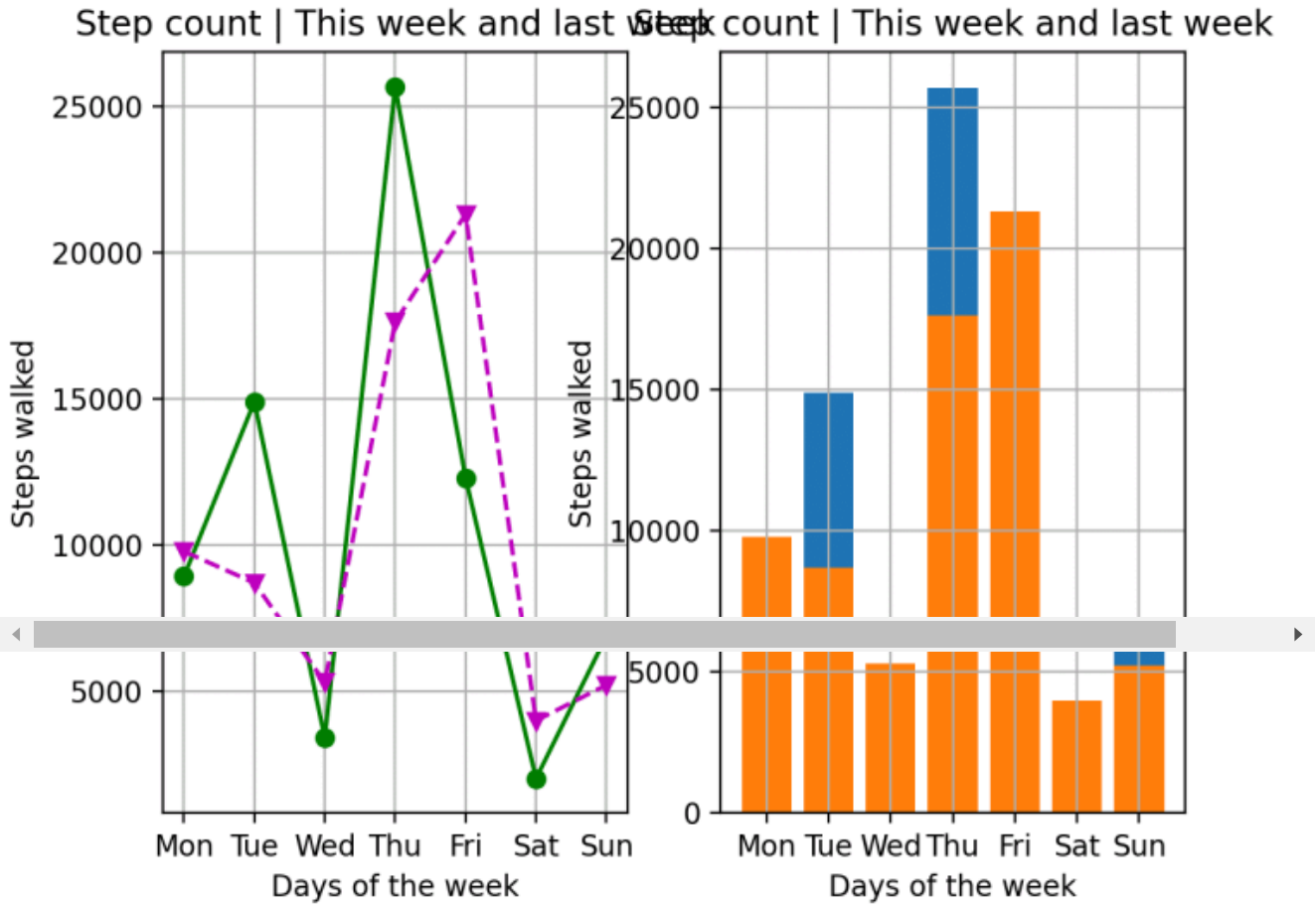


You can now represent the same data using a bar chart on the right-hand side:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, axs = plt.subplots(1, 2)

# Plot line chart
axs[0].plot(days, steps_walked, "o-g")
axs[0].plot(days, steps_last_week, "v--m")
axs[0].set_title("Step count | This week and last week")
axs[0].set_xlabel("Days of the week")
axs[0].set_ylabel("Steps walked")
axs[0].grid(True)

# Plot bar chart
axs[1].bar(days, steps_walked)
axs[1].bar(days, steps_last_week)
axs[1].set_title("Step count | This week and last week")
```

```
    axs[1].set_xlabel("Days of the week")
    axs[1].set_ylabel("Steps walked")
    axs[1].grid(True)

    plt.show()
```

On the second set of axes, you're now using the `Axes` method `bar()` to draw two bar charts, one for the steps walked for the current week and another for the previous week. The output from this code is the following:



You'll note that there are a couple of issues with this figure. Firstly, everything is cramped and there's overlap between elements of both subplots. You'll fix this by changing the size of the figure as you did earlier using the parameter `figsize`.

However, the main problem is that the second bar chart you plotted is drawn on top of the first one. This means that, for some of the days, the data from the previous week is obscuring the information from the current week. This issue happens for the data on Monday, Wednesday, Friday, and Saturday in this example.

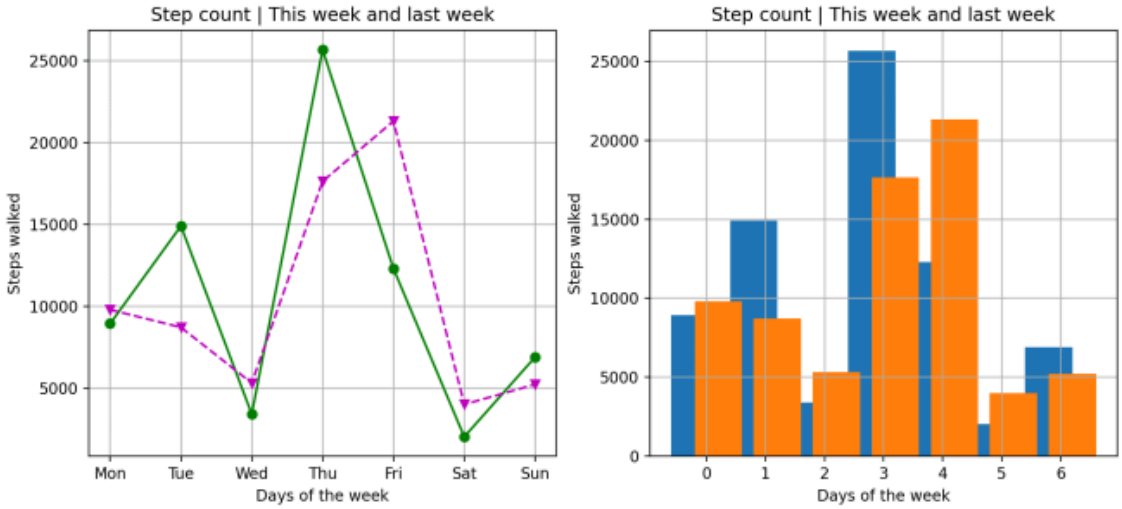
Customising the bar plot

You can fix this by shifting each plot sideways so that they don't overlap. Up until now, you used the list `days` as the data in the *x*-axis. You can get more control over where the bars are plotted by using a list of numbers instead. Start by creating two sets of *x*-coordinates for the two sets of bars. You can then use these lists as the first argument in the calls to `bar()`. You can also fix the cramming problem at this stage by setting the figure size when you call `plt.subplots()`:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Plot line chart
axs[0].plot(days, steps_walked, "o-g")
axs[0].plot(days, steps_last_week, "v--m")
axs[0].set_title("Step count | This week and last week")
axs[0].set_xlabel("Days of the week")
axs[0].set_ylabel("Steps walked")
axs[0].grid(True)

# Plot bar chart
x_range_current = [-0.2, 0.8, 1.8, 2.8, 3.8, 4.8, 5.8]
x_range_previous = [0.2, 1.2, 2.2, 3.2, 4.2, 5.2, 6.2]

axs[1].bar(x_range_current, steps_walked)
axs[1].bar(x_range_previous, steps_last_week)
axs[1].set_title("Step count | This week and last week")
axs[1].set_xlabel("Days of the week")
axs[1].set_ylabel("Steps walked")
axs[1].grid(True)

plt.show()
```

