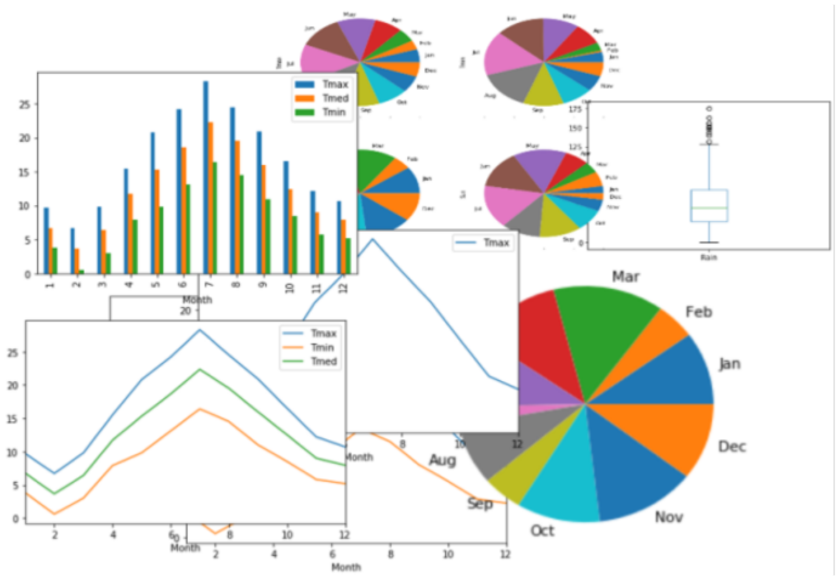
Plotting with Pandas An Introduction to Data Visualization in Python

If you are a Data Scientist or Data Journalist, being able to visualize your data gives you the ability to better understand and communicate it.



Alan Jones

Getting started with data visualization with Python and Pandas

Visualizing data gives you the opportunity to gain insights into the relationships between elements of that data, to spot correlations and dependencies, and to communicate those insights to others.

In this book, you will learn how to easily plot impressive graphic representations of your data using Python and Pandas.

You don’t need to be an expert in Python to be able to create graphics with Pandas, although some exposure to programming in Python would be very useful, as would be a basic understanding of DataFrames in Pandas (fundamentally, they are tables).

If you are familiar with Jupyter Notebooks then that might be a good platform on which to follow this book but if you are happier with a straightforward Python editor or IDE then that’s fine, too.

We are going to explore the data visualization capabilities of Pandas. We’ll start by introducing the basics—line graphs, bar charts, scatter diagrams and pie charts—and then we’ll take a look at the more statistical views with histograms and box plots. Lastly, we’ll see how we can create multiple plots in one chart, how we save charts as images, so we can utilize them in our own reports, documents and web pages and, finally, we look at how to customize our charts to change the size, colours and so on.

Throughout the tutorial, you will use a dataset about the weather in London, UK, and you’ll create a number of charts using that data. I have created this dataset from public domain information that is available from the UK Meteorological Office.

Plotting with Pandas

Fundamentally, plotting with Pandas is using a set of methods that are associated with a Pandas DataFrame to plot various graphs with the data contained in that DataFrame. It relies on a Python plotting library called **matplotlib**.

The purpose of Pandas plotting is to simplify the creation of graphs and plots, so you don’t need to know the details of how mathplotlib works. However, you will need to know one or two matplotlib commands but they are very simple and I’ll explain them as we get to them.

You can think of matplotlib as being a ‘backend’ for Pandas that takes care of the mechanics of creating a plot.

Importing the libraries

The first thing you need to do is import the Python libraries that we are going to use. There are only two libraries that you need: **pandas** gives us ways of storing and manipulating data in *dataframes* and **matplotlib**, as mentioned above, provides the basic plotting functionality that Pandas uses to produce charts and graphs.

It's very possible that you already have the libraries. If not you need to install them with *pip*, or *conda*, e.g.

```
pip install pandas
pip install mathplotlib
```

Now you are ready to start programming!

My own preference for this type of work is to use a Jupyter Notebook. If you are familiar with them then you can follow the tutorial by typing each of the code blocks into a new Notebook cell and run them individually.

However, if you prefer to use a normal editor or IDE to create a single Python program, you can add the code blocks one after the other to create a program.

Now you can start up a Jupyter notebook or a new file in a Python editor and type in the following:

```
# Only use the first line if you are using a Jupyter Notebook
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
```

This, of course, imports the libraries that we need.

But you may not be familiar with the first line. *%matplotlib inline* is specific to Jupyter Notebooks. It's a so-called *magic* command that ensures that the figures that we are going to plot will show up properly in the notebook when you run a cell. It only needs to be included in the Notebook, once. If you are not using Jupyter Notebooks **do not include the first line**.

Getting the data

Before you start visualization you need to get some data.

I've created a couple of csv files of weather data from London. This is a simple data set that is derived from historical data from the UK Met Office. It records the maximum and minimum temperatures, the rainfall and the number of hours of sun for each month over a few decades.

There are two files: one is a record of several decades of data and, the other, a subset of that data for the year 2018, only. You’ll be using the subset for the first few exercises and the larger one later, when we encounter statistical plots.

Both files are included with this book. You should copy them into the folder where you are running your Python program.

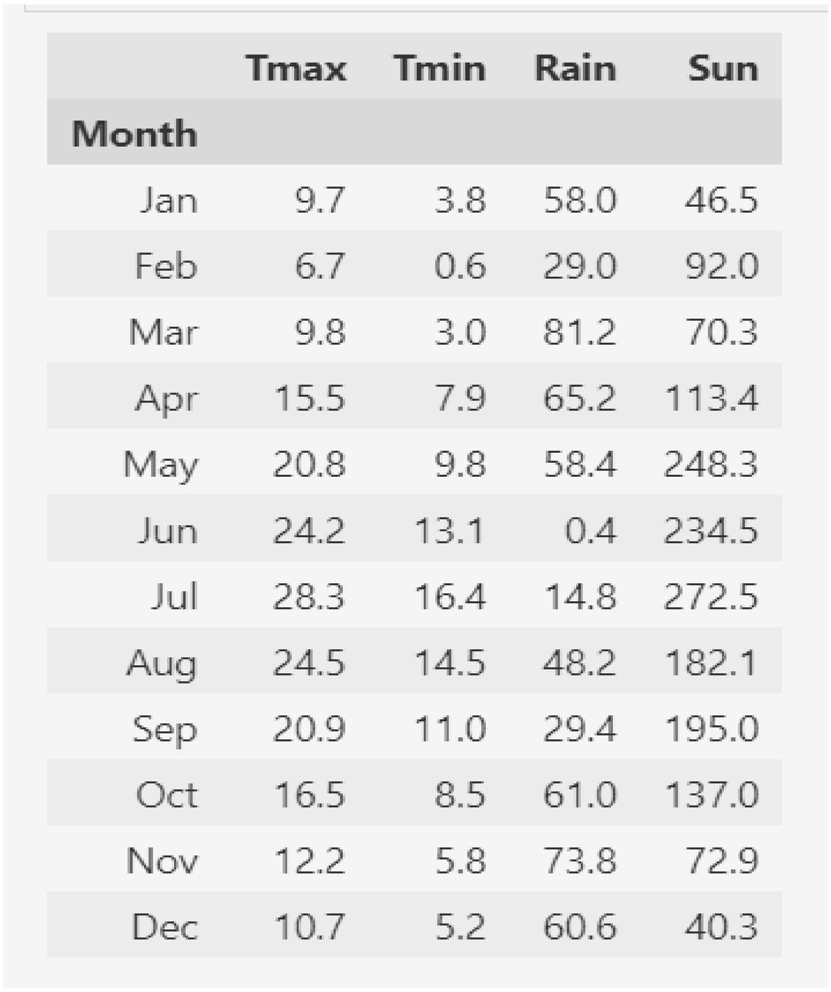
The snippet of code below uses a Pandas DataFrame, *weather*, to hold the weather data and it loads the csv file into that DataFrame from a url.

The second line of code prints the DataFrame and this displays the data as a table.

Looking at the table you can see that the columns are labelled *Month*, *Tmax* (maximum temperature), *Tmin* (minimum temperature), *Rain* (rainfall in millimetres) and *Sun* (hours of sunlight). We have specified that the *Month* column is the index of the table - we'll see how this is used later.

```
url = 'londonweather2018.csv'

weather = pd.read_csv(url, index_col='Month')
print(weather)
```



	Tmax	Tmin	Rain	Sun
Month				
Jan	9.7	3.8	58.0	46.5
Feb	6.7	0.6	29.0	92.0
Mar	9.8	3.0	81.2	70.3
Apr	15.5	7.9	65.2	113.4
May	20.8	9.8	58.4	248.3
Jun	24.2	13.1	0.4	234.5
Jul	28.3	16.4	14.8	272.5
Aug	24.5	14.5	48.2	182.1
Sep	20.9	11.0	29.4	195.0
Oct	16.5	8.5	61.0	137.0
Nov	12.2	5.8	73.8	72.9
Dec	10.7	5.2	60.6	40.3

A first Pandas graph

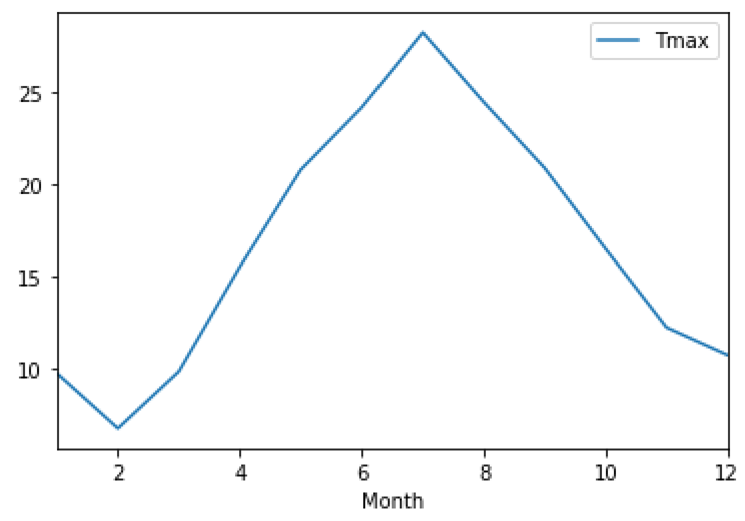
So, *weather* is a Pandas DataFrame. This is essentially a table for storing data, but, in addition, Pandas provides us with all sorts of functionality associated with a DataFrame.

