Python Code Text: An Overview of Fundamentals, Machine Learning and Financial Analysis

By Evan Okin

Last Updated: May 27, 2020 (WIP)

This version of the text contains dataframes, arrays, hypothesis testing, interacting with a sql database, many machine learning algorithms, natural language processing, recommender systems, time series analysis, deep learning and financial applications.

Later versions of the text will contain web scraping, big data technologies and further fleshed out financial applications using calcbench.

About the Author

Evan holds a BS and MS degree in Applied Mathematics from The Whiting School of Engineering at The Johns Hopkins University, and an MBA in Quantitative Finance from NYU Stern School of Business. He also completed a Data Science bootcamp at The FlatIron School. Evan is passionate about math, statistics, programming, and education, and wants to "give back" by helping others.

Preface

The objective of this text is to help others learn key python syntax. It can help aspiring data scientists, data analysis, financial analysts or anyone who wants to improve their python skills with a straight-forward way to apply everyday code snippets. The course is split into three modules. Module 1 encompasses fundamentals and data analysis, including key libraries such as numpy and pandas. Module 2 encompasses predictive modeling and machine learning, including key algorithms in sklearn. Module 3 encompasses financial applications and portfolio management.

Code snippets were inspired by numerous sources that need to be acknowledged for making this happen. These include: Professor Benjamin Zweig (NYU Stern's course "Data Bootcamp"), Professor Foster Provost (NYU Stern's course "Data Science for Business - Technical"), Professor Dan Gode (NYU Stern's course "Financial Statement Analysis Using Python"), and Jose Portilla (Udemy courses "Data Science Bootcamp" and "Python for Finance and Algorithmic Trading").

The idea to create this text came to me during Quarantine during the Coronavirus Pandemic, upon soul-searching and wanting to do something that would both (1) help solidify my skillset and (2) can be used by others.

Module 1: Fundamentals and Data Analysis

Section: Getting Set Up

We will want to know our working directory so that we can store associated code and data sources into the correct directory. Thus, we can use the following command to print the working directory, so that we know where to put spreadsheets that we will read in during later sections.

In [1]:
```
!pwd
```

```
/Users/evanokin/Desktop/Snippets
```

Warnings can be cumbersome to sift through. We can turn off warnings so that they don't take out up a substantial amount of space with our output.

In [2]:
```
import warnings
warnings.filterwarnings("ignore")
```

We might want to run write a comment within the text of code. This is important because we will often write code that others will look at later. Comments are important because they can help others understand what you were thinking when you wrote code. It can also be helpful for the code writer, who might be away from the code for a while. We can do that by using a # for sporadic comments.

In [3]:
```
#This is a comment, but the rest of this code cell will still run
!pwd
```

```
/Users/evanokin/Desktop/Snippets
```

While we can use # for several lines, we can also comment out a bigger block by using triple quotation marks.

In [4]:
```
'''
This
is
a
comment
'''
!pwd
```

```
/Users/evanokin/Desktop/Snippets
```

Section: Performing Arithmetic

We can perform basic arithmetic, which follows the standard notation that we would expect. We can add numbers together.

In [5]:
```
4 + 2
```
Out[5]: 6

Notice that we used one space between each character. However, we can add numbers together, or perform any similar operations, without worrying about spacing (we can leave out space between our numbers and the addition operator, or we can put in as many as we want).

In [6]:
```
4+     2
```
Out[6]: 6

As a general rule of thumb, one space is used between characters as it is easy to read. Therefore, we will generally utilize one space between characters going forward. We can subtract numbers.

In [7]:
```
4 - 2
```
Out[7]: 2

We can multiply two numbers together.

In [8]:
```
4 * 2
```
Out[8]: 8

We can divide two numbers.

In [9]:
```
4 / 2
```
Out[9]: 2.0

We can use parantheses, to showcase that we can perform operations with more than two numbers. This displays the standard order of operations.

In [10]:
```
(4 + 2) * (8 + 8)
```
Out[10]: 96

We can raise a number to an exponent, or "take its power."

In [11]:
```
4 ** 2
```
Out[11]: 16

We can also raise a number to an exponent by using the "math" library. This library, like most others, can be easily imported and often can perform code in a more efficient way than we would be able to program it.

In [12]:
```
import math
math.pow(4,2)
```
Out[12]:
```
16.0
```

We can find the remainder, or "mod", of a number, when divided by another number.

In [13]:
```
4 % 3
```
Out[13]:
```
1
```

We can therefore confirm that a number is even, by taking the number mod 2, and confirming that there is no remainder.

In [14]:
```
4 % 2
```
Out[14]:
```
0
```

Similarly, we can confirm that a number is odd, by taking the number mod 2, and confirming that there is a remainder.

In [15]:
```
3 % 2
```
Out[15]:
```
1
```

We can find the number of groups of a set size that you can create.

In [16]:
```
4 // 2
```
Out[16]:
```
2
```

Section: Strings

We can display a string of text like we displayed numbers above.

In [17]:
```
'This is a string.'
```
Out[17]:
```
'This is a string.'
```

Notice that we used single quotation marks. We can also use double quotation marks for strings.

In [18]:
```
"This is a string."
```
Out[18]:
```
'This is a string.'
```

We can also use triple quotation marks for strings.