You're using the numbers `0` to `6` to represent the days of the week. The numbers in `x_range_current` are shifted by `-0.2` from the numbers in the range `0` to `6`. The

numbers in `x_range_previous` are shifted by `+0.2`. Therefore, when you use these values in the two calls to `bar()`, the bar charts plotted are shifted with respect to each other:

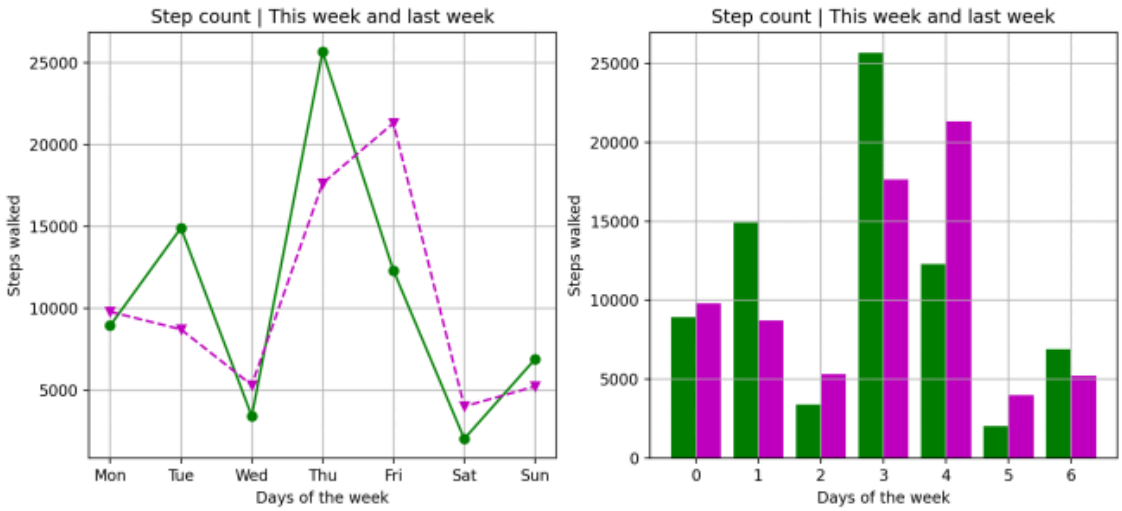


Although you can see the separate bars because of the shift, the bars are still overlapping. The default width of each bar is still too large. You can change the width of the bars to prevent them from overlapping. Since you shifted each set of bars by `0.2` from the centre, you can set the width of each bar to `0.4`. You can also change the colour of the bars so that you're using the same colour scheme as in the plot on the left-hand side:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Plot line chart
axs[0].plot(days, steps_walked, "o-g")
axs[0].plot(days, steps_last_week, "v--m")
axs[0].set_title("Step count | This week and last week")
axs[0].set_xlabel("Days of the week")
axs[0].set_ylabel("Steps walked")
axs[0].grid(True)

# Plot bar chart
x_range_current = [-0.2, 0.8, 1.8, 2.8, 3.8, 4.8, 5.8]
```

```
x_range_previous = [0.2, 1.2, 2.2, 3.2, 4.2, 5.2, 6.2]

axs[1].bar(x_range_current, steps_walked, width=0.4, col
axs[1].bar(x_range_previous, steps_last_week, width=0.4,
axs[1].set_title("Step count | This week and last week")
axs[1].set_xlabel("Days of the week")
axs[1].set_ylabel("Steps walked")
axs[1].grid(True)

plt.show()
```

This code gives the following figure:



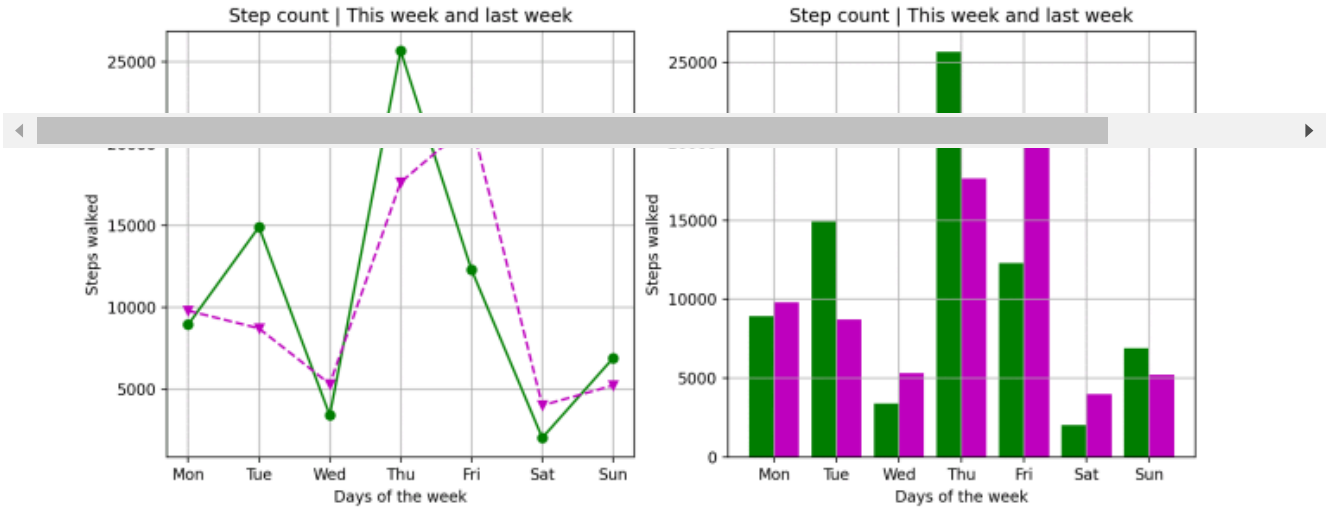
The problem now is that the labels on the *x*-axis no longer show the days of the week. You'll also notice that the ticks on the *x*-axis are not the values you're using in either of the bar charts you plotted. Matplotlib works out where to place the

You can do so using the `set_xticks()` method. You can also change the labels for these ticks using `set_xticklabels()`:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Plot line chart
axs[0].plot(days, steps_walked, "o-g")
```

```
axs[0].plot(days, steps_last_week, "v--m")
axs[0].set_title("Step count | This week and last week")
axs[0].set_xlabel("Days of the week")
axs[0].set_ylabel("Steps walked")
axs[0].grid(True)

# Plot bar chart
x_range_current = [-0.2, 0.8, 1.8, 2.8, 3.8, 4.8, 5.8]
x_range_previous = [0.2, 1.2, 2.2, 3.2, 4.2, 5.2, 6.2]

axs[1].bar(x_range_current, steps_walked, width=0.4, col
axs[1].bar(x_range_previous, steps_last_week, width=0.4,
axs[1].set_title("Step count | This week and last week")
axs[1].set_xlabel("Days of the week")
axs[1].set_ylabel("Steps walked")
axs[1].grid(True)
axs[1].set_xticks(range(7))
axs[1].set_xticklabels(days)

plt.show()
```

The call to `set_xticks()` determines where the ticks are placed on the *x*-axis. You'll recall that `range(7)` represents the integers between `0` and `6`. The call to `set_xticklabels()` then maps the strings in `days` to these ticks on the *x*-axis. This gives the following figure:



Before finishing off this figure, let's tidy up this code to make it more Pythonic

Making the code more Pythonic!