We don't need to go into all of the clever things that you can do with a Pandas DataFrame for our purposes here, instead we will be concentrating on the various methods used to plot a graph.

To plot a graph we use the method `weather.plot()` and this, by default, will create a line graph.

We need to specify *y* coordinate, and we do this by referencing the column names from the DataFrame. Unless we specify otherwise, the horizontal axis of the graph will be the index of the table, i.e. month. So, in the code below, the vertical axis (the y coordinate) will be *Tmax*, the maximum temperature for each month. The code looks like this:

```
weather.plot(y='Tmax')
plt.show()
```



When you run this code, you'll see the result like the graph shown here where the maximum temperature, *Tmax*, for each month of the year is plotted against 12 months.

The first line of the code above is the one that does the work of creating the plot. It calls the DataFrame method *.plot()* from the DataFrame *weather* and passes one parameter to that function, the value for the vertical axis.

The second line of code refers to the *mathplotlib* library - *plt*. *plt.show()* is a function from that library and does what you would expect, it displays the graph.

The *y* parameter is fundamental to drawing a line chart but *plot()* can take a number of other parameters, some of which you will come across later.

Simple charts

We are going to discover a few types of chart. We'll start with the simple ones.

Line charts

Line charts are suitable for visualizing data that changes continuously over time, so are a good way to show temperature changes as these tend to be gradual.

You can create more than just one type of plot and you can specify which type you want in two ways: you can pass a parameter, or you can modify the function call. The code above does not specify the type of plot because the default in Pandas is a line plot. But if you wanted to be specific you could pass the type of plot in the kind parameter like this:

```
weather.plot(kind='line', y='Tmax')
```

Alternatively, there is a specific method for a line plot that you can use like this:

```
weather.plot.line(y='Tmax')
```

These two alternatives produce exactly the same result.

Later, you will see how to produce bar charts, pie charts, scatter diagrams, histograms and box plots. In each case, you can specify the type of plot using the kind parameter or use the method call for that type of plot.

Multiple line plots

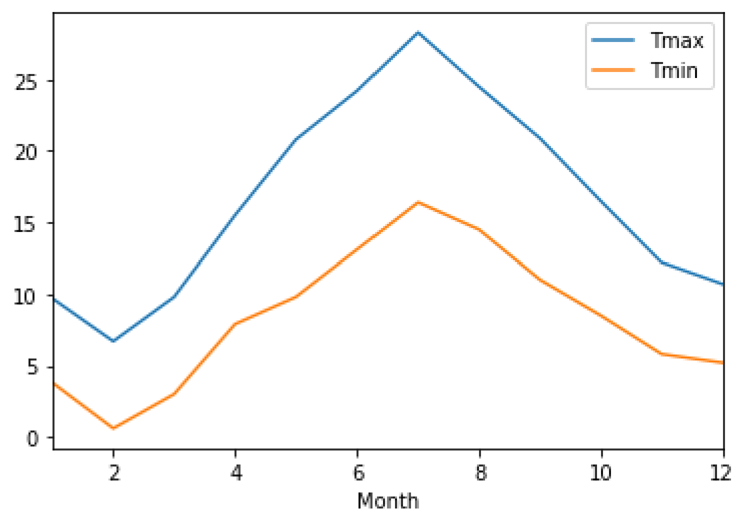
What if you wanted to plot both the maximum and minimum temperatures in the same figure? It's a reasonable thing to want to do and it's easy to do. You create a list of *y* values like this:

```
['Tmax','Tmin']
```

and assign this to the *y* parameter in the plot function.

Take a look at the following code and the resulting plot.

```
weather.plot(y=['Tmax','Tmin'])
plt.show()
```



You can see that the code is almost identical to the first plot but the *y* parameter now contains values for both the maximum and minimum temperatures (*Tmax* and *Tmin*) in a list. The result is a graph above.

You could add more values to the list that are assigned to *y*, if we wished to, but in this data set we don’t have any more suitable values (it wouldn’t make sense to plot, say, temperatures and rainfall on the same graph because they are different measurements with different units i.e. degrees Celsius and millimeters).

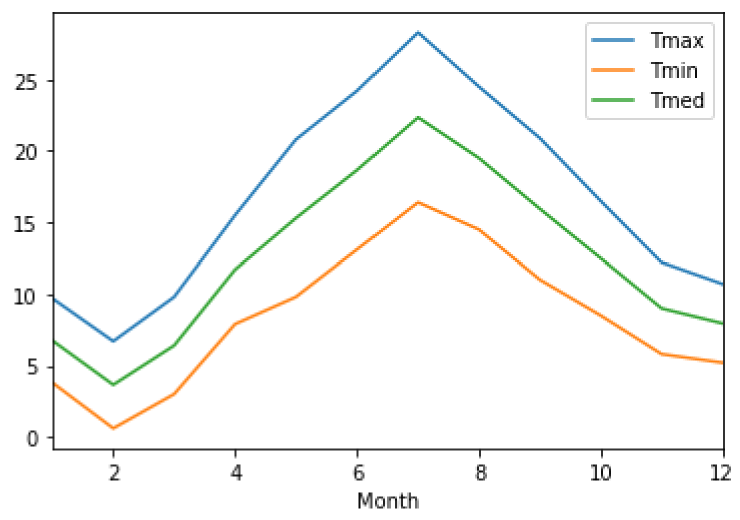
So, just for illustrative purposes, we’ll use a little Pandas magic to create a new column and make a Pandas plot of that, too. In the code, below, we create a column *Tmed* which is the average of *Tmax* and *Tmin* (the sum of *Tmax* and *Tmin* divided by 2).

```
weather['Tmed'] = (weather['Tmax'] + weather['Tmin'])/2
```

If you want to see what it looks like you can *print(weather)* and you’ll see the extra column with the average values.

Now we plot as before but with the extra column added to the list of *y* values.

```
weather.plot(y=['Tmax','Tmin','Tmed'])
plt.show()
```



Unsurprisingly, the new *y* value draws a line drawn halfway between the maximum and minimum temperatures.

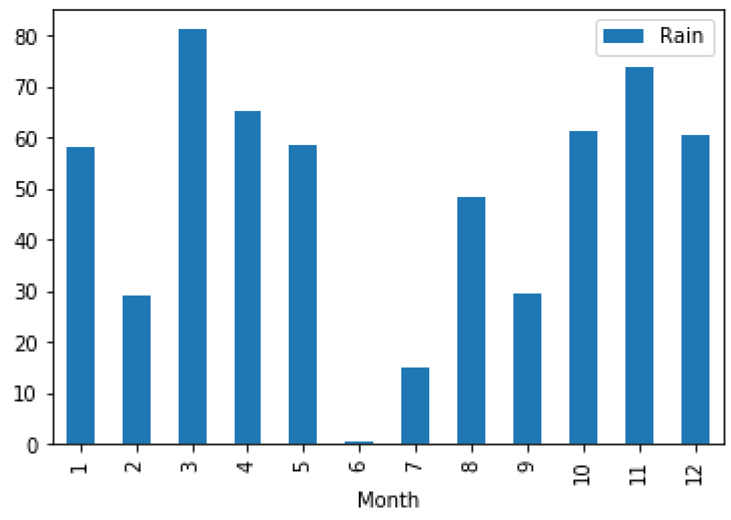
While line charts are great for plotting continuous values, some measurements are more discrete in their nature. Temperatures tend to change gradually without sudden leaps from one value to another but rainfall is a different matter.

It's not unusual for it to rain one day and not the next. Rain doesn't change gradually but rather it can start and stop quite abruptly. A line chart is not really suitable for visualizing that sort of behavior—better to use a bar chart.

Bar Charts

If you draw a bar chart of the rainfall data for 2018, you can see that the changes over time are more abrupt than the temperature data and, so, a bar chart representation is quite appropriate. Here's what it looks like:

```
weather.plot(kind='bar', y='Rain')
plt.show()
```



There is quite a difference in rainfall between May and June, and it is very clear from the bar chart that this is the case. A line chart would make it look like there was a smooth transition between the two months whereas this is unlikely to be the case.

This sort of representation makes to easier to spot phenomena like *April showers* - April being the month when it typically rains a lot in the UK. Except that, in 2018, April was less wet than March, and not much worse than May.

Ah well, that's folklore for you.

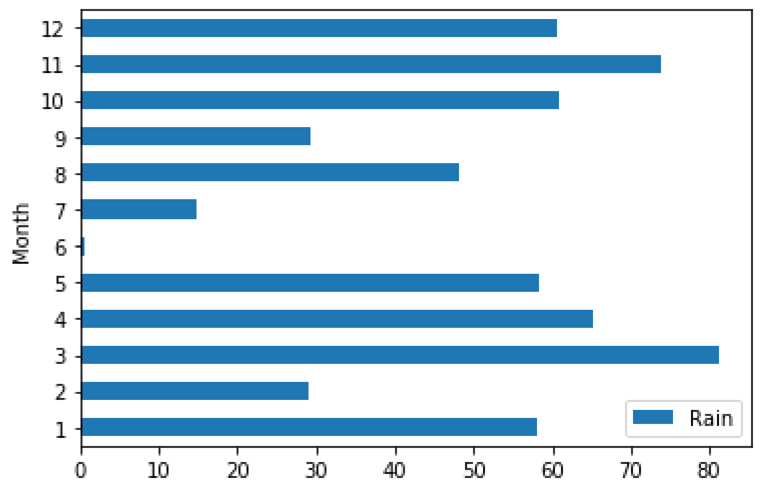
Looking at the code you can see that it is very similar to the line chart code. The only difference is that the *kind* parameter is '*bar'*, rather than '*line'*. As we discussed above, the same result would be obtained if you called the specific bar chart method, like this:

```
weather.plot.bar(y='Rain', x='Month')
```

I won't continue to talk about the two ways of specifying a particular chart, suffice it to say that in all of the charts you can use either construction.

