Exploratory Data Analysis and Visualization

Neba Nfonsang

University of Denver

In [1]:
```
# import necessary packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib
import statistics
import pydataset
import seaborn as sns
from scipy import stats


import warnings
warnings.filterwarnings("ignore")
```

Exploratory Data Analysis

In exploratory data analysis (EDA) and visualization, we will cover descriptive statistics and data visualization.

- Specifically, we will look at how to use different statistical packages to measure central tendency, variability and the shapes of distributions.
- We will examine how boxplots and histograms can be used to explore the normality of your data.
- Other visual plots such as scatter plots, line plot, bar charts, pie charts, normal distribution curves will be covered.

Descriptive Statistics

Descriptive statistics involves summarizing our data using numerical measures such as:

- Central tendency (mean, median and mode),
- Variability (variance, standard deviation, range, minimum value, and maximum value, etc) and
- The shape of the distribution of data (kurtosis and skewness).

Let's Use the Tips Dataset

In [2]:
```
# Read the tips dataset from the internet
url = r"https://raw.githubusercontent.com/mwaskom/seaborn-data/master/tips.csv"
data = pd.read_csv(url)
data.head()
```

Out[2]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

In [3]:
```python
# based on APA, let's change the column "sex" to "gender"
data = data.rename(columns={"sex": "gender"})
data.head()
```

Out[3]:

	total_bill	tip	gender	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Use pandas Methods to Generate Descriptive Statistics

Note that when we apply descriptive statistics methods to the entire data set, descriptive statistics will be generated only for the columns that have quantitative or numerical values.

Generate Descriptive Statistics with .describe()

In [4]:
```
data.describe()
```

Out[4]:

	total_bill	tip	size
count	244.000000	244.000000	244.000000
mean	19.785943	2.998279	2.569672
std	8.902412	1.383638	0.951100
min	3.070000	1.000000	1.000000
25%	13.347500	2.000000	2.000000
50%	17.795000	2.900000	2.000000
75%	24.127500	3.562500	3.000000
max	50.810000	10.000000	6.000000

Generate Descriptive Statistics Separately

In [5]:
```
# compute the mean
data.mean()
```

Out[5]:
```
total_bill    19.785943
tip            2.998279
size           2.569672
dtype: float64
```

In [6]:
```
# compute the median
data.median()
```

Out[6]:
```
total_bill    17.795
tip            2.900
size           2.000
dtype: float64
```

In [7]:
```
# compute minimum value
data[["total_bill", "tip", "size"]].min()
```

Out[7]:
```
total_bill    3.07
tip           1.00
size          1.00
dtype: float64
```

In [8]:
```
# compute maximum value
data[["total_bill", "tip", "size"]].max()
```

Out[8]:
```
total_bill    50.81
tip           10.00
size           6.00
dtype: float64
```

In [9]:
```
# compute the sum
data[["total_bill", "tip", "size"]].sum()
```

Out[9]:
```
total_bill    4827.77
tip            731.58
size           627.00
dtype: float64
```

In [10]:
```
# compute variance
data.var()
```

Out[10]:
```
total_bill    79.252939
tip            1.914455
size           0.904591
dtype: float64
```

In [11]:
```
# compute standard deviation
data.std()
```

Out[11]:
```
total_bill    8.902412
tip           1.383638
size          0.951100
dtype: float64
```

In [12]:
```
# compute skewness
data.skew()
```

Out[12]:
```
total_bill    1.133213
tip           1.465451
size          1.447882
dtype: float64
```

In [13]:
```
# compute kurtosis
# data.kurtosis() also works

data.kurt()
```

Out[13]:
```
total_bill    1.218484
tip           3.648376
size          1.731700
dtype: float64
```

Generate Multiple Descriptive Statistics

- We can use the .apply() and .agg() methods with multiple statistics functions to generate several descriptive statistics at once.
- We specifically need to select the columns with numerical data to compute the descriptive statistics.
- We could as well define a function that return the descriptive statistics.

In [14]:

```
# use the .apply() function
data[['total_bill', 'tip', 'size']].apply(["count", "mean", "median",
                                           "min", "max", "var", "std",
                                           "skew", "kurt"])
```

Out[14]:

	total_bill	tip	size
count	244.000000	244.000000	244.000000
mean	19.785943	2.998279	2.569672
median	17.795000	2.900000	2.000000
min	3.070000	1.000000	1.000000
max	50.810000	10.000000	6.000000
var	79.252939	1.914455	0.904591
std	8.902412	1.383638	0.951100
skew	1.133213	1.465451	1.447882
kurt	1.218484	3.648376	1.731700

In [15]:

```
# use the .agg() function
data[['total_bill', 'tip', 'size']].apply(["count", "mean", "median",
                                            "min", "max", "var", "std",
                                            "skew", "kurt"])
```

Out[15]:

	total_bill	tip	size
count	244.000000	244.000000	244.000000
mean	19.785943	2.998279	2.569672
median	17.795000	2.900000	2.000000
min	3.070000	1.000000	1.000000
max	50.810000	10.000000	6.000000
var	79.252939	1.914455	0.904591
std	8.902412	1.383638	0.951100
skew	1.133213	1.465451	1.447882
kurt	1.218484	3.648376	1.731700

In [16]:

```
# compute descriptive statistics by category
# use agg() or apply() on a GroupBy object

data.groupby("gender")['total_bill'].agg(["count", "mean", "median",
                                           "min", "max", "var", "std", "skew"])
```

Out[16]:

	count	mean	median	min	max	var	std	skew
gender								
Female	87	18.056897	16.40	3.07	44.30	64.147429	8.009209	1.134052
Male	157	20.744076	18.35	7.25	50.81	85.497185	9.246469	1.103269

Generate Descriptive Statistics Using Numpy

In [17]:
```
# compute mean
np.mean(data)
```

Out[17]:
```
total_bill    19.785943
tip            2.998279
size           2.569672
dtype: float64
```

In [18]:
```
# compute minimum
np.min(data[['total_bill', 'tip', 'size']])
```

Out[18]:
```
total_bill    3.07
tip           1.00
size          1.00
dtype: float64
```

In [19]:
```
# compute maximum
np.max(data[['total_bill', 'tip', 'size']])
```

Out[19]:
```
total_bill    50.81
tip           10.00
size           6.00
dtype: float64
```

In [20]:
```
# compute standard deviation
np.std(data[['total_bill', 'tip', 'size']])
```

Out[20]:
```
total_bill    8.884151
tip           1.380800
size          0.949149
dtype: float64
```

In [21]:
```
# compute variance
np.var(data[['total_bill', 'tip', 'size']])
```

Out[21]:
```
total_bill    78.928131
tip            1.906609
size           0.900883
dtype: float64
```

Generate Descriptive Statistics Using the Statistics Package

In [22]:
```
# view the features inside the package
print(dir(statistics))
```

```
['Decimal', 'Fraction', 'StatisticsError', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loade
r__', '__name__', '__package__', '__spec__', '_coerce', '_convert', '_counts', '_exact_ratio', '_fail_neg', '_find_
lteq', '_find_rteq', '_isfinite', '_ss', '_sum', 'bisect_left', 'bisect_right', 'chain', 'collections', 'decimal',
'groupby', 'harmonic_mean', 'math', 'mean', 'median', 'median_grouped', 'median_high', 'median_low', 'mode', 'numbe
rs', 'pstdev', 'pvariance', 'stdev', 'variance']
```

The descriptive statistics methods that we will use in this section could take series or lists as the arguments.

In [23]:
```
# compute mean
statistics.mean(data.tip)
```

Out[23]:
```
2.9982786885245902
```

In [24]:
```
# compute median
statistics.median(data.total_bill)
```

Out[24]:
```
17.795
```

In [25]:
```python
# compute mode
statistics.mode(data["size"])
```

Out[25]: 2

In [26]:
```python
# compute population standard deviation
statistics.pstdev(data["tip"])
```

Out[26]: 1.3807999538298952

In [27]:
```python
# compute sample standard deviation
statistics.stdev(data["tip"])
```

Out[27]: 1.383638189001182

In [28]:
```python
# compute population variance
statistics.pvariance(data.tip)
```

Out[28]: 1.9066085124966408

In [29]:
```python
# compute sample variance
statistics.variance(data.tip)
```

Out[29]: 1.9144546380624705

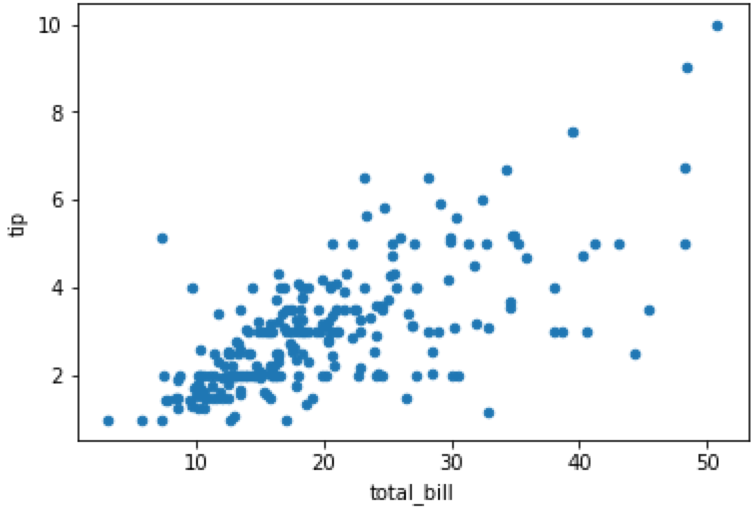
Visualiuzation

Data Visualization using pandas Methods

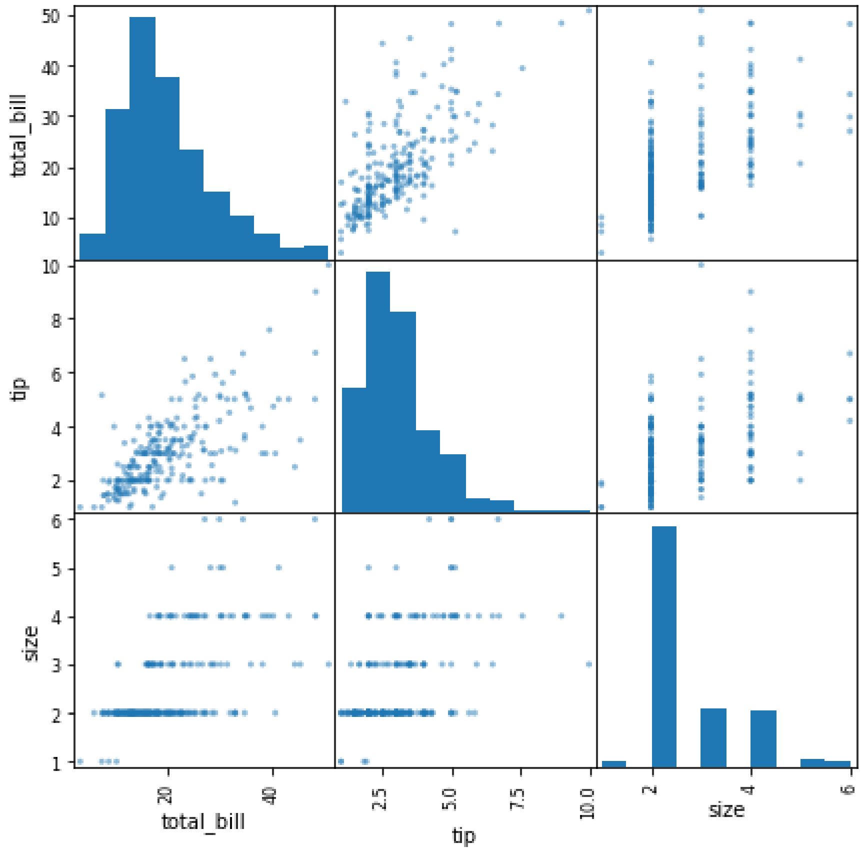
- Let's take a look at how to plot scatter plot, histogram, boxplot, bar chart and pie chart.
- The DataFrame's .plot() method will be used and the column names of interest would be used as arguments

In [30]:
```
# create a scatter plot
data.plot.scatter("total_bill", "tip")
```

Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0x28d6ee5deb8>



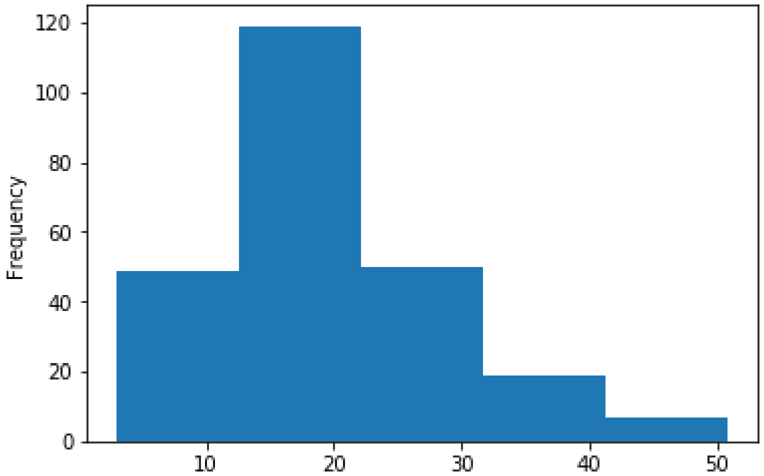
In [31]:
```python
# creat a scatter matrix
pd.plotting.scatter_matrix(data, figsize=(7, 7))
plt.show()
```



In [32]:

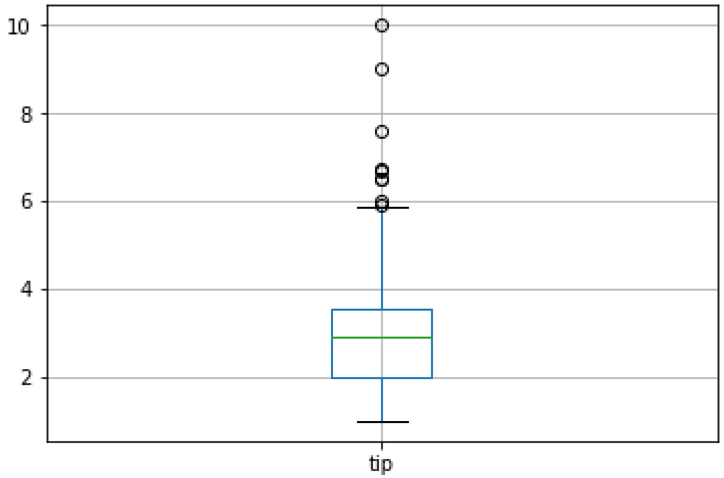
```python
# create a histogram of the total bill
data["total_bill"].plot.hist(bins=5)
```

Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x28d711d24e0>



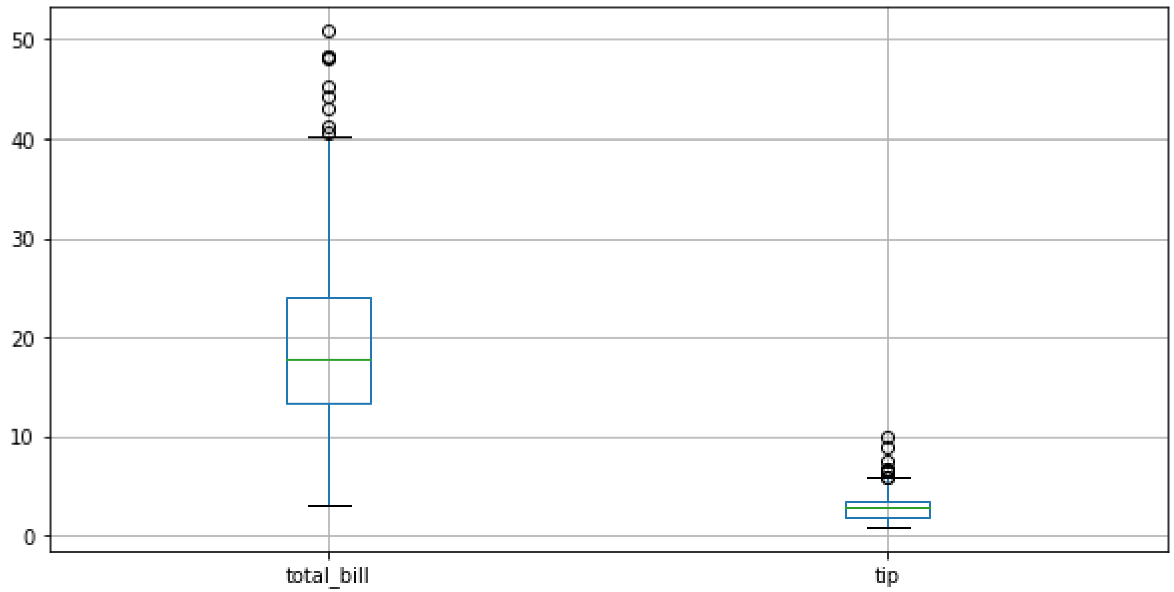
In [33]:
```
# create a boxplot
data.boxplot("tip")
```

Out[33]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x28d6ee47d68>
```



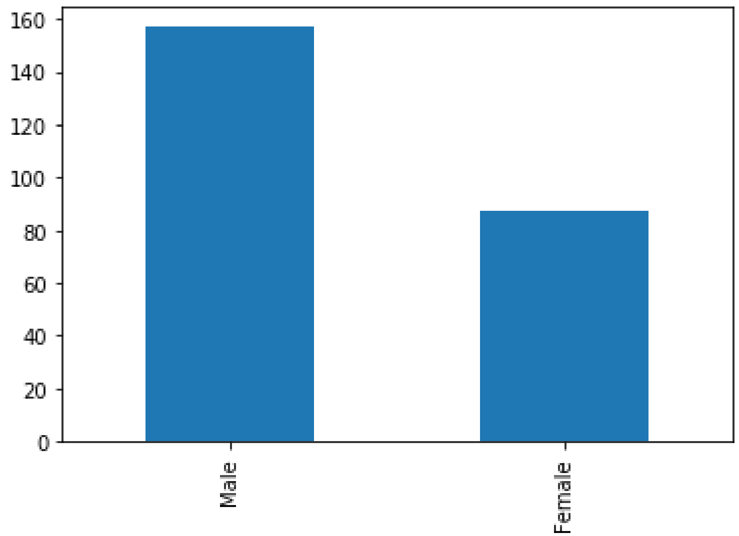
In [34]:
```
# plot multiple boxplots
data.boxplot(["total_bill", "tip"], figsize=(10, 5))
```

Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0x28d712a36d8>



In [35]:
```
# create a bar chart
gender_counts = data["gender"].value_counts()
gender_counts.plot.bar()
```

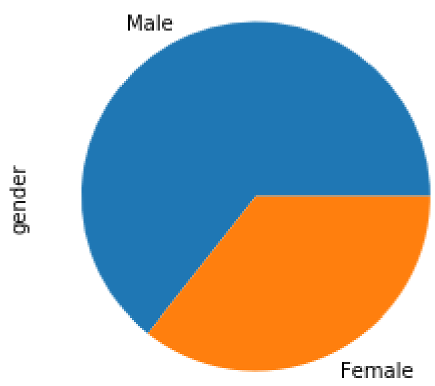
Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x28d7130ab38>



In [36]:
```
# create a bar chart
print(gender_counts)
gender_counts.plot.pie()
```

```
Male      157
Female     87
Name: gender, dtype: int64
```

Out[36]: `<matplotlib.axes._subplots.AxesSubplot at 0x28d711800f0>`

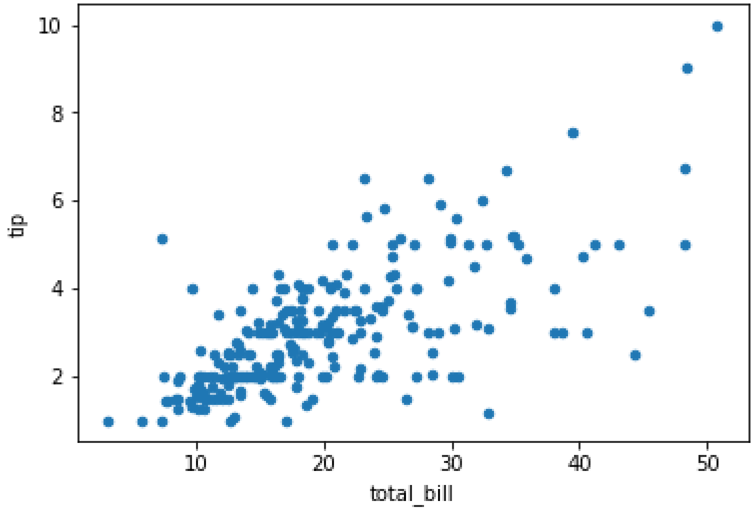


An Alternative Way of Plotting with pandas

- In the previous plots, we used .plot.scatter(), .plot.hist(), .plot.bar(), .plot.pie(),
- Alternatively we could use .plot(kind="scatter"), .plot(kind="hist"), .plot(kind="bar), .plot(kind="pie")

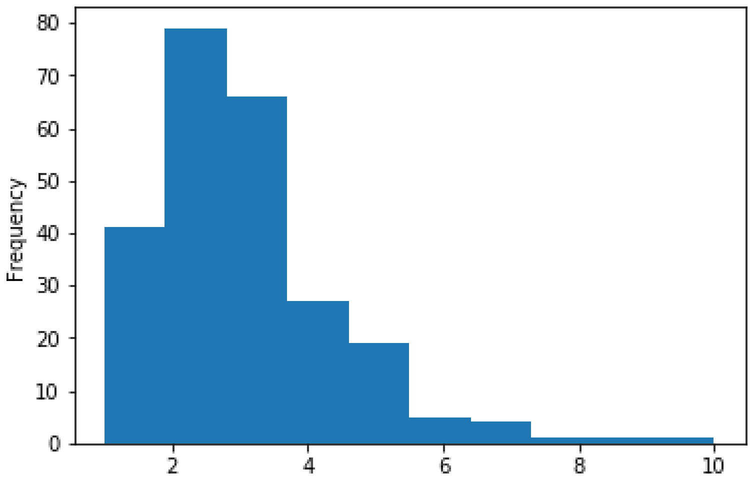
In [37]:
```
# create a scatter plot
data.plot("total_bill", "tip", kind="scatter")
```

Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x28d71190da0>



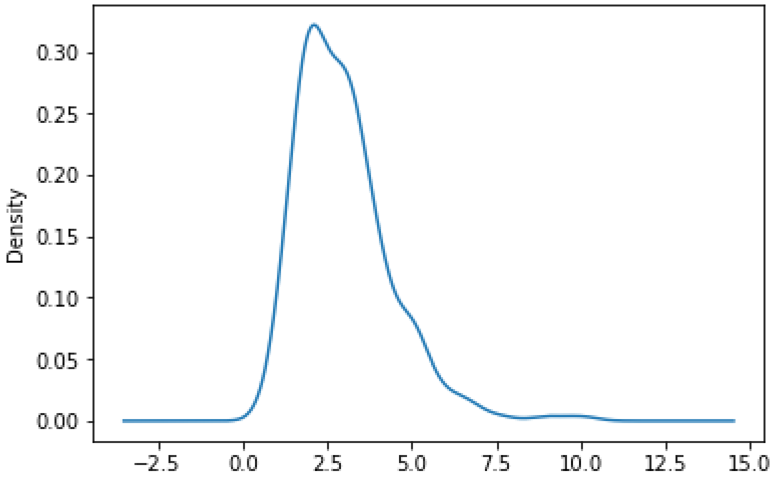
In [38]:
```
# create a histogram
data.tip.plot(kind="hist")
```

Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x28d71314860>



In [39]:
```
# plot the kernel density of the "tip" variable
data.tip.plot(kind="kde")
```

Out[39]: <matplotlib.axes._subplots.AxesSubplot at 0x28d710a2d30>



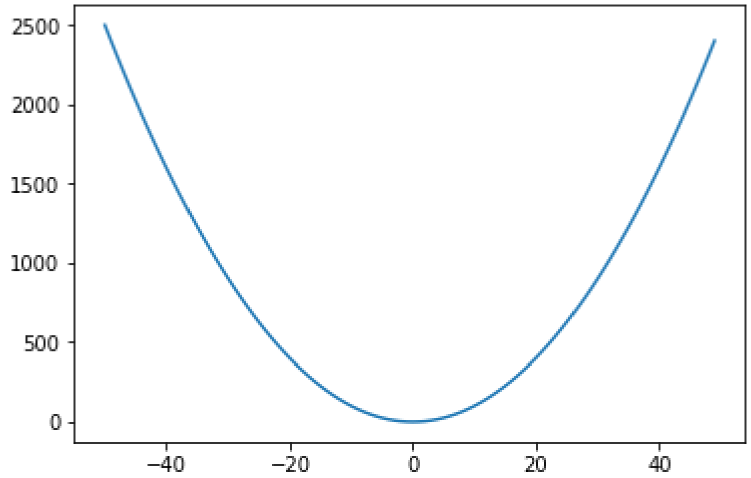
Data Visualization with matplotlib

- matplotlib is a Python 2D plotting library for producing publication quality figures in a variety of formats.
- matplotlib's plyplot module provides a MATLAB-like interface especially when combined with the IPython shell.

A Line Plot

- A line plot is a line connecting the (x, y) points in a coordinate plane

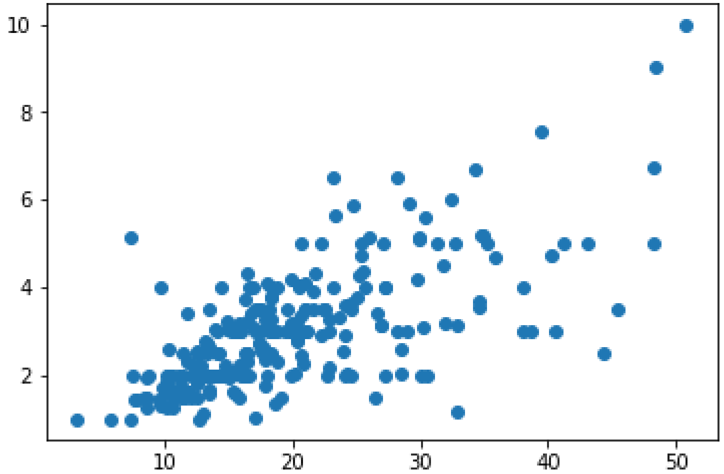
In [40]:
```
x = np.arange(-50, 50)
y = x**2

plt.plot(x, y)
plt.show()
```



A Scatter Plot

- We can use a scatter plot to visualize the relationship or correlation between two numerical variables.
- A scatter plot is usually required as a preliminary analysis for correlation analysis

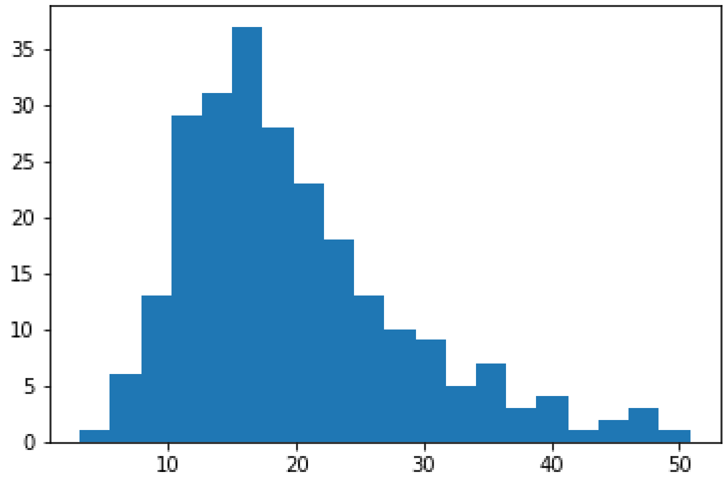
In [41]:
```
# a scatter plot of "total bill" and "tip"
plt.scatter(x=data["total_bill"], y=data["tip"])

# to display the plot
plt.show()
```



Histogram

- A histogram is the graphical display of the counts of various ranges of numerical data.
- The data is grouped into bins of equal lengths and the number of data points within each bin are displayed.
- Histograms are usually used to check the normality of data distribution.
- The data can be considered to be approximately noramally distributed if the histogram looks like a bell shape.
- It is advisable to use additional statistical measures such as skewness/kurtosis or statistical tests such as Shapiro-Wilks test to draw conclusion about the normality of the data.

In [42]:
```
# create a histogram
plt.hist(data.total_bill, bins=20)
plt.show()
```



Bar Chart

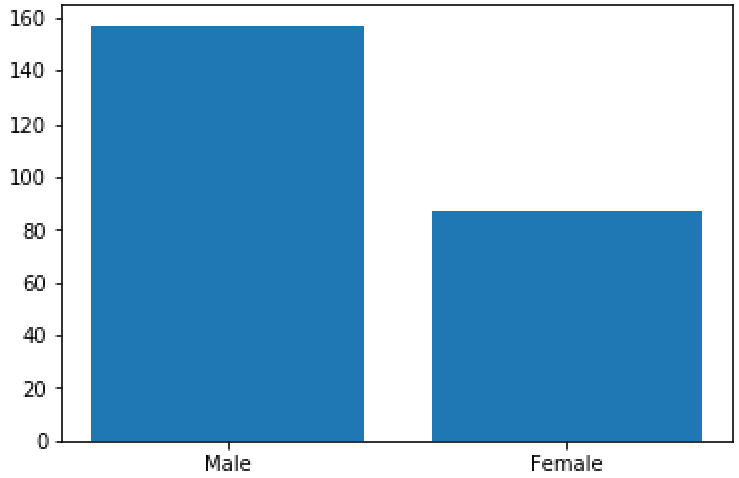
- A bar chart is a graphical display of the frequency or count of categorical data
- To plot a bar chart, we may first need to compute the frequencies of the categories for the variable of interest.

In [43]:
```
gender_counts = data["gender"].value_counts()
gender_counts
```

Out[43]:
```
Male      157
Female     87
Name: gender, dtype: int64
```

In [44]:

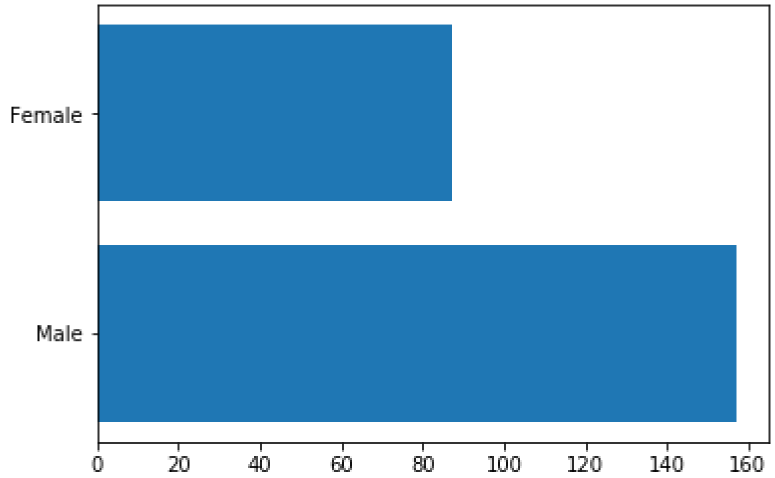
```
# extract the index labels and the counts
x = gender_counts.index
y = gender_counts.values