When writing code, it's often convenient to hardwire values in the code as you try out and explore options. However, you should aim to refactor your code to tidy it up when possible. Refactoring means making some changes to how the code looks, but not what it does, to make the code more future-proof and easier to read and maintain.

Now that you know the width of the bars and how much to shift them, you can refactor your code as follows:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Plot line chart
axs[0].plot(days, steps_walked, "o-g")
axs[0].plot(days, steps_last_week, "v--m")
axs[0].set_title("Step count | This week and last week")
axs[0].set_xlabel("Days of the week")
axs[0].set_ylabel("Steps walked")
axs[0].grid(True)

# Plot bar chart
bar_width = 0.4
x_range_current = [idx - bar_width/2 for idx in range(7)
x_range_previous = [idx + bar_width/2 for idx in range(7

axs[1].bar(x_range_current, steps_walked, width=bar_widt
axs[1].bar(x_range_previous, steps_last_week, width=bar_
axs[1].set_title("Step count | This week and last week")
axs[1].set_xlabel("Days of the week")
axs[1].set_ylabel("Steps walked")
axs[1].grid(True)
axs[1].set_xticks(range(7))
axs[1].set_xticklabels(days)

plt.show()
```

You define a variable called `bar_width` and then use it within list comprehensions to generate the shifted *x*-coordinate values for the two sets of bars. The figure displayed by this code is unchanged.

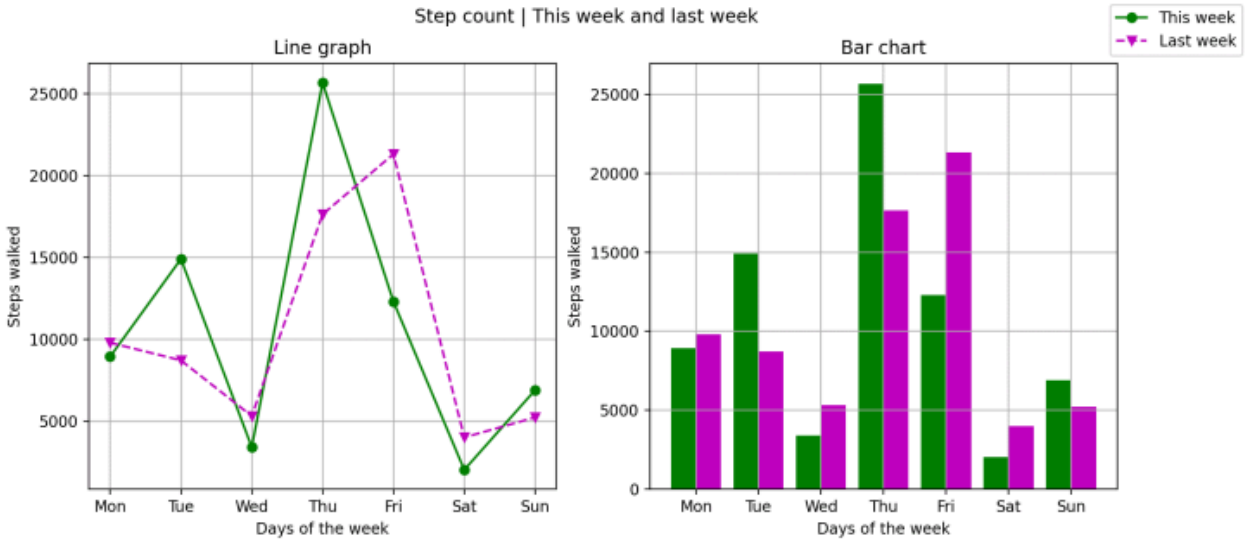
Choosing `Figure` and `Axes` components

You can now decide which components should be figure-wide and which are specific to one of the `Axes` objects. You can start by adding a legend to the figure. Since the legend should be the same for both subplots, you can use the `Figure` method `legend()` rather than the one that belongs to the `Axes` class. You can also move the separate `Axes` titles, which are identical to a `Figure` title, and replace the `Axes` titles with something more specific:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Plot line chart
axs[0].plot(days, steps_walked, "o-g")
axs[0].plot(days, steps_last_week, "v--m")
axs[0].set_title("Line graph")
axs[0].set_xlabel("Days of the week")
axs[0].set_ylabel("Steps walked")
axs[0].grid(True)

# Plot bar chart
bar_width = 0.4
x_range_current = [idx - bar_width/2 for idx in range(7)
x_range_previous = [idx + bar_width/2 for idx in range(7

axs[1].bar(x_range_current, steps_walked, width=bar_widt
axs[1].bar(x_range_previous, steps_last_week, width=bar_
axs[1].set_title("Bar chart")
axs[1].set_xlabel("Days of the week")
axs[1].set_ylabel("Steps walked")
axs[1].grid(True)
axs[1].set_xticks(range(7))
```

```
axs[1].set_xticklabels(days)

# Figure-wide components
fig.suptitle("Step count | This week and last week")
fig.legend(["This week", "Last week"])
```

The separate control you have over the `Figure` object and `Axes` objects allows you to customise the figure in any way you wish. The code displays the following figure:

