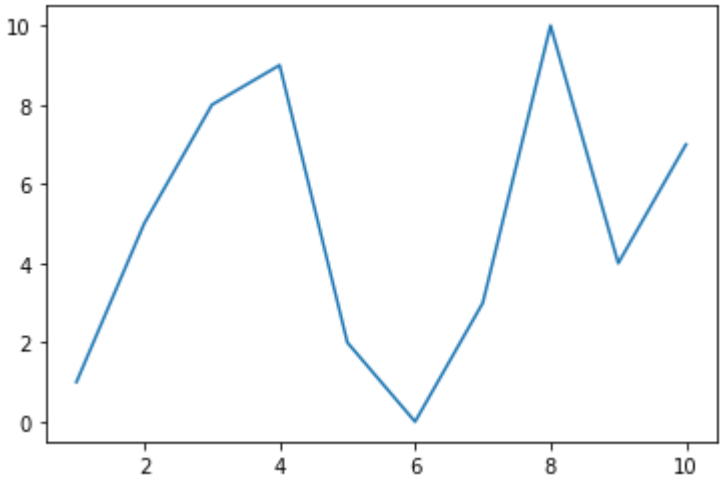
In [1]:
```
##Starting with Plots
```

In [ ]:
```
###Defining the Plot
```

Plots show graphically what you've defined numerically. To define a plot, you need some values, the matplotlib.pyplot module, and an idea of what you want to display, as shown in the following code.

In [4]:
```
import matplotlib.pyplot as plt
values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
x=range(1,11)
plt.plot(x, values)
plt.show()
```

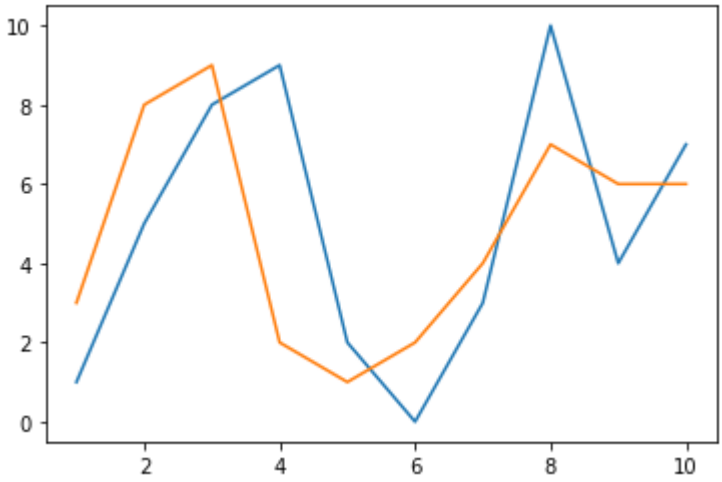


In this case, the code tells the plt.plot() function to create a plot using x-axis values between 1 and 11 and y-axis values as they appear in values. Calling plot.show() displays the plot in a separate dialog box

In [5]:
```
###Drawing multiple lines and plots
```

You encounter many situations in which you must use multiple plot lines, such as when comparing two sets of values. To create such plots using MatPlotLib, you simply call plt.plot() multiple times — once for each plot line, as shown in the following example.

In [6]:
```
values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
values2 = [3, 8, 9, 2, 1, 2, 4, 7, 6, 6]
import matplotlib.pyplot as plt
plt.plot(range(1,11), values)
plt.plot(range(1,11), values2)
plt.show()
```

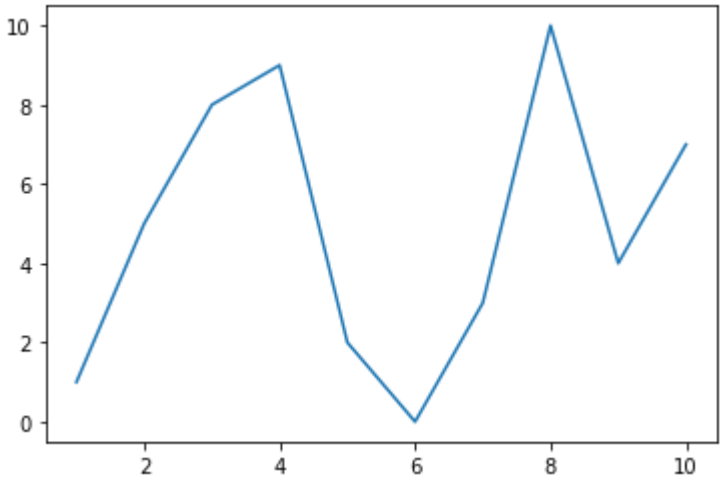


When you run this example, you see two plot lines. The line graphs are different colors so that you can tell them apart.

In [7]:
```
###Saving your work to disk
```

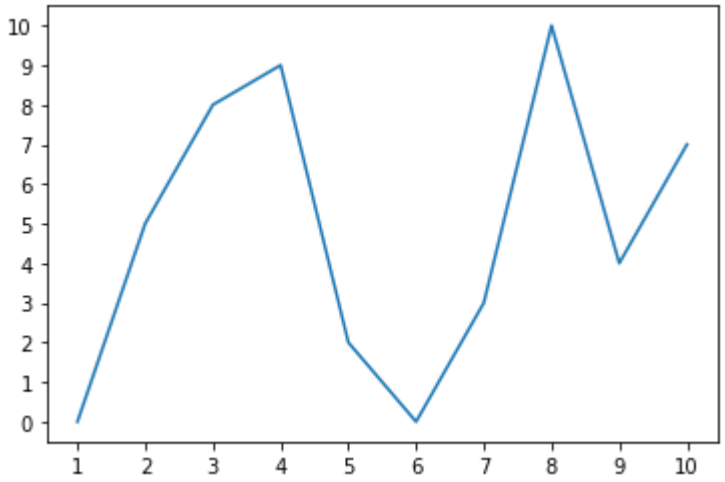
Sometimes you need to save the graphic automatically. In this case, you can save it programmatically using the plt.savefig() function, as shown in the following code:

In [8]:
```
values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
import matplotlib.pyplot as plt
plt.plot(range(1,11), values)
plt.savefig('MySamplePlot.png', format='png')
```



In this case, you must provide a minimum of two inputs. The first input is the filename. You may optionally include a path for saving the file. The second input is the file format. In this case, the example saves the file in Portable Network Graphic (PNG) format, but you have other options: Portable Document Format (PDF), Postscript (PS), Encapsulated Postscript (EPS), and Scalable Vector Graphics (SVG).

In [14]:
```
values = [0, 5, 8, 9, 2, 0, 3, 10, 4, 7]
import matplotlib.pyplot as plt
plt.xticks([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
plt.yticks([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
plt.plot(range(1,11), values)
plt.show()
```



The xticks() and yticks() calls change the ticks used to display data.

In [15]:
```
##Defining the Line Appearance
```

Just drawing lines on a page won't do much for you if you need to help the viewer understand the importance of your data. In most cases, you need to use different line styles to ensure that the viewer can tell one data grouping from another. However, to emphasize the importance or value of a particular data grouping, you need to employ color. The use of color communicates all sorts of ideas to the viewer. For example, green often denotes that something is safe, while red communicates danger.

In [16]:
```
###Woking with Line Styles
```

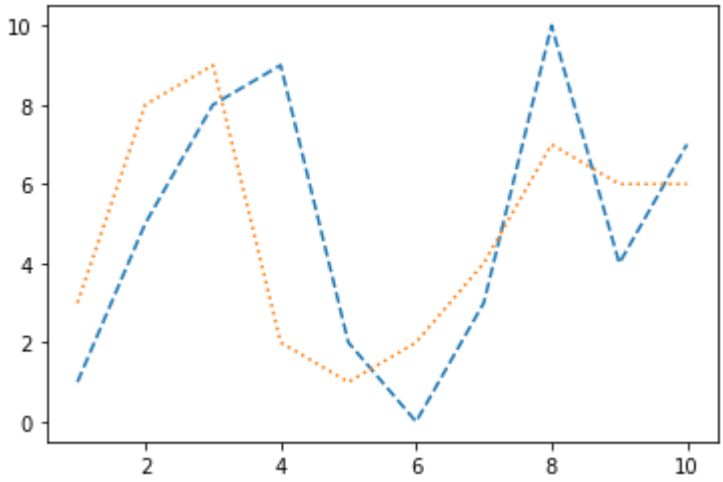
Line styles help differentiate graphs by drawing the lines in various ways. Using a unique presentation for each line helps you distinguish each line so that you can call it out (even when the printout is in shades of gray). You could also call out a particular line graph by using a different line style for it (and using the same style for the other lines).

MatPlotLib Line Styles

Character	*Line Style*
'-'	Solid line
'--'	Dashed line
'-.'	Dash-dot line
':'	Dotted line

The line style appears as a third argument to the plot() function call. You simply provide the desired string for the line type, as shown in the following example.

In [28]:
```
import matplotlib.pyplot as plt
values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
values2 = [3, 8, 9, 2, 1, 2, 4, 7, 6, 6]
plt.plot(range(1,11), values, '--')
plt.plot(range(1,11), values2, ':')
plt.show()
```



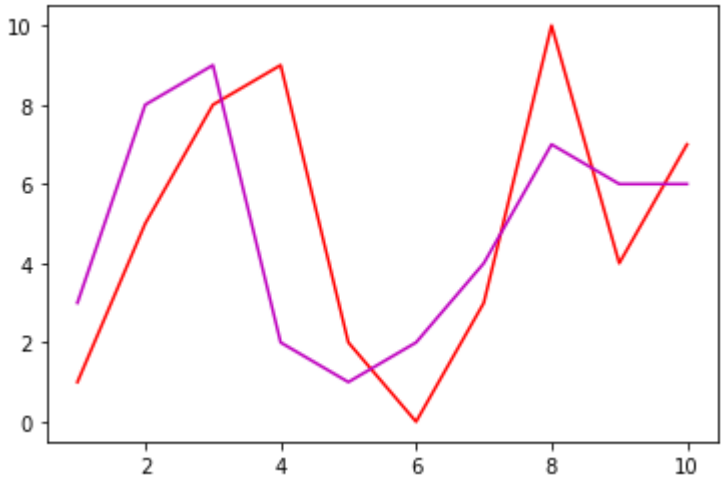
In this case, the first line graph uses a dashed line style, while the second line graph uses a dotted line style.

In [30]:
```
# Colours Supported by MatPlotLib
```

Character	Color
'b'	Blue
'g'	Green
'r'	Red
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

As with line styles, the color appears in a string as the third argument to the plot() function call. In this case, the viewer sees two lines — one in red and the other in magenta.

In [31]:
```
import matplotlib.pyplot as plt

values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
values2 = [3, 8, 9, 2, 1, 2, 4, 7, 6, 6]
plt.plot(range(1,11), values, 'r')
plt.plot(range(1,11), values2, 'm')
plt.show()
```



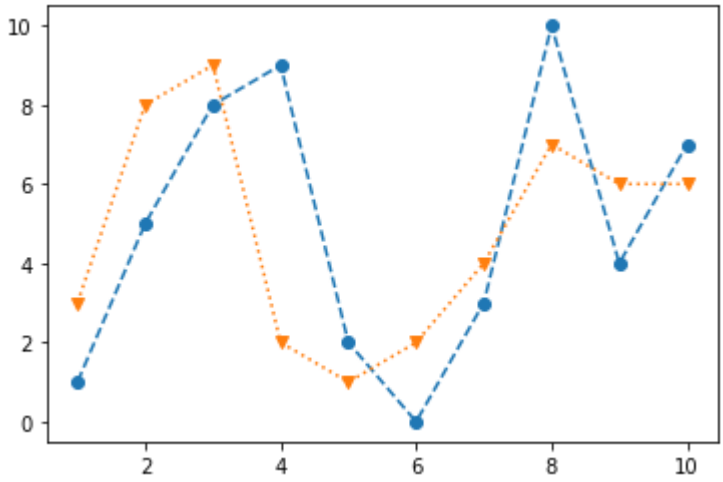
In [32]:
```
##Adding Markers
```

Markers add a special symbol to each data point in a line graph.