# plot the bar chart
plt.bar(x, y)
plt.show()
```



Horizontal Bar Chart

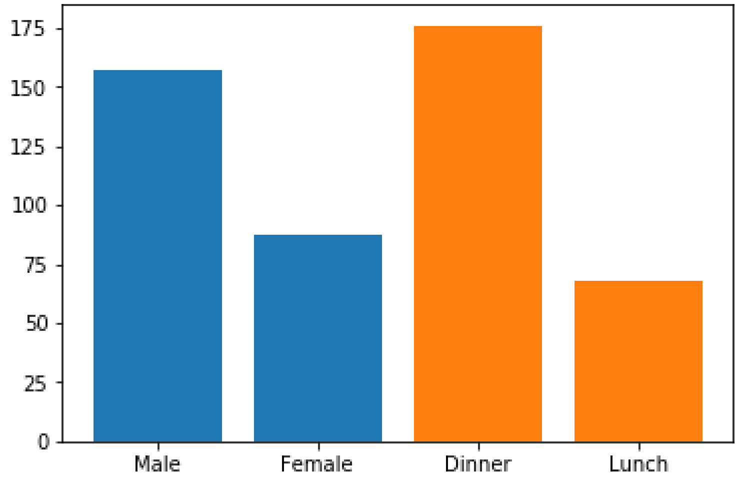
In [45]:

```
x = gender_counts.index
y = gender_counts.values

plt.barh(x, y)
plt.show()
```



Multiple Bar Charts on the same Figure

In [46]:
```
# generate counts
gender_counts = data["gender"].value_counts()
time_counts = data["time"].value_counts()

# generate plots
plt.bar(gender_counts.index, gender_counts.values)
plt.bar(time_counts.index, time_counts.values)
plt.show()
```



Multiple Bar Charts for Variables that Interact

In [47]:
```
ctab = pd.crosstab(data.gender, data.time)
ctab
```

Out[47]:

time	Dinner	Lunch
gender		
Female	52	35
Male	124	33

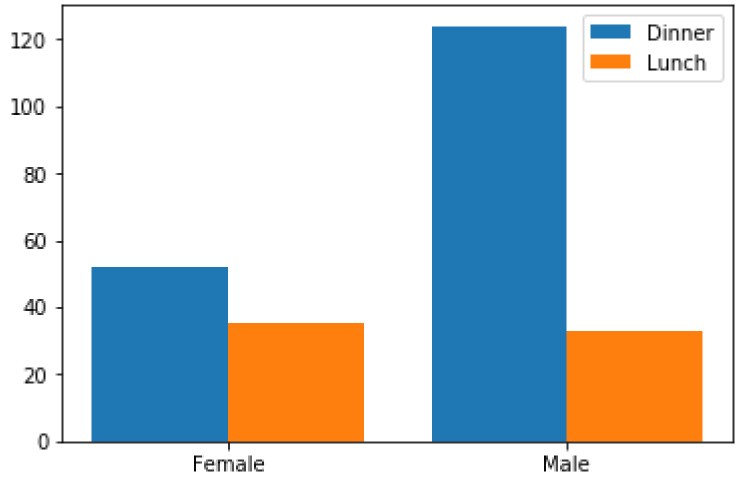
In [48]:
```
ctab.Dinner
```

Out[48]:
```
gender
Female     52
Male      124
Name: Dinner, dtype: int64
```

In [49]:
```
ctab.Lunch
```

Out[49]:
```
gender
Female    35
Male      33
Name: Lunch, dtype: int64
```

In [50]:
```
x = np.arange(len(ctab.Dinner))

plt.bar(x, ctab.Dinner.values, width=0.4)
plt.bar(x + 0.4, ctab.Lunch.values, width=0.4)

plt.xticks([.2, 1.2], ctab.Dinner.index)
plt.legend(ctab.columns)
plt.show()
```



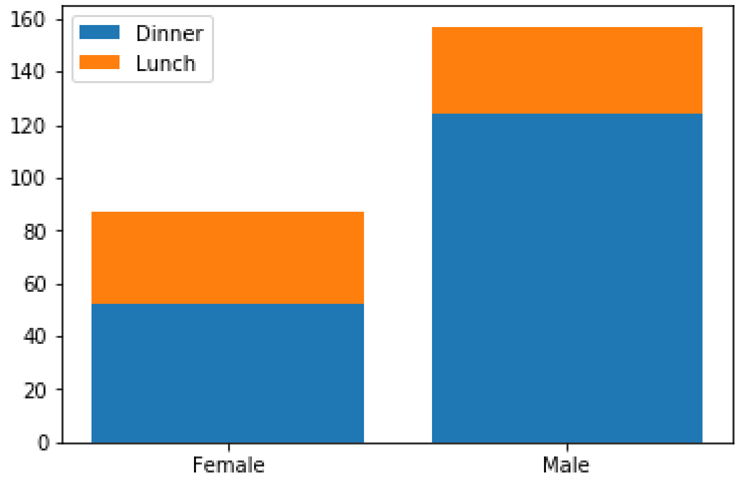
- This multiple bar chart is good for comparing counts of male or female who went for dinner or launch.

Stacked Bar Charts

- The height of each sub-bar represent the frequency of a category
- The height of the entire bar represent the total frequency of a category
- Stacked bar charts help us to compare sub-categories such as number of female who went for launch vs number of female who went for dinner

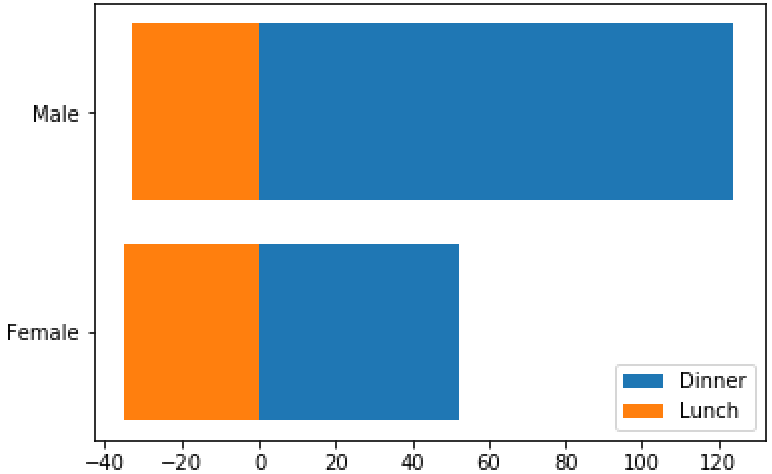
In [51]:

```
x = ctab.Dinner.index
# or x=ctab.Lunch.index
# or x = ["female", "male"]

plt.bar(x, ctab.Dinner.values)
plt.bar(x, ctab.Lunch.values, bottom=ctab.Dinner.values)

plt.xticks(ctab.Dinner.index)
plt.legend(ctab.columns)
plt.show()
```



Back to Back Bar Chart

In [52]:
```
x = ctab.Dinner.index
# or x=ctab.Lunch.index
# or x = ["female", "male"]

plt.barh(x, ctab.Dinner.values)
plt.barh(x, -ctab.Lunch.values)

plt.legend(ctab.columns)
plt.show()
```



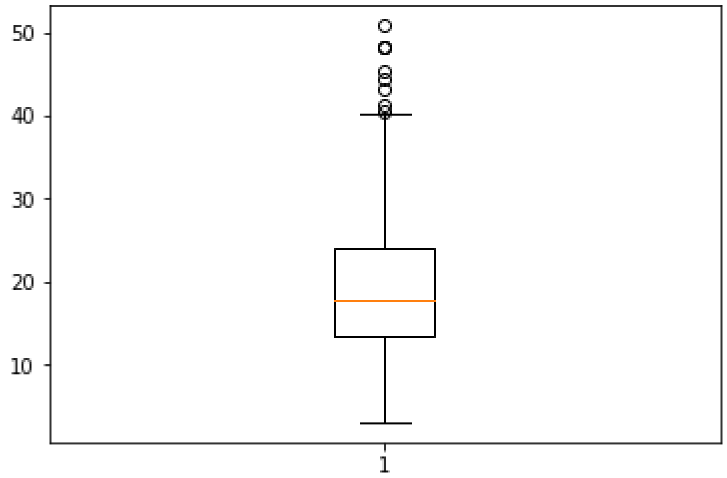
Back-to-back bar charts are like stacked bar charts but are horizontal. The sub-categories such as frequency of male who went for launch versus frequency of male who went for dinner, can be compared using the lengths of the bars corresponding to the frequencies or counts of the sub-categories.

Box Plot

- A boxplot is a graphic display of the distribution of data using a five-number summary.
- The five-point summary consist of minimum value, maximum value, first quartile, second quartile, and third quartile.

- The is dataset is sorted and split into four equal parts or quartiles.
- The first quartile (Q1) is the data point below which 25% of the data lies.
- The second quartile (Q2) is the data point below or above which 50% of the data lies (it is also called the median).
- The third quartile (Q3) is the data point below which 75% of the data lies.
- The interquartile range (IQR) is the distance from the first quartile to the third quartile: IQR = Q3 - Q1.

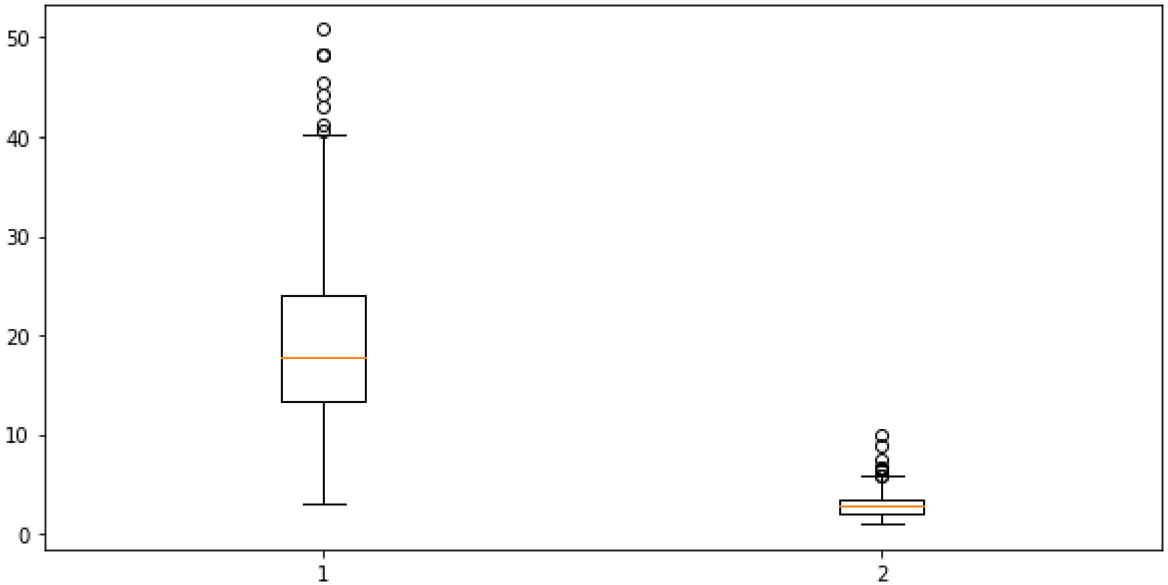
In [53]:
```
# a single box plot
plt.boxplot(data.total_bill)
plt.show()
```



Multiple Boxplots

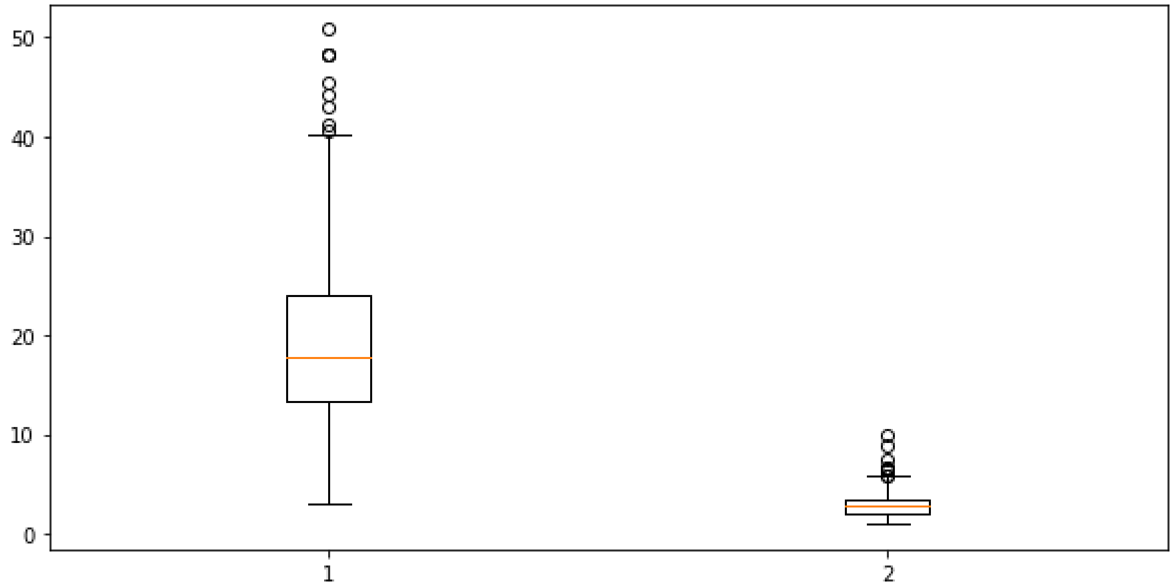
In [54]:

```
# set figure size
plt.figure(figsize=(10, 5))

x = data.total_bill, data.tip
plt.boxplot(x)
plt.show()
```



In [55]:

```
# an alternative way to create multiple boxplots

# extract and transpose the DataFrames
x = data[["total_bill", "tip"]].T

plt.figure(figsize=(10,5))
plt.boxplot(x)
plt.show()
```



Customizing the Color and Styles

We will take a look at:

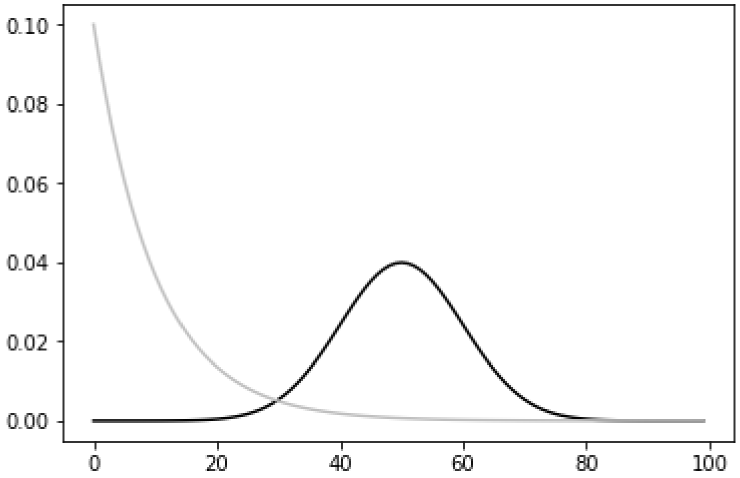
- how to set the color of plots,
- set linestyle and thickness for line plots,
- add markers/marker properties for plots,

- custom color/cmap for scatter plots,
- custom color for bar plots
- custom color for boxplots

Set the Colors of Plots

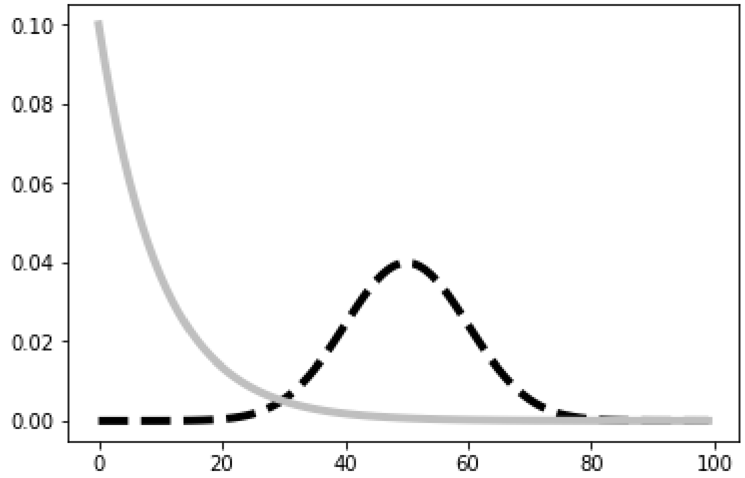
- Generally, predefined colors can be used or you could define your own colors.
- The name of predefined colors or their symbols can be used as argument in the plot function.
- Examples of predefined colors and their symbols are **blue (b), green (g), red (r), cyan (c), magenta (m), yellow (y), black (k), white (w)** .
- Html color codes can also be used. For example, balck (#000000), gray (#808080), white(#ffffff), etc.
- Sting values between 0 to 1 can be used where "0" is black and "1" is white.

In [56]:
```
x = np.arange(0, 100)