In [19]:
```
'''This is a string.'''
```

Out[19]:
```
'This is a string.'
```

One of the benefits of using triple quotation marks over single or double quotation marks is that triple quotation marks allow for printing on multiple lines.

In [20]:
```
print('''This
is
a
string.''')
```

```
This
is
a
string.
```

We can use a built-in function, called the print function, to print out a string.

In [21]:
```
print('This is a string.')
```

```
This is a string.
```

Alternatively, we can set a string to a specific variable, and then print the variable.

In [22]:
```
x = 'This is a string.'
print(x)
```

```
This is a string.
```

Strings are sequenced, and as a consequence, strings have indices for the characters in them. We can take a slice of the string.

In [23]:
```
x = 'This is a string.'
x[0]
```

Out[23]:
```
'T'
```

What we see above is that the first character of the string is a "T." Notice that the first element of indexing starts at 0, not 1. We can take a subset of multiple characters as well.

In [24]:
```
x[0:3]
```

Out[24]:
```
'Thi'
```

Notice that this prints out three characters - corresponding to indices 0, 1, and 2, even though the range went up to 3. This is because the last number is not inclusive.

We can slice the entire string by leaving off the last end point after the colon.

In [25]:
```
x[0:]
```
Out[25]:
```
'This is a string.'
```

We can take a slice of the last element of the string.

In [26]:
```
x[-1]
```
Out[26]:
```
'.'
```

We can lower-case all elements of a string.

In [27]:
```
x.lower()
```
Out[27]:
```
'this is a string.'
```

We can see below that this does not actually change the value of x, which is still capitalized.

In [28]:
```
x
```
Out[28]:
```
'This is a string.'
```

We can, however, save the variable with the command, so that it saves the change we made.

In [29]:
```
x = x.lower()
print(x)
```
```
this is a string.
```

We can upper-case all elements of a string.

In [30]:
```
x.upper()
```
Out[30]:
```
'THIS IS A STRING.'
```

We can upper-case each word of a string.

In [31]:
```
x.title()
```
Out[31]:
```
'This Is A String.'
```

We can upper-case the first letter of a string.

In [32]:
```
x.capitalize()
```
Out[32]:
```
'This is a string.'
```

We can take the reverse of a string.

In [33]:
```
x[::-1]
```
Out[33]:
```
'.gnirts a si siht'
```

We can find the length of a string, which corresponds to the number of characters in it.

In [34]:
```
len(x)
```
Out[34]:
```
17
```

We can see that this includes spaces in the calculation of characters - 4 characters in "This", 2 characters in "Is", 1 character in "A", 6 characters in "String", 1 character in "." and 3 spaces, for 17 total characters.

We can split a string's elements into a list of its elements (ie, words).

In [35]:
```
x.split()
```
Out[35]:
```
['this', 'is', 'a', 'string.']
```

Alternatively, we can do this another way.

In [36]:
```
x.rsplit()
```
Out[36]:
```
['this', 'is', 'a', 'string.']
```

We can split a string using a conditional, and return a portion of it by slicing it.

In [37]:
```
x='This is a #string.'
x.split('#')[1]
```
Out[37]:
```
'string.'
```

Section: Variables

We can set a string to be a variable, as we saw above.

In [38]:
```
s = "This is a string."
```

We can also set a number to a variable.

In [39]:
```
s = 4.2
```

We can display the variable by using the print function.

In [40]:
```
print(s)
```

```
4.2
```

Alternatively, we can display the variable by calling it.

In [41]:
```
s
```

Out[41]:
```
4.2
```

If we save numbers to variables, we can perform arithmetic on the variables, such as adding them together.

In [42]:
```
x = 4.2
y = 8.8
x + y
```

Out[42]:
```
13.0
```

We can re-assign a variable.

In [43]:
```
x = 4.2
x = 8.8
print(x)
```

```
8.8
```

Notice that the variable takes on the most recent assignment (second line), and ignores the previous assignment (first line).

We can increment a variable, which means adding onto it.

In [44]:
```
x = 1
x = x + 1
x
```

Out[44]:
```
2
```

Alternatively, we can use a different notation, which is more concise (and the standard way to write it).

In [45]:
```
x = 1
x += 1
x
```

Out[45]:
```
2
```

We can increment a variable down as well.

In [46]:
```
x = 89
x -= 1
print(x)
```

```
88
```

We can confirm the type of a variable that corresponds to an integer.

In [47]:
```
x=88
type(x)
```

Out[47]:
```
int
```

Alternatively, we can confirm the type of a variable that corresponds to a floating point number, which is a number with a decimal.

In [48]:
```
x = 4.2
type(x)
```

Out[48]:
```
float
```

We can see that if change a variable, it will only show the last variable saved. This is done with storing variables in different IDs of memory. We can check a variable's memory ID.

In [49]:
```
x = 24
id(x)
```

Out[49]:
```
4483651712
```

If we change the variable, we see that it will (always) utilize a new memory ID.

In [50]:
```
x = 88
id(x)
```

Out[50]:
```
4483653760
```

Section: Lists

We can input a list of numbers.

In [51]:
```
[4,8,15,16,23,42]
```

Out[51]:
```
[4, 8, 15, 16, 23, 42]
```

We can also input a list of strings.