Character	*Marker Type*
'.'	Point
','	Pixel
'o'	Circle
'v'	Triangle 1 down
'^'	Triangle 1 up
'<'	Triangle 1 left
'>'	Triangle 1 right
'1'	Triangle 2 down
'2'	Triangle 2 up
'3'	Triangle 2 left
'4'	Triangle 2 right
's'	Square
'p'	Pentagon
'*'	Star
'h'	Hexagon style 1
'H'	Hexagon style 2
'+'	Plus
'x'	X
'D'	Diamond
'd'	Thin diamond
'|'	Vertical line
'_'	Horizontal line

As with line style and color, you add markers as the third argument to a plot() call. In the following example, you see the effects of combining line style with a marker to provide a unique line graph presentation.

In [34]:
```
import matplotlib.pyplot as plt

values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
values2 = [3, 8, 9, 2, 1, 2, 4, 7, 6, 6]
plt.plot(range(1,11), values, 'o--')
plt.plot(range(1,11), values2, 'v:')
plt.show()
```



In [35]:
```
##Using Labels, Annotations, and Legends
```

To fully document your graph, you usually have to resort to labels, annotations, and legends. Each of these elements has a different purpose, as follows:

✓✓Label: Provides positive identification of a particular data element or grouping. The purpose is to make it easy for the viewer to know the name or kind of data illustrated.

✓✓Annotation: Augments the information the viewer can immediately see about the data with notes, sources, or other useful information. In contrast to a label, the purpose of annotation is to help extend the viewer's knowledge of the data rather than simply identify it.

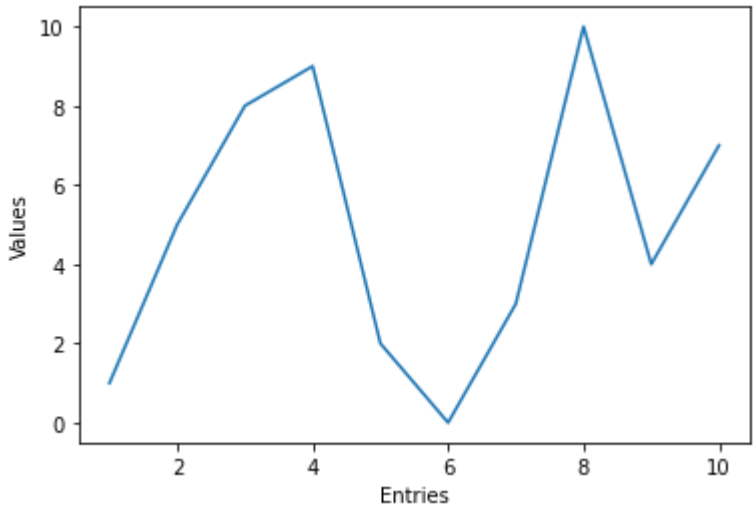
✓✓Legend: Presents a listing of the data groups within the graph and often provides cues (such as line type or color) to make identification of the data group easier. For example, all the red points may belong to group A, while all the blue points may belong to group B.

In [36]:
```
###Adding Labels
```

Labels help people understand the significance of each axis of any graph you create. Without labels, the values portrayed don't have any significance. In addition to a moniker, such as rainfall, you can also add units of measure, such as inches or centimeters, so that your audience knows how to interpret the data shown. The following example shows how to add labels to your graph:

In [37]:
```
values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
import matplotlib.pyplot as plt
plt.xlabel('Entries')
plt.ylabel('Values')
```

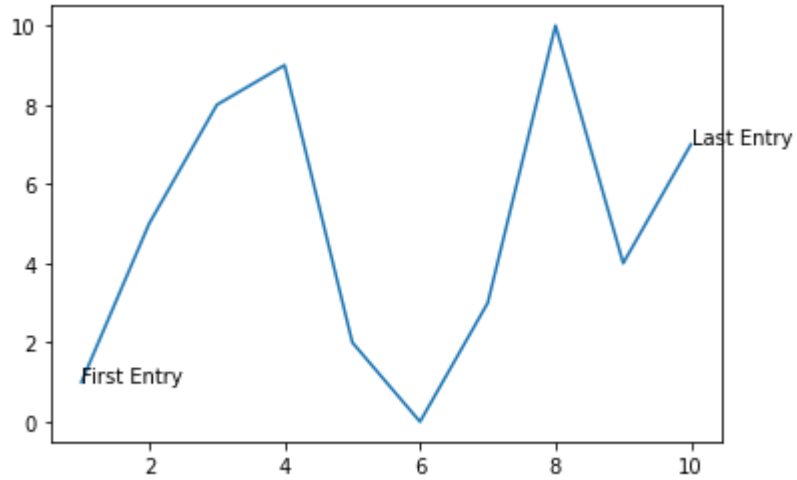
```
plt.plot(range(1,11), values)
plt.show()
```



In [38]:
```
###Annotating the chart
```

You use annotation to draw special attention to points of interest on a graph. For example, you may want to point out that a specific data point is outside the usual range expected for a particular dataset. The following example shows how to add annotation to a graph.

In [49]:
```
import matplotlib.pyplot as plt
values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
xvalues=range(1,11)
plt.annotate(text='First Entry',xy=[1,1])
plt.annotate(text='Last Entry',xy=[10,7])
plt.plot(xvalues, values)
plt.show()
```

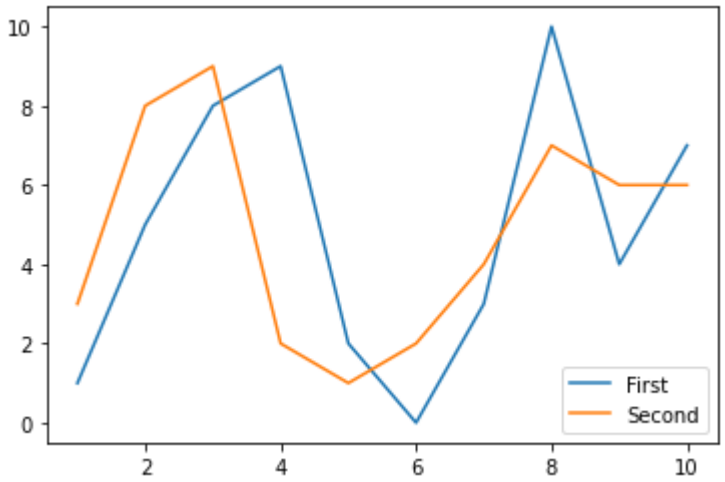


The call to annotate() provides the labeling you need. You must provide a location for the annotation by using the xy parameter, as well as provide text to place at the location by using the s parameter. The annotate() function also provides other parameters that you can use to create special formatting or placement onscreen.

In [50]:
```
###Creating a legend
```

A legend documents the individual elements of a plot. Each line is presented in a table that contains a label for it so that people can differentiate between each line. For example, one line may represent sales in 2014 and another line may represent sales in 2015, so you include an entry in the legend for each line that is labeled 2014 and 2015. The following example shows how to add a legend to your plot.

In [51]:
```
import matplotlib.pyplot as plt
values = [1, 5, 8, 9, 2, 0, 3, 10, 4, 7]
values2 = [3, 8, 9, 2, 1, 2, 4, 7, 6, 6]

line1 = plt.plot(range(1,11), values)
line2 = plt.plot(range(1,11), values2)
plt.legend(['First', 'Second'], loc=0)
plt.show()
```



The call to legend() occurs after you create the plots, not before, as with some of the other functions described in this chapter. You must provide a handle to each of the plots. Notice how line1 is set equal to the first plot() call and line2 is set equal to the second plot() call. The default location for the legend is the upper-right corner of the plot, which proved inconvenient for this particular example. Adding the loc parameter lets you place the legend in a different location.

Location Code The string 'best' places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

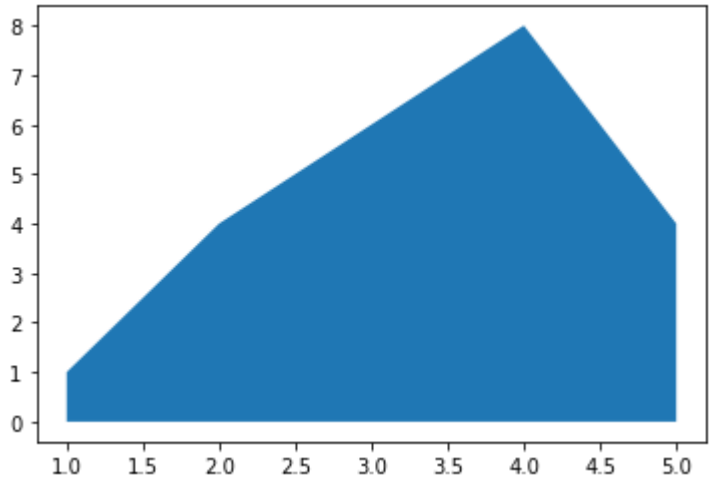
In [52]:
```
## Area plots/charts
```

An area chart combines the line chart and bar chart to show how one or more groups' numeric values change over the progression of a second variable, typically that of time. An area chart is distinguished from a line chart by the addition of shading between lines and a baseline, like in a bar chart.

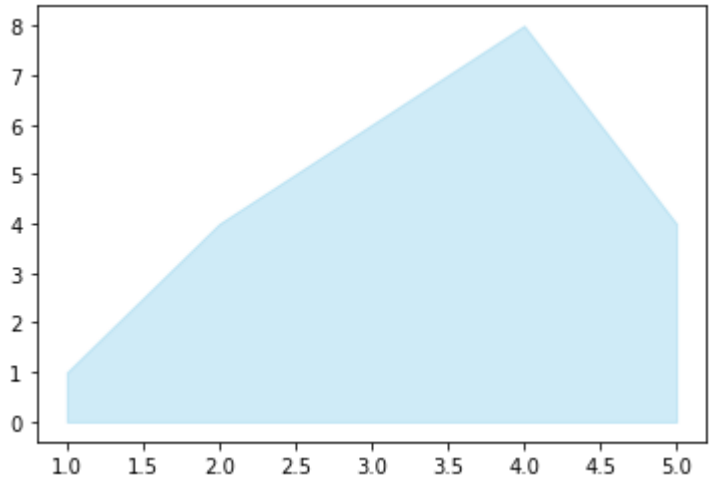
An area chart is typically used with multiple lines to make a comparison between groups (aka series) or to show how a whole is divided into component parts.