What if we want our bars to be horizontal? In that case we specify *barh* as the type of chart, as below:

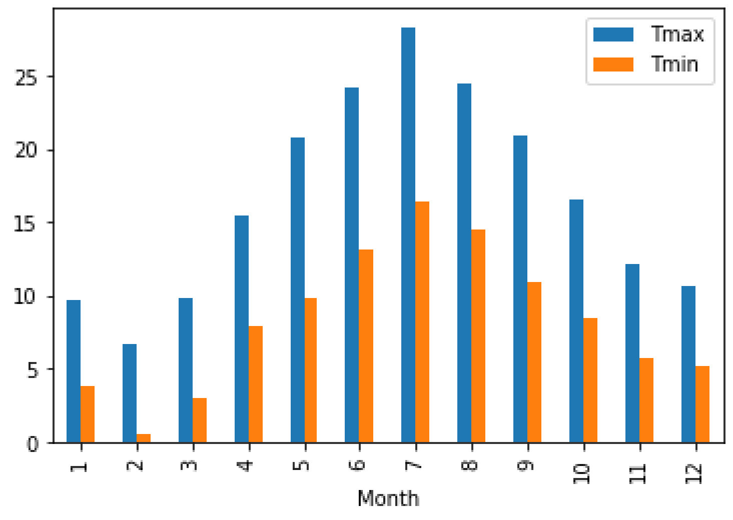
```
weather.plot(kind='barh', y='Rain')
plt.show()
```



And, as you might expect, you can create multiple bars by adding a list of *y* values in the same way as the line chart.

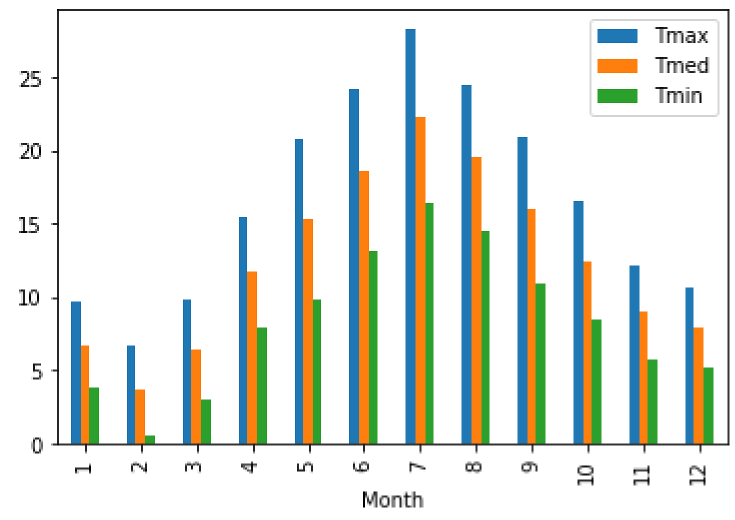
Clearly, if you are going to plot two values on the same chart, the type of data needs to be similar. It would not make sense to have rainfall and temperature on the same chart as they are measured in different units. So, to illustrate a multiple bar chart we are going back to temperatures. Here is a bar chart that plots both *Tmax* and *Tmin*:

```
weather.plot(kind='bar', y=['Tmax','Tmin'])
plt.show()
```



And one that plots the three temperatures, as you did with an earlier line chart:

```
weather.plot(kind='bar', y=['Tmax','Tmed','Tmin'])
plt.show()
```



Scatter diagram

A scatter diagram plots a series of points that correspond to two variables and allows us to determine if there is a relationship between them.

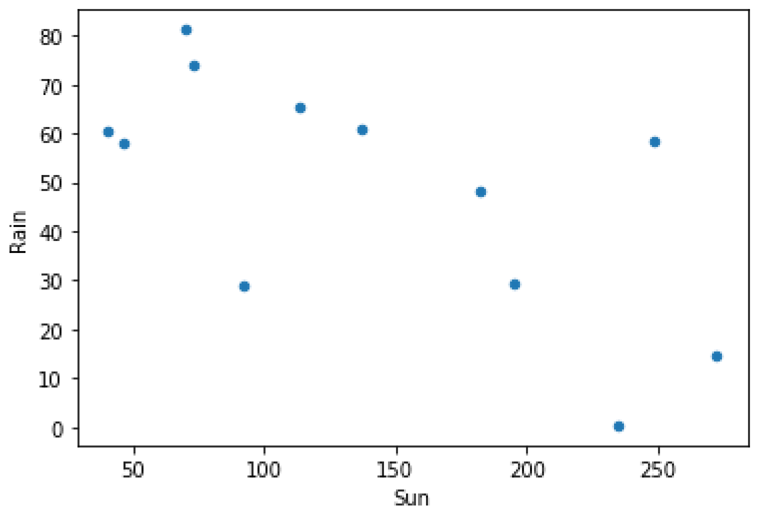
Is there a relationship between the amount of sunshine in any particular month and the level of rainfall? Probably there is.

The scatter plot below plots Sun against Rain. There are 12 points, one for each row in the table, and the points plot the value of Rain on the vertical axis against Sun on the horizontal one.

It's not particularly clear but you can see a vague linear relationship.

A straight line that was the best fit through the twelve points would start somewhere high up on the left and end up low on the right. That tells us that when *Rain* has a high level, *Sun* has a low one, and vice versa. Which common sense tells us is probably right - there is, generally speaking, an inverse relationship between the amount of sunshine and the amount of rain. Because when it's raining it's also cloudy and so there is less sunshine.

```
weather.plot(kind='scatter', x='Sun', y='Rain')
plt.show()
```



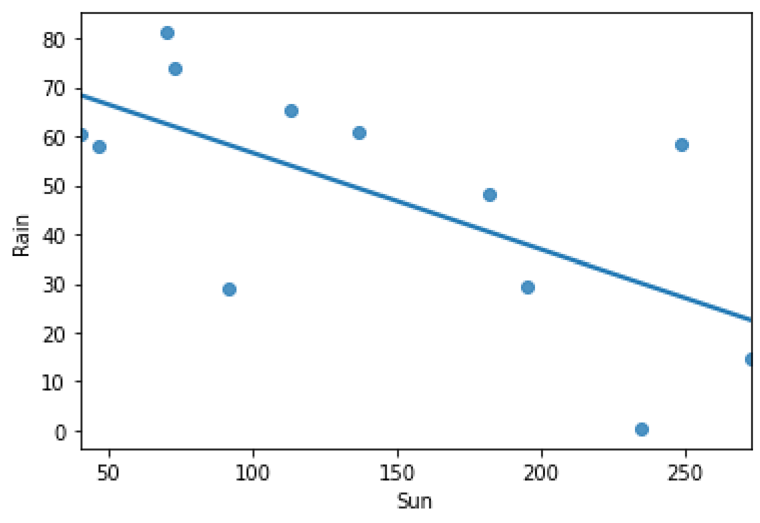
In this example, the relationship between the values is fairly obvious and if, from some hypothetical data set, we were to plot rainfall against umbrella sales, we might also see what we expected. But it can be useful to be able to demonstrate such relationships when they are not so obvious.

Regression Plots

You can get a better idea of patterns in a scatter plot if you plot a regression line.

Unfortunately, Pandas doesn't have a built-in regression plot type. Instead, you would need to either build a regression model using a statistics library or, more simply, use *regplot* from the Seaborn plotting library. We won't cover Seaborn here but to give an idea here is a simple regression plot.

```
import seaborn as sns
sns.regplot(data=weather, x='Sun', y='Rain', ci=False)
```



You can see that the straight line graph gives you a better idea of what the relationship looks like.

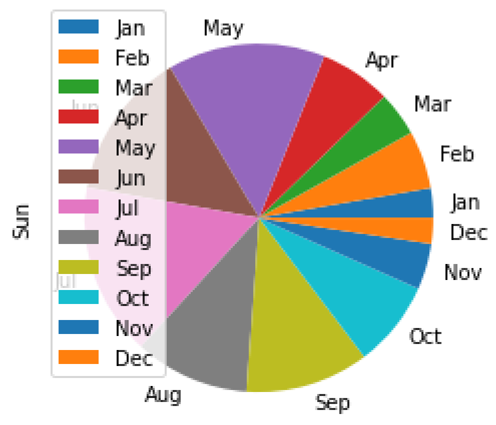
Pie charts

Pie charts are typically used to show what proportion of some value can be associated with a particular group or category. You might, for example, show peoples' preferences for different types of fast food—80% like pizza and 20% prefer burgers—or the market share of various types, or makes, of automobile.

Here we are going to see what proportion of the annual sunshine happens in each month.

So, the pie chart only takes one parameter for the data, in the case *Sun*. This is then plotted against the index of the DataFrame. Here's a first try at creating the pie chart:

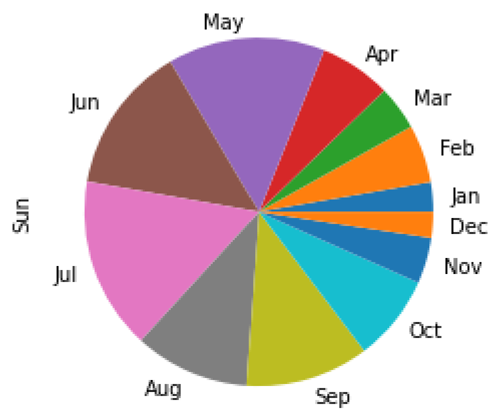
```
weather.plot(kind='pie', y='Sun')
plt.show()
```



This is the sort of thing that we want but, frankly, it could be better.