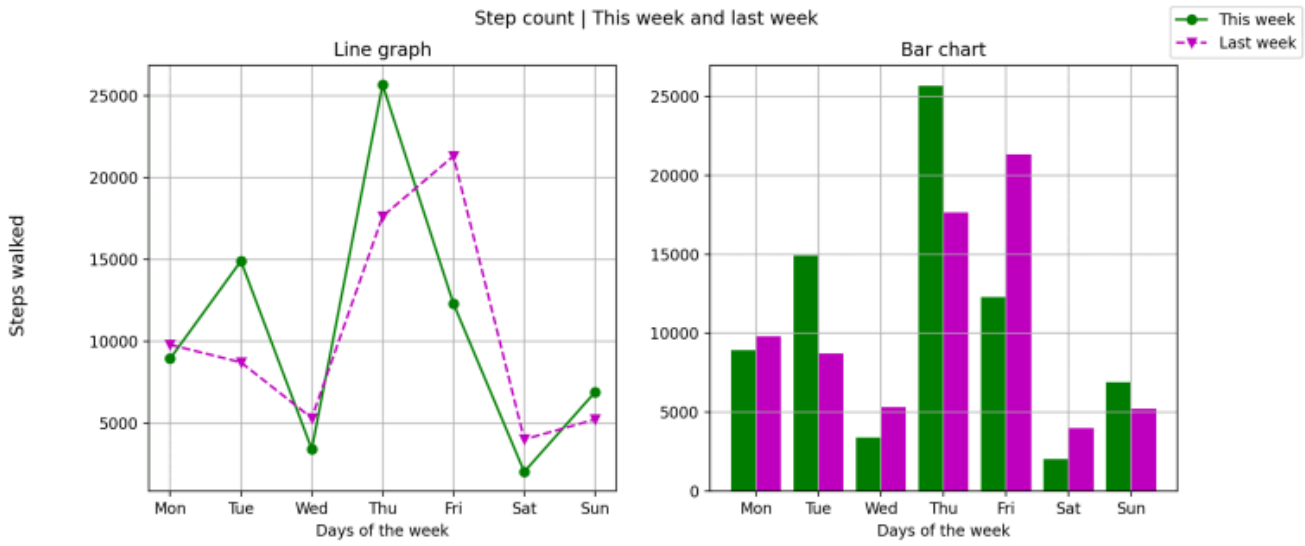


To demonstrate this further, you can also remove the separate *y*-axis labels from each `Axes` object and add a single figure-wide *y*-axis label:

```
import matplotlib.pyplot as plt

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
steps_walked = [8934, 14902, 3409, 25672, 12300, 2023, 6
steps_last_week = [9788, 8710, 5308, 17630, 21309, 4002,

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Plot line chart
axs[0].plot(days, steps_walked, "o-g")
axs[0].plot(days, steps_last_week, "v--m")
axs[0].set_title("Line graph")
axs[0].set_xlabel("Days of the week")
axs[0].grid(True)

# Plot bar chart
```

```
bar_width = 0.4
x_range_current = [idx - bar_width/2 for idx in range(7)
x_range_previous = [idx + bar_width/2 for idx in range(7

axs[1].bar(x_range_current, steps_walked, width=bar_widt
axs[1].bar(x_range_previous, steps_last_week, width=bar_
axs[1].set_title("Bar chart")
axs[1].set_xlabel("Days of the week")
axs[1].grid(True)
axs[1].set_xticks(range(7))
axs[1].set_xticklabels(days)

# Figure-wide components
fig.suptitle("Step count | This week and last week")
fig.legend(["This week", "Last week"])
fig.supylabel("Steps walked")

fig.savefig("steps_comparison.png")

plt.show()
```

You've also added a call to the `Figure` method `savefig()`, which allows you to save the figure to file. The final output from this example is the following figure:



You'll also find a PNG file named steps_comparison.png in your Project folder.

In the Snippets section, there are additional examples of more complex subplot grids.

Comparison Between The Two Matplotlib Interfaces

You've learned about the two ways of creating figures in Matplotlib. In the simpler option, you use functions within the submodule `matplotlib.pyplot` directly. You use calls such as `plt.plot()` and `plt.title()`. Matplotlib will automatically create and manage the objects for you. This option is useful as it's quicker and easier to use. However, it gives you less flexibility to customise your figures.

In the alternative way, you create `Figure` and `Axes` objects using `plt.subplots()` and then you call methods of those two classes. Dealing with instances of `Figure` and `Axes` directly gives you more control over your figure.

Which option should you use? Both interfaces are available to you when using Matplotlib, and therefore, you can use whichever one you're more comfortable with. The more direct approach is easier to start with. However, once you understand the anatomy of a figure, in particular how you have a `Figure` object that contains one or more `Axes` objects, you may prefer to use the object-oriented version in most cases.

The Matplotlib documentation recommends using the simpler version when creating quick graphs in interactive environments like the Console or Jupyter and when exploring your data. However, for all other purposes, the object-oriented interface may be preferable.

You can find all the functions available to use in the direct approach on the `pyplot` documentation page. If you're using the object-oriented approach, you can find all the methods you need in the `Figure` class documentation page and in the `Axes` class documentation page.

As you become more proficient with Matplotlib, and if you require more complex plots, you can also dive further into other classes defined in Matplotlib. However, for the time being, the functions available in `pyplot` and the methods of the `Figure` and `Axes` classes are more than enough!

The rest of this Chapter will give a brief overview of some other plots you can create with Matplotlib.

Displaying Images Using Matplotlib

An image is an array of numbers. Therefore, it is possible to deal with images using the same tools as when dealing with any array of numbers. In this section, you'll see how you can perform basic image manipulation using Matplotlib and NumPy.

There are other libraries in Python to deal with images and, in particular, to deal with image processing, machine vision, and related fields. We will not cover any of these in this book. This section aims is to give you a basic introduction to dealing with images from within a computer program.

You'll use a PNG image in this example, but you can use images of most standard formats in the same manner. You can download the image you'll use in this example from **The Python Coding Book File Repository**. You'll need the file named `balconies.png`, and *you should place the file in your Project folder*.

> **Download The Python Coding Book File Repository**
>
> Through the link above, you can download the folder you need directly to your computer. I would recommend this option which is the most straightforward. But if you prefer, you can also access the repository through Github.

You can read in the image using `plt.imread()` and explore what data type the function returns:

```
import matplotlib.pyplot as plt

img = plt.imread("balconies.png")

print(type(img))
print(img.shape)
print(img[100, 100, 0])
```

The output from the three calls to `print()` is the following:

```
<class 'numpy.ndarray'>
(453, 456, 4)
0.7294118
```

`type(img)` shows that `imread()` returns a NumPy `ndarray`. The shape of the array is `(453, 456, 4)`. The first two values in this tuple show that this image is `453x456` pixels large. The `4` in the final position in the tuple shows that there are four *layers* of numbers. You'll learn about what these four layers are soon.

The final call to `print()` returns the value of one of the cells in the array. In this case you're printing the value of the pixel at `(100, 100)` in the first layer out of the four layers present. The values in this image range from `0` to `1`. In some images, the image values can range from `0` to `255`, too.

Representing an image using an array

An image can be represented using one of three types of arrays:

1. an `(MxN)` array represents a **grayscale image** that's `MxN` pixels large.

- This array has just one layer, which represents the grayscale values of the image. Each value in the array represents the grayscale value of that pixel.
- Images typically either have values ranging from `0` to `1` or from `0` to `255`.
- On the `0...1` scale, `0` represents black, and `1` represents white. On the `0...255` scale, `0` represents black, and `255` represents white.

2. an `(MxNx3)` array represents a **colour image** that's `MxN` pixels large.

- This array has three layers, each layer having `MxN` pixels.
- The first layer represents the amount of red in the image, the second layer represents green, and the third layer represents the level of blue. This is the **RGB colour model** of images.
- This means that each pixel of an image can represent over 16 million different colours (256 x 256 x 256).

3. an `(MxNx4)` array represents a **colour image with transparency**. The image is `MxN` pixels large.

- The array has four layers, each layer having `MxN` pixels.
- The first three layers are the RGB layers as described above.

- The fourth layer represents the alpha value. This indicates what level of transparency the pixel has, ranging from fully transparent to fully opaque. This is the **RGBA colour model** of images.

What's special about the number 256? Eight computer bits are used for each pixel for each colour. A single bit can take one of two values, either `0` or `1`. Therefore, eight bits can represent 2^8 values, which is equal to 256.