In [52]:
```
['Person A','Person B','Person C']
```
Out[52]:
```
['Person A', 'Person B', 'Person C']
```

Like strings, lists are sequenced - the order matters (a list of [A,B] is different from a list of [B,A]). Because lists are sequenced, we can index. Indexing on lists starts at 0 (similar to strings), not 1. In addition to being sequenced, lists are mutable, which means we can change the content of them without changing the memory ID. We can perform indexing on a list to find (or change) a certain element.

In [53]:
```
l = [1,2,3,4,5]
l[2] = 'Re-assign the third value of this list, to this string'
l
```
Out[53]:
```
[1, 2, 'Re-assign the third value of this list, to this string', 4, 5]
```

We can reverse a list.

In [54]:
```
l.reverse()
print(l)
```
```
[5, 4, 'Re-assign the third value of this list, to this string', 2, 1]
```

We can add an element to the end of a list.

In [55]:
```
l = ['Person A','Person B','Person C']
l.append('Person D')
l
```
Out[55]:
```
['Person A', 'Person B', 'Person C', 'Person D']
```

We can remove the last element of a list.

In [56]:
```
l.pop()
```
Out[56]:
```
'Person D'
```

We can remove a specific element of a list.

In [57]:
```
l=[1,3,5]
l.pop(2)
```
Out[57]:
```
5
```

We can find the number of elements of a list.

In [58]:
```
l = [1,2,3,4,5]
len(l)
```

Out[58]: 5

We can find the set of unique elements of a list.

In [59]:
```
l = [1,1,1,2,3]
set(l)
```

Out[59]: {1, 2, 3}

We can find out how many repeats or duplicates there are in a list.

In [60]:
```
len(l) - len(set(l))
```

Out[60]: 2

We can concatenate or combine lists together.

In [61]:
```
list_even=[2,4,6]
list_odd=[1,3,5]
list_even + list_odd
```

Out[61]: [2, 4, 6, 1, 3, 5]

We can add a list onto another.

In [62]:
```
list_even_more=[8,10]
list_even.extend(list_even_more)
print(list_even)
```

```
[2, 4, 6, 8, 10]
```

We can create a nested list, or a list within a list.

In [63]:
```
l = [1,2,[3,4]]
print(l)
```

```
[1, 2, [3, 4]]
```

We can slice a nested list similar to a regular list, using multi-level indexing to slice inner elements.

In [64]:
```
l[2][1]
```

Out[64]: 4

We can create a list using the range function to create a collection of increasing numbers.

In [65]:
```
list(range(0,10))
```

Out[65]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Notice that the range does not include the last number, similar to when we sliced a string. Alternatively, we don't need to specify the starting range.

In [66]:
```
list(range(10))
```

Out[66]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can also create a list using the range function but specifying the step size.

In [67]:
```
list(range(0,10,2))
```

Out[67]:
```
[0, 2, 4, 6, 8]
```

We can create a list by splitting a string.

In [68]:
```
list_from_string= 'US Canada Mexico'.split()
list_from_string
```

Out[68]:
```
['US', 'Canada', 'Mexico']
```

We can perform a list comprehension, which operates on each element of a list.

In [69]:
```
x = [1,4,7]
x_adjusted = [i**3 for i in x]
print(x_adjusted)
```

```
[1, 64, 343]
```

We can use mapping with a lambda function on a list to perform the same thing.

In [70]:
```
x = [1,4,7]
list(map(lambda i: i**3,x))
```

Out[70]:
```
[1, 64, 343]
```

We can use a lambda function to filter on a list. For example, we can filter out to find the even values in our list.

In [71]:
```
l = [1,64,343]
list(filter(lambda i: i%2 ==0,l))
```

Out[71]:
```
[64]
```

We can sort a list of numbers sequentially.

In [72]:
```
l=[1,3,5,2,4]
l=sorted(l)
l
```

Out[72]:
```
[1, 2, 3, 4, 5]
```

We can sort a list of strings as well. For example, if we have a list of names, we can sort based on first name.

In [73]:
```
names = ['Michael Jordan','Lebron James','Larry Bird','Kobe Bryant','Scotti
sorted(names, key=lambda x: x.split()[0])
```

Out[73]:
```
['Kobe Bryant',
 'Larry Bird',
 'Lebron James',
 'Michael Jordan',
 'Scottie Pippen']
```

We can also sort based on last name, by sorting on the second word instead of the first.

In [74]:
```
sorted(names, key=lambda x: x.split()[1])
```

Out[74]:
```
['Larry Bird',
 'Kobe Bryant',
 'Lebron James',
 'Michael Jordan',
 'Scottie Pippen']
```

We can confirm the type of a list.

In [75]:
```
type(l)
```

Out[75]:
```
list
```

Section: Tuples

We can input a tuple and print it. Tuples are immutable, meaning that they cannot be changed, unlike lists.

In [76]:
```
t=(1,2,3)
print(t)
```

```
(1, 2, 3)
```

Similar to a list, we can slice an element of a tuple, with indexing starting at 0, because tuples are sequenced.

In [77]:
```
t[0]
```
Out[77]:
```
1
```

We can confirm the type of a tuple.

In [78]:
```
type(t)
```
Out[78]:
```
tuple
```

If you input elements without specifying a data structure, it will create a tuple. We can create a tuple from elements.

In [79]:
```
"324","hello"
```
Out[79]:
```
('324', 'hello')
```

We can sort a tuple like we sorted a list.