The problem is that the legend, being quite big, obscures part of the chart. To fix this, you can simply dispose of the legend, as it doesn't really add any value to the plot. You do this by adding a new parameter to the plot method. The parameter is *legend* and you simply set it to *False*.

```
weather.plot(kind='pie', y='Sun', legend=False)
plt.show()
```



Statistical charts and spotting unusual events

The second set of charts that we will look at are concerned with statistics.

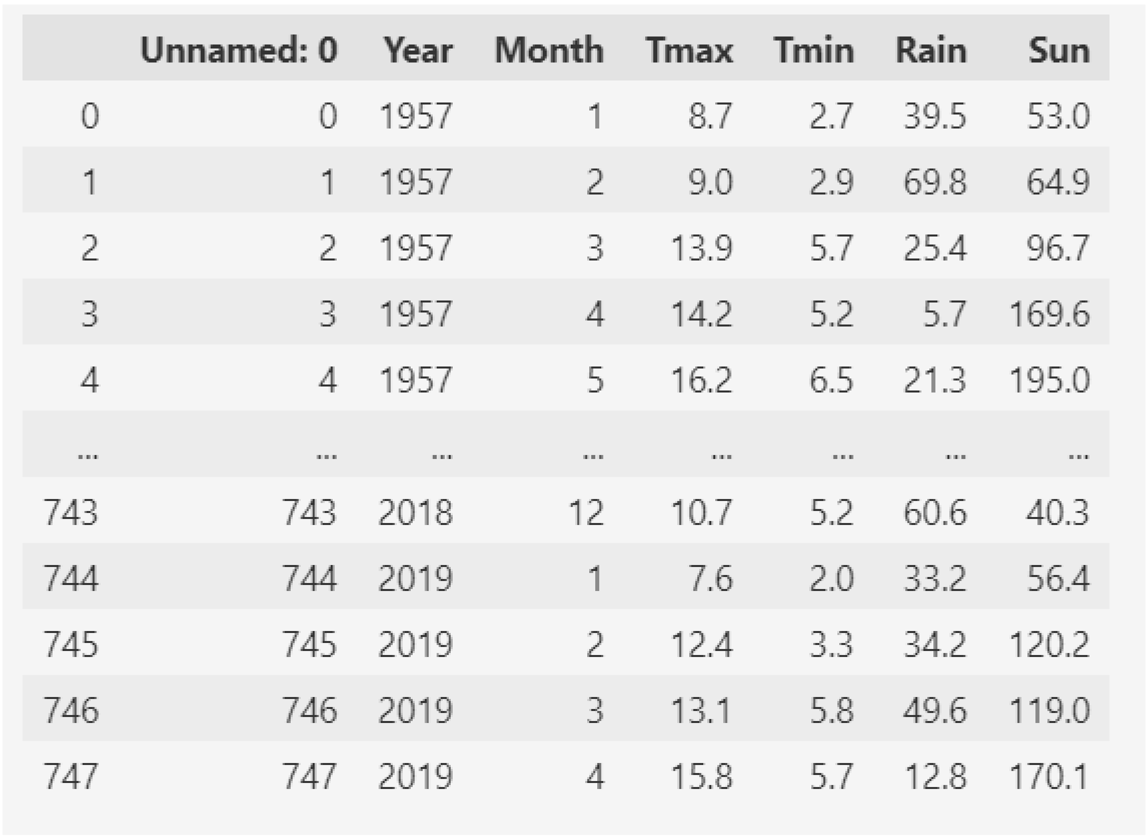
Getting more data

You've been using a small Dataset up to now and the simple charts that you've used have been entirely appropriate.

Now, you are going to download a larger dataset. It's a similar format as before but covers several decades, not just one year.

Again, it is a *csv* file. The following code downloads the data and stores it as a DataFrame, *more_weather*. It then print the first four years (48 rows) of data:

```
url='londonweather.csv'
more_weather = pd.read_csv(url)
print(more_weather)
```



	Unnamed: 0	Year	Month	Tmax	Tmin	Rain	Sun
0	0	1957	1	8.7	2.7	39.5	53.0
1	1	1957	2	9.0	2.9	69.8	64.9
2	2	1957	3	13.9	5.7	25.4	96.7
3	3	1957	4	14.2	5.2	5.7	169.6
4	4	1957	5	16.2	6.5	21.3	195.0
...	...	...	...	...	...	...	...
743	743	2018	12	10.7	5.2	60.6	40.3
744	744	2019	1	7.6	2.0	33.2	56.4
745	745	2019	2	12.4	3.3	34.2	120.2
746	746	2019	3	13.1	5.8	49.6	119.0
747	747	2019	4	15.8	5.7	12.8	170.1

To get a feel for this new set of data you can use the Pandas method *describe*. The following code prints out a description of the *Rain* column:

```
count    748.000000
mean      50.408957
std       29.721493
min        0.300000
25%       27.800000
50%       46.100000
75%       68.800000
max      174.800000
Name: Rain, dtype: float64
```

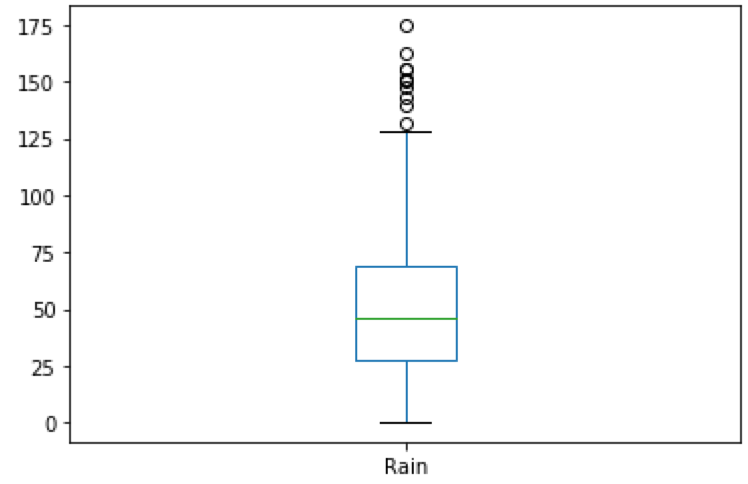
You can see from this that there are 748 rows of data (representing 748 months, that's over 62 years), the mean monthly rainfall over that time was a little over 50mm, the minimum in any month was 0.3mm and the maximum over 174mm.

But if you want to communicate that data graphically, the charts that you've seen, so far, are not much help. A *boxplot*, however, gives you a great summary of the data in one simple graphic.

Box plots

The code below is a box plot of the *Rain* data and it contains a great deal of information in a single graphic.

```
more_weather.plot.box(y='Rain')
plt.show()
```



This is also called a *box and whisker* plot because of the lines, or whiskers, coming from the top and bottom of the plot.

Here's how you interpret the graphic.

The box itself represents the range of the *Rain* data between the first and third quartiles. Quartiles are simply the boundaries when you split the data into quarters. So the first quartile (Q1) is at the 25% mark—25% of the data points are below Q1 and 75% is above it. The third quartile (Q3) is at the 75% mark and so has 25% of the data points above Q3 and 75% below.

And as you probably realize, the second quartile (Q2) is the one halfway through the data, 50% the data points are above Q2 and 50% below.

So, the box itself represents 50% of the data: the top of the box is Q3 and the bottom of the box is Q1. The horizontal line inside the box is Q2, which is also the median: the middle value of all the data points

By convention, the whiskers are set to a value calculated from the *inter quartile range (IQR)*. The IQR is the value of Q3 minus Q1 and the ends of the whiskers are set at 1.5 times the IQR, from the top and the bottom of the boxes.

The idea is that the bulk of the data is represented by the box and whiskers and anything beyond then are considered *outliers*.

In the graph, each outlier is represented by a circle—there are 8 outliers in the plot above.

The numbers down the left are, of course, the values that we are measuring, in this case, monthly rainfall in millimeters.

The outliers are when it was really, really wet!

There’s a lot of information in a box plot and you can see at a glance the shape of the data that you are looking at: where the bulk of the values lie and what outliers there are in the data.

You can see that the monthly rainfall varies quite a lot, from less than 1mm to around 125mm. That, by London standards, is a range from really very dry to really rather wet. The outliers represent extremely wet months but in the 748 months for which we have data, there are only 8 of them.

To summarize, then, in the box plot:

- The centre line is the second quartile (Q2) and is also the median of the data
- The bottom of box is the first quartile (Q1) and is also the median of the bottom half of the data
- The top of the box is the third quartile (Q3), and is also the the median of the top half of the data
- The whiskers extend to 1.5 times inter quartile range (IQR) , which is Q3-Q1, from edge of box
- The circles are outliers, the individual values that lie beyond the end of the whiskers

Histograms

You could see from the box plot of rainfall that half the rainfall was in the range around 25 to around 75 millimeters, that the bulk was between roughly 0 and 125 and that there were a handful of outliers when Londoners got completely soaked.

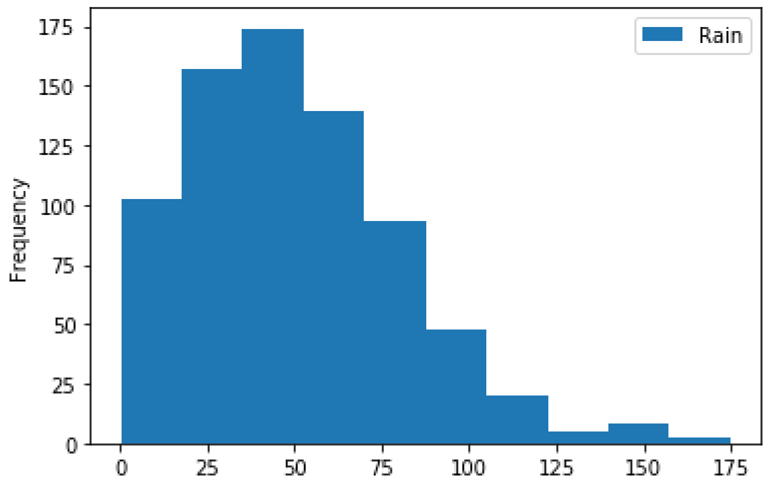
But you can look at this sort of distribution in more detail with a *histogram*.