In [53]:
```
### Simple Area Chart
```

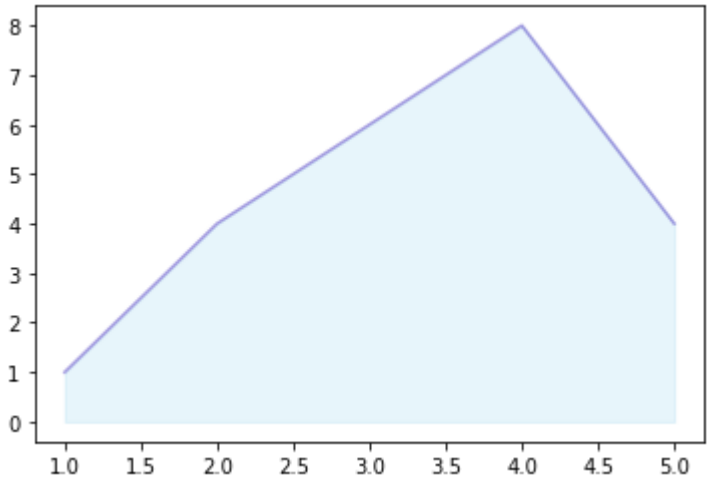
In [54]:
```
import numpy as np
import matplotlib.pyplot as plt

# Create data
x=range(1,6)
y=[1,4,6,8,4]

# Area plot
plt.fill_between(x, y)
plt.show()
```



In [56]:
```
# Change the color and its transparency
plt.fill_between( x, y, color="skyblue", alpha=0.4)

# Show the graph
plt.show()
```



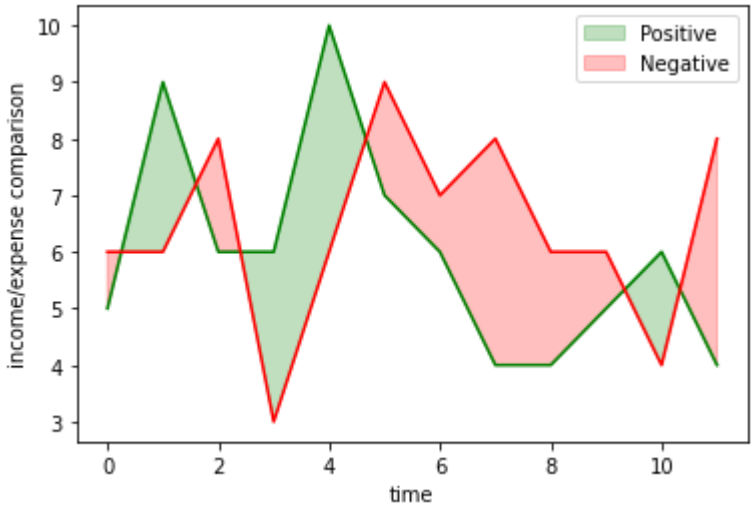
In [57]:
```

# Same, but add a stronger line on top (edge)
plt.fill_between( x, y, color="skyblue", alpha=0.2)
plt.plot(x, y, color="Slateblue", alpha=0.6)
# See the line plot function to learn how to customize the plt.plot function

# Show the graph
plt.show()
```



In [59]:
```
###Area fill between two lines in Matplotlib
```

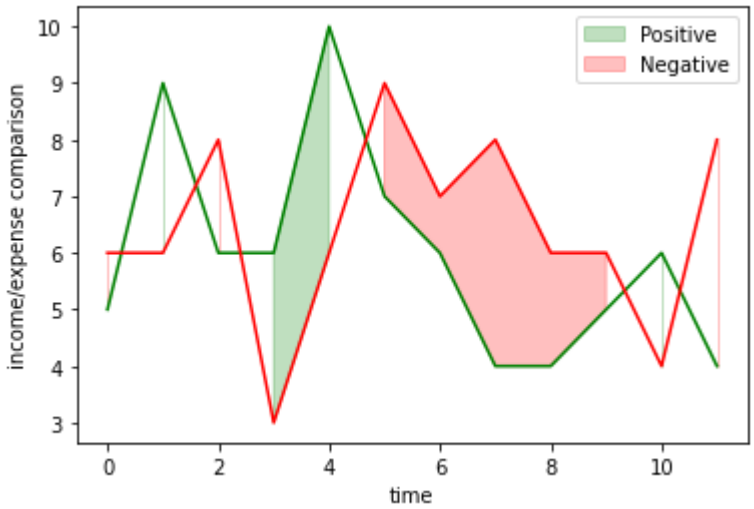
NumPy arange() is one of the array creation routines based on numerical ranges. It creates an instance of ndarray with evenly spaced values and returns the reference to it.

In [62]:
```
import numpy as np
import matplotlib.pyplot as plt

time = np.arange(12)
income = np.array([5, 9, 6, 6, 10, 7, 6, 4, 4, 5, 6, 4])
expenses = np.array([6, 6, 8, 3, 6, 9, 7, 8, 6, 6, 4, 8])

# Plot lines
plt.plot(time, income, color="green")
plt.plot(time, expenses, color="red")

# Fill area when income > expenses with green
plt.fill_between(time, income, expenses, where=(income > expenses),
    color="green", alpha=0.25,label="Positive", interpolate = True)

# Fill area when income <= expenses with red
plt.fill_between(time, income, expenses, where=(income <= expenses),
    color="red", alpha=0.25,label="Negative", interpolate = True)

plt.xlabel('time')
plt.ylabel('income/expense comparison')
plt.legend()
plt.show()
```



In [64]:

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

time = np.arange(12)
income = np.array([5, 9, 6, 6, 10, 7, 6, 4, 4, 5, 6, 4])
expenses = np.array([6, 6, 8, 3, 6, 9, 7, 8, 6, 6, 4, 8])

# Plot lines
plt.plot(time, income, color="green")
plt.plot(time, expenses, color="red")

# Fill area when income > expenses with green
plt.fill_between(time, income, expenses, where=(income > expenses),
    color="green", alpha=0.25,label="Positive")

# Fill area when income <= expenses with red
plt.fill_between(time, income, expenses, where=(income <= expenses),
    color="red", alpha=0.25, label="Negative")

plt.xlabel('time')
plt.ylabel('income/expense comparison')
plt.legend()
plt.show()
```

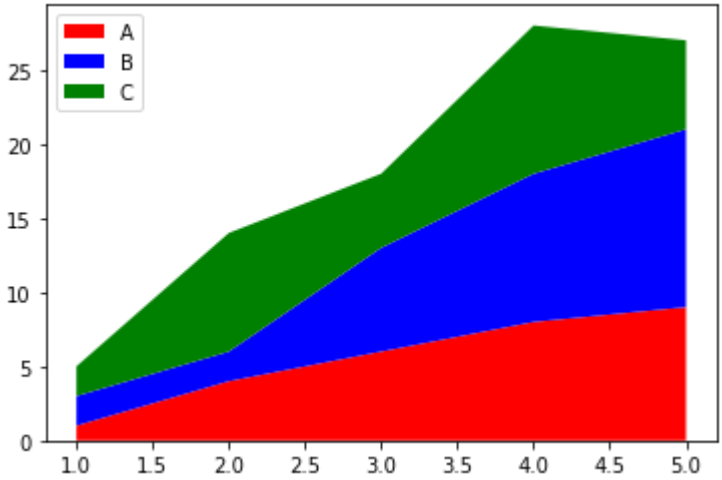


Note: interpolate=True perform color change even between the two points and fill between that part as per income vs expense.

In [65]:
```
###Stacked area Chart
```

A stacked area chart displays the evolution of a numeric variable for several groups of a dataset. Each group is displayed on top of each other, making it easy to read the evolution of the total, but hard to read each group value accurately. In python, stacked area charts are mainly done thanks to the stackplot() function

In [66]:
```
import numpy as np
import matplotlib.pyplot as plt

# Create data
x=range(1,6)
y1=[1,4,6,8,9]
y2=[2,2,7,10,12]
y3=[2,8,5,10,6]

# Basic stacked area chart.
plt.stackplot(x,y1, y2, y3, labels=['A','B','C'], colors = ['red', 'blue', 'green'])
plt.legend(loc='upper left')
plt.show()
```



Note: for y2 graph value is y1+y2 and for y3 it is y1+y2+y3

In [67]:
```
#################### Box Plot ######################################
```

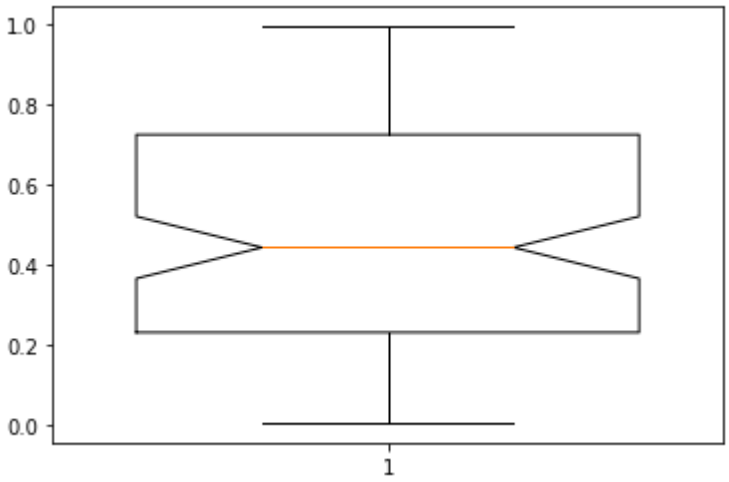
In [68]:
```
##Depicting groups using box plots
```

Box plots provide a means of depicting groups of numbers through their quartiles (three points dividing a group into four equal parts). A box plot may also have lines, called whiskers, indicating data outside the upper and lower quartiles. The spacing shown within a box plot helps indicate the skew and dispersion of the data. The following example shows how to create a box plot with randomized data.

In [71]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
data = np.random.rand(100)
print(data)
plt.boxplot(data, widths=0.75, notch=True)
plt.show()
```

```
[0.05696773 0.73247623 0.18983201 0.29808896 0.48338626 0.61533219
 0.44277213 0.22824219 0.61436516 0.70185612 0.29860231 0.45330848
 0.92558963 0.99545391 0.80476006 0.16357082 0.79069643 0.19596974
 0.79504901 0.4489259  0.25730755 0.22038543 0.43727524 0.60863878
 0.41003498 0.34413426 0.6003342  0.09293531 0.62941014 0.56276311
 0.94893973 0.21376589 0.28053783 0.60915324 0.94586954 0.61573887
 0.30644242 0.30763695 0.83278631 0.32255384 0.99359387 0.51428912
 0.78956885 0.77754291 0.12849036 0.06080509 0.72967117 0.2742372
 0.18875642 0.99519173 0.74763425 0.40604346 0.18902969 0.79591518
 0.40172664 0.86666002 0.3179361  0.14260269 0.32874723 0.79928609
 0.02247766 0.30226323 0.94348646 0.7869913  0.29206653 0.78646531
 0.08184198 0.17938259 0.67881264 0.51683186 0.20550475 0.67038367
 0.7166741  0.72425675 0.30482212 0.7168436  0.24733586 0.33003413
 0.94161021 0.73534001 0.79347489 0.23260851 0.45820358 0.12510472
 0.02853348 0.53438873 0.50913282 0.55528267 0.32047926 0.61140689
 0.04001705 0.93801424 0.42559377 0.08776877 0.00613248 0.11836615
 0.23485833 0.06413681 0.44464059 0.1511735 ]
```



In [72]:
```
#To check values of Q1 ,Q3 and Low_val and High_val to find outliers
df = pd.DataFrame(data)
print(df.describe())
```

```
                0
count  100.000000
mean     0.470924
std      0.283385
min      0.006132
25%      0.231517
50%      0.443706
75%      0.725610
max      0.995454
```

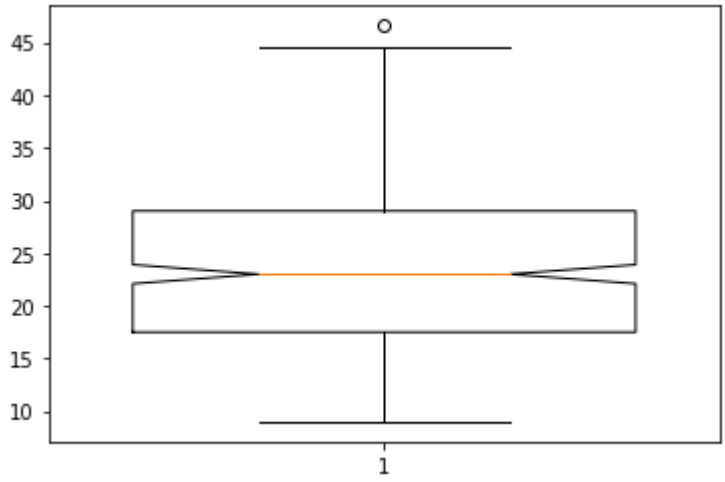
The call to boxplot()requires only data as input. All other parameters have default settings. In this case, the code sets the presentation of outliers to green Xs by setting the sym parameter. You use widths to modify the size of the box (made extra large in this case to make the box easier to see). Finally, you can create a square box or a box with a notch using the notch parameter (which normally defaults to False).