# create densities for normal and exponential distributions
y1 = stats.norm(loc=50, scale=10).pdf(x)
y2 = stats.expon.pdf(x, scale=10)

# plot the distributions
plt.plot(x, y1, color="0") # grey color
plt.plot(x, y2, color=".75") # black color

plt.show()
```



Set Linestyle and Thickness for Line Plots

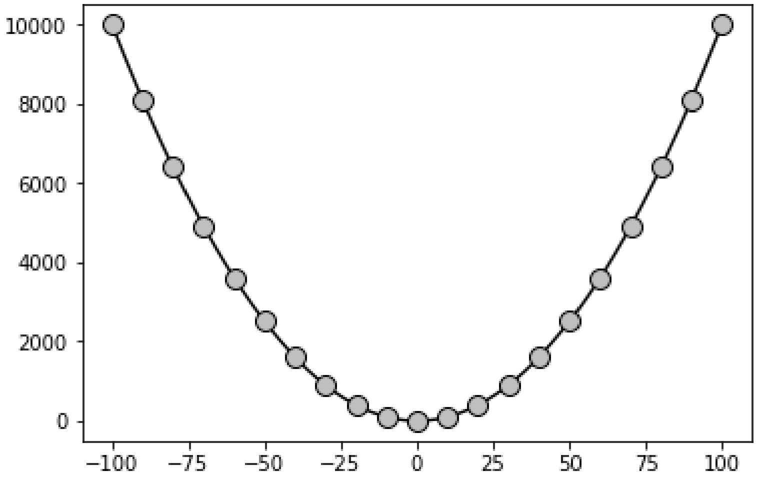
In [57]:
```
x = np.arange(0, 100)

# create densities for normal and exponential distributions
y1 = stats.norm(loc=50, scale=10).pdf(x)
y2 = stats.expon.pdf(x, scale=10)

# plot the distributions
plt.plot(x, y1, color="0", linewidth=4, linestyle="dashed") # grey color
plt.plot(x, y2, color="0.75", linewidth=4, linestyle="solid") # black color

plt.show()
```



Add Markers/Marker Properties for Plots

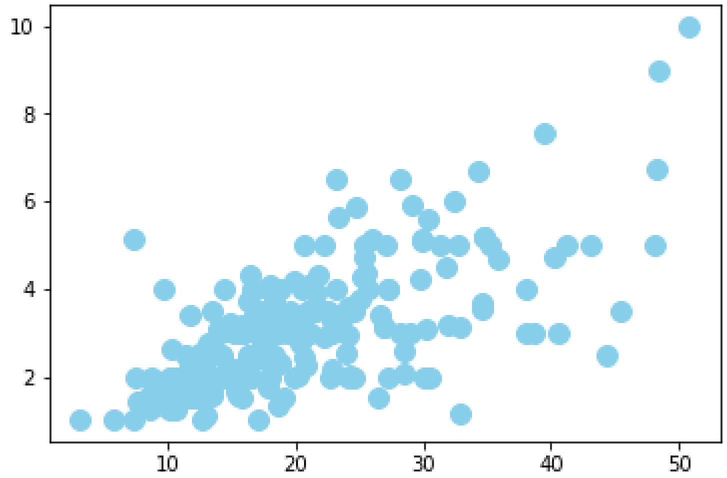
In [58]:

```
x = np.arange(-100, 101)
y1 = x**2

plt.plot(x, y1,
         c="k",
         marker="o",
         markevery=10,
         markersize=10,
         markerfacecolor=".75",
         markeredgecolor="k")

plt.show()
```



Custom Color for Scatter Plots

In [59]:
```
# Let's use the tips dataset

x1 = data.total_bill
x2 = data.tip

plt.scatter(x1, x2, color="skyblue", s=100)
plt.show()
```

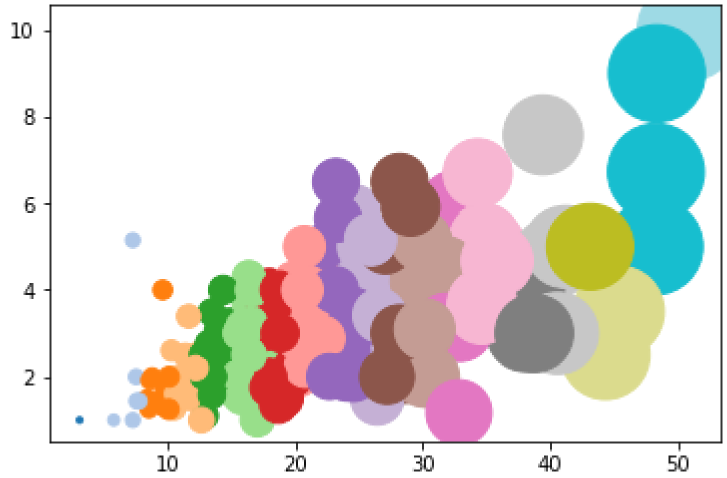


Note that the size of each point can be set using the "s" (size) parameter.

Custom Colormap for Scatter Plots

In [60]:

```
# let's apply a colormap

x1 = data.total_bill
x2 = data.tip

plt.scatter(x1, x2, c=x1, s=x1**2, cmap=plt.cm.tab20)
plt.show()
```



Here, we set the colors to change for different values of x1. The size of the marker is also set to change with with the values of x1

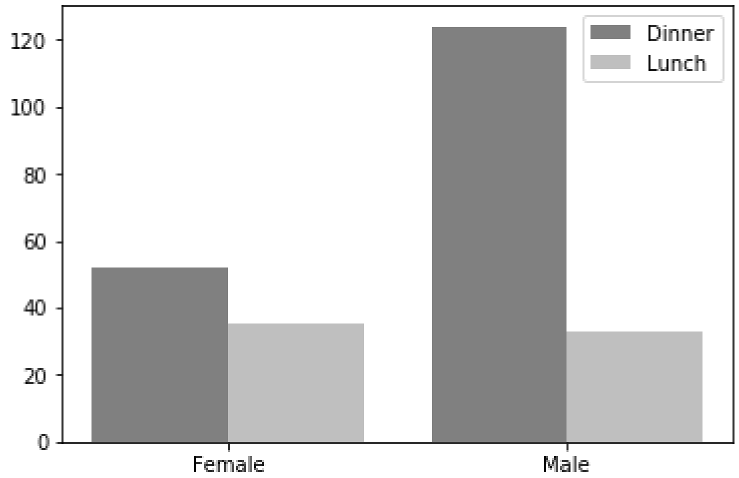
Custom Color for Bar Charts

In [61]:
```
# use the tips dataset
ctab = pd.crosstab(data.gender, data.time)
ctab
```

Out[61]:

time	Dinner	Lunch
gender		
Female	52	35
Male	124	33

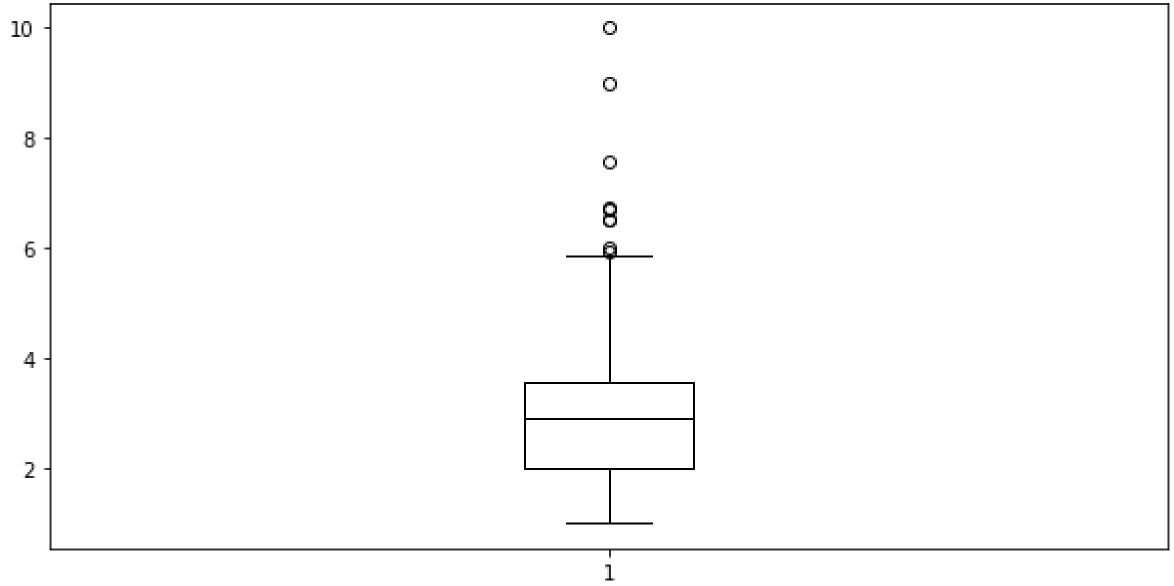
In [62]:
```
x = np.arange(len(ctab.Dinner))

plt.bar(x, ctab.Dinner.values, width=0.4,  color="0.5")
plt.bar(x + 0.4, ctab.Lunch.values, width=0.4, color="0.75")

plt.xticks([.2, 1.2], ctab.Dinner.index)
plt.legend(ctab.columns)
plt.show()
```



Custom Color for Boxplots

In [63]:

```
# set color of boxplot to black
# this is required sometimes for publishing

x = data.tip
plt.figure(figsize=(10, 5))

for name, line_list in plt.boxplot(x).items():
    for line in line_list:
        line.set_color("k")
```



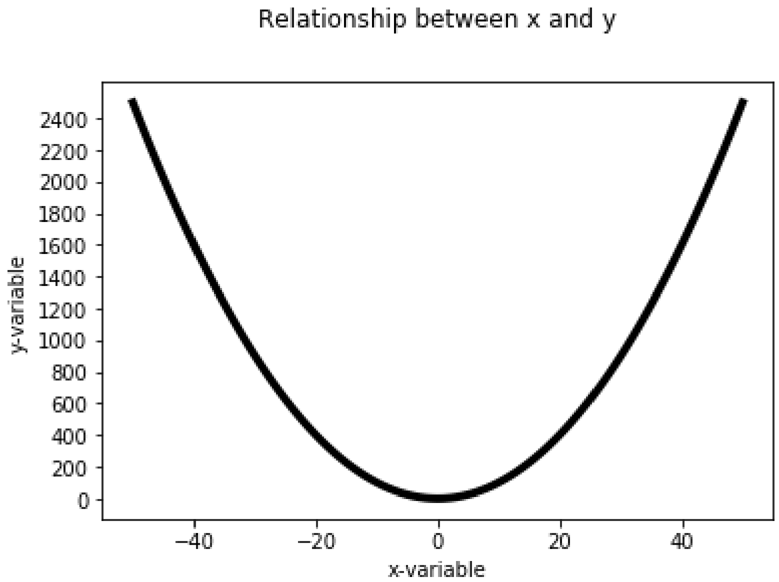
Working with Annotations

In this section, we will see how to add:

- title, axis labels,
- text and arrows,
- ticks and ticks labels,
- plot labels and legends,
- grid

Add Title, Axis Labels

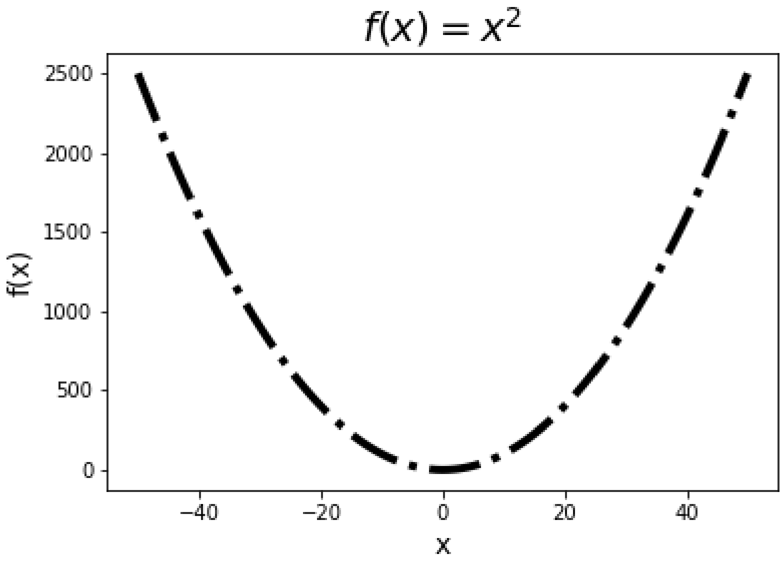
In [64]:
```
x = np.arange(-50, 51)
y = x**2

plt.plot(x, y, lw=4, c="k")

plt.title("Relationship between x and y", y=1.1) # add title
plt.xlabel("x-variable") # add a label on the x-axis
plt.ylabel("y-variable") # add a label on the y-axis
plt.yticks(np.arange(y.min(), y.max(), 200)) # add ticks on the y-axis

plt.show()
```



Note that you can increase the space between the title and the figure using the "y" parameter, for example y=1.1

Add tile using Latex

In [65]:
```
x = np.arange(-50, 51)
y = x**2

plt.plot(x, y, lw=4, ls="-.", c="k")

plt.title("$f(x) = x^2$", fontsize=20)
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)

plt.show()
```

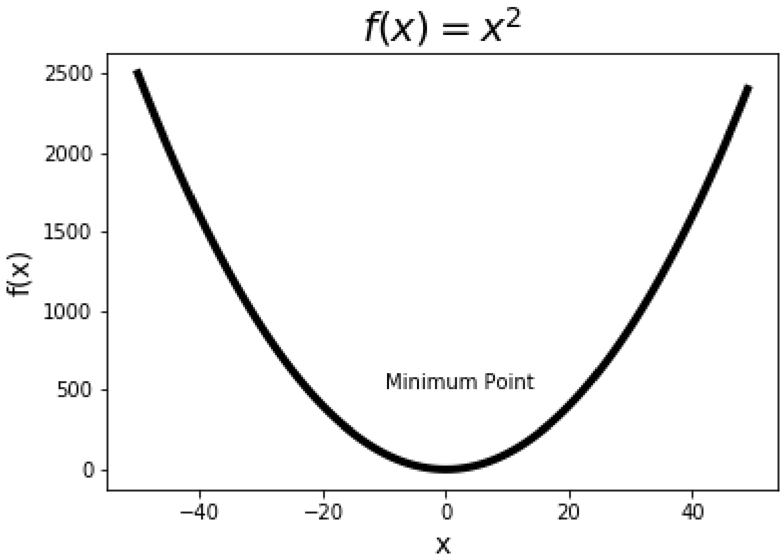


Note: the fontsize parameter can be used to change the font sizes of titles and labels

Add Text

In [66]:

```
x = np.arange(-50, 50)
y = x**2

plt.plot(x, y, lw=4, ls="-", c="k")

plt.title("$f(x) = x^2$", fontsize=20)
plt.xlabel("x", fontsize=14)
plt.ylabel("f(x)", fontsize=14)