In [80]:
```
sorted((1,3,5,2,4))
```
Out[80]:
```
[1, 2, 3, 4, 5]
```

Notice that when we sort a tuple, we get back a list. In general, one benefit of using tuples over lists is that we use up less memory. We can confirm the amount of memory that we use on a list or tuple.

In [81]:
```
list_check = [1,2,3,4,5]
tuple_check = (1,2,3,4,5)
print('Memory used on list: ' , list_check.__sizeof__())
print('Memory used on tuple: ' , tuple_check.__sizeof__())
```
```
Memory used on list:  80
Memory used on tuple:  64
```

Section: Sets

We can input a set, which is defined only by unique elements.

In [82]:
```
{1,1,2,2,3}
```
Out[82]:
```
{1, 2, 3}
```

Alternatively, we can take a list and turn it into a set.

In [83]:
```
set([1,1,2,2,3])
```
Out[83]:
```
{1, 2, 3}
```

We can add an element to a set.

In [84]:
```
s = {1,2,3}
s.add(4)
print(s)
```

```
{1, 2, 3, 4}
```

We can confirm the type of a set.

In [85]:
```
type(s)
```

Out[85]:
```
set
```

We can find the union (or total collection of elements) between two sets.

In [86]:
```
A=set([1,2,3,4,5])
B=set([4,5,6,7,8])
A | B
```

Out[86]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

We can find the intersection (or overlap of common elements) between two sets.

In [87]:
```
A & B
```

Out[87]:
```
{4, 5}
```

Unlike lists, sets are not sequenced, so we can not take a slice of a set without obtaining an error. Also, if we can create an empty set.

In [88]:
```
a_set = set()
```

We need to use parentheses and not brackets, because brackets are used to create dictionaries which are discussed in the next section.

Section: Dictionaries

We can input a dictionary, which is a collection of key-value pairs. For example, we might want a dictionary of the type of graduate school and the admissions test required for that type of school.

In [89]:
```
d={'Business School':'GMAT','Doctoral Program':'GRE','Med School':'MCAT'}
print(d)
```

```
{'Business School': 'GMAT', 'Doctoral Program': 'GRE', 'Med School': 'MCA
T'}
```

We can return the items of a dictionary.

In [90]:
```
d.items()
```

Out[90]:
```
dict_items([('Business School', 'GMAT'), ('Doctoral Program', 'GRE'), ('M
ed School', 'MCAT')])
```

We can return the keys of a dictionary.

In [91]:
```
d.keys()
```

Out[91]:
```
dict_keys(['Business School', 'Doctoral Program', 'Med School'])
```

We can return the values of a dictionary.

In [92]:
```
d.values()
```

Out[92]:
```
dict_values(['GMAT', 'GRE', 'MCAT'])
```

The keys that we use have to be unique, although the values can be duplicated. We can return an element of a dictionary.

In [93]:
```
d['Business School']
```

Out[93]:
```
'GMAT'
```

We can add an element to a dictionary.

In [94]:
```
d['Law School']='LSAT'
print(d)
```

```
{'Business School': 'GMAT', 'Doctoral Program': 'GRE', 'Med School': 'MCA
T', 'Law School': 'LSAT'}
```

We can delete an element of a dictionary.

In [95]:
```
del d['Doctoral Program']
print(d)
```

```
{'Business School': 'GMAT', 'Med School': 'MCAT', 'Law School': 'LSAT'}
```

We can find the number of elements of a dictionary.

In [96]:
```
len(d)
```

Out[96]:
```
3
```

While a dictionary has length, we can't slice it because unlike a list, a dictionary has no order.

We can confirm the type of a dictionary.

In [97]:
```
type(d)
```
Out[97]:
```
dict
```

Like sets, dictionaries are not sequenced, and as a result we can't slice on them.

Section: Booleans

We can return a boolean with a True value.

In [98]:
```
True
```
Out[98]:
```
True
```

We can return a boolean with a False value.

In [99]:
```
False
```
Out[99]:
```
False
```

We can test if a condition is true or false.

In [100]:
```
4 > 2
```
Out[100]:
```
True
```

We can do the same to test if two numbers are equal to each other.

In [101]:
```
4 == 2
```
Out[101]:
```
False
```

Notice that we used two equal signs to test equality, whereas we used one equal sign above to set a variable equal to something. We can also test an inequality.

In [102]:
```
4 != 2
```
Out[102]:
```
True
```

We can test several conditions at once to see if all conditions hold.

In [103]:
```
(1<2) and (2>3)
```
Out[103]:
```
False
```

We can test several conditions at once to see if one condition holds.

In [104]:
```
(1<2) or (2>3)
```

Out[104]:
```
True
```

We can test if a string ends with a certain condition.

In [105]:
```
a='baseball'
a.endswith('ball')
```

Out[105]:
```
True
```

We can confirm the type of a boolean.

In [106]:
```
type(True)
```

Out[106]:
```
bool
```

Section: If-Statements

We can perform an if-statement.

In [107]:
```
temperature=60
if temperature<32:
    print('It is freezing')
else:
    print('It is not freezing')
```

```
It is not freezing
```

Similarly, we can perform an if-statement with many conditionals.

In [108]:
```
temperature=60
if temperature<32:
    print('It is freezing')
elif temperature<50:
    print('It is cold')
elif temperature<70:
    print('It is warm')
else:
    print('It is hot')
```

```
It is warm
```

Notice that we used several elif statements here. In general, we will use an if statement for the first statement, an else statement for the last statement, and an elif statement for all statements in between.