The box shows the three data points as the box, with the red line in the middle being the median. The two black horizontal lines connected to the box by whiskers show the upper and lower limits

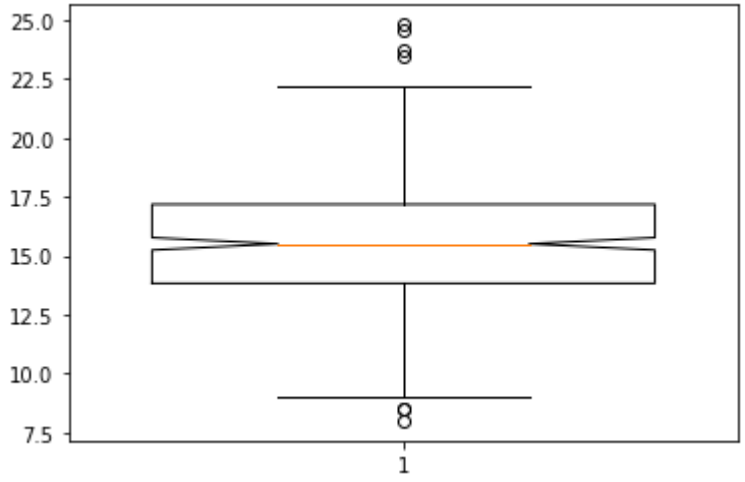
(for four quartiles). The outliers appear above and below the upper and lower limit lines as green Xs.

Boxplots can be useful for detecting outliers too. As seen in the previous chapter in the auto-mpg dataset, the mpg and acceleration columns had outliers. They can be clearly seen by creating boxplots.

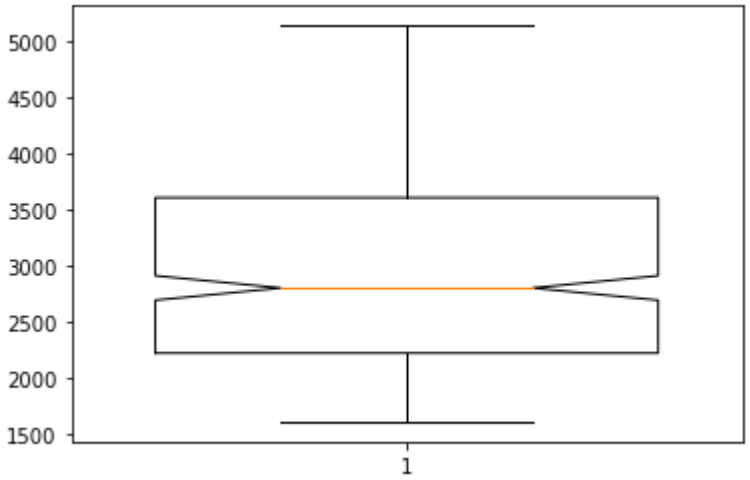
In [73]:
```
import matplotlib.pyplot as plt
import pandas as pd

dataset = pd.read_csv('auto-mpg.csv')
plt.boxplot(dataset['mpg'], widths=0.75, notch=True)
plt.show()
```



In [74]:
```
plt.boxplot(dataset['acceleration'], widths=0.75, notch=True)
plt.show()
```



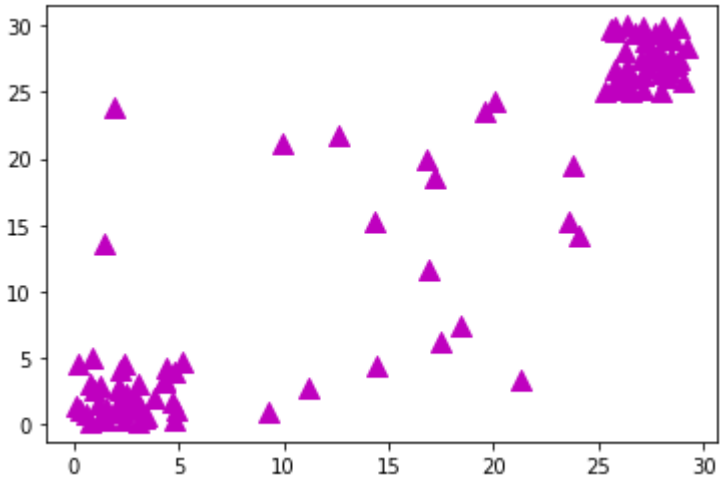
In [75]:
```
plt.boxplot(dataset['weight'], widths=0.75, notch=True)
plt.show()
```



In [77]:
```
########### Seeing data patterns using scatterplots  ######################
```

Scatterplots show clusters of data rather than trends (as with line graphs) or discrete values (as with bar charts). The purpose of a scatterplot is to help you see data patterns. The following example shows how to create a scatterplot using randomized data:

In [78]:
```
import numpy as np
import matplotlib.pyplot as plt
x1 = 5 * np.random.rand(40)
x2 = 5 * np.random.rand(40) + 25
x3 = 25 * np.random.rand(20)
x = np.concatenate((x1, x2, x3))
y1 = 5 * np.random.rand(40)
y2 = 5 * np.random.rand(40) + 25
y3 = 25 * np.random.rand(20)
y = np.concatenate((y1, y2, y3))
plt.scatter(x, y, s=[100], marker='^', c='m')
plt.show()
```



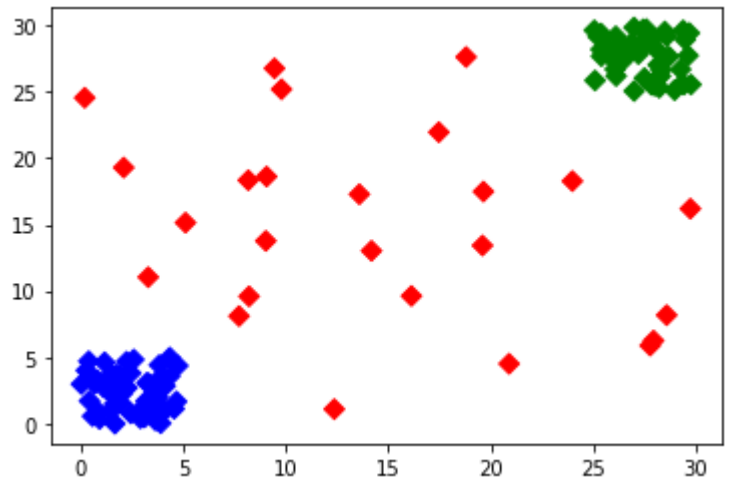
In [ ]:
```
##Creating Advanced Scatterplots
```

Scatterplots are especially important for data science because they can show data patterns that aren't obvious when viewed in other ways. You can see data groupings with relative ease and help the viewer understand when data belongs to a particular group. You can also show overlaps between groups and even demonstrate when certain data is outside the expected range. Showing

these various kinds of relationships in the data is an advanced technique that you need to know in order to make the best use of MatPlotLib.

Color is the third axis when working with a scatterplot. Using color lets you highlight groups so that others can see them with greater ease. The following example shows how you can use color to show groups within a scatterplot:

In [79]:
```
import numpy as np
import matplotlib.pyplot as plt
x1 = 5 * np.random.rand(50)
x2 = 5 * np.random.rand(50) + 25
x3 = 30 * np.random.rand(25)
x = np.concatenate((x1, x2, x3))
y1 = 5 * np.random.rand(50)
y2 = 5 * np.random.rand(50) + 25
y3 = 30 * np.random.rand(25)
y = np.concatenate((y1, y2, y3))
color_array = ['b'] * 50 + ['g'] * 50 + ['r'] * 25
plt.scatter(x, y, s=[50], marker='D', c=color_array)
plt.show()
```



The example works essentially the same as the scatterplot example in the previous section, except that this example uses an array for the colors. The first group is blue, followed by green for the second group. Any outliers appear in red.

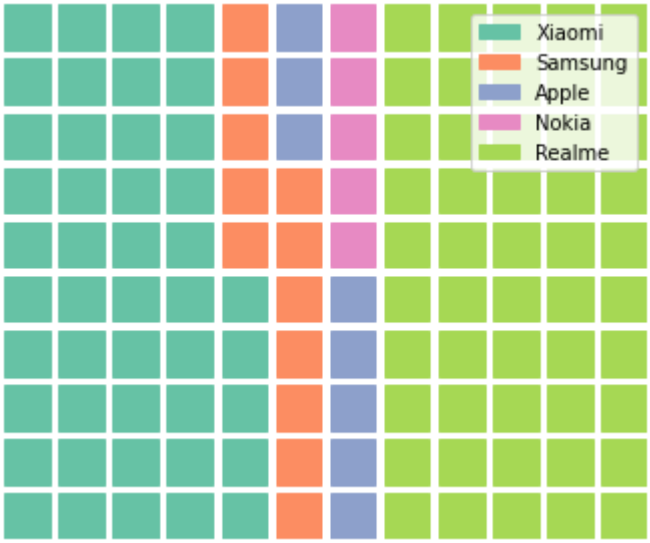
In [80]:
```
###################### Waffle Charts ########################
```

A waffle chart is an interesting visualization that is normally created to display progress toward goals. It is commonly an effective option when you are trying to add interesting visualization features to a visual that consists mainly of cells, such as an Excel dashboard.

In [ ]:
```
# Command to Install pywaffle library
!pip install pywaffle
```

In [88]:
```
import pandas as pd
import matplotlib.pyplot as plt
from pywaffle import Waffle

# creation of a dataframe
data ={'phone': ['Xiaomi', 'Samsung','Apple', 'Nokia', 'Realme'],
```

```
        'stock': [45, 12, 8, 5, 50]
      }

df = pd.DataFrame(data)

# To plot the waffle Chart
fig = plt.figure(FigureClass = Waffle, rows = 10,
    values = df.stock, labels = list(df.phone))
```



In [89]:
```
import matplotlib.pyplot as plt
import pandas as pd
from pywaffle import Waffle

data={'phone':['Realme','Poco','Samsung','Apple','Vivo','Oppo'],'stock':[44,15,20,50,60
df=pd.DataFrame(data)
print(df)
fig=plt.figure(FigureClass=Waffle,rows=10,values=df.stock,labels=list(df.phone))
```

```
     phone  stock
0   Realme     44
1     Poco     15
2  Samsung     20
3    Apple     50
4     Vivo     60
5     Oppo     25
```

In [90]:
```
############## Word Cloud ######################
```

In [91]:
```
# install wordcloud
!pip3 install wordcloud==1.8.1
```

```
Collecting wordcloud==1.8.1
  Downloading wordcloud-1.8.1-cp38-cp38-win_amd64.whl (155 kB)
Requirement already satisfied: matplotlib in c:\programdata\anaconda3\lib\site-packages
(from wordcloud==1.8.1) (3.3.2)
Requirement already satisfied: pillow in c:\programdata\anaconda3\lib\site-packages (fro
m wordcloud==1.8.1) (8.0.1)
Requirement already satisfied: numpy>=1.6.1 in c:\users\tejas\appdata\roaming\python\pyt
hon38\site-packages (from wordcloud==1.8.1) (1.24.3)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.3 in c:\programdat
a\anaconda3\lib\site-packages (from matplotlib->wordcloud==1.8.1) (2.4.7)
Requirement already satisfied: cycler>=0.10 in c:\programdata\anaconda3\lib\site-package
s (from matplotlib->wordcloud==1.8.1) (0.10.0)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\programdata\anaconda3\lib\site-pa
ckages (from matplotlib->wordcloud==1.8.1) (1.3.0)
Requirement already satisfied: certifi>=2020.06.20 in c:\programdata\anaconda3\lib\site-
packages (from matplotlib->wordcloud==1.8.1) (2020.6.20)
Requirement already satisfied: python-dateutil>=2.1 in c:\users\tejas\appdata\roaming\py
thon\python38\site-packages (from matplotlib->wordcloud==1.8.1) (2.8.2)
Requirement already satisfied: six in c:\programdata\anaconda3\lib\site-packages (from c
ycler>=0.10->matplotlib->wordcloud==1.8.1) (1.15.0)
Installing collected packages: wordcloud
  Attempting uninstall: wordcloud
    Found existing installation: wordcloud 1.9.1.1
    Uninstalling wordcloud-1.9.1.1:
      Successfully uninstalled wordcloud-1.9.1.1
Successfully installed wordcloud-1.8.1
```

`Word` clouds are commonly used to perform high-level analysis and visualization of text data. Let's try to analyze a short novel written by **Lewis Carroll** titled *Alice's Adventures in Wonderland*. Let's go ahead and download a *.txt* file of the novel.