# specify position to place text using -10, 500
plt.text(-10, 500, s="Minimum Point")
plt.show()
```



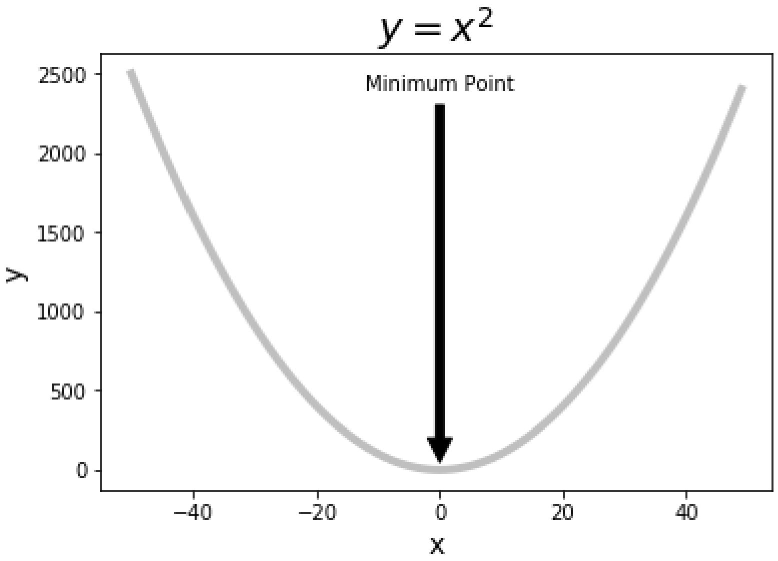
Add Arrow

- The **plt.annotate()** is used to add an arrow. The following parameters are used to specify the properties of the arrow.

- The **xy** parameter specifies the position of the arrow.
- The **xytext** specifies the position of the text
- The text is aligned throught the **horizontal and vertical** alignment parameter (ha, va).
- The **shrink** parameter controls the gap between the arrow itself and the end point.
- The **arrowprops** parameter contains a dictionary of arrow properties including **arrowstyle, facecolor, edgecolor, and alpha** .

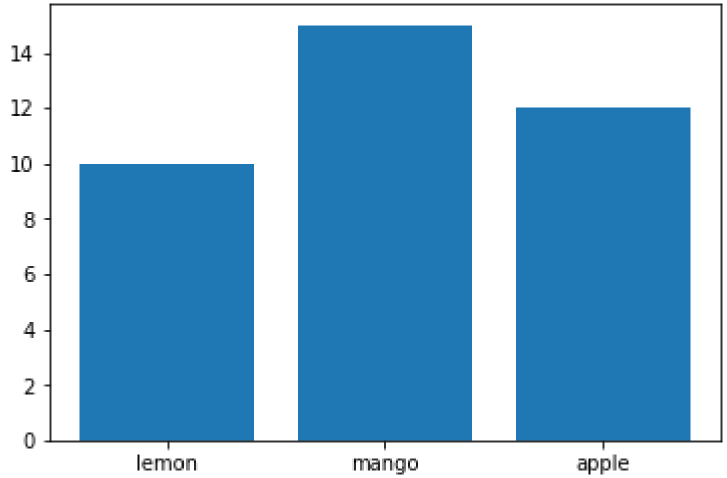
In [67]:

```
x = np.arange(-50, 50)
y = x**2

plt.plot(x, y, lw=4, ls="-", c="0.75")

plt.title("$ y = x^2$", fontsize=20)
plt.xlabel("x", fontsize=14)
plt.ylabel("y", fontsize=14)

# add arrow
plt.annotate("Minimum Point",
             ha="center", va="top", # align text
             xytext=(0, 2500), # text position
             xy=(0, 0), # point to annotate
             arrowprops ={"facecolor": "black", "shrink":.02})
plt.show()
```



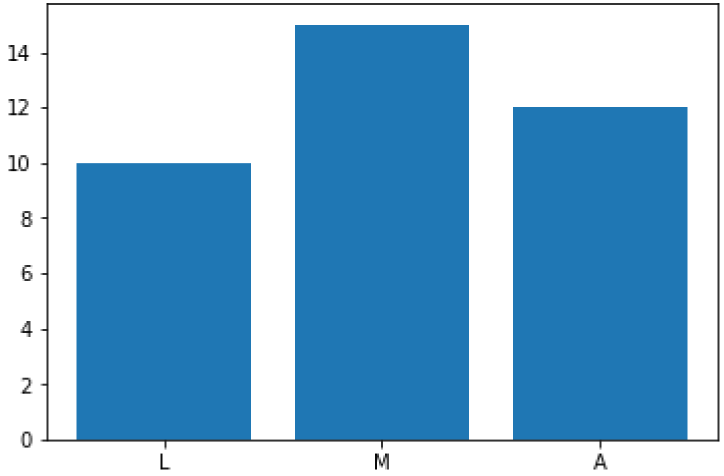
Ticks and Tick Labels

In [68]:
```
quantity = [10, 15, 12]
fruits = ["lemon", "mango", "apple"]

plt.bar(x=fruits, height=quantity)
plt.show()
```



In [69]:

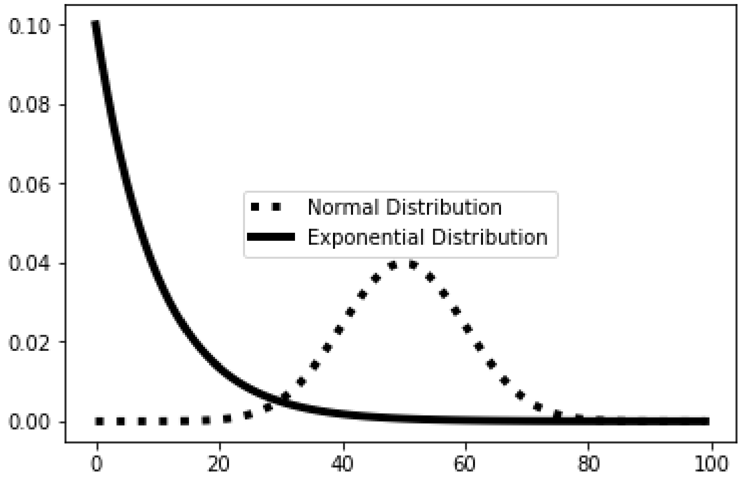
```
# use first letter of fruit to lable the x-axis

quantity = [10, 15, 12]
fruits = ["lemon", "mango", "apple"]

plt.bar(x=fruits, height=quantity)
plt.xticks(ticks=fruits, labels=["L", "M", "A"])
plt.show()
```



Plot Labels and Legends

In [70]:

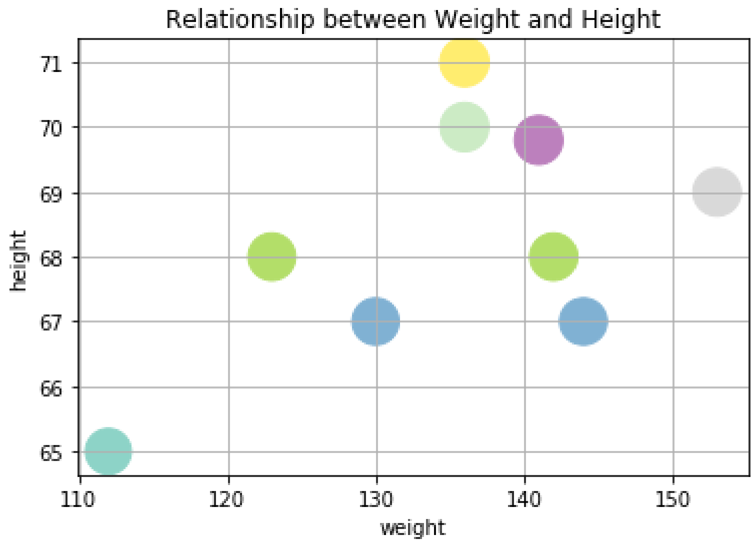
```
# add plot labels and legend

x = np.arange(0, 100)
y1 = stats.norm(loc=50, scale=10).pdf(x)
y2 = stats.expon.pdf(x, scale=10)

label = ["Normal Distribution", "Exponential Distribution"]

plt.plot(x, y1, linestyle="dotted", color="k", linewidth=4)
plt.plot(x, y2, linestyle="solid", color="k", linewidth=4 )

# position the legend at the center of the figure
plt.legend(label, loc="center")
plt.show()
```



Grid

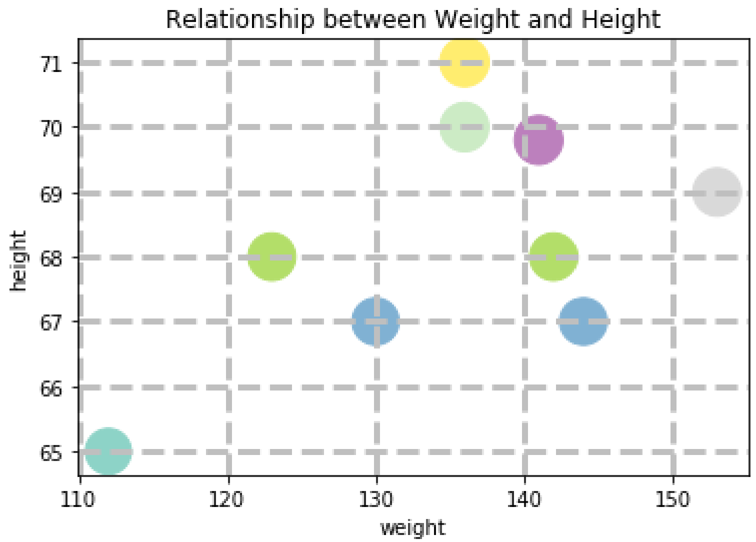
In [71]:

```
# add grid

height = np.array([65, 71, 69, 68, 67, 68, 69.8, 70, 67])
weight = np.array([112, 136, 153, 142, 144, 123, 141, 136, 130])

plt.title("Relationship between Weight and Height")
plt.xlabel("weight")
plt.ylabel("height")

plt.scatter(weight, height, c=height,
            cmap=plt.cm.Set3, s=height**1.5)

plt.grid()

plt.show()
```



In [72]:

```python
# add grid and style

height = np.array([65, 71, 69, 68, 67, 68, 69.8, 70, 67])
weight = np.array([112, 136, 153, 142, 144, 123, 141, 136, 130])

plt.title("Relationship between Weight and Height")
plt.xlabel("weight")
plt.ylabel("height")

plt.scatter(weight, height, c=height,
            cmap=plt.cm.Set3, s=height**1.5)

plt.grid(lw=3, ls="--", c="0.75")

plt.show()
```



The pyplot API Versus the Object-Oriented API

- There are two main approaches for plotting with matplotlib: by using functions in the pyplot module or through an object-oriented API
- matplotlib.pyplot is a module that contains a collection of functions used for creating figures, creating plotting areas, plotting and formatting plots.
- The pyplot API is generally less-flexible than the object-oriented API.
- Most of matplotlib.pyplot's function calls can also be implemented as methods of the **axes** object.
- In relation to the object oriented approach, matplotlib is object-oriented and you can work directly with it's object if you need more control in customizing your plots.
- All the plots we have seen so far were created using the pyplot API. We will now see how to create plots using the object oriented approach.
- https://matplotlib.org/api/api_overview.html#id2 (https://matplotlib.org/api/api_overview.html#id2)
- https://matplotlib.org/3.1.0/tutorials/introductory/pyplot.html (https://matplotlib.org/3.1.0/tutorials/introductory/pyplot.html)

Using the Methods of the Figure and Axes Objects for Plotting

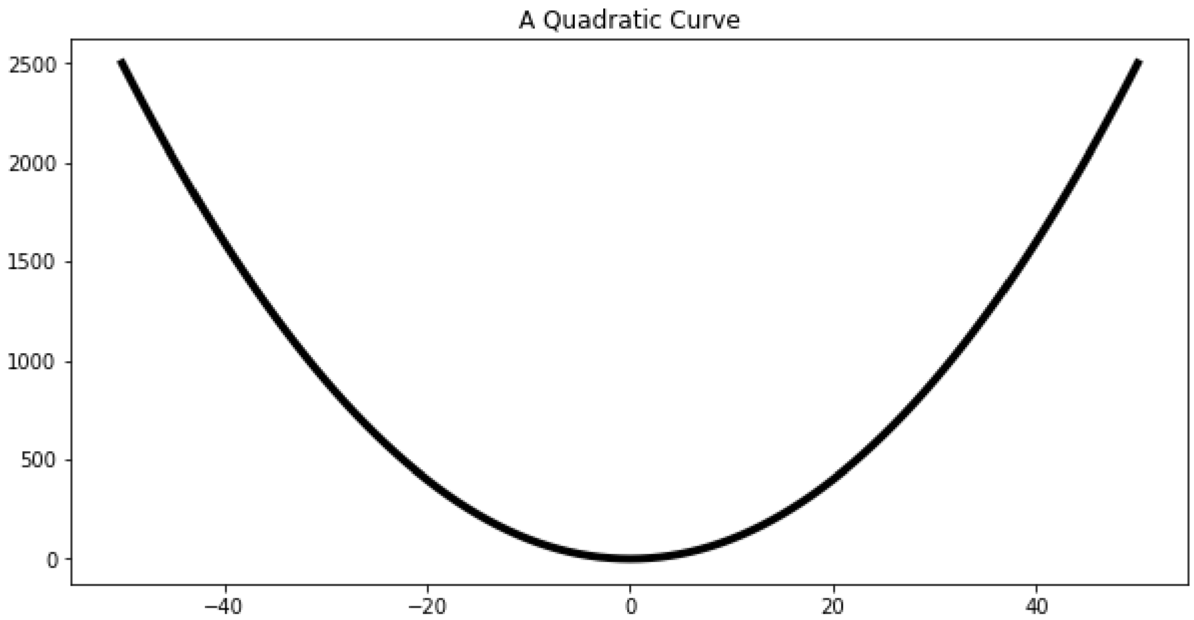
- Note that matplotlib is object oriented. It's major objects are the figure and axes objects.
- A matplotlib figure is a high-level object that contains the axes object.
- The Axes object representing a sub-section of a figure where the graph is plotted.
- A figure could have more than one axes and each axes further has attributes such as title, xlabel, ylabel, xticks, yticks, legend, grid, etc.
- That is, axes contain most of the figure's elements: https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes (https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes)

Instantiate the Figure and Axes Objects

- The figure and axes objects can be instantitated in different ways
- Sometimes the figure and axes objects are instantiated implicitly (or behind the scene).
- For example, when matplotlib's pyplot function such as plt.plot() or plt.scatter() are called, figure and axes objects are instantiated behind the scene.

In [73]:

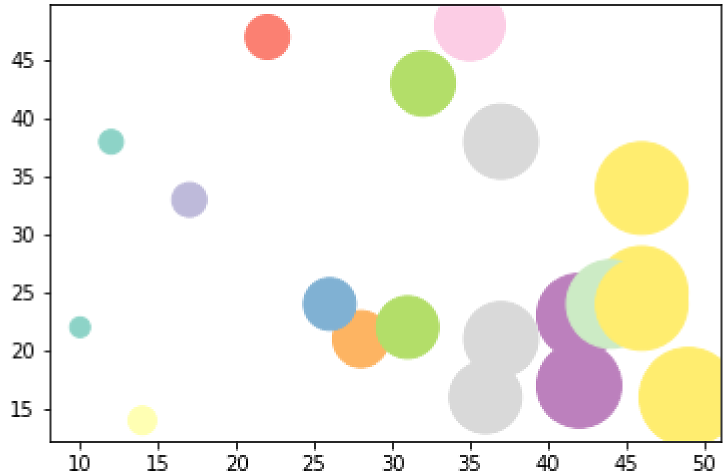
```
# instantiating figure and axes objects behind the scene with pyplot
x = np.arange(-50, 51)
y = x**2