Section: Loops

We can use a for loop to loop through elements of a list and perform an operation on all of the elements.

In [109]:
```
l = [1,2,3,4,5]
for i in l:
    print(i)
```

```
1
2
3
4
5
```

Notice that we used "i" in the loop, but it doesn't matter what we use. We can use anything, shown below.

In [110]:
```
l = [1,2,3,4,5]
for element in l:
    print(element)
```

```
1
2
3
4
5
```

We can create a nested loop, or a loop within a loop.

In [111]:
```
for i in range(1,2):
    for j in range(1,3):
        print('i',i,'j',j)
```

```
i 1 j 1
i 1 j 2
```

We can create a while loop as well.

In [112]:
```
i = 1
while i < 5:
    print('Number ',i)
    i+=2
```

```
Number  1
Number  3
```

We can use a break statement within a loop, which would stop the loop.

In [113]:
```
for i in list(range(0, 5)):
    if True:
        print(i)
        break
    print("Never executed because of the break statement")
```

```
0
```

Section: Functions

We can write functions to perform operations for us that we can call on as many times as we want after we run it. For example, we can write a function to cube a number.

In [114]:
```
def cuber(n):
    return n**3
cuber(10)
```

Out[114]:
```
1000
```

We can then apply the function to a list of numbers.

In [115]:
```
list_of_nums_to_cube = list(range(0,11))
for i in list_of_nums_to_cube:
    print('Number to cube is ' , i, ': ', cuber(i))
```

```
Number to cube is  0 :  0
Number to cube is  1 :  1
Number to cube is  2 :  8
Number to cube is  3 :  27
Number to cube is  4 :  64
Number to cube is  5 :  125
Number to cube is  6 :  216
Number to cube is  7 :  343
Number to cube is  8 :  512
Number to cube is  9 :  729
Number to cube is  10 :  1000
```

We can write a function to take the mean, or average, of a list of numbers.

In [116]:
```
def mean(l):
    length = len(l)
    total = sum(l)
    return total/length
mean([3,2,4])
```

Out[116]:
```
3.0
```

We can write a function to return the factorial of a number.

In [117]:
```
def fact(n):
    tot=1
    while n>1:
        tot*=n
        n-=1
    return tot
fact(5)
```

Out[117]: 120

We can confirm that this is doing what we want it to, based on the definition of factorials.

In [118]:
```
fact(5) == (5*4*3*2*1)
```

Out[118]: True

We can write a function that counts the number of times a word appears in a string.

In [119]:
```
def countWord(word,s):
    count=0
    for i in s.lower().split():
        if i==word:
            count += 1
    return count
countWord('i','I want to test how many times I appears in this sentence.')
```

Out[119]: 2

We can write a function that returns the sample variance from a list of numbers.

In [120]:
```
def get_variance(sample):
    n = len(sample)
    total = sum(sample)
    sample_mean = total/n
    value=0
    for i in sample:
        value += (i - sample_mean)**2
    variance = value / (n - 1)
    return variance
print(get_variance([1,2,3,4,5]))
```

```
2.5
```

We can combine mapping with a function, to map all elements of a list.

In [121]:
```
l=[1,2,3,4]
list(map(cuber,l)) #easier than a for-loop
```

Out[121]: [1, 8, 27, 64]

We can assess how long it takes to cube a number using mapping with a function, a list comprehension, and a for-loop.

In [122]:
```
%%time
l=[1,2,3,4,5,6,7,8]
list(map(cuber,l))
```

```
CPU times: user 8 µs, sys: 0 ns, total: 8 µs
Wall time: 11.2 µs
```

In [123]:
```
%%time
l=[1,2,3,4,5,6,7,8]
[i**3 for i in l]
```

```
CPU times: user 8 µs, sys: 0 ns, total: 8 µs
Wall time: 11.9 µs
```

In [124]:
```
%%time
l=[1,2,3,4,5,6,7,8]
l_cube=[]
for i in l:
    l_cube.append(i)
```

```
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 6.91 µs
```

Section: Formating

We can round a number to a certain number of decimals.

In [125]:
```
x=12.3456
round(x,2)
```

Out[125]:
```
12.35
```

If we don't specify any parameters, it rounds an input to the nearest integer.

In [126]:
```
round(x)
```

Out[126]:
```
12
```

We can round a number to the nearest ten, hundred, thousandth, etc. as well. For example we can round a number to the nearest thousand.

In [127]:
```
x=104320
round(x,-3)
```

Out[127]:
```
104000
```

We can combine with for-loops. For example,

In [128]:
```
import math
for i in range(10):
    print('Digits of pi ' + str(i) + ' ' + str(round(math.pi,i)))
```

```
Digits of pi 0 3.0
Digits of pi 1 3.1
Digits of pi 2 3.14
Digits of pi 3 3.142
Digits of pi 4 3.1416
Digits of pi 5 3.14159
Digits of pi 6 3.141593
Digits of pi 7 3.1415927
Digits of pi 8 3.14159265
Digits of pi 9 3.141592654
```

We can format a string using .format.

In [129]:
```
name = 'Evan'
favorite_number = 42
print('My name is {one} and my favorite number is {two}.'.format(one=name,
```

```
My name is Evan and my favorite number is 42.
```

Alternatively, we can use f-strings.