The diagram, below, is a histogram of the monthly rainfall in our data—a histogram is plotted by setting the *kind* parameter to '*hist*'.

You can see that the data is split up into ranges, or *bins*, each being represented by a bar. The width of the bar is the range of values and the height of the bar is the number of times that values in that range have occurred.

The default number of bins is 10, so when you run the code below, you will get 10 bars.

```
more_weather.plot(kind='hist', y='Rain')
plt.show()
```



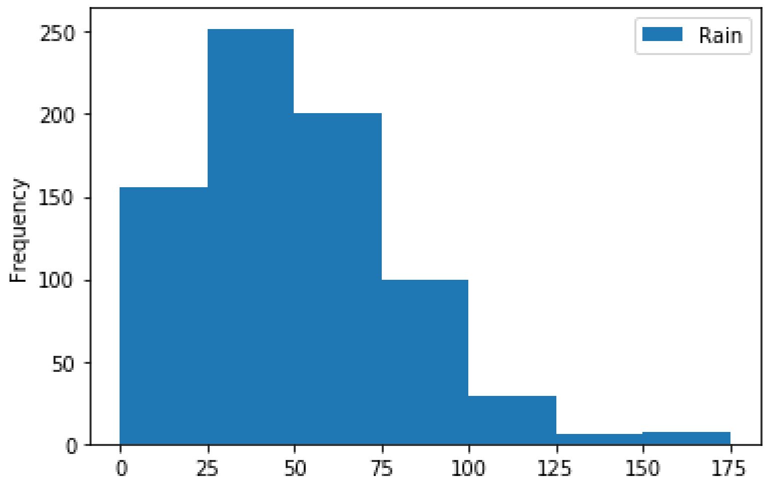
More bins

You can adjust the number of bins by setting the *bins* parameter. You can set this to a number such as *bins=15* or with a list of values which represent the boundaries of the bins.

For example, say you wanted to take a closer look at the outliers that you found in the box plot earlier. Those outliers are in the range of about 125 to 175mm, so you could make sure that your bins match those ranges.

The code below sets the bins parameter to a list of 8 values which, because these represent the boundaries, gives us 7 bins each representing a range of 25mm. The last two bins represent the outliers and you can see in the new histogram that they are not particularly significant.

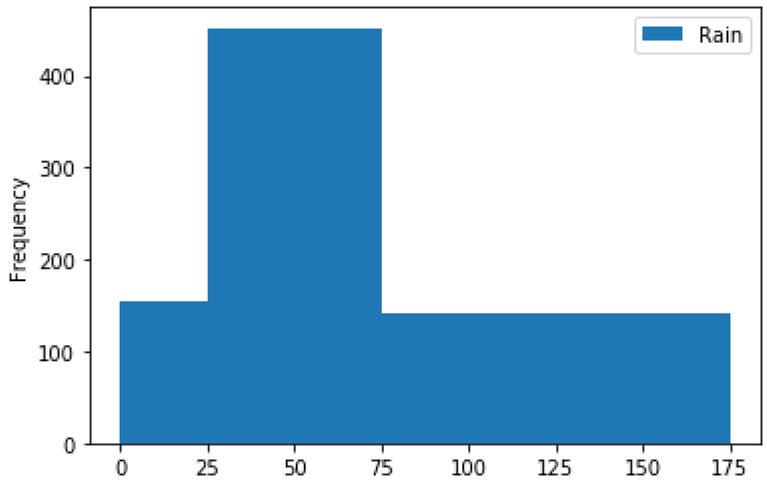
```
more_weather.plot(kind='hist', y='Rain',
bins=[0,25,50,75,100,125,150,175])
plt.show()
```



Let’s take another view. You could decide that, according to the previous plots, *normal* rainfall is in the range 25 to 75mm and that everything else is unusual. So, to identify the frequency of *unusual* events you could display 3 bins, one representing unusually dry weather, a second for *normal* weather, and a third that records unusually wet weather.

The code below gives you 3 bins, representing 0 to 25mm (unusually dry), 25 to 75mm (normal) and 75 to 175mm (unusually wet).

```
more_weather.plot.hist(y='Rain', bins=[0,25,75,175])
plt.show()
```



The resulting graph shows us that there are around 450 normal months, and a similar number of unusually wet and unusually dry months—about 150 of each.

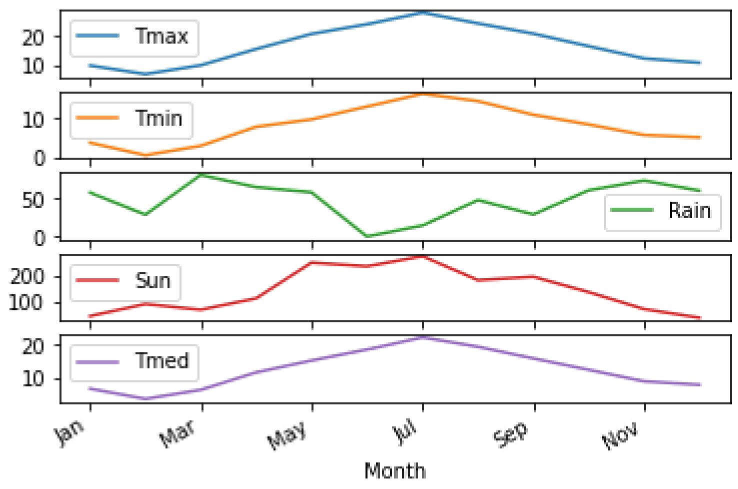
Pandas Plot utilities

Here are a few useful things you can do with Pandas charts.

Multiple charts

You can also create a set of separate charts for each series of data points. The simple code below will display a line plot for every column in weather in separate graphs.

```
weather.plot(subplots=True)
plt.show()
```



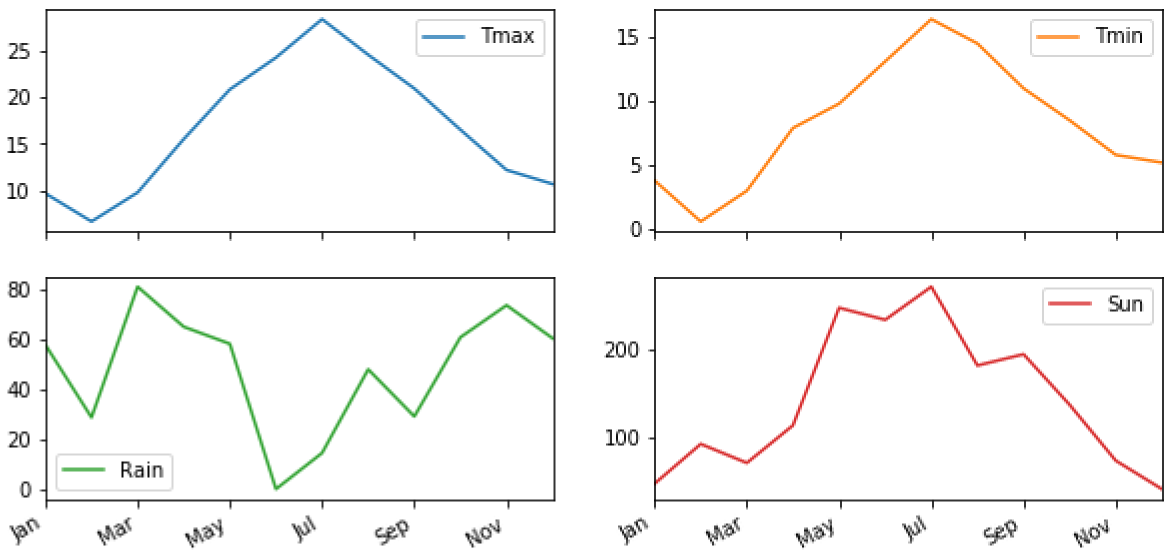
As you can see, we specify a parameter `subplots=True` (the default value is *False*) and this draws a plot for each column. The default behaviour is to draw each graph one after the other but we can change the layout easily.

In the next example, we'll plot four of the columns and set the layout to be 2 x 2 - we'll also set the size of the plot.

As you can see below the layout and the size of the plot using the *layout* and *figsize* parameters. Each of these parameters takes a list, or a tuple, of two values. In the case of layout the first value in the list specifies the number of rows and the second one the number of figures in each row. For *figsize*, the first value is the width of the figure and the second its height.