In [38]:
```
alice_novel=open("alice.txt")
alice_novel=alice_novel.read()
#print(alice_novel) ---- IT WILL SHOW WHOLE FILE DETAILS STORED IN alice_novel variable
```

Next, let's use the stopwords that we imported from `word_cloud` . We use the function *set* to remove any redundant stopwords.

Create a word cloud object and generate a word cloud. For simplicity, let's generate a word cloud using only the first 2000 words in the novel.

In [46]:
```
# import package and its set of stopwords
from wordcloud import WordCloud, STOPWORDS

alice_novel=open("alice.txt")
alice_novel=alice_novel.read()
# print(alice_novel) ##---- IT WILL SHOW WHOLE FILE DETAILS STORED IN alice_novel varia

stopwords = set(STOPWORDS)
# stopwords  # it will show set of stopwords

# instantiate a word cloud object
alice_wc = WordCloud(background_color='white',max_words=2000,
                     stopwords=stopwords)

# generate the word cloud
alice_wc.generate(alice_novel)
```

Out[46]:
```
<wordcloud.wordcloud.WordCloud at 0x2689a292490>
```

Awesome! Now that the `word` cloud is created, let's visualize it.

In [47]:
```
# display the word cloud
import matplotlib.pyplot as plt
plt.imshow(alice_wc)
plt.axis('off')
plt.show()
```

The matplotlib function imshow() creates an image from a 2-dimensional numpy array.

Interesting! So in the first 2000 words in the novel, the most common words are **Alice**, **said**, **little**, **Queen**, and so on. Let's resize the cloud so that we can see the less frequent words a little better.

In [51]:
```
fig = plt.figure(figsize=(14, 18))

# display the cloud
plt.imshow(alice_wc)
plt.axis('off')
plt.show()
```

Much better! However, **said** isn't really an informative word. So let's add it to our stopwords and re-generate the cloud.

In [52]:
```
stopwords.add('said') # add the words said to stopwords

# re-generate the word cloud
```

```
alice_wc.generate(alice_novel)

# display the cloud
fig = plt.figure(figsize=(14, 18))

plt.imshow(alice_wc)
plt.axis('off')
plt.show()
```

In [54]:
```
##################### Regression Plot ##########################
```

Regression Plots

> Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics. You can learn more about *seaborn* by following this link and more about *seaborn* regression plots by following this link.

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn helps resolve the two major problems faced by Matplotlib; the problems are ?

1. Default Matplotlib parameters
2. Working with data frames

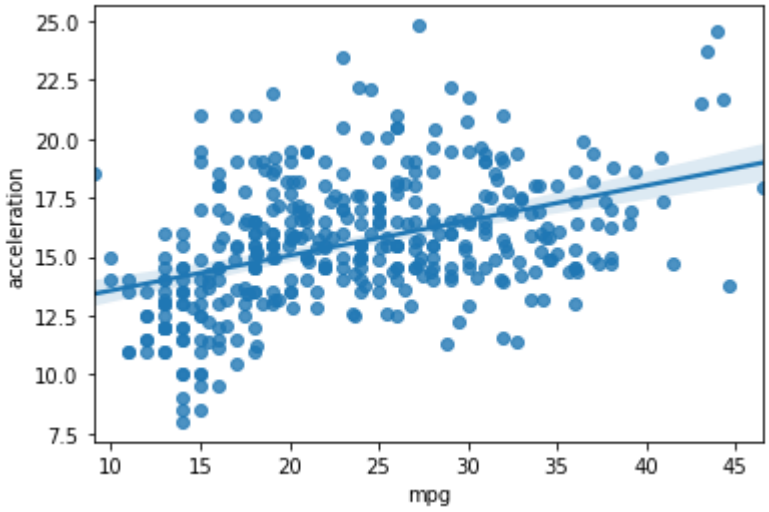
As Seaborn compliments and extends Matplotlib, the learning curve is quite gradual. If you know Matplotlib, you are already half-way through Seaborn.

With seaborn, generating a regression plot is as simple as calling the regplot function.

This method is used to plot data and a linear regression model fit. There are a number of mutually exclusive options for estimating the regression model.

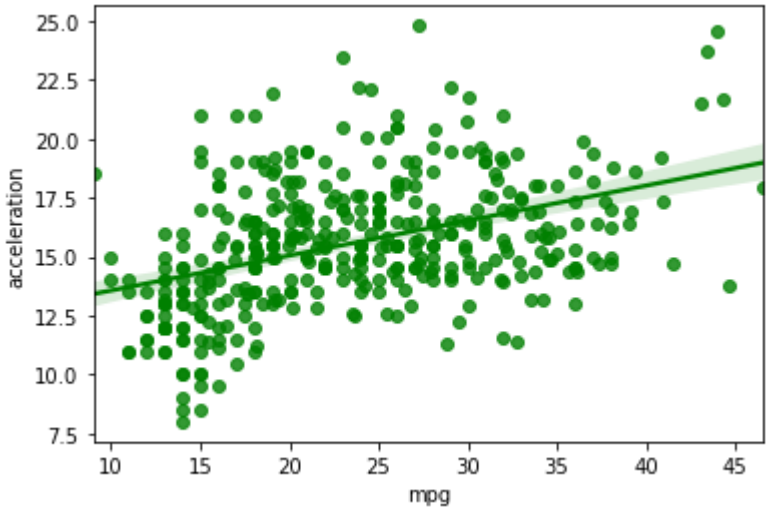
In [4]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

```
import pandas as pd
dataset = pd.read_csv('auto-mpg.csv')
sns.regplot(x='mpg', y='acceleration', data=dataset)
plt.show()
```



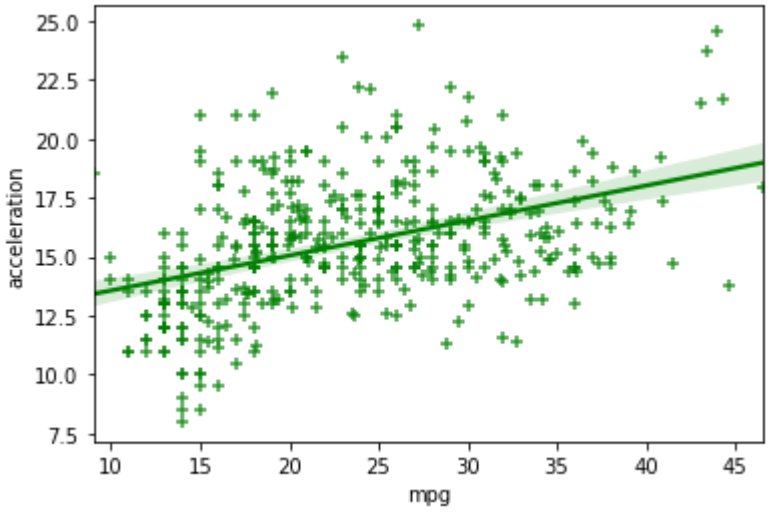
This is not magic; it is seaborn! You can also customize the color of the scatter plot and regression line. Let's change the color to green.

In [58]:
```
sns.regplot(x='mpg', y='acceleration', data = dataset, color='green')
plt.show()
```

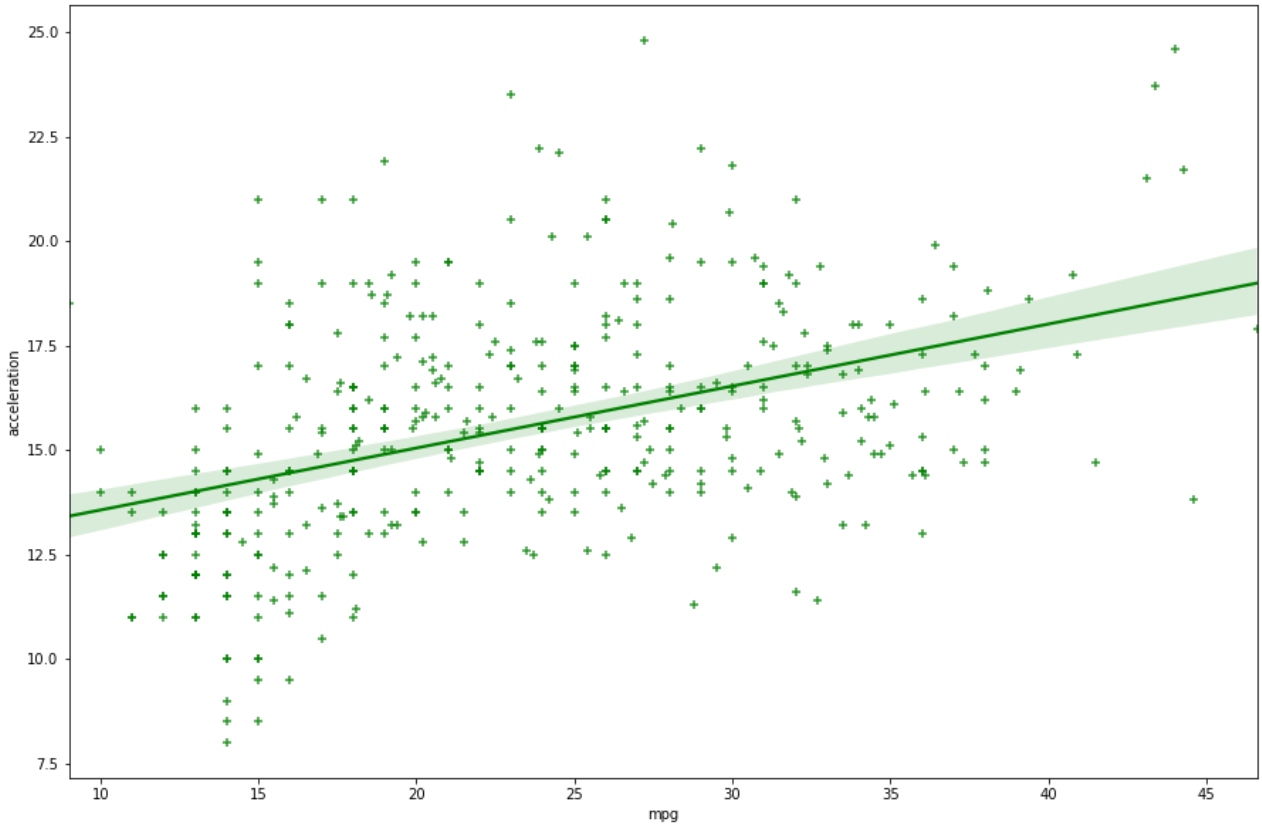


You can always customize the marker shape, so instead of circular markers, let's use `+`.