In [130]:
```
name = 'Evan'
favorite_number = 42
print(f'My name is {name} and my favorite number is {favorite_number}.')
```

```
My name is Evan and my favorite number is 42.
```

We can format a number with a specified number of decimal points.

In [131]:
```
my_num=3.241989
print ("%.3f" % my_num)
```

```
3.242
```

We can format a number with commas.

In [132]:
```
num = 123456789
print('Normal format: ' , num)
print('Formatted properly: ' f" {num:,.0f}")
```

```
Normal format:  123456789
Formatted properly:  123,456,789
```

We can format a number with underscores in place of commas.

In [133]:
```
num = 1234.56
print(f" {num:_.2f}")
```

```
 1_234.56
```

We can format a string to be right-aligned.

In [134]:
```
txt1 = "This is the first text"
txt2 = "This is the second text"
print(f"{txt1:>25}")
print(f"{txt2:>25}")
```

```
   This is the first text
  This is the second text
```

We can format a string to be left-aligned.

In [135]:
```
txt1 = "This is the first text"
txt2 = "This is the second text"
print(f"{txt1:<25}")
print(f"{txt2:<25}")
```

```
This is the first text
This is the second text
```

We can format a string to be center-aligned.

In [136]:
```
print(f"{txt1:^25}")
print(f"{txt2:^25}")
```

```
 This is the first text
 This is the second text
```

We can format a string to be center-aligned padded with dashes.

In [137]:
```
print(f"{txt1:-^25}")
print(f"{txt2:-^25}")
```

```
-This is the first text--
-This is the second text-
```

Section: Numpy

We utilize the NumPy library for many types of numerical problems. We can import the NumPy library.

In [138]:
```
import numpy as np
```

We can convert a list into an array.

In [139]:
```
l=[1,2,3,4,5]
import numpy as np
np.array(l)
```

Out[139]:
```
array([1, 2, 3, 4, 5])
```

We can create an array using the arange function with one parameter.

In [140]:
```
import numpy as np
np.arange(10)
```

Out[140]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Alternatively, we can create an array using the range function and turning it into a list.

In [141]:
```
import numpy as np
np.array(list(range(10)))
```

Out[141]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

We can create an array using the arange function with several parameters, including the step size between numbers.

In [142]:
```
import numpy as np
np.arange(0,10,2)
```

Out[142]:
```
array([0, 2, 4, 6, 8])
```

We can create an array of linearly spaced numbers with several parameters.

In [143]:
```
import numpy as np
np.linspace(0,5,10)
```

Out[143]:
```
array([0.        , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
       2.77777778, 3.33333333, 3.88888889, 4.44444444, 5.        ])
```

We can create an array of one specified number.

In [144]:
```
import numpy as np
np.full(10,3)
```

Out[144]:
```
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3])
```

We can create an empty array.

In [145]:
```
import numpy as np
np.empty(5)
```

Out[145]:
```
array([5.e-323, 0.e+000, 0.e+000, 0.e+000, 0.e+000])
```

We can create a one-dimensional array of zeros.

In [146]:
```
import numpy as np
np.zeros(5)
```

Out[146]:
```
array([0., 0., 0., 0., 0.])
```

We can create a one-dimensional array of ones.

In [147]:
```
import numpy as np
np.ones(5)
```

Out[147]:
```
array([1., 1., 1., 1., 1.])
```

We can create a two-dimensional array of zeros.

In [148]:
```
import numpy as np
np.zeros((2,5))
```

Out[148]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

We can create a two-dimensional array of ones.

In [149]:
```
import numpy as np
np.ones((3,4))
```

Out[149]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

We can create a matrix with 3 rows and 3 columns.

In [150]:
```
import numpy as np
three_by_three = [[1,2,3],[4,5,6],[7,8,9]]
np.array(three_by_three)
```

Out[150]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

We can take a transpose of a matrix.

In [151]:
```
import numpy as np
np.transpose(three_by_three)
```

Out[151]:
```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

We can concatenate arrays.

In [152]:
```
import numpy as np
np.concatenate((np.arange(3),np.arange(4)),axis=0)
```

Out[152]:
```
array([0, 1, 2, 0, 1, 2, 3])
```

Alternatively,

In [153]:
```
import numpy as np
np.concatenate((np.eye(2),np.eye(2)),axis=1)
```

Out[153]:
```
array([[1., 0., 1., 0.],
       [0., 1., 0., 1.]])
```

We can insert an element into an array.

In [154]:
```
import numpy as np
np.insert(np.arange(3),1,5)
```

Out[154]:
```
array([0, 5, 1, 2])
```

We can delete an element within an array.

In [155]:
```
import numpy as np
np.delete(np.arange(3),[0])
```

Out[155]:
```
array([1, 2])
```

We can create an integer data type.

In [156]:
```
import numpy as np
np.int64
```

Out[156]:
```
numpy.int64
```

We can create a float data type.

In [157]:
```
import numpy as np
np.float32
```

Out[157]:
```
numpy.float32
```

We can create a complex number type.

In [158]:
```
import numpy as np
np.complex
```

Out[158]:
```
complex
```

We can create a boolean type.

In [159]:
```
import numpy as np
np.bool
```

Out[159]:
```
bool
```

We can create an object type.

In [160]:
```
import numpy as np
np.object
```

Out[160]:
```
object
```

We can create a string type.

In [161]:
```
import numpy as np
np.string_
```

Out[161]:
```
numpy.bytes_
```

We can create a unicode type.

In [162]:
```
import numpy as np
np.unicode_
```

Out[162]:
```
numpy.str_
```

We can create an identity matrix.

In [163]:
```
import numpy as np
np.eye(4)
```

Out[163]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Alternatively,

In [164]:
```
import numpy as np
np.identity(4,dtype=float)
```

Out[164]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

We can also set the type of the elements to an integer.

In [165]:
```
import numpy as np
np.identity(4, dtype=int)
```

Out[165]:
```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

We can create an identity matrix from scratch using a for-loop.