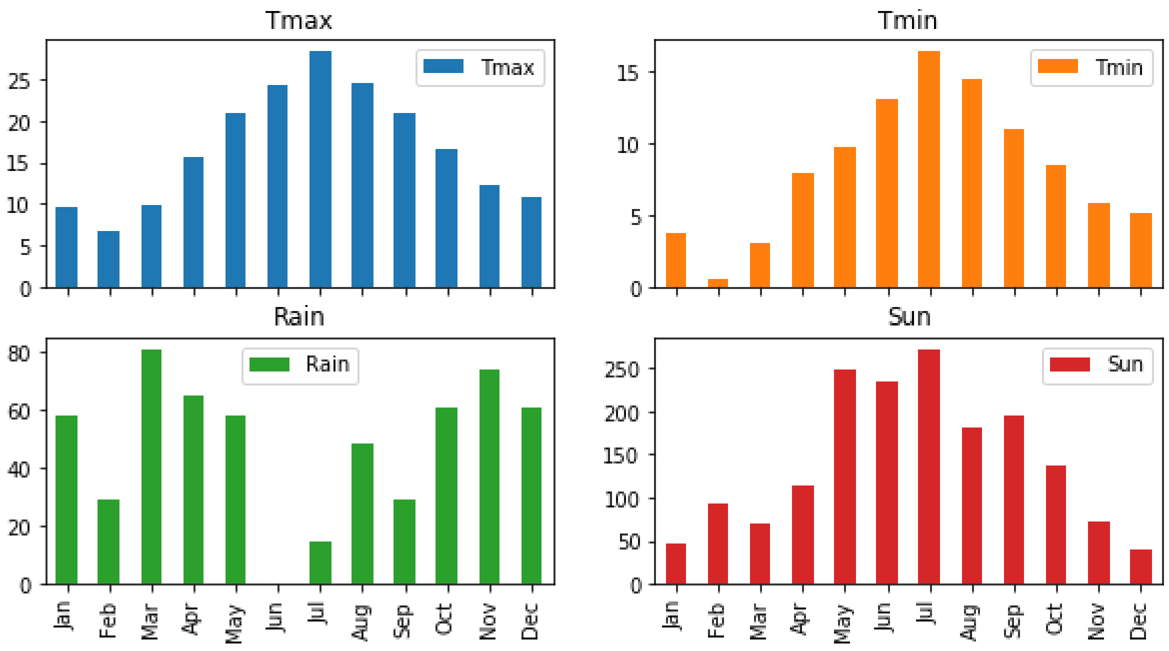
Try changing the values in the list to see what effect they have.

```
weather.plot(y=['Tmax', 'Tmin','Rain','Sun'], subplots=True,
   layout=(2,2), figsize=(10,5))
plt.show()
```



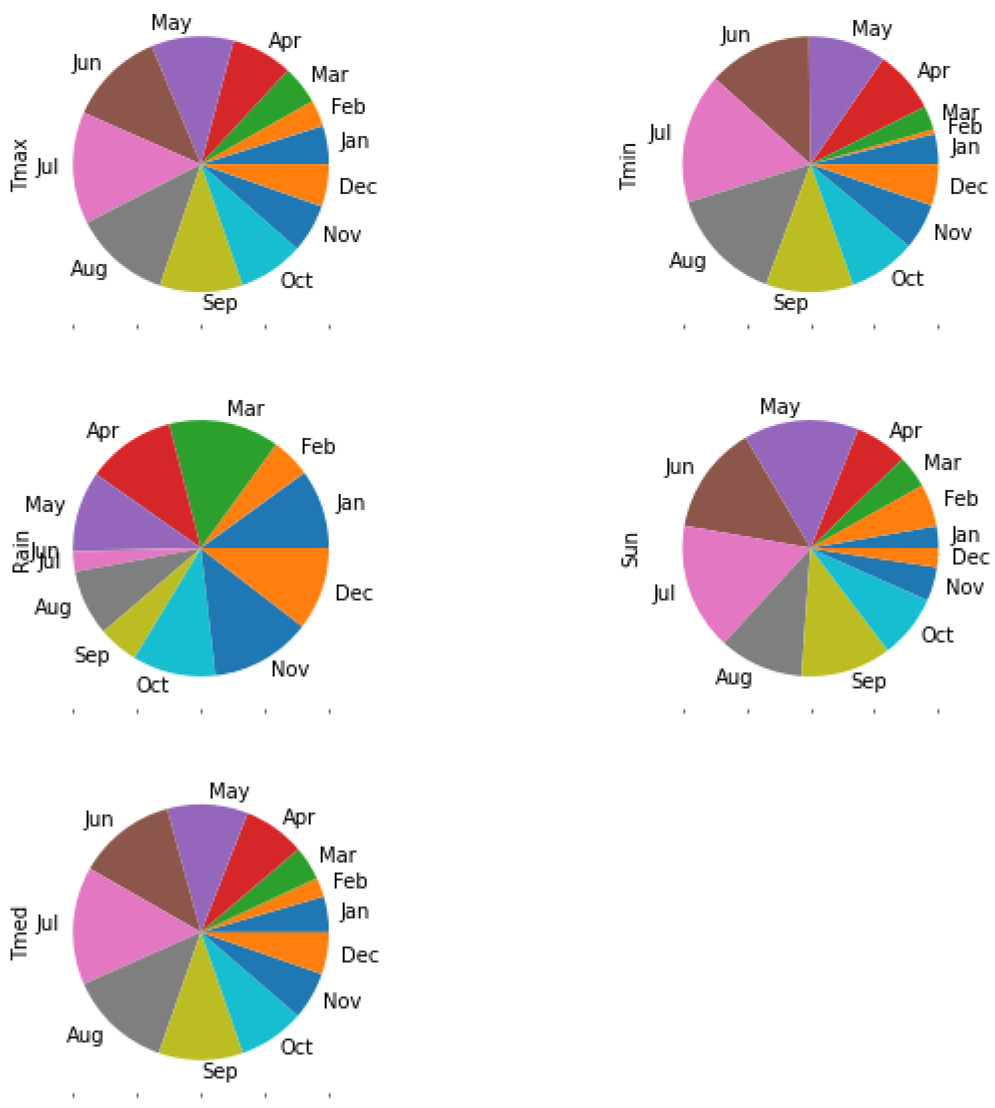
Here’s a set of bar charts:

```
weather.plot(kind='bar', y=['Tmax', 'Tmin','Rain','Sun'], subplots=True,
layout=(2,2), figsize=(10,5))
plt.show()
```



And a set of pie charts:

```
weather.plot(kind='pie', subplots=True, legend=False, layout=(3,2),
   figsize=(10,10))
plt.show()
```

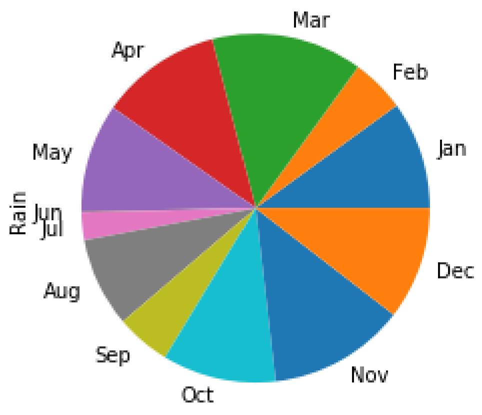


You should note that, in the case of pie charts, all of the columns are drawn and you cannot specify a list as with the other chart types we have seen.

Saving the Charts

This is all very well but maybe you want to be able to use the charts that you produce. If you want to use them in a presentation or document, then it would be useful to be able to export them as image files that you can include in another file. The simple way of saving the images is like this:

```
weather.plot(kind='pie', y='Rain', legend=False)
plt.show()
plt.savefig('pie.png')
```



We’ve used functions from *mathplotlib* before and this is just another one called *savefig()*. This function You can see that the name of the file is specified as a parameter and the type of image that you save is assumed from the file extension—in this case a *png* file called *“pie.png”*. As we haven’t specified a path, the file will be saved in the same directory as the notebook or program.

Customizing Pandas Plots and Charts

Pandas gives you a simple and attractive way of producing plots and charts from your data. But sometimes you want something a little different. Here are some suggestions.

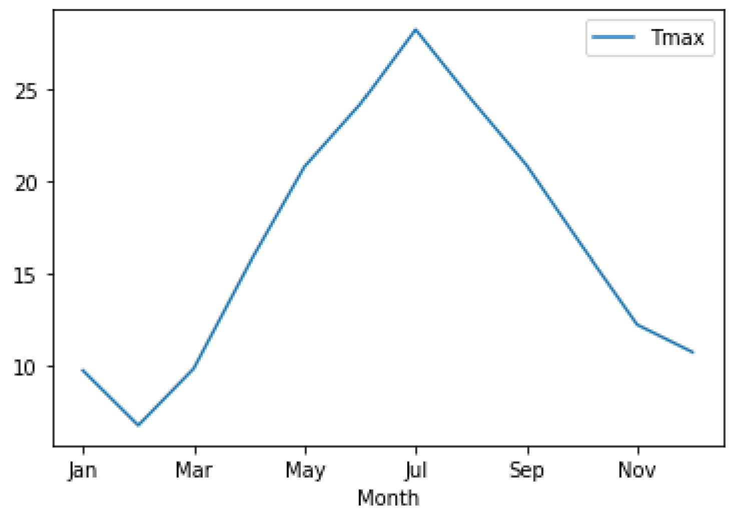
Perhaps you are a data journalist putting a new story together, or a data scientist preparing a paper or presentation. You've got a nice set of charts that are looking good but a bit of tweaking would be helpful. Maybe you want to give them all titles. Maybe some would be improved with a grid, or the ticks are in the wrong places or too small to easily read.

You now know how to produce standard line plots, bar charts, scatter diagrams, and so on but you don't have to stick with what you are given. There are quite a lot of parameters that allow you to change various aspects of your diagrams. You can change labels, add grids, change colors and so on.

Using the underlying *matplotlib* library you can change just about every aspect of what your plots look like and it can get complicated. However, we are going to look at some of the easier things we can do just with Pandas.

Let's just remind ourselves of the simple plot of how the maximum temperature changed over the year

```
weather.plot(y='Tmax')
plt.show()
```



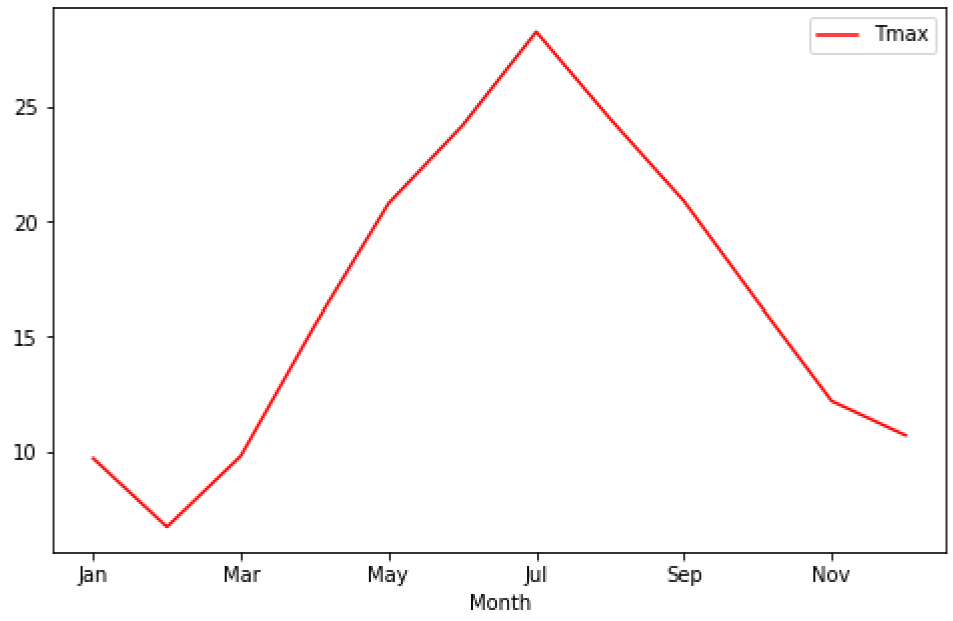
This is the default chart and it is quite acceptable. But we can change a few things to make it more so.