In this example, you can ignore the alpha channel. You can discard the information in the fourth layer:

```
import matplotlib.pyplot as plt

img = plt.imread("balconies.png")
img = img[:, :, :3]

print(img.shape)
```

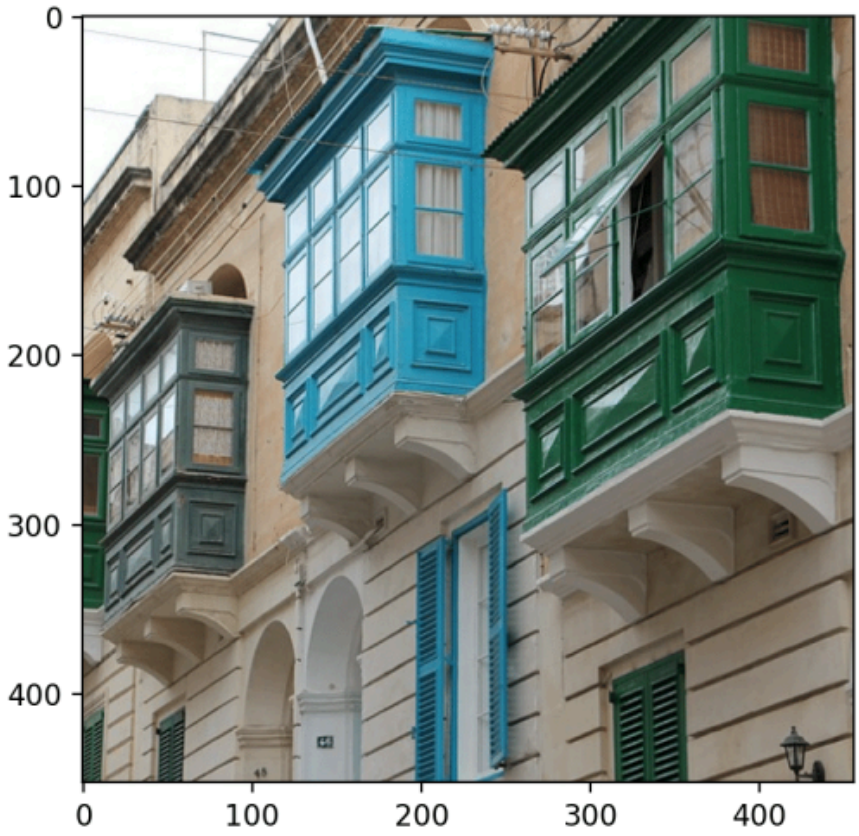
You're keeping all the pixels in the layers `0, 1, 2` since the slice `:3` which you use for the third dimension of the array represents the indices from `0` up to but excluding `3`. The output confirms that the shape of the array `img` is now `(453, 456, 3)`.

Showing the images and the separate colour channels

You can use Matplotlib to display the image directly in a Matplotlib figure:

```
import matplotlib.pyplot as plt

img = plt.imread("balconies.png")
img = img[:, :, :3]

plt.imshow(img)
plt.show()
```

You use the function `imshow()` in `matplotlib.pyplot`. This gives the following figure:

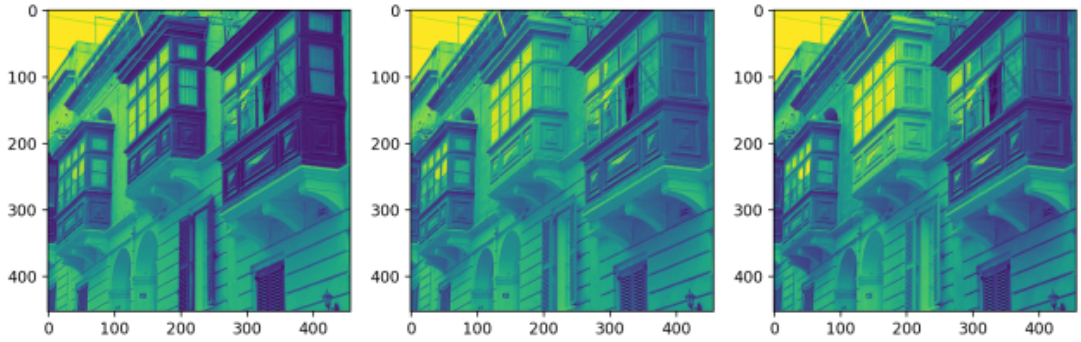


You can now create a series of subplots to show the red, green, and blue components of this image separately. You'll shift to using the object-oriented way of creating figures:

```
import matplotlib.pyplot as plt

img = plt.imread("balconies.png")
img = img[:, :, :3]

fig, axs = plt.subplots(1, 3, figsize=(12, 4))
axs[0].imshow(img[:, :, 0])
axs[1].imshow(img[:, :, 1])
axs[2].imshow(img[:, :, 2])

plt.show()
```

You learned earlier that `axs` is a NumPy `ndarray` containing `Axes` objects. Therefore, `axs[0]` is the `Axes` object for the first subplot. And the same applies to the other two subplots.

The image you get is not quite what you might have expected:



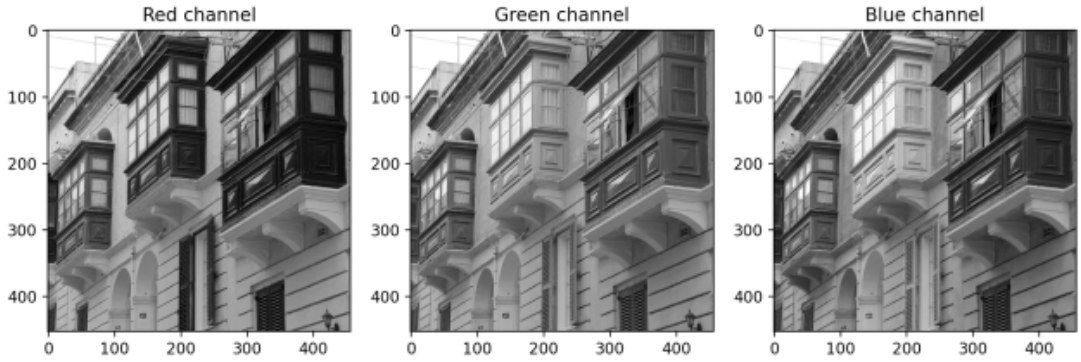
The three images are not the same. You can see this when comparing the first one (the red channel) compared to the other two.

`imshow()` uses a default colour map to represent the three images as each image only has one layer now. Therefore, these are grayscale images. A colour map is a mapping between colours and values. Earlier, when you displayed the `MxNx4` array `img`, Matplotlib recognised this as a colour image and therefore displayed it as a colour image.

You can change the colour map for each subplot to grayscale:

```
import matplotlib.pyplot as plt

img = plt.imread("balconies.png")
img = img[:, :, :3]

fig, axs = plt.subplots(1, 3, figsize=(12, 4))
axs[0].imshow(img[:, :, 0], cmap="gray")
axs[0].set_title("Red channel")

axs[1].imshow(img[:, :, 1], cmap="gray")
axs[1].set_title("Green channel")

axs[2].imshow(img[:, :, 2], cmap="gray")
axs[2].set_title("Blue channel")

plt.show()
```

You use the keyword parameter `cmap` to switch to a grayscale colour map. You've also added titles to each subplot to identify each colour channel in the image. The output now shows the three channels displayed in grayscale:



Understanding the three colour channels

Let's make sense of what you see in these images. When you look at the original colour image, you see that the balcony at the centre of the screen has a light blue colour and the one on the right has a dark green colour. Let's focus on the central balcony first. Light blue consists of high values for blue and green and a low value in the red channel.

When you look at the three separate channels, the middle balcony appears dark in the red-channel image. This shows that there isn't a lot of red in those pixels. The green and blue channels show the middle balcony in a lighter colour, showing that there's a lot of green and blue in the pixels that make up the middle balcony. The balcony appears nearly white in the blue channel because these pixels are almost at their maximum level in the blue channel.

Still looking at the middle balcony, if you look at the windows, you'll notice that these are shown in a bright shade in all three colour channels. In the colour image, the reflection from these windows makes them look white, and white is represented by maximum values for all three channels: red, green, and blue.

The balcony on the right has a dark green colour. In the three separate subplots, you can see that the balcony appears almost black in the red channel. There's very little red in these pixels. This balcony appears brightest in the green channel. However, as the balcony is dark green, it only appears as an intermediate shade of grey in the green channel.

When dealing with data visualisation in Python, you may have images as part of your data set. You can now start exploring any image using Matplotlib.