In [61]:
```
sns.regplot(x='mpg', y='acceleration', data = dataset, color='green', marker = '+')
plt.show()
```



In [6]:
```
plt.figure(figsize=(15, 10))

sns.regplot(x='mpg', y='acceleration', data = dataset, color='green', marker = '+')
plt.show()
```



And let's increase the size of markers so they match the new size of the figure, and add a title and x- and y-labels.

In [79]:
```
plt.figure(figsize=(15, 10))
sns.set(font_scale=1.5)   # to increase font scale

ax=sns.regplot(x='mpg', y='acceleration', data = dataset, color='green', marker = '+',s
ax.set(xlabel='mpg', ylabel='acceleration') # add x- and y-labels
ax.set_title('mpg vs acceleration') # add title
plt.show()
```

In [80]:
```
dataset.corr()
```

Out[80]:

	mpg	cylinders	displacement	weight	acceleration	model year	origin
mpg	1.000000	-0.775396	-0.804203	-0.831741	0.420289	0.579267	0.563450
cylinders	-0.775396	1.000000	0.950721	0.896017	-0.505419	-0.348746	-0.562543
displacement	-0.804203	0.950721	1.000000	0.932824	-0.543684	-0.370164	-0.609409
weight	-0.831741	0.896017	0.932824	1.000000	-0.417457	-0.306564	-0.581024
acceleration	0.420289	-0.505419	-0.543684	-0.417457	1.000000	0.288137	0.205873
model year	0.579267	-0.348746	-0.370164	-0.306564	0.288137	1.000000	0.180662
origin	0.563450	-0.562543	-0.609409	-0.581024	0.205873	0.180662	1.000000

Let's create a regression plot for attributes with a good correlation

In [ ]:
```
#Weight Vs Displacement
```

In [83]:
```
plt.figure(figsize=(15, 10))
sns.set(font_scale=1.5)

ax = sns.regplot(x='weight', y='displacement', data = dataset, color='green', marker =
ax.set(xlabel='weight', ylabel='displacement') # add x- and y-labels
ax.set_title('weight vs displacement') # add title
plt.show()
```

In [ ]:
```
# Bad Correlation Weight Vs model year
```

In [84]:
```
plt.figure(figsize=(15, 10))
sns.set(font_scale=1.5)

ax = sns.regplot(x='weight', y='model year', data = dataset, color='green', marker = '+
ax.set(xlabel='weight', ylabel='model year') # add x- and y-labels
ax.set_title('weight vs model year') # add title
plt.show()
```

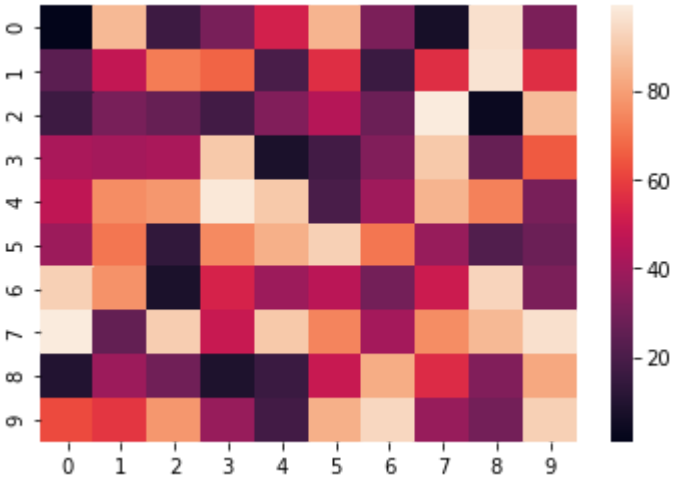
In [85]:
```
############################  Heatmaps  ##########################################
```

Heatmap is defined as a graphical representation of data using colors to visualize the value of the matrix. In this, to represent more common values or higher activities brighter colors basically reddish colors are used and to represent less common or activity values, darker colors are preferred. Heatmap is also defined by the name of the shading matrix. Heatmaps in Seaborn can be plotted by using the seaborn.heatmap() function.

In [1]:
```
# importing the modules
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# generating 2-D 10x10 matrix of random numbers from 1 to 100
data = np.random.randint(low = 1,high = 100,size = (10, 10))

print("The data to be plotted:\n")
print(data)

# plotting the heatmap
hm=sns.heatmap(data)

# displaying the plotted heatmap
plt.show()
```

```
The data to be plotted:

[[ 1 86 17 31 52 85 32  7 96 32]
 [24 48 72 67 20 56 16 56 97 56]
 [17 31 27 18 33 45 28 99  4 87]
 [42 41 42 90  8 18 33 90 27 65]
```

```
[47 76 78 98 90 20 40 85 73 31]
[39 71 14 75 84 92 71 38 22 28]
[92 77  8 53 39 46 30 50 93 32]
[99 26 91 49 90 74 41 76 86 96]
[10 39 29  9 16 49 83 55 33 82]
[62 58 78 38 18 84 94 38 30 92]]
```



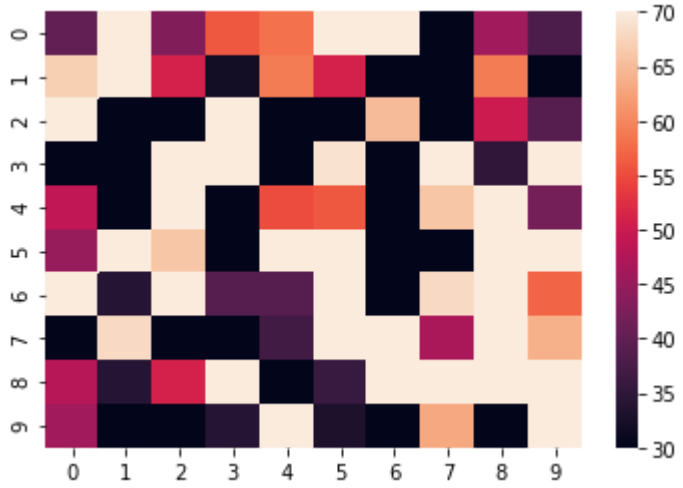
Anchoring the colormap

If we set the vmin value to 30 and the vmax value to 70, then only the cells with values between 30 and 70 will be displayed. This is called anchoring the colormap.

In [5]:

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

# generating 2-D 10x10 matrix of random numbers
# from 1 to 100
data = np.random.randint(low=1,high=100,size=(10, 10))
print(data)

# plotting the heatmap
hm = sns.heatmap(data,vmin=30,vmax=70)

# displaying the plotted heatmap
plt.show()
```

```
[[40 71 43 56 58 77 89 26 46 38]
 [67 94 51 32 59 51 14 21 59 29]
 [72 15 28 81 28 29 65 22 50 39]
 [ 5  9 78 96 13 69 26 73 35 74]
 [49 14 99  4 55 56 15 66 80 42]
 [45 75 66 14 95 84  5  7 94 76]
 [90 34 72 39 39 99  2 68 91 57]
 [21 68 11 15 37 86 95 47 88 64]
 [48 34 51 81 29 36 75 89 77 80]
 [46 14 17 34 72 33 29 63  3 97]]
```



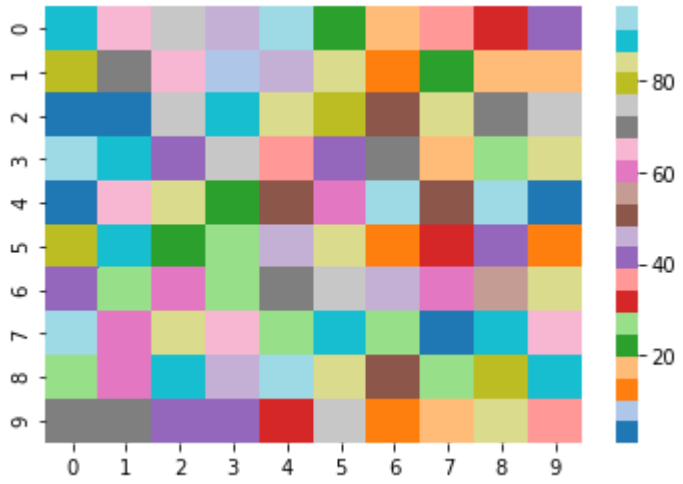
In [119…

```
###Choosing the colormap
```

In this, we will be looking at the cmap parameter. Matplotlib provides us with multiple colormaps, you can look at all of them here https://matplotlib.org/stable/tutorials/colors/colormaps.html. In our example, we'll be using tab20.

In [7]:

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

# generating 2-D 10x10 matrix of random numbers frpm 1 to 100
data = np.random.randint(low=1,high=100,size=(10, 10))

# plotting the heatmap
hm = sns.heatmap(data,cmap="tab20")

# displaying the plotted heatmap
plt.show()
```



In [121…

```
###Displaying the cell values
```

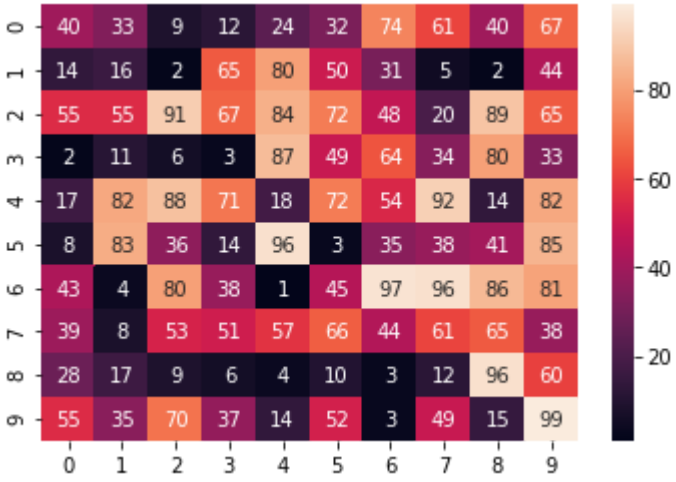
If we want to display the value of the cells, then we pass the parameter annot as True.

In [12]:

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

```
data = np.random.randint(low=1,high=100,size=(10, 10))
print(data)
# plotting the heatmap
hm = sns.heatmap(data,annot=True)

# displaying the plotted heatmap
plt.show()
```

```
[[40 33  9 12 24 32 74 61 40 67]
 [14 16  2 65 80 50 31  5  2 44]
 [55 55 91 67 84 72 48 20 89 65]
 [ 2 11  6  3 87 49 64 34 80 33]
 [17 82 88 71 18 72 54 92 14 82]
 [ 8 83 36 14 96  3 35 38 41 85]
 [43  4 80 38  1 45 97 96 86 81]
 [39  8 53 51 57 66 44 61 65 38]
 [28 17  9  6  4 10  3 12 96 60]
 [55 35 70 37 14 52  3 49 15 99]]
```



Customizing the separating line

We can change the thickness and the color of the lines separating the cells using the linewidths and linecolor parameters respectively.

Hiding the colorbar

We can disable the colorbar by setting the cbar parameter to False.

Removing the labels

We can disable the x-label and the y-label by passing False in the xticklabels and yticklabels parameters respectively.