# figure and axes are created when this code is run
plt.plot(x, y, lw=4, c="k")

# get the current axes
ax = plt.gca()

# get the current figure
fig = plt.gcf()

# call a method on the axes object to set the title
ax.set_title("A Quadratic Curve")

# call a method on the figure object to set the figure size
fig.set_size_inches(10, 5)
plt.show()
```



- When a pyplot plotting function is run and a figure object and an axes object is created under the hood, we can use the plt.gca() and plt.gcf() to get the current axes and current figure respectively.
- The methods of the figure and axes objects can then be called on the objects to perform certain tasks such as formatting.
- Running a pyplot plotting function as shown above can only create a single figure and a single axes.

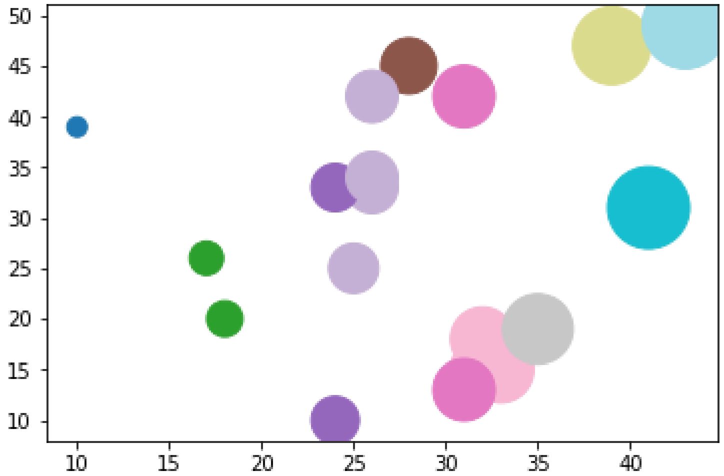
In [74]:

```
# creating one axes in a single figure using plt.subplot()

x1 = np.random.randint(10, 50, size=20)
x2 = np.random.randint(10, 50, size=20)

# create one axes in a figure.
fig, ax = plt.subplots(1, 1)
ax.scatter(x1, x2, c=x1, s=x1**2, cmap=plt.cm.Set3)

plt.show()
```



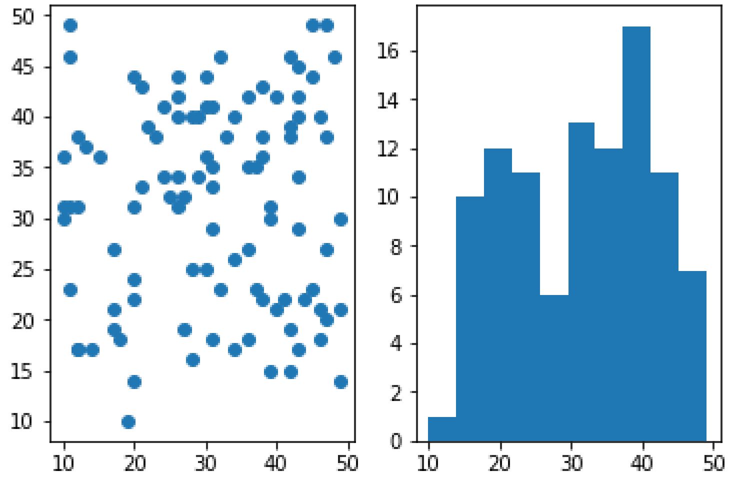
In [75]:

```
# first create a figure, then the axes

x1 = np.random.randint(10, 50, size=20)
x2 = np.random.randint(10, 50, size=20)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(x1, x2, c=x1, s=x1**2, cmap=plt.cm.tab20)

plt.show()
```



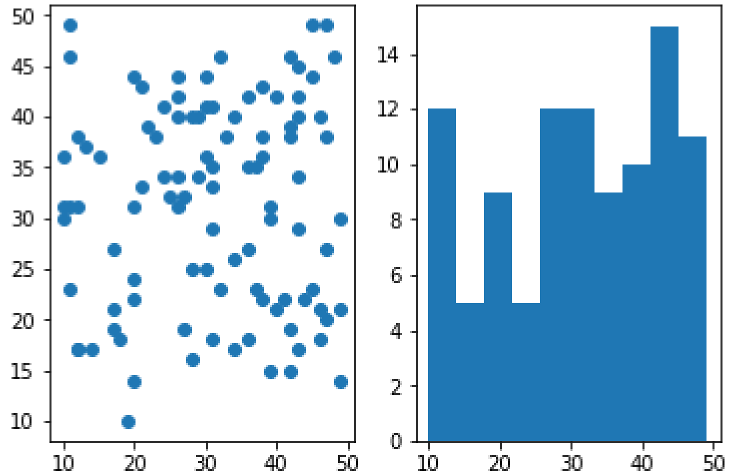
In [76]:

```
# creating several axes in a single figure using plt.subplot()

np.random.seed(3)
x1 = np.random.randint(10, 50, size=100)
x2 = np.random.randint(10, 50, size=100)

# plt.subplot takes (nrow, ncol) as arguments
fig, (ax1, ax2) = plt.subplots(1, 2)

ax1.scatter(x1, x2)
ax2.hist(x2)

plt.show()
```



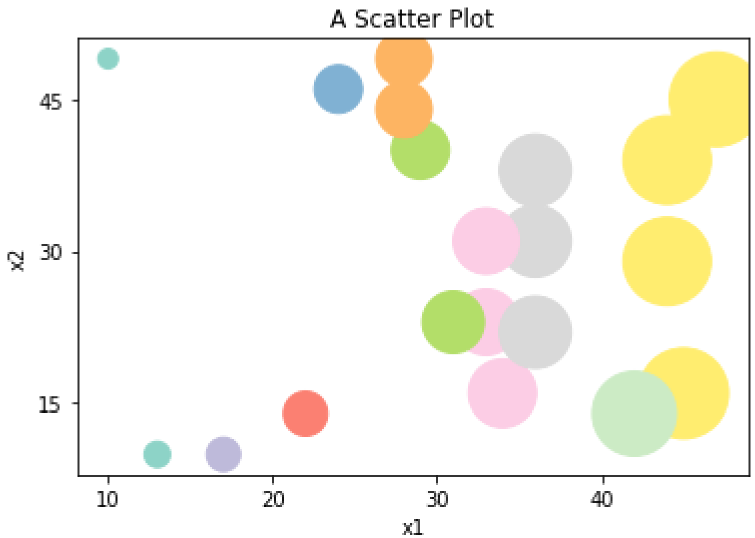
In [77]:

```
# another way to create multiple axes in one figure

np.random.seed(3)
x1 = np.random.randint(10, 50, size=100)
x2 = np.random.randint(10, 50, size=100)

# plt.subplot takes (nrow, ncol) as arguments
fig, ax = plt.subplots(1, 2)

ax[0].scatter(x1, x2)
ax[1].hist(x1)

plt.show()
```



Applying More Methods of the Axes Object

In [78]:
```
# creating one axes in a single figure using plt.subplot()

x1 = np.random.randint(10, 50, size=20)
x2 = np.random.randint(10, 50, size=20)

# create one axes in a figure.
fig, ax = plt.subplots(1, 1)
ax.scatter(x1, x2, c=x1, s=x1**2, cmap=plt.cm.Set3)

ax.set_title("A Scatter Plot")
ax.set_xlabel("x1")
ax.set_ylabel("x2")

# set ticks as multiple of an integer
ax.xaxis.set_major_locator(matplotlib.ticker.MultipleLocator(10))
ax.yaxis.set_major_locator(matplotlib.ticker.MultipleLocator(15))

plt.show()
```



There are more methods of the axes object that make it flexible to plot graphs: https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes (https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes)

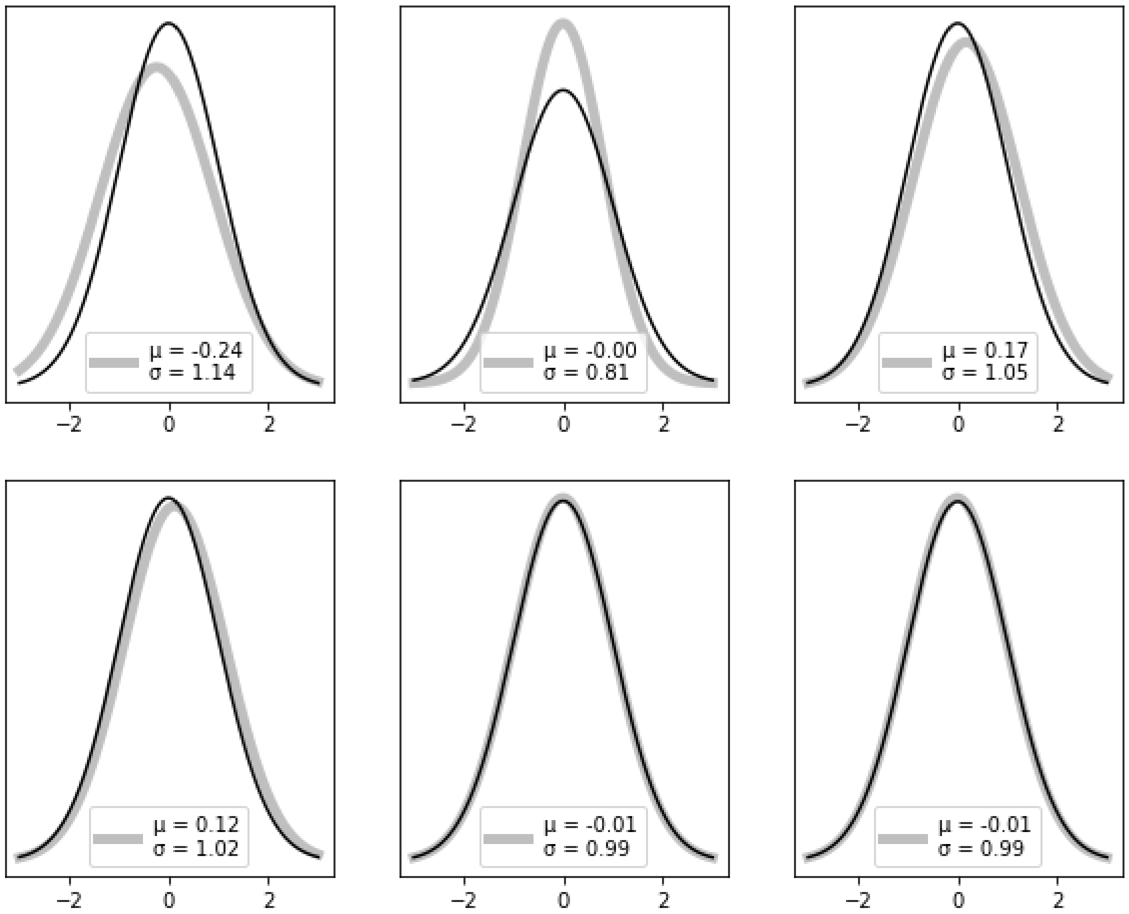
Use a Loop to Create Subplots

In [79]:

```
# create a normal density function
def norm_pdf(x, mu, sigma):

    """
    Generate a density for a normal random variable
    """

    a = 1/(np.sqrt(2*np.pi*sigma**2))
    b = np.square(x-mu)/(2*sigma**2)
    f = a*np.exp(-b)

    return f
```

In [80]:

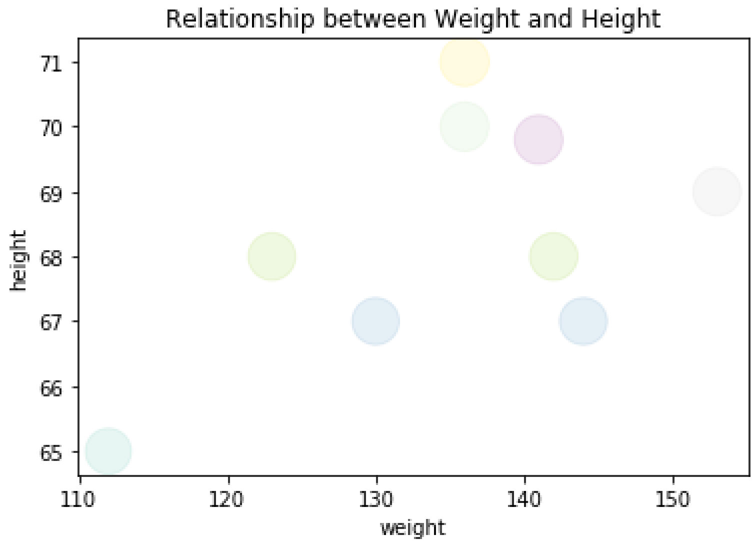
```
# create normal distributions of various sample sizes on subplots

X = np.linspace(-3, 3, 1000)
n = [15, 30, 60, 100, 1000, 10000]

fig, ax = plt.subplots(2, 3, figsize=(10,8))
counter=0
for i in range(2):
    for j in range(3):
        sample = np.random.randn(n[counter])
        mu = np.mean(sample)
        sigma = np.std(sample)

        label = "μ = {:3.2f}\nσ = {:3.2f}".format(mu, sigma)
        ax[i, j].plot(X,
                      norm_pdf(X, mu, sigma),
                      linewidth=5,
                      linestyle="solid",
                      label=label,
                      c=".75")

        ax[i, j].plot(X, norm_pdf(X, 0, 1), color = "k")
        ax[i, j].set_yticks([])
        ax[i, j].legend(loc="best")

        counter = counter + 1

plt.show()
```



Make Plot Transparent Using Alpha

In [81]:

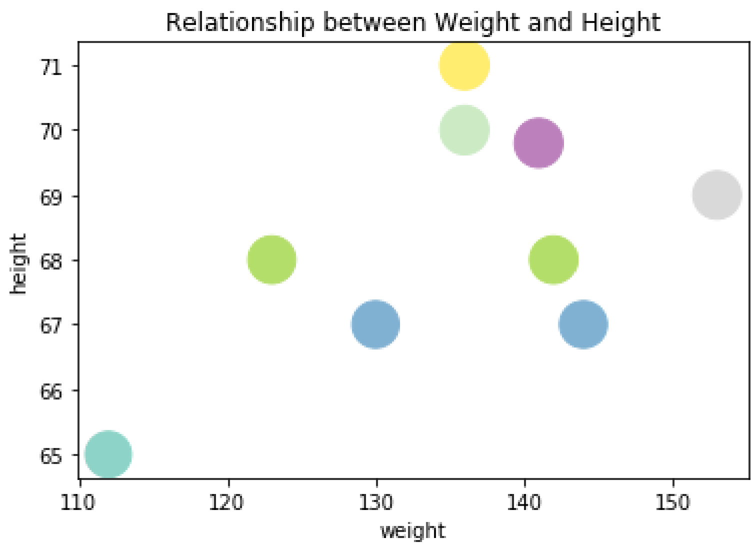
```
height = np.array([65, 71, 69, 68, 67, 68, 69.8, 70, 67])
weight = np.array([112, 136, 153, 142, 144, 123, 141, 136, 130])

plt.title("Relationship between Weight and Height")
plt.xlabel("weight")
plt.ylabel("height")

plt.scatter(weight, height, c=height, alpha=0.2,
            cmap=plt.cm.Set3, s=height**1.5)

plt.show()
```



Working with File Output

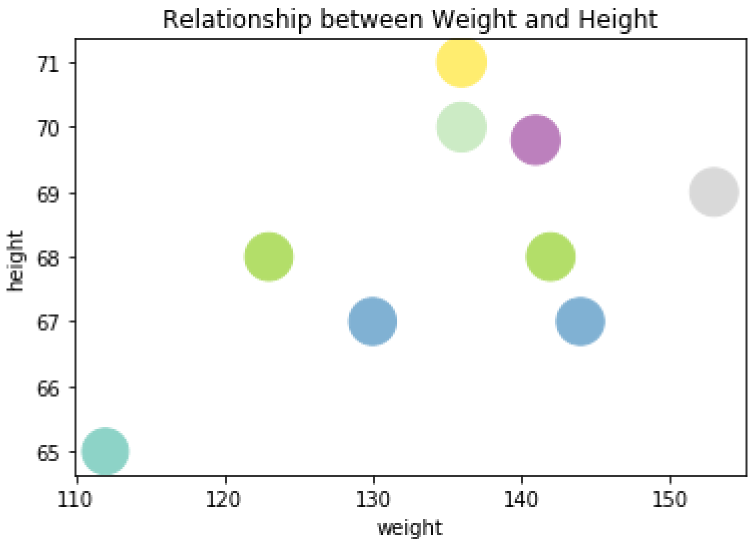
In [82]:

```python
# save figure as picture (.png)

height = np.array([65, 71, 69, 68, 67, 68, 69.8, 70, 67])
weight = np.array([112, 136, 153, 142, 144, 123, 141, 136, 130])

plt.title("Relationship between Weight and Height")
plt.xlabel("weight")
plt.ylabel("height")

plt.scatter(weight, height, c=height,
            cmap=plt.cm.Set3, s=height**1.5)

plt.savefig(r"C:\Users\nnfon\Desktop\scatter_plot.png")
```



In [83]:

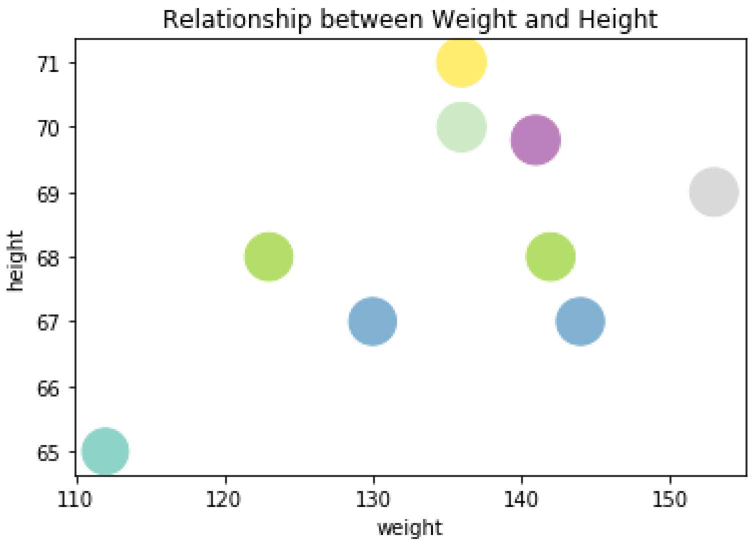
```python
# specify resolution for the picture

height = np.array([65, 71, 69, 68, 67, 68, 69.8, 70, 67])
weight = np.array([112, 136, 153, 142, 144, 123, 141, 136, 130])

plt.title("Relationship between Weight and Height")
plt.xlabel("weight")
plt.ylabel("height")

plt.scatter(weight, height, c=height,
            cmap=plt.cm.Set3, s=height**1.5)

plt.savefig(r"C:\Users\nnfon\Desktop\scatter_plot1.png", dpi=300)
```



- Use a large resolution if the picture will be used for a large poster.

In [84]:

```
# save to figure to a pdf document

height = np.array([65, 71, 69, 68, 67, 68, 69.8, 70, 67])
weight = np.array([112, 136, 153, 142, 144, 123, 141, 136, 130])

plt.title("Relationship between Weight and Height")
plt.xlabel("weight")
plt.ylabel("height")

plt.scatter(weight, height, c=height,
            cmap=plt.cm.Set3, s=height**1.5)

plt.savefig(r"C:\Users\nnfon\Desktop\scatter_plot1.pdf")
```



Data Visualization with Seaborn

In [85]:
```
# we can also get the tips data from the pydataset package

tips_data = pydataset.data("tips")
tips_data = tips_data.rename(columns={"sex": "gender"})
tips_data.head()
```

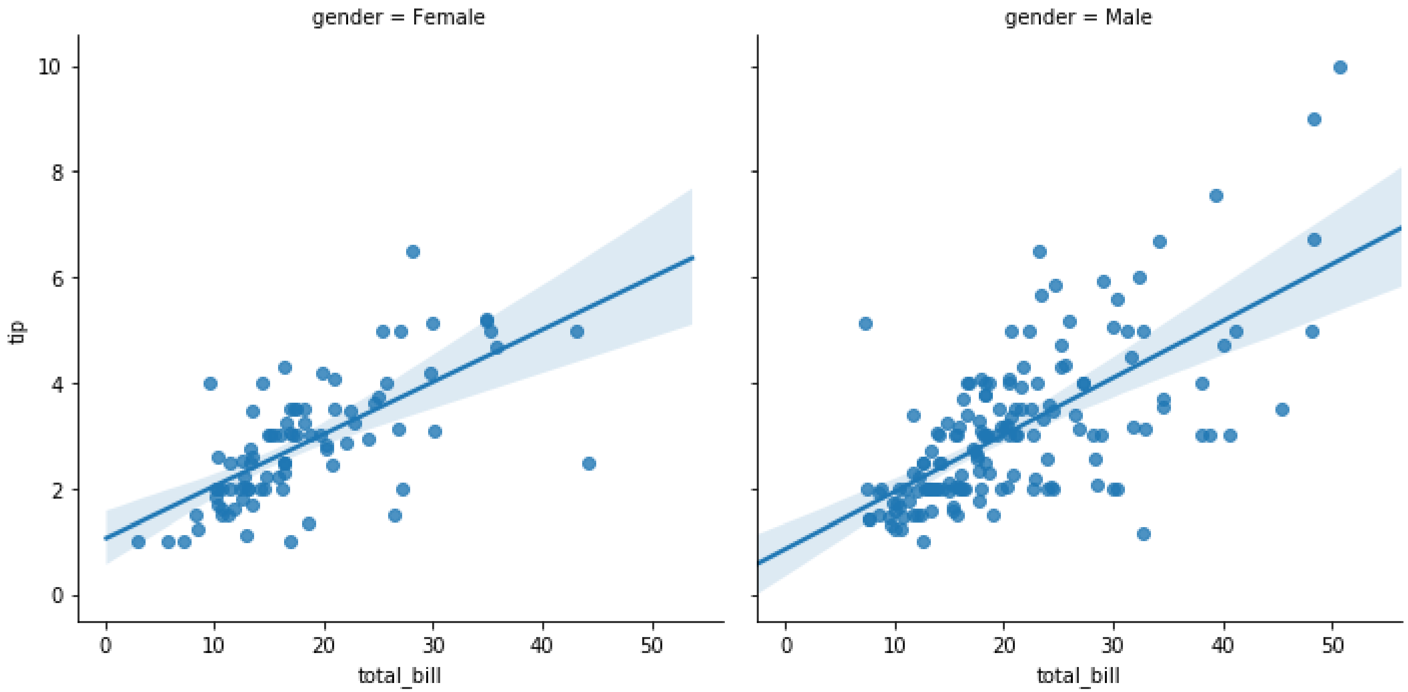
Out[85]:

	total_bill	tip	gender	smoker	day	time	size
1	16.99	1.01	Female	No	Sun	Dinner	2
2	10.34	1.66	Male	No	Sun	Dinner	3
3	21.01	3.50	Male	No	Sun	Dinner	3
4	23.68	3.31	Male	No	Sun	Dinner	2
5	24.59	3.61	Female	No	Sun	Dinner	4

Linear Plots by Category

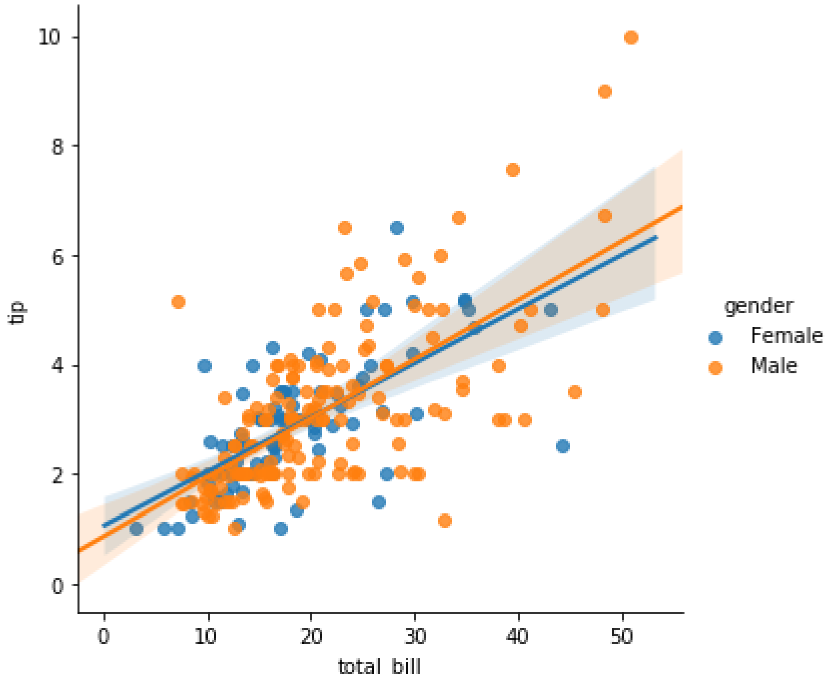
In [86]:

```
# plot a linear plot where total bill predicts tips by gender
sns.lmplot(x="total_bill", y="tip", col="gender", data=tips_data)
plt.show()
```



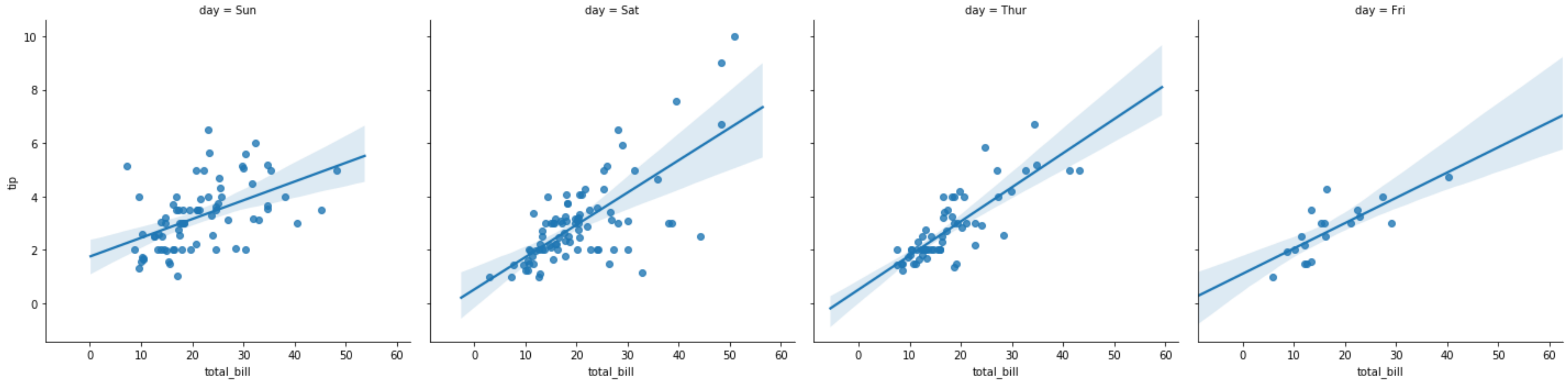
In [87]:

```
# plot a linear plot where total bill predicts tips by gender
# plot on the same axes

sns.lmplot(x="total_bill", y="tip", hue="gender", data=tips_data)
plt.show()
```

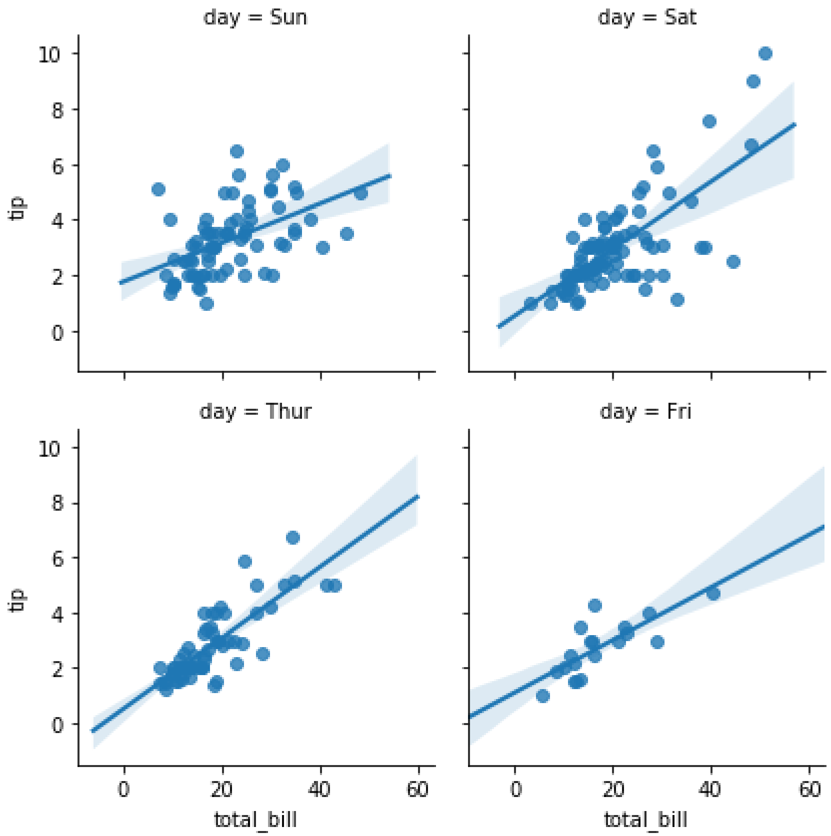


In [88]:

```
# plot a linear plot where total bill predicts tips by gender
sns.lmplot(x="total_bill", y="tip", col="day", data=tips_data)
plt.show()
```

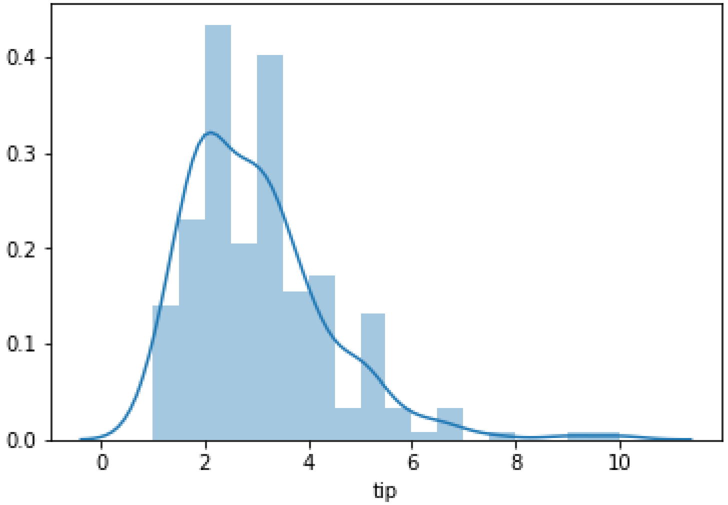


In [89]:
```
# wrap levels of categorical variable into multiple rows
sns.lmplot(x="total_bill", y="tip", col="day", data=tips_data, col_wrap=2, height=3)
plt.show()
```



Histogram and Kernel Density Estimate

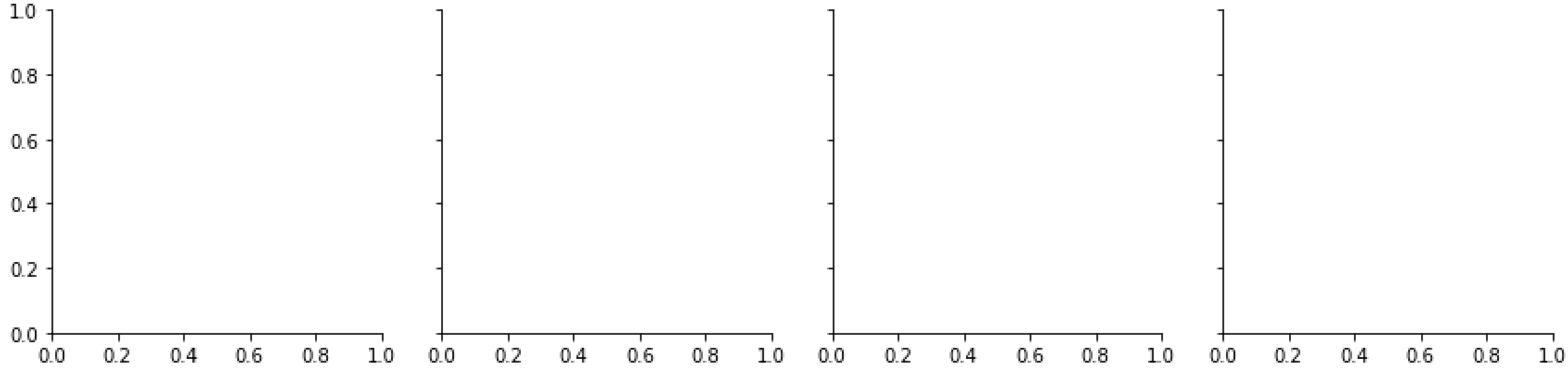
In [90]:
```
# distribution with a kernel desity estimate
sns.distplot(tips_data.tip)
plt.show()
```



Using FacetGrid to Plot by Category

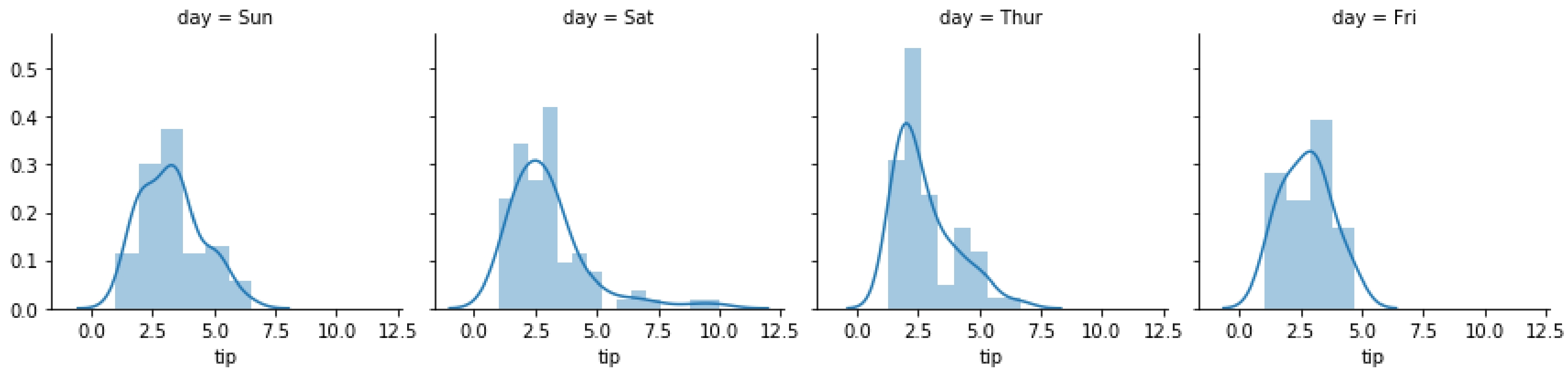
In [91]:

```
# to create a plot by levels of a categorical variable or column
sns.FacetGrid(data=tips_data, col="day")
plt.show()
```



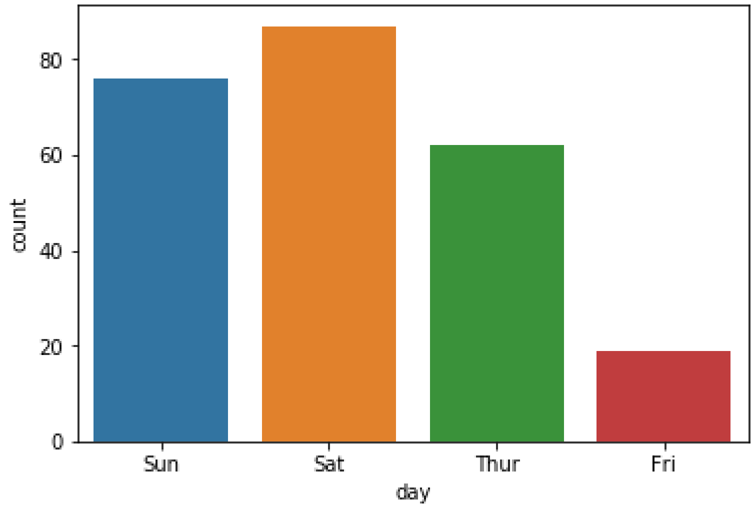
In [92]:

```
# to create a plot by levels of a categorical variable or column
facetgrid = sns.FacetGrid(data=tips_data, col="day")
facetgrid.map(sns.distplot, "tip")
plt.show()
```



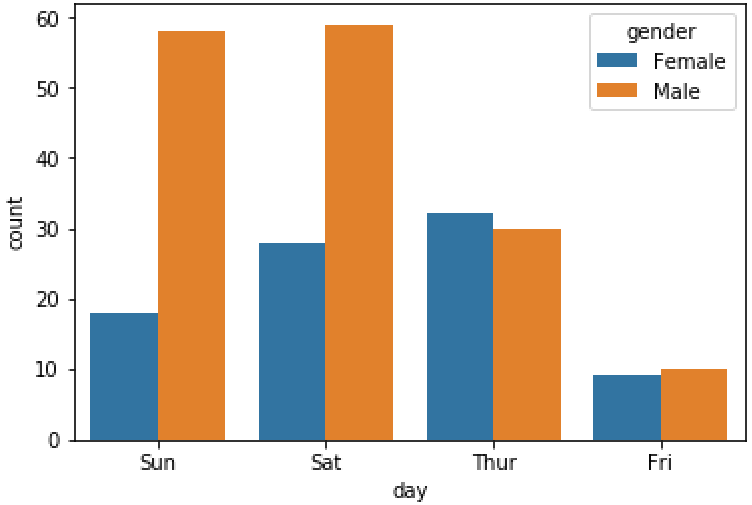
Bar Charts

In [93]:
```
# count plot is a fequency plot of categorical data (bar chart)
sns.countplot("day", data=tips_data)
plt.show()
```



In [94]:

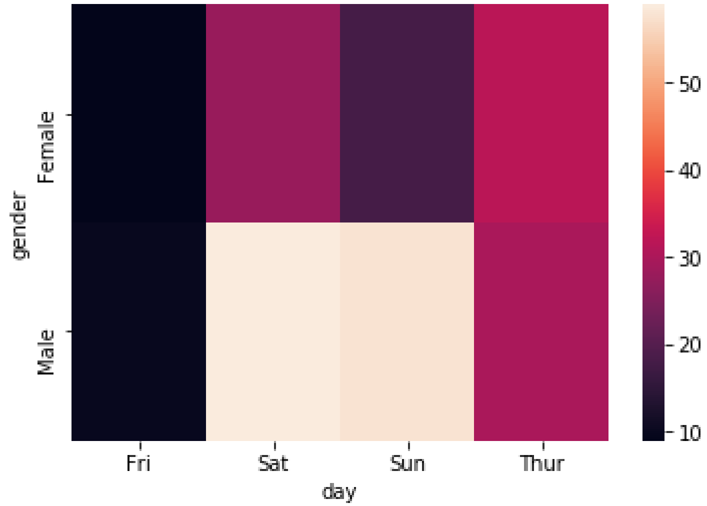
```
# use "hue" to specify second categorical variable
sns.countplot("day", hue="gender", data=tips_data)
plt.show()
```



Heat Map

In [95]:

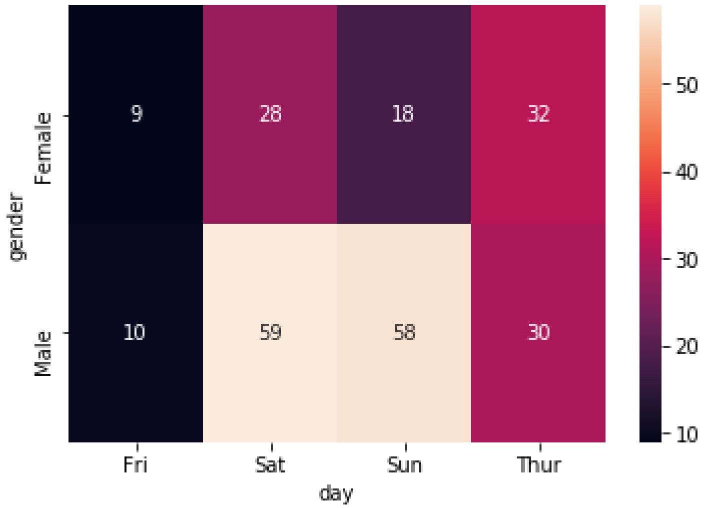
```
# heatmap plots rectangular data
# each value corresponds to a color
# can be useful to compare frequencies when variables interact

gender_day = pd.crosstab(tips_data.gender, tips_data.day)
sns.heatmap(gender_day)
plt.show()

print(gender_day)
```




```
day     Fri  Sat  Sun  Thur
gender
Female    9   28   18    32
Male     10   59   58    30
```

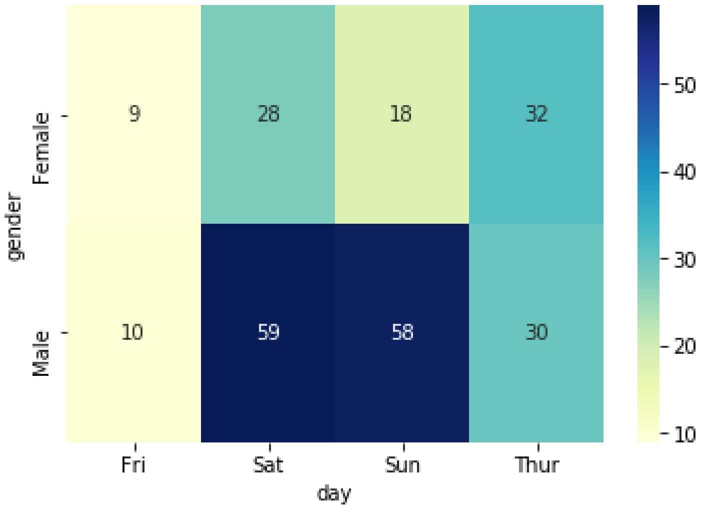
In [96]:

```python
# include values to be more precise
gender_day = pd.crosstab(tips_data.gender, tips_data.day)
sns.heatmap(gender_day, annot=True)
plt.show()
```



In [97]:

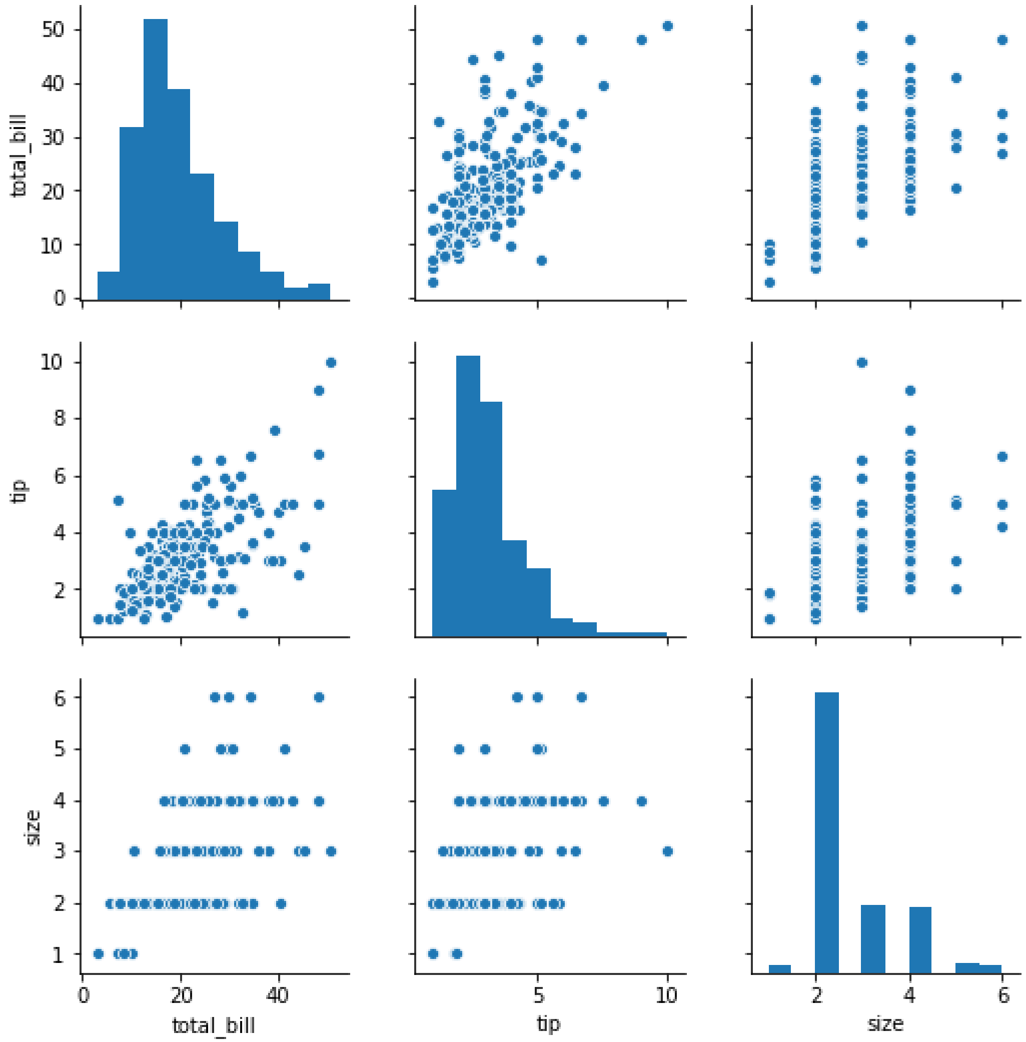
```
# specify a different color map
gender_day = pd.crosstab(tips_data.gender, tips_data.day)
sns.heatmap(gender_day, annot=True, cmap="YlGnBu")
plt.show()
```



Scatter Matrix

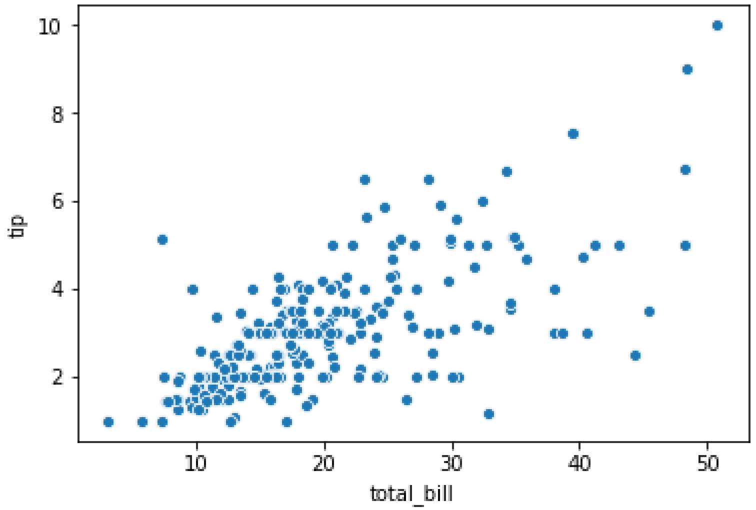
In [98]:

```
sns.pairplot(data=tips_data)
plt.show()
```



Scatter Plot

In [99]:
```
# scatter plot
sns.scatterplot(x="total_bill", y="tip", data=tips_data)
plt.show()
```



Linear Regression Model Fit Plot

In [100]:

```
# linear regression model fit plot
sns.regplot(x="total_bill", y="tip", data=tips_data)
plt.show()
```