Plotting in 3D

Another common requirement in data visualisation in Python is to display 3d plots. You can plot data in 3D using Matplotlib.

In the Chapter about using NumPy, the final section dealt with representing equations in Python. You'll use the same example in this section, but you'll convert the equation you used in that section from 1D into 2D.

Note: *I'm avoiding using the term function to refer to mathematical functions to avoid confusion with a Python function. Although there are similarities between a mathematical function and a Python function, there are also significant differences. You'll read more about this topic in the Chapter on Functional Programming.*

In the Chapter about NumPy you plotted the following equation:

$$y = \frac{\sin(x - a)}{x}$$

You can simplify this by making `a = 0`:

$$y = \frac{\sin(x)}{x}$$

This is known as the sinc function in mathematics. You can consider a 2D version of this equation:

$$z = \frac{\sin(x)}{x}\frac{\sin(y)}{y}$$

You can create the arrays that represent the *x*-axis and *y*-axis:

```
import numpy as np

x = np.linspace(-10, 10, 1000)
y = np.linspace(-10, 10, 1000)
```

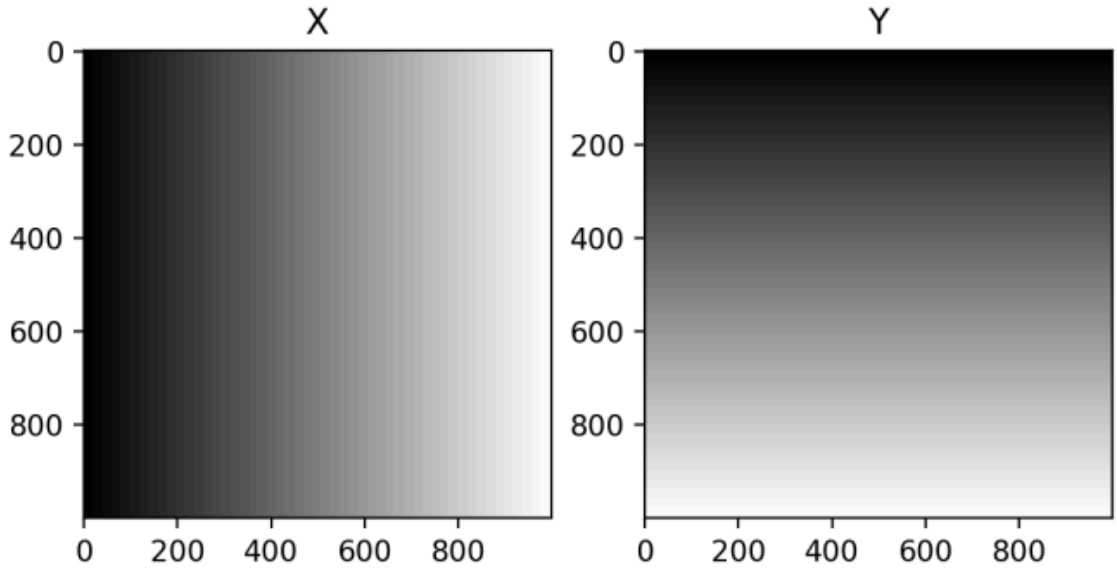
1D arrays such as `x` and `y` were sufficient to create 1D equations. However, *z* depends on two variables. Therefore, the data structure that will hold the values of *z* needs to be a 2D array.

Using `meshgrid()`

You can convert the 1D arrays `x` and `y` into their 2D counterparts using the function `np.meshgrid()`:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-10, 10, 1000)
y = np.linspace(-10, 10, 1000)

X, Y = np.meshgrid(x, y)

# Temporary code: Visualising X and Y returned by meshgri
fig, axs = plt.subplots(1, 2)
axs[0].imshow(X, cmap="gray")
axs[0].set_title("X")
axs[1].imshow(Y, cmap="gray")
axs[1].set_title("Y")

plt.show()
```

The `meshgrid()` function returns two 2D `ndarrays` which you're naming `X` and `Y`. They extend the 1D arrays `x` and `y` into two dimensions. You show these arrays as images using `imshow()` to get the following figure:

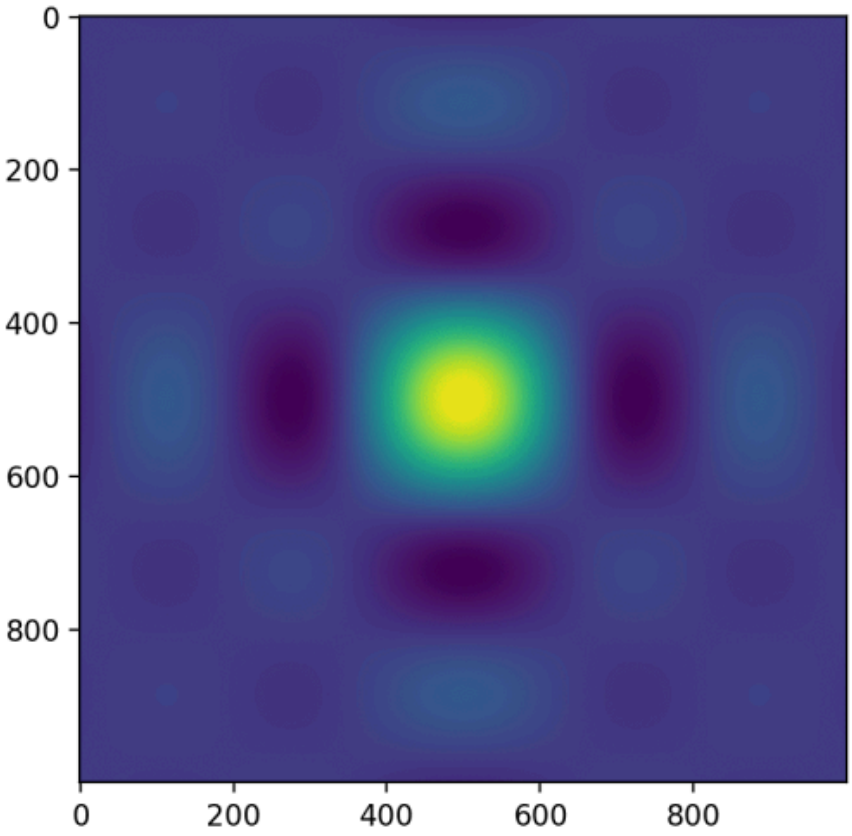


The values of `X` and `Y` range from `-10` to `10`. Black represents `-10`, and white represents `10` in these plots. You can see from the plots that `X` varies from `-10` to `10` from left to right, and `Y` varies across the same range from top to bottom.

You can now use `X` and `Y` to create `Z` using the equation above:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-10, 10, 1000)
y = np.linspace(-10, 10, 1000)

X, Y = np.meshgrid(x, y)

Z = (np.sin(X) / X) * (np.sin(Y) / Y)

plt.imshow(Z)
plt.show()
```

You create `Z` from the 2D arrays `X` and `Y`, and therefore, `Z` is also a 2D array. The figure created when you use `imshow()` is the following:



The colour in the 2D image represents the third dimension. In this colour map, the yellow colour represents the highest values, and the dark purple colours are the lowest values in the array `Z`.