In [13]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

data =np.random.randint(low='1',high='100',size=(10,10))
print(data)

hm=sns.heatmap(data,vmin=10,vmax=70,cmap='Pastel1',annot=True,linewidths=2,linecolor='b
```

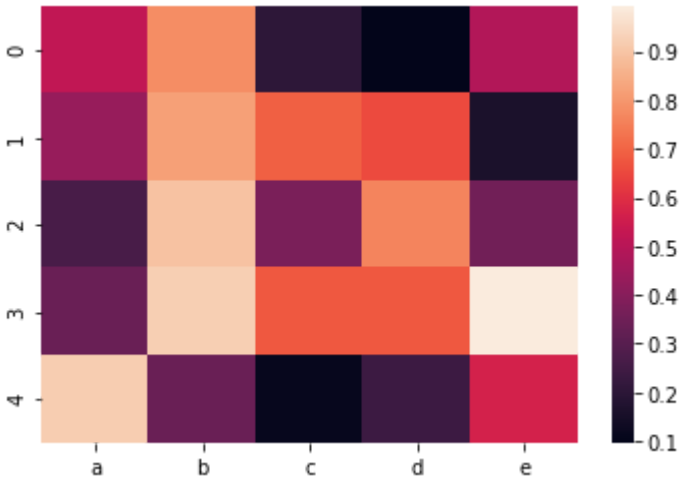
```
                cbar=False,xticklabels=False,yticklabels=False)
plt.show()
```

```
[[ 3 87 71 92 81 34 76 70 86 80]
 [46 41 52  9 36 31 82 51 64 26]
 [64 17 34 30 66 71 73 95 84  7]
 [21 12 25 29 61 82 38 26 30 64]
 [92 82 97 20 11 69 37 33 34 21]
 [49 17 26 63 76 82 72 72 56 96]
 [ 1 85 60 21 29 35 80 91 44 31]
 [12 84 92 49 46 68 89 19 71 30]
 [50 19 55 34 38 66 94 81 49 75]
 [21 56 94 29 80 53 44 59 13 63]]
```

3	87	71	92	81	34	76	70	86	80
46	41	52	9	36	31	82	51	64	26
64	17	34	30	66	71	73	95	84	7
21	12	25	29	61	82	38	26	30	64
92	82	97	20	11	69	37	33	34	21
49	17	26	63	76	82	72	72	56	96
1	85	60	21	29	35	80	91	44	31
12	84	92	49	46	68	89	19	71	30
50	19	55	34	38	66	94	81	49	75
21	56	94	29	80	53	44	59	13	63

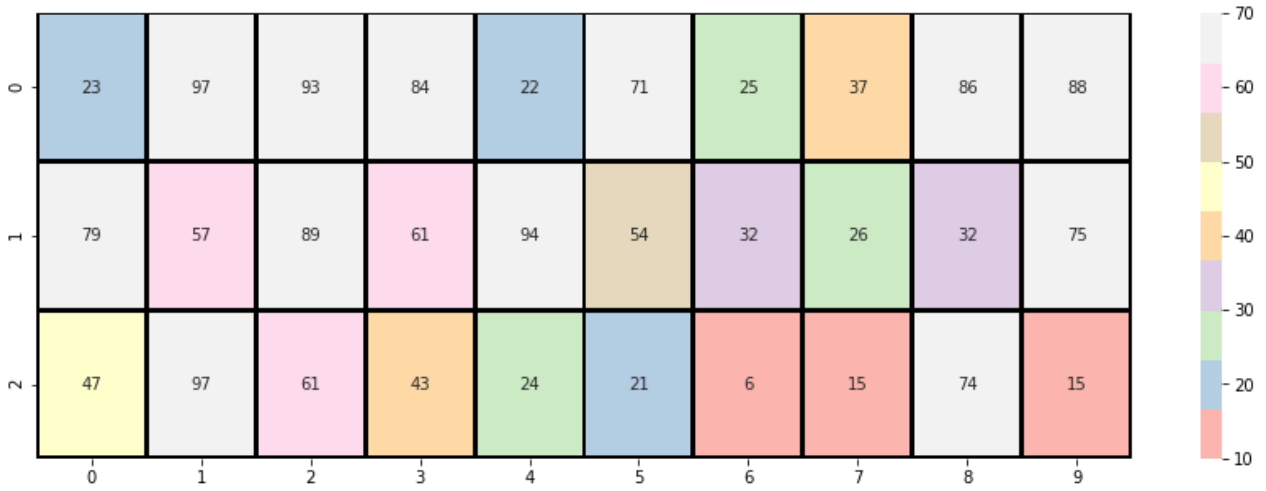
In [14]:
```
import seaborn as sns
import pandas as pd
import numpy as np

# Create a dataset
df = pd.DataFrame(np.random.random((5,5)), columns=["a","b","c","d","e"])
print(df)

p1 = sns.heatmap(df)
```

```
          a         b         c         d         e
0  0.522722  0.778470  0.205366  0.097253  0.490032
1  0.435878  0.817716  0.693385  0.653779  0.161540
2  0.268337  0.896706  0.374350  0.761522  0.357460
3  0.338911  0.921946  0.677166  0.679242  0.991134
4  0.919345  0.338503  0.113200  0.239097  0.565430
```



In [15]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(15,5))
data = np.random.randint(low='1',high='100',size=(3,10))
print(data)

hm=sns.heatmap(data=data,vmin=10,vmax=70,cmap='Pastel1',annot=True,linewidths=2,linecol
plt.show()
```

```
[[23 97 93 84 22 71 25 37 86 88]
 [79 57 89 61 94 54 32 26 32 75]
 [47 97 61 43 24 21  6 15 74 15]]
```



In [ ]:
```
####################3 Folium Map ###################################
```

In [16]:
```
#Folium Map
!pip install folium
```

```
Requirement already satisfied: folium in c:\programdata\anaconda3\lib\site-packages (0.1
4.0)
Requirement already satisfied: branca>=0.6.0 in c:\programdata\anaconda3\lib\site-packag
es (from folium) (0.6.0)
Requirement already satisfied: numpy in c:\users\tejas\appdata\roaming\python\python38\s
ite-packages (from folium) (1.24.3)
Requirement already satisfied: jinja2>=2.9 in c:\programdata\anaconda3\lib\site-packages
(from folium) (2.11.2)
Requirement already satisfied: requests in c:\programdata\anaconda3\lib\site-packages (f
rom folium) (2.24.0)
Requirement already satisfied: MarkupSafe>=0.23 in c:\programdata\anaconda3\lib\site-pac
kages (from jinja2>=2.9->folium) (1.1.1)
Requirement already satisfied: certifi>=2017.4.17 in c:\programdata\anaconda3\lib\site-p
ackages (from requests->folium) (2020.6.20)
Requirement already satisfied: chardet<4,>=3.0.2 in c:\programdata\anaconda3\lib\site-pa
ckages (from requests->folium) (3.0.4)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in c:\programdata
\anaconda3\lib\site-packages (from requests->folium) (1.25.11)
Requirement already satisfied: idna<3,>=2.5 in c:\programdata\anaconda3\lib\site-package
s (from requests->folium) (2.10)
```

Generating the world map is straightforward in Folium. You simply create a Folium Map object, and then you display it. What is attractive about Folium maps is that they are interactive, so you can zoom into any region of interest despite the initial zoom level.

In [19]:
```
import folium
# define the world map
world_map = folium.Map()

# display world map
world_map
```

Out[19]: Make this Notebook Trusted to load map: File -> Trust Notebook

You can customize this default definition of the world map by specifying the centre of your map, and the initial zoom level.

All locations on a map are defined by their respective Latitude and Longitude values. So you can create a map and pass in a center of Latitude and Longitude values of [0, 0].

For a defined center, you can also define the initial zoom level into that location when the map is rendered. The higher the zoom level the more the map is zoomed into the center.

Let's create a map centered around Canada and play with the zoom level to see how it affects the rendered map.

In [18]:
```
# define the world map centered around Canada with a low zoom level
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4)

# display world map
world_map
```

Out[18]: Make this Notebook Trusted to load map: File -> Trust Notebook

In [20]:
```
# define the world map centered around Canada with a higher zoom level
world_map = folium.Map(location=[56.130, -106.35], zoom_start=8)

# display world map
world_map
```

Out[20]: Make this Notebook Trusted to load map: File -> Trust Notebook

As you can see, the higher the zoom level the more the map is zoomed into the given center.

Another cool feature of Folium is that you can generate different map styles.

A. Stamen Toner Maps

These are high-contrast B+W (black and white) maps. They are perfect for data mashups and exploring river meanders and coastal zones.

Let's create a Stamen Toner map of canada with a zoom level of 4.

In [22]:
```
# create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Toner')

# display map
world_map
```

Out[22]: Make this Notebook Trusted to load map: File -> Trust Notebook

Feel free to zoom in and out to see how this style compares to the default one.

B. Stamen Terrain Maps

These are maps that feature hill shading and natural vegetation colors. They showcase advanced labeling and linework generalization of dual-carriageway roads.

Let's create a Stamen Terrain map of Canada with zoom level 4.

In [23]:
```
# create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Terrain'

# display map
world_map
```

Out[23]: Make this Notebook Trusted to load map: File -> Trust Notebook

Feel free to zoom in and out to see how this style compares to Stamen Toner, and the default style.

Zoom in and notice how the borders start showing as you zoom in, and the displayed country names are in English.

Other types of styles of maps can be found in the list below:

The following tilesets are built-in to Folium. Pass any of the following to the "tiles" keyword:

1. "OpenStreetMap"
2. "Mapbox Bright" (Limited levels of zoom for free tiles)
3. "Mapbox Control Room" (Limited levels of zoom for free tiles)
4. "Stamen" (Terrain, Toner, and Watercolor)

5. "Cloudmade" (Must pass API key)

6. "Mapbox" (Must pass API key)

7. "CartoDB" (positron and dark_matter)

Cloudmade and Mapbox cannot be used without an API key to Folium

##Map with Markers

Let's download and import the data on police department incidents using pandas read_csv() method.

Download the dataset and read it into a pandas dataframe:

In [27]:
```
import pandas as pd
df_incidents = pd.read_csv('Police_Department_Incidents_-_Previous_Year__2016_.csv')
```

In [28]:
```
df_incidents.head()
```

Out[28]:

	IncidntNum	Category	Descript	DayOfWeek	Date	Time	PdDistrict	Resolution	Add
0	120058272	WEAPON LAWS	POSS OF PROHIBITED WEAPON	Friday	01/29/2016 12:00:00 AM	11:00	SOUTHERN	ARREST, BOOKED	800 B BRY
1	120058272	WEAPON LAWS	FIREARM, LOADED, IN VEHICLE, POSSESSION OR USE	Friday	01/29/2016 12:00:00 AM	11:00	SOUTHERN	ARREST, BOOKED	800 B BRY
2	141059263	WARRANTS	WARRANT ARREST	Monday	04/25/2016 12:00:00 AM	14:59	BAYVIEW	ARREST, BOOKED	KEIT SHA
3	160013662	NON-CRIMINAL	LOST PROPERTY	Tuesday	01/05/2016 12:00:00 AM	23:50	TENDERLOIN	NONE	JONE OFAR
4	160002740	NON-CRIMINAL	LOST PROPERTY	Friday	01/01/2016 12:00:00 AM	00:30	MISSION	NONE	16TH MISS

So each row consists of 13 features:

1. **IncidntNum**: Incident Number
2. **Category**: Category of crime or incident
3. **Descript**: Description of the crime or incident
4. **DayOfWeek**: The day of week on which the incident occurred
5. **Date**: The Date on which the incident occurred
6. **Time**: The time of day on which the incident occurred
7. **PdDistrict**: The police department district
8. **Resolution**: The resolution of the crime in terms whether the perpetrator was arrested or not
9. **Address**: The closest address to where the incident took place
10. **X**: The longitude value of the crime location
11. **Y**: The latitude value of the crime location
12. **Location**: A tuple of the latitude and the longitude values
13. **PdId**: The police department ID

Let's find out how many entries there are in our dataset.