In [166]:
```
m=np.zeros((4,4))
for i in list(range(4)):
    m[i][i]=1
m
```

Out[166]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

We can calculate the inverse of a matrix.

In [167]:
```
import numpy as np
A = np.array([[4,2,1],[4,8,3],[1,1,0]])
A_inv = np.linalg.inv(A)
A_inv
```

Out[167]:
```
array([[ 0.3, -0.1,  0.2],
       [-0.3,  0.1,  0.8],
       [ 0.4,  0.2, -2.4]])
```

We can round the elements of a matrix.

In [168]:
```
import numpy as np
np.matrix.round(A_inv,0)
```

Out[168]:
```
array([[ 0., -0.,  0.],
       [-0.,  0.,  1.],
       [ 0.,  0., -2.]])
```

We can find the dot product of matrices.

In [169]:
```
import numpy as np
A = np.array([[1, 2], [3, 4], [5, 6]])
v = np.array([0.5, 0.5])
C = A.dot(v)
C
```

Out[169]:
```
array([1.5, 3.5, 5.5])
```

We can find the cross product of matrices.

In [170]:
```
import numpy as np
x = np.array([0,0,1])
y = np.array([0,1,0])
Z = np.cross(x,y)
Z
```

Out[170]:
```
array([-1,  0,  0])
```

We can solve a system of linear equations.

In [171]:
```
import numpy as np
# Suppose we want to solve:
# 2 a + 1 b = 35
# 3 a + 4 b = 65
A = np.matrix([[2, 1], [3, 4]])
B = np.matrix([35,65])
np.linalg.solve(A,B.T)
```

Out[171]:
```
matrix([[15.],
        [ 5.]])
```

We can then confirm that the math is doing what we want it to.

In [172]:
```
import numpy as np
print('First equation: ',(2*np.linalg.solve(A,B.T)[0]+1*np.linalg.solve(A,B
print('Second equation: ',(3*np.linalg.solve(A,B.T)[0]+4*np.linalg.solve(A,
```

```
First equation:  [[ True]]
Second equation:  [[ True]]
```

NumPy allows us to work with random numbers. In many cases, we will want to set the random seed to be a fixed number. This allows us to generate the same output for a code that utilizes random numbers.

In [173]:
```
import numpy as np
np.random.seed(88)
```

We can generate a random number between 0 and 1.

In [174]:
```
import numpy as np
np.random.rand()
```

Out[174]:
```
0.6475510493530234
```

We can generate an array of specified amount of random numbers between 0 and 1.

In [175]:
```
import numpy as np
np.random.rand(5)
```

Out[175]:
```
array([0.50714969, 0.52834138, 0.8962852 , 0.69999119, 0.7142971 ])
```

We can also generate a matrix of a specified amount of random numbers between 0 and 1.

In [176]:
```
import numpy as np
np.random.rand(5,5)
```

Out[176]:
```
array([[0.71733838, 0.22281946, 0.17515452, 0.45684149, 0.92873843],
       [0.00988589, 0.08992219, 0.85020027, 0.48562106, 0.87683559],
       [0.30733394, 0.38796555, 0.58144574, 0.11368718, 0.76788018],
       [0.31256604, 0.64601046, 0.03269892, 0.16639558, 0.43030303],
       [0.22780647, 0.96630747, 0.90493941, 0.86566291, 0.03216278]])
```

We can confirm that if we run millions of random numbers, the average should be close to 0.50 because the random numbers are uniformly distributed.

In [177]:
```
import numpy as np
np.random.rand(1000000).mean()
```

Out[177]:
```
0.500516865047794
```

All of the above can also be done with standard normal random variables as well. For example, we can generate a standard normal random variable.

In [178]:
```
import numpy as np
np.random.randn()
```

Out[178]:
```
0.5051479448557485
```

We can generate an array of a specified amount of standard normal random variables.

In [179]:
```
import numpy as np
np.random.randn(5)
```

Out[179]:
```
array([-1.38831478, -0.599593  ,  0.16769117, -0.8223701 ,  1.3854854 ])
```

We can also generate a matrix of a specified amount of standard normal random variables.

In [180]:
```
import numpy as np
np.random.randn(5,5)
```

Out[180]:
```
array([[ 0.98648339, -1.59624255, -0.61894522, -0.00793183,  0.69409221],
       [-0.02177052, -0.63091489,  1.44987162, -0.21929341, -0.33585627],
       [-0.24545345,  0.24691046, -0.29704841, -0.16921025, -1.13733868],
       [-0.65825252, -0.58066084, -1.83285334,  0.15594937,  0.90661332],
       [ 0.64209131, -0.13903365,  0.43787719,  0.95718837,  0.5131072
5]])
```

We can confirm that if we run millions of standard normal random variables, the average should be close to 0 because the random numbers are distributed with mean 0.