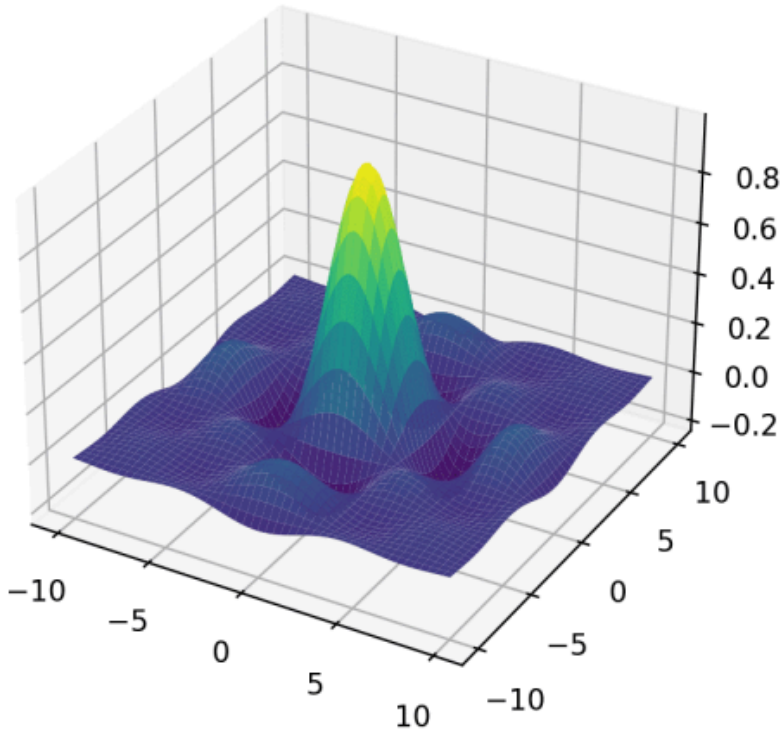
Plotting in 3D

However, another way of visualising the 2D equation is by using a 3D plot. Earlier in this Chapter, you created `Axes` objects that were 2D. You can also create 3D axes:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-10, 10, 1000)
y = np.linspace(-10, 10, 1000)

X, Y = np.meshgrid(x, y)

Z = (np.sin(X) / X) * (np.sin(Y) / Y)

fig, ax = plt.subplots(subplot_kw={'projection': "3d"})
```

```
    ax.plot_surface(X, Y, Z, cmap="viridis")

    plt.show()
```

The `subplot_kw` parameter in `plt.subplots()` allows you to pass keyword parameters into the creation of subplots. In this case, you're choosing the projection of the plot to be 3D. This creates an `Axes3D` object. One of its methods is `plot_surface()` which plots the array `Z` as a 3D surface. You use the `"viridis"` colour map:



Although the colour in the colour map still represents the third dimension as before, the plot is now also displayed in 3D, making it easier to visualise, understand, and study.

Creating Animations Using Matplotlib

The final example in this Chapter will extend the static plots into dynamic animations. You'll use the following equation for this exercise:

$$z = \sin(x^2 + y^2)$$

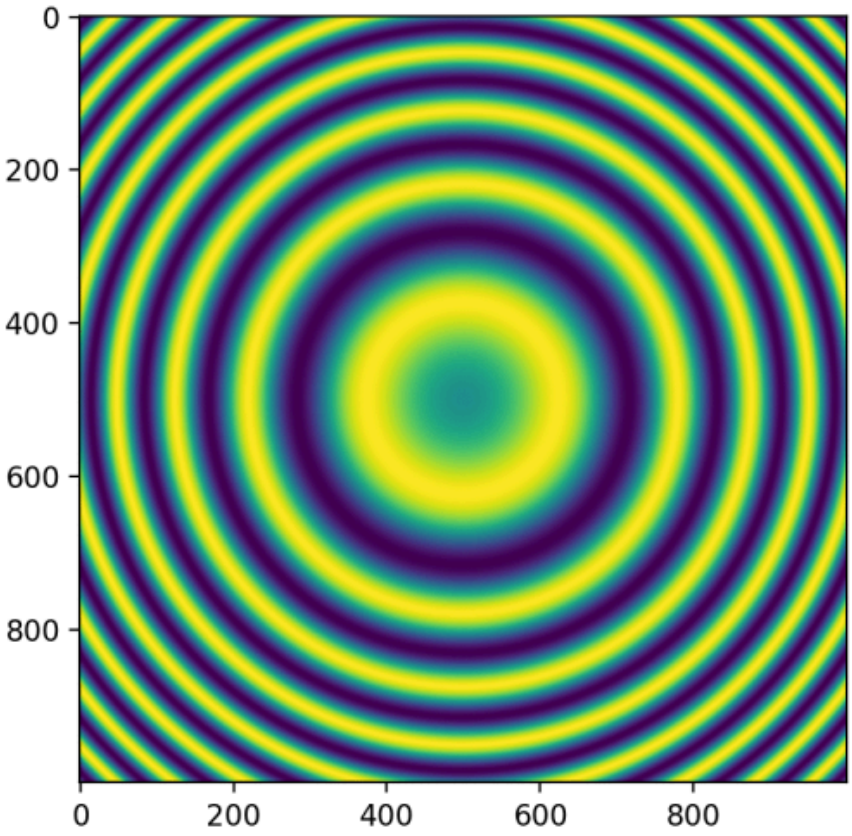
You'll extend this by adding a shift to the sine function:

$$z = \sin(x^2 + y^2 - a)$$

Let's see what this equation looks like:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 1000)
y = np.linspace(-5, 5, 1000)

X, Y = np.meshgrid(x, y)

fig, ax = plt.subplots()

a = 0
Z = np.sin(X ** 2 + Y ** 2 - a)

ax.imshow(Z)
plt.show()
```

With `a = 0`, this equation looks like this:



You want to explore how this equation changes as you use different values for a. Visually, an animation would be the best solution to explore this.

You can achieve this using a `for` loop:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 1000)
y = np.linspace(-5, 5, 1000)

X, Y = np.meshgrid(x, y)

fig, ax = plt.subplots()

for a in np.linspace(-np.pi, np.pi, 50):
    Z = np.sin(X ** 2 + Y ** 2 - a)
    ax.imshow(Z)
    plt.pause(0.001)
plt.show()
```

You're iterating through an `ndarray` you create using `linspace()`. This array contains `50` values ranging from $-\pi$ to π. You assign these values to the parameter `a` in the `for` loop statement. The function `plt.pause()` can be used to display the plot and introduce a short delay which you can use to partially control the speed of the animation.

The speed of the animation displayed when you run this code will depend on the computer you're using and what processes you have running on your device. However, the animation will likely be rather slow. You can reduce the amount of time in the `pause()` function, but this will not make much difference as the bottleneck is elsewhere in the loop. Each iteration needs to work out the new value of `Z` and display it. This slows things down.

There are several ways you can resolve this problem. You'll look at two of these solutions in the next two subsections.

Saving the images to file

One option is to save the images to file as JPG or PNG images and then use external software to create a movie from the series of images. Yes, this option relies on external software. However, if you're comfortable using other software that can create videos from static images, this option can be very useful.

You can save the images to file as you iterate in the loop. For simplicity, I'm saving the files directly in the Project folder in the example below. If you prefer, you can create a subfolder in your Project folder, say one called `Images`, and then add `"Images/"` (Mac) or `"Images\"` (Windows) to the file name in the code:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 1000)
y = np.linspace(-5, 5, 1000)

X, Y = np.meshgrid(x, y)
fig, ax = plt.subplots()

file_number = 0
for a in np.linspace(-np.pi, np.pi, 50):
```