Change the size and color

The first thing that you might want to do is change the size (we saw this briefly, earlier). To do this we add the *figsize* parameter and give it the sizes of *x*, and *y* (in inches). The values are given a tuple, as below.

To change the color we set the *color* parameter. The easiest way to do this is with a string that represents a valid web color such as ‘Red’, ‘Black’ or ‘Teal’. (Note: you can find a list of web colors in Wikipedia.)

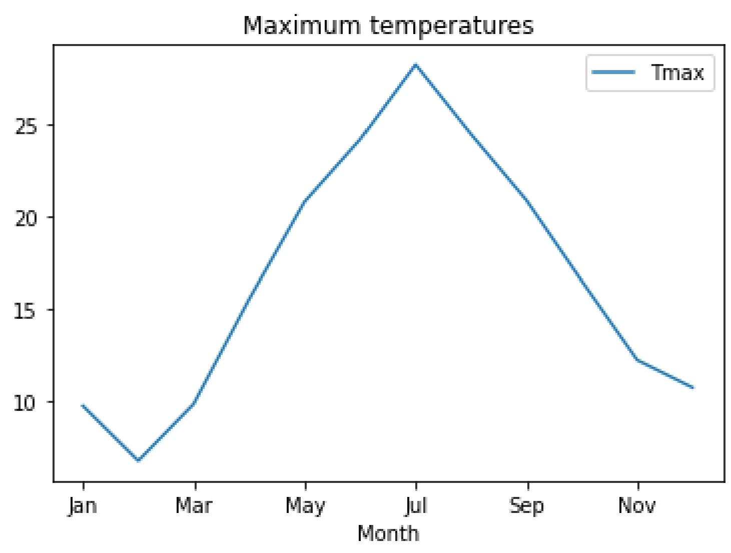
```
weather.plot(y='Tmax', figsize=(8,5), color='Red')
plt.show()
```



Setting a title

It’s very likely that for an article, paper or presentation, you will want to set a title for your chart. As you’ve probably gathered, much of this is knowing what the correct parameters are and setting them properly. The parameter to set a title is *title*. Of course! Here’s the code:

```
weather.plot(y='Tmax', title='Maximum temperatures')
plt.show()
```

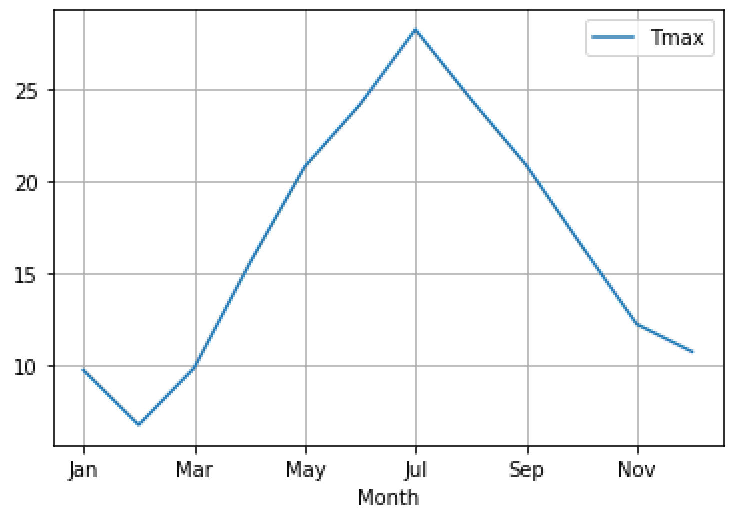


Display a grid

While the default charts are fine, sometimes you want your audience to more easily see what certain values are in your chart. Drawing grid lines on your plot can help.

To draw the grid, simply set the *grid* parameter to *True*. Pandas defaults to *False*.

```
weather.plot(y='Tmax', grid=True)
plt.show()
```

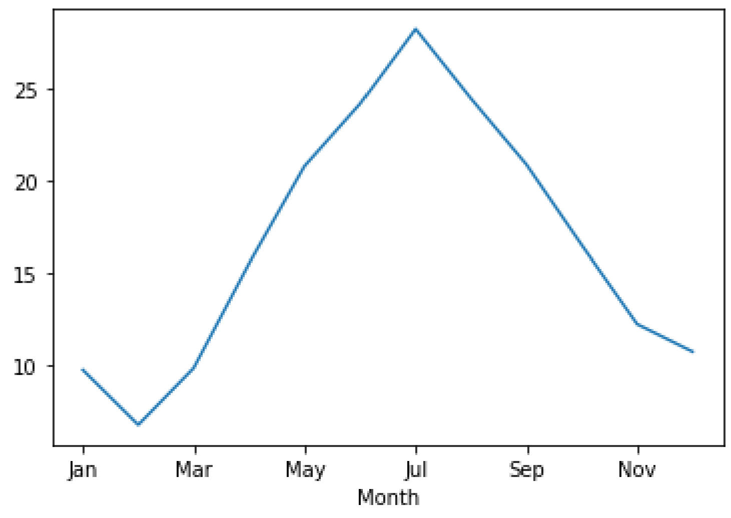


Change the legend

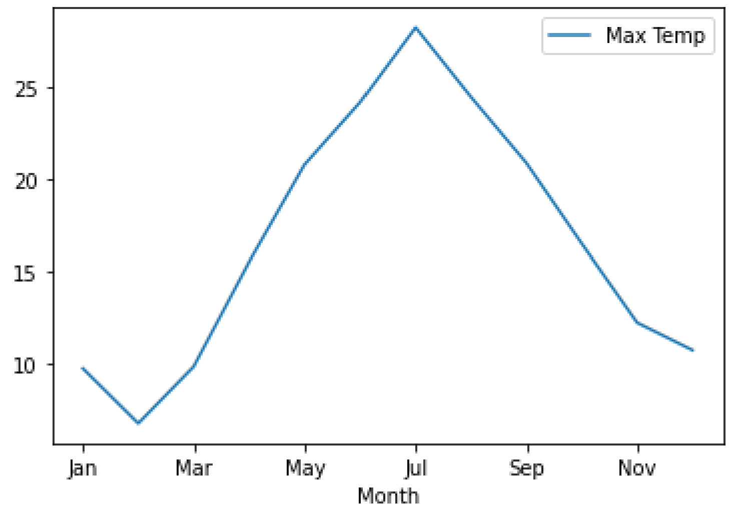
The legend is given the name of the column that represents the *y* axis. If this is not an acceptably descriptive name, you can change it. Or, indeed, you can eliminate it altogether!

If we want to remove it we set the parameter *legend* to *False*. If we want to change the label we incorporate the *label* parameter and set it to the string that we want displayed.

```
weather.plot(y='Tmax', legend=False)
plt.show()
```



```
weather.plot(y='Tmax', label='Max Temp')
plt.show()
```

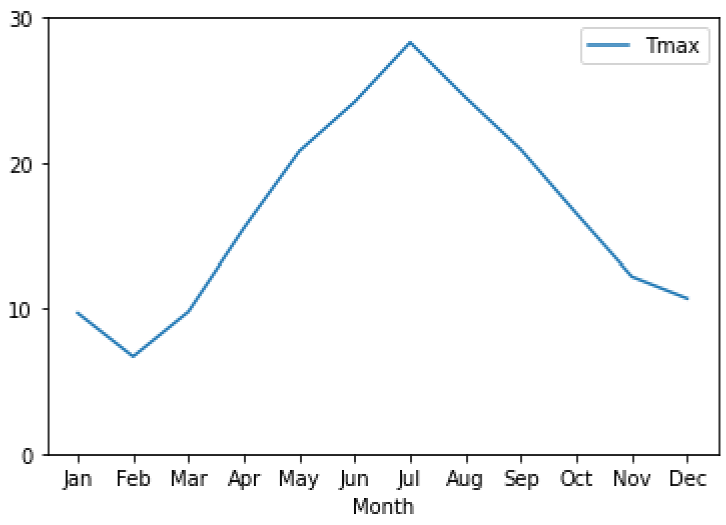


Customize the ticks

Ticks are the divisions on the *x* and *y* axes. You can see that on our charts they are labelled from *10*to *25*on the *y* axis and *Jan* to *Nov* on the *x* axis. We can set the tick labels with tuples. If we want to display all twelve months we would set the parameter *xticks* to