In [29]:
```
df_incidents.shape
```

Out[29]:
```
(150500, 13)
```

So the dataframe consists of 150,500 crimes, which took place in the year 2016. In order to reduce computational cost, let's just work with the first 100 incidents in this dataset.

In [30]:
```
# get the first 100 crimes in the df_incidents dataframe

df= df_incidents.iloc[0:100]
# print(df_incidents)
```

In [31]:
```
df.shape
```

Out[31]:
```
(100, 13)
```

Now that we reduced the data a little, let's visualize where these crimes took place in the city of San Francisco. We will use the default style, and we will initialize the zoom level to 12.

In [32]:
```
# San Francisco latitude and longitude values
latitude = 37.77
longitude = -122.42
```

In [33]:
```
import folium

# create map and display it
sanfran_map = folium.Map(location=[latitude, longitude], zoom_start=12)

# display the map of San Francisco
sanfran_map

# sanfran_map.save('sanfran.html') # using this command we can save map in html
```

Out[33]: Make this Notebook Trusted to load map: File -> Trust Notebook

Now let's superimpose the locations of the crimes onto the map. The way to do that in Folium is to create a feature group with its own features and style and then add it to the sanfran_map.

In [34]:
```
# instantiate a feature group for the incidents in the dataframe
incidents = folium.map.FeatureGroup()

# loop through the 100 crimes and add each to the incidents feature group
for lat, lng, in zip(df.Y, df.X):
    incidents.add_child(
        folium.CircleMarker([lat, lng],radius=5, # define how big you want the circle m
            color='yellow',
            fill=True,
            fill_color='black',
            fill_opacity=0.6)
    )

# add incidents to map
sanfran_map.add_child(incidents)
```

Out[34]: Make this Notebook Trusted to load map: File -> Trust Notebook

You can also add some pop-up text that would get displayed when you hover over a marker. Let's make each marker display the category of the crime when hovered over.

In [35]:
```
# instantiate a feature group for the incidents in the dataframe
incidents = folium.map.FeatureGroup()

# loop through the 100 crimes and add each to the incidents feature group
for lat,lng,labels in zip(df.Y, df.X,df.Category):
    incidents.add_child(
        folium.CircleMarker([lat, lng],radius=5,
                            color='yellow',
                            fill=True,
                            fill_color='blue',
                            fill_opacity=0.6),
```

```
        folium.Marker([lat,lng],popup=labels).add_to(sanfran_map)
    )
sanfran_map.add_child(incidents)
```

Out[35]: Make this Notebook Trusted to load map: File -> Trust Notebook

In [36]:
```
# Understanding of Zip Function Example
a = ("John", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica")

x = zip(a, b)
for i,j in x:
    print(i,j)
```

```
John Jenny
Charles Christy
Mike Monica
```

Choropleth Maps

In [37]:
```
state_unemp = pd.read_csv("US_Unemployment_Oct2012.csv")
state_geo = "us-states.json"
```

In [38]:
```
usa_state = folium.Map(location=[48, -102], zoom_start=3)
folium.Choropleth(
    geo_data = state_geo,                  #json
    name ='choropleth',
    data = state_unemp,
    columns = ['State', 'Unemployment'], #columns to work on
    key_on ='feature.id',
    fill_color ='YlGnBu',     #I passed colors Yellow,Green,Blue
    fill_opacity = 0.7,
    line_opacity = 0.2,
   legend_name = "Unemployment scale"
    ).add_to(usa_state)
usa_state
```

Out[38]: Make this Notebook Trusted to load map: File -> Trust Notebook

In [1]:
```
################## Visualizing Graphs using NetworkX Library ##########################
```

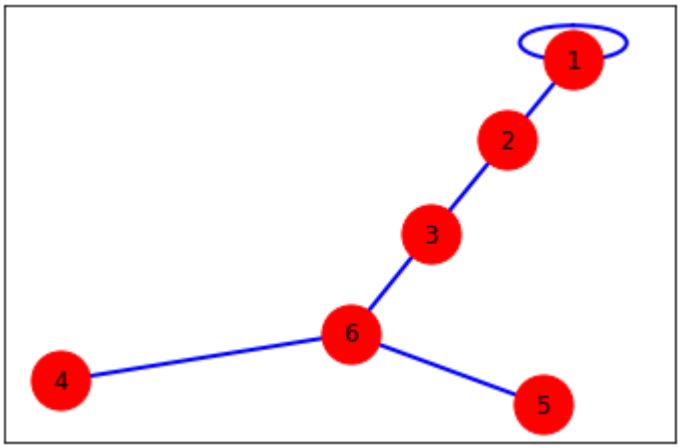
#Visualizing Graphs with NetworkX

Graphs in data structures are non-linear data structures made up of a finite number of nodes or vertices and the edges that connect them. Graphs in data structures are used to address real-world problems in which it represents the problem area as a network like telephone networks, circuit networks, and social networks.

Undirected Graphs

An undirected graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction.

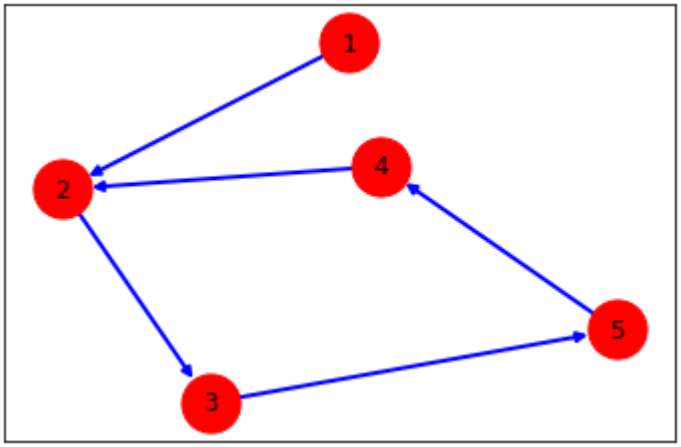
In [56]:
```
import networkx as nx
import matplotlib.pyplot as plt

# Unidirected Graph so nx.Graph()
G = nx.Graph()

#Different methods to add nodes
G.add_node(1)
G.add_nodes_from([2, 3])
G.add_nodes_from(range(4, 7))

#Different Methods to add edges
G.add_edge(1, 2)
G.add_edge(1, 1)
G.add_edges_from([(2,3), (3,6), (4,6), (5,6)])

nx.draw_networkx(G, node_size=850, node_color='red', width=2, edge_color='blue')
plt.show()
```



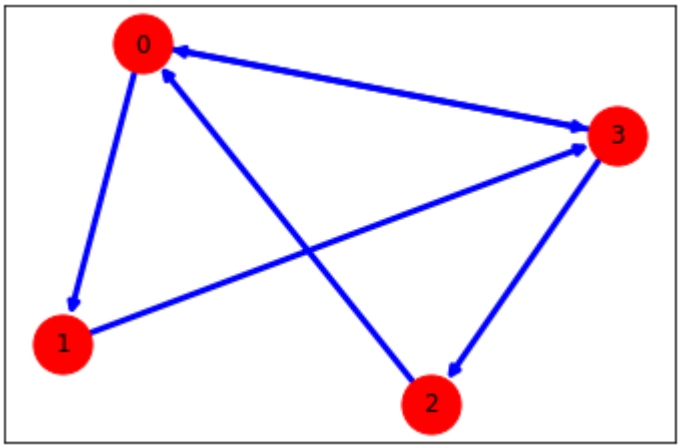
Directed Graphs

A directed graph is a data structure that stores data in vertices or nodes and these nodes or vertices are connected and also directed by edges (one vertex is directed towards another vertex through an edge). Directed graph is also called a digraph or directed network.

In [3]:
```
import networkx as nx
import matplotlib.pyplot as plt

# Directed Graph so nx.DiGraph()
g = nx.DiGraph()

g.add_nodes_from([1,2,3,4,5])

g.add_edge(1,2)
g.add_edge(4,2)
g.add_edge(3,5)
g.add_edge(2,3)
g.add_edge(5,4)

nx.draw_networkx(g, node_size=850, node_color='red', width=2, edge_color='blue')

# nx.draw(g,with_labels=True, node_size=1000, node_color='red', width=5, edge_color='bl
# plt.draw()
plt.show()
```



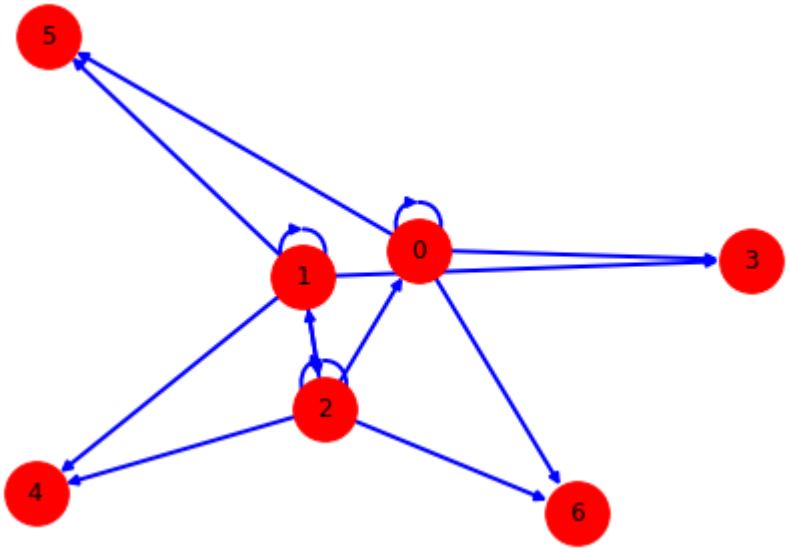
In [58]:
```
import networkx as nx
import matplotlib.pyplot as plt

G=nx.DiGraph()

G.add_nodes_from(range(4))
l=[[0,1,0,1],[0,0,0,1],[1,0,0,0],[1,0,1,0]]

for i in range(4):
    for j in range(4):
        if l[i][j]==1:
            G.add_edge(i,j)

nx.draw_networkx(G,node_size=850,node_color='red',edge_color='blue',width=3)
plt.show()
```



Creating a Directed graph from an Adjacency Matrix

Directed Graph from Adjacency Matrix with 3 nodes

In [1]:
```
import networkx as nx
import matplotlib.pyplot as plt

g = nx.DiGraph()
L = [[1,0,0,1,0,1,1],[0,1,1,1,1,1,0],[1,1,1,0,1,0,1]]

g.add_nodes_from([0,1,2])

for i in range(0,3):
    for j in range(0,7):
        if L[i][j] == 1:
            g.add_edge(i,j)

nx.draw(g, with_labels=True, node_size=1000, node_color='red', width=2, edge_color='blu
plt.show()
```



Directed Graph from Adjacency Matrix with 7 nodes

In [62]:

```
import networkx as nx
import matplotlib.pyplot as plt

g = nx.DiGraph()
L = [[1,0,0,1,0,1,1],[0,1,1,1,1,1,0],[1,1,1,0,1,0,1],[1,0,0,0,1,0,1],[1,1,1,0,0,0,1],[1

g.add_nodes_from([0,1,2,3,4,5,6])

for i in range(0,7):
    for j in range(0,7):
        if L[i][j] == 1:
            g.add_edge(i,j)

nx.draw(g, with_labels=True, node_size=1000, node_color='red', width=2, edge_color='blu
plt.show()
```