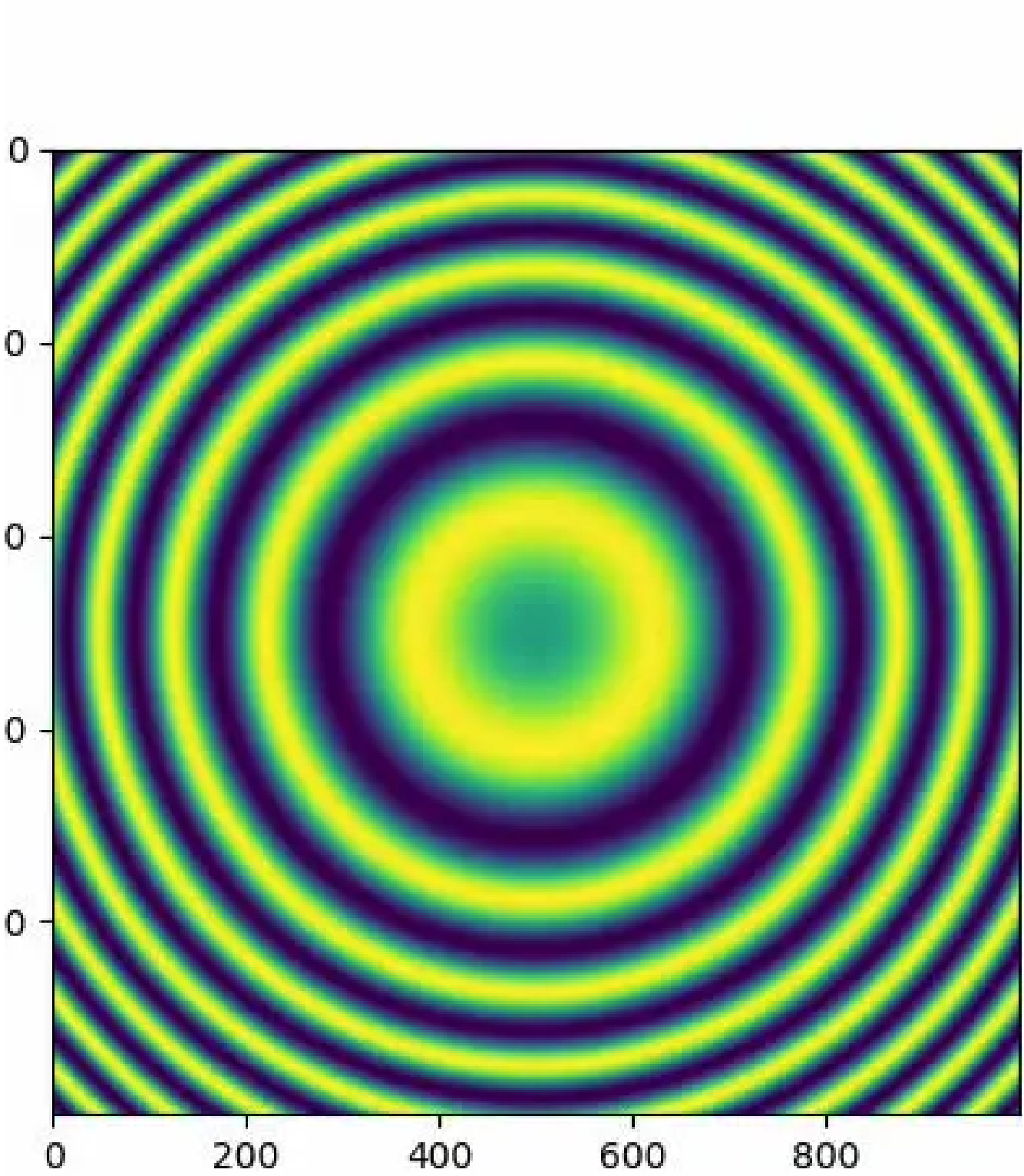
```
        Z = np.sin(X ** 2 + Y ** 2 - a)
        ax.imshow(Z)
        print(f"Saving image {file_number + 1}")
        fig.savefig(f"image_{file_number}.png")
        file_number += 1
```

Rather than displaying the images on screen, you're creating the figure 'behind the scenes' and saving each figure to a PNG file using `fig.savefig()`. You increment the file number using the variable `file_number`.

There is a more Pythonic way of incrementing the file number using Python's built-in `enumerate()` function. I'll show this option below without dwelling on how `enumerate()` works. You can read more about `enumerate()` in the Snippets section at the end of this Chapter:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 1000)
y = np.linspace(-5, 5, 1000)

X, Y = np.meshgrid(x, y)
fig, ax = plt.subplots()

for file_number, a in enumerate(np.linspace(-np.pi, np.p
    Z = np.sin(X ** 2 + Y ** 2 - a)
    ax.imshow(Z)
    print(f"Saving image {file_number + 1}")
    fig.savefig(f"image_{file_number}.png")
```

I've used Quicktime Player on a Mac and its *Open Image Sequence...* option to create the video below. There are also several web-based, free platforms that will allow you to upload an image sequence to generate this movie file:



Using `matplotlib.animation`

One of the submodules in `matplotlib` is the `animation` submodule that provides functionality to create and customise your animations directly in your Python code. This section will briefly demonstrate one way of using this submodule:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import matplotlib.animation as animation

x = np.linspace(-5, 5, 1000)
y = np.linspace(-5, 5, 1000)

X, Y = np.meshgrid(x, y)
fig, ax = plt.subplots()

images = []
for a in np.linspace(-np.pi, np.pi, 50):
    Z = np.sin(X ** 2 + Y ** 2 - a)
    img = ax.imshow(Z)
    images.append([img])

output = animation.ArtistAnimation(fig, images, interval=
plt.show()
```

You've imported the submodule `matplotlib.animation` using the alias `animation`.

The `for` loop is similar to the one you used earlier, iterating from $-\pi$ to π in `50` steps. This time, instead of displaying the image or saving the figure to file, you append the image to a list in each iteration of the loop. Note that each item you append is itself a list with the image within it. Therefore, `images` is a list of lists.

You then create an `ArtistAnimation` object which is one of the objects that allows Matplotlib to deal with animations. The arguments you use when you create the instance of `ArtistAnimation` are the following:

- `fig` refers to the `Figure` object to be used.
- `images` is a list of lists. Each list within `images` contains the images to be included in a single frame of the animation. In this case, each frame only has one image within it, but you can have several images or plots combined into a single frame.
- `interval` determines the delay between the frames of the animation in milliseconds.
- `blit` turns on functionality that optimises the drawings to make the animation smoother.

When you run this code, you'll see the same animation shown earlier, but in this case, the animation runs directly in a Matplotlib figure.

Conclusion

You're now familiar with how to get started with data visualisation in Python using Matplotlib. This library provides a lot of functionality that allows you to customise your plots. If you plan to dive deeper into data visualisation in Python, you'll need to bookmark the Matplotlib documentation pages. The documentation also contains many examples covering several types of visualisations.

In this Chapter, you've learned:

- the **fundamentals** of plotting figures
- when and how to use the **two interfaces** in Matplotlib
- how to plot **2D figures**, including using **subplots**
- how to **display images**
- how to plot **3D figures**
- how to **create animations**

You can now start exploring data visualisation in Python with any of your own data sets.

Additional Reading

- Some useful links to the Matplotlib documentation:
- Documentation homepage
- Main user guide
- List of tutorials on matplotlib.org
- More customisation examples
- List of functions in the `pyplot` interface
- Documentation on the `Figure` class
- Documentation on the `Axes` class
- You can read the article about how to create any image using only sine functions using the 2D Fourier Transform in Python for another tutorial that uses many of the Matplotlib functionality you read about in this Chapter.

Image Credit: Malta Balconies Image by Alex B from Pixabay

[This article uses KaTeX By Thomas Churchman]

Next Chapter

Browse Zeroth Edition

Snippets

Coming soon...