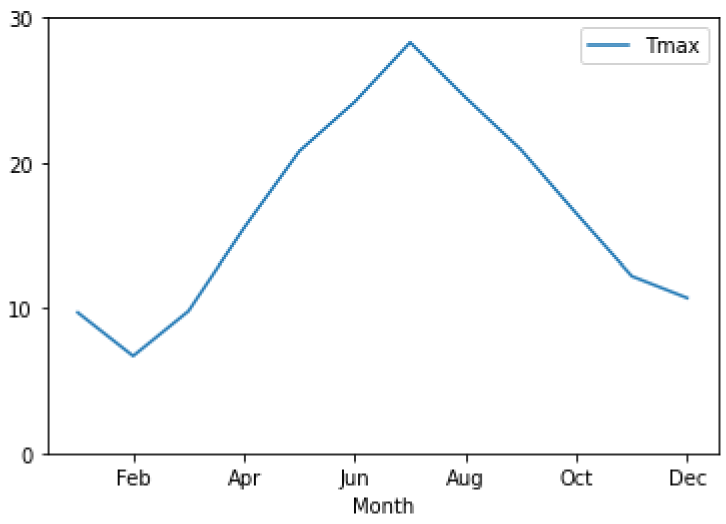
(0,1,2,3,4,5,6,7,8,9,10,11). You can do a similar thing with the *yticks* parameter. Take a look at this code:

```
weather.plot(y='Tmax', xticks=(0,1,2,3,4,5,6,7,8,9,10,11),
   yticks=(0,10,20,30))
plt.show()
```



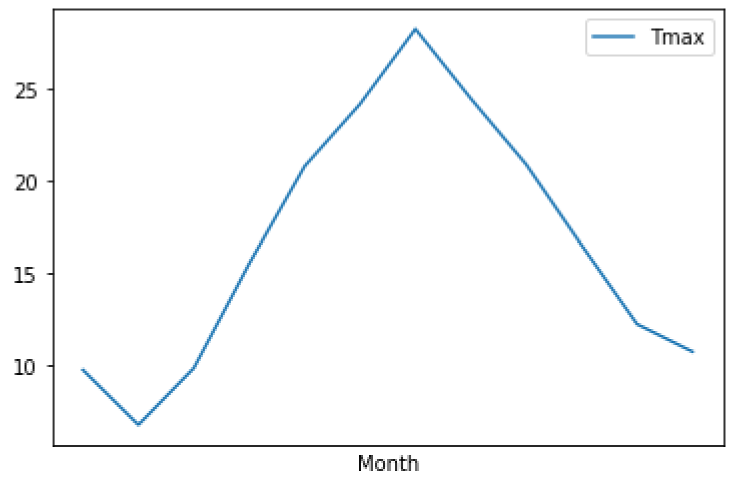
As you can see I've set both sets of ticks explicitly. The *y* ticks now start at 0 and go up in tens to 30, and the *x* ticks show every month. We could do something similar if we only wanted to show every other month:

```
weather.plot(y='Tmax', xticks=(1,3,5,7,9,11), yticks=(0,10,20,30))
plt.show()
```



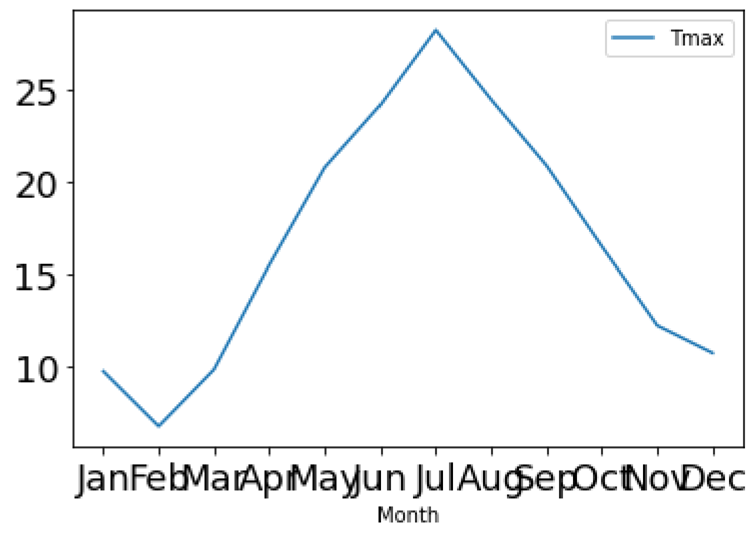
If you wanted to remove the ticks altogether, it would be a simple matter to set either parameter to an empty tuple, e.g. `xticks=()`.

```
weather.plot(y='Tmax', xticks=())
plt.show()
```



If you want to emphasize the ticks even more you can change the font size. In the example below, you can see how.

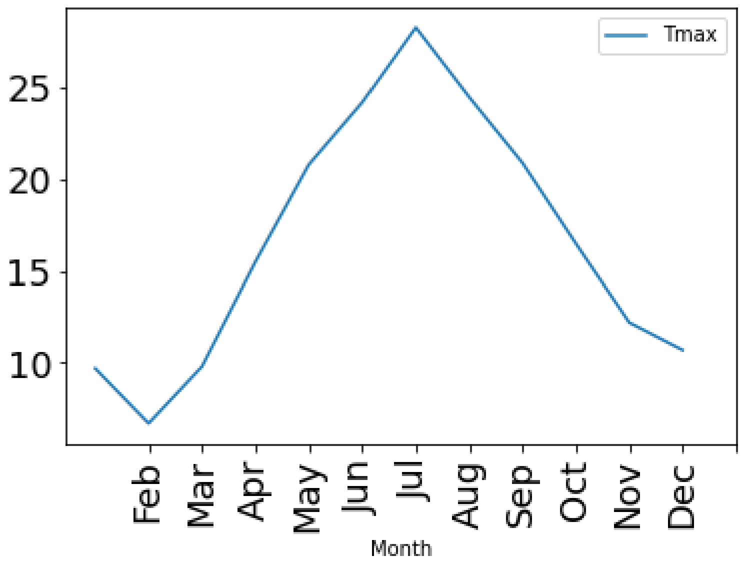
```
plot = weather.plot(y='Tmax', xticks=range(0,12), fontsize=18)
plt.show()
```



Oh, dear, that's not very readable. Let's try rotating the ticks by 90 degrees using the *rot* parameter.

```
plot = weather.plot(y='Tmax', xticks=range(0,12), fontsize=18, rot=90)
```

```
plt.show()
```



By the way, notice how I used the Python *range* function to produce the ticks, instead of explicitly typing them all in a list. This function produces a list starting at the value of the first parameter and ending at one less than the value of the second one.

It could get messy

I mean your code could get messy. What if you wanted to set the ticks, a title, labels, a grid and so on. First, the line of code to do the plot would be very long and, second, if you have several plots to make you find yourself repeating it.

Here's a solution.

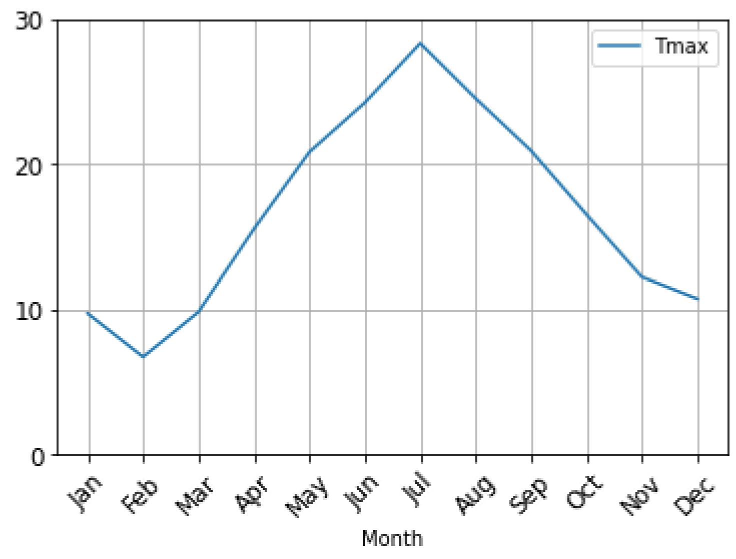
Assume that you want all of your plots to look the same. What you do is define a dictionary of all of the parameters that you want to apply to all of your charts, like this:

```
plot_kwargs={'xticks':range(0,12),
  'yticks':(0,10,20,30),
  'grid':True,
  'fontsize':12,
  'rot':45}
```

then instead of typing in all of the parameters explicitly, you can take advantage of Python's ** operator which will expand a dictionary into a list of keyword arguments. I've called the variable *plot_kwargs* as *kwargs* is the conventional name given to a variable containing keyword parameters (which is what these are).

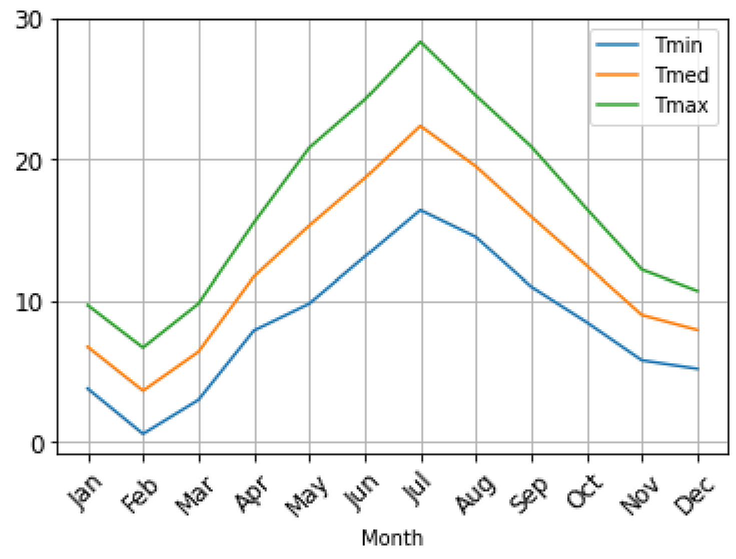
Use them like this:

```
weather.plot(y='Tmax', **plot_kwargs)
plt.show()
```



Now, you can use the same dictionary for other plots, too, for example:

```
weather.plot(y=['Tmin','Tmed','Tmax'], **plot_kwargs)
plt.show()
```



Here, I’ve used the *plot_kwargs* parameter to set the default parameters but explicitly set the ones for the individual plot.

That’s it

Well, no. It isn’t really. There is a lot you can do to customize your plots more, both with Pandas and matplotlib. You can also make changes when you save the plots to a file. There is just far too much to cover in a short introductory text like this. But I hope you have found it useful.