In [181]:
```
import numpy as np
np.random.randn(1000000).mean()
```
Out[181]:
```
-7.731962380773469e-06
```

We an generate a number for a normal random variables where you set the parameters for mean and standard deviation.

In [182]:
```
import numpy as np
np.random.normal(0,10)
```
Out[182]:
```
6.605891681300666
```

For example, we can set standard deviation to zero.

In [183]:
```
import numpy as np
np.random.normal(5,0)
```
Out[183]:
```
5.0
```

We can generate an array of normal random variables.

In [184]:
```
import numpy as np
np.random.normal(0,10,5)
```
Out[184]:
```
array([16.51284211, 16.53958985,  4.0969922 , -5.87356118, -0.65091718])
```

We can generate a random integer between two numbers. For example,

In [185]:
```
import numpy as np
np.random.randint(1,100)
```
Out[185]:
```
36
```

We can generate an array of random integers between two numbers.

In [186]:
```
import numpy as np
np.random.randint(1,100,12)
```
Out[186]:
```
array([95, 18, 75, 97,  5, 74, 19, 60, 66, 42, 84, 96])
```

We can generate a number from the binomial distribution.

In [187]:
```
import numpy as np
np.random.binomial(10,.5)
```
Out[187]:
```
5
```

We can generate an array of numbers from the binomial distribution.

In [188]:
```
import numpy as np
np.random.binomial(10,.5,5)
```

Out[188]:
```
array([4, 3, 7, 6, 7])
```

We can generate one number from a uniform distribution.

In [189]:
```
import numpy as np
np.random.uniform(-10,10)
```

Out[189]:
```
-1.673826080106144
```

We can generate an array of numbers from the uniform distribution.

In [190]:
```
import numpy as np
np.random.uniform(-10,10,5)
```

Out[190]:
```
array([ 5.48989328, -1.09023124,  1.96273596, -4.56224804,  2.55094554])
```

We can re-shape an array into a matrix by specifying the number of rows and columns that we want.

In [191]:
```
import numpy as np
r=np.random.randint(1,100,12)
r=r.reshape(3,4)
r
```

Out[191]:
```
array([[34,  2, 74, 22],
       [64, 22, 86, 25],
       [19, 12, 57, 70]])
```

We can find (and confirm) the shape of a matrix.

In [192]:
```
r.shape
```

Out[192]:
```
(3, 4)
```

We can find the number of dimensions of a matrix.

In [193]:
```
r.ndim
```

Out[193]:
```
2
```

We can find the number of elements of a matrix.

In [194]:
```
r.size
```

Out[194]:
```
12
```

We can find the maximum value of an array.

In [195]:
```
r.max()
```
Out[195]: 86

Alternatively,we can use NumPy to find the maximum value of an array.

In [196]:
```
import numpy as np
np.max(r)
```
Out[196]: 86

We can find the minimum value of an array.

In [197]:
```
r.min()
```
Out[197]: 2

Alternatively, we can use NumPy to find the minimum value of an array.

In [198]:
```
import numpy as np
np.min(r)
```
Out[198]: 2

Often we will care not just about what the maximum element of an array is, but also it's corresponding index. We can display the corresponding index of the maximum value.

In [199]:
```
r.argmax()
```
Out[199]: 6

We can display the corresponding index of the minimum value.

In [200]:
```
r.argmin()
```
Out[200]: 1

We can find the sum of all elements of an array.

In [201]:
```
r.sum()
```
Out[201]: 487

Alternatively, we can use NumPy to find the sum of all elements of an array.

In [202]:
```
import numpy as np
np.sum(r)
```

Out[202]:
```
487
```

We can find the sum of all rows of a matrix.

In [203]:
```
r.sum(axis=1)
```

Out[203]:
```
array([132, 197, 158])
```

We can find the sum of all columns of a matrix.

In [204]:
```
r.sum(axis=0)
```

Out[204]:
```
array([117,  36, 217, 117])
```

We can find the cumulative sum across columns of a matrix.

In [205]:
```
r.cumsum(axis=0)
```

Out[205]:
```
array([[ 34,   2,  74,  22],
       [ 98,  24, 160,  47],
       [117,  36, 217, 117]])
```

We can find the cumulative sum across rows of a matrix.

In [206]:
```
r.cumsum(axis=1)
```

Out[206]:
```
array([[ 34,  36, 110, 132],
       [ 64,  86, 172, 197],
       [ 19,  31,  88, 158]])
```

We can find the standard deviation of an array.

In [207]:
```
r.std()
```

Out[207]:
```
26.787616334584328
```

Alternatively, we can use NumPy to find the standard deviation of an array.

In [208]:
```
import numpy as np
np.std(r)
```

Out[208]:
```
26.787616334584328
```

Similar to what we did on lists, we can slice an array.

In [209]:
```
import numpy as np
r=np.random.randint(1,100,5)
r[0:2]
```

Out[209]:
```
array([44, 89])
```

We can return a specific element of an array using slicing.

In [210]:
```
import numpy as np
r = np.array([[1,2],[3,4]])
r[0][1]
```

Out[210]:
```
2
```

We can change specific values of an array using slicing.

In [211]:
```
r[0:1]=5
r
```

Out[211]:
```
array([[5, 5],
       [3, 4]])
```

We can create a copy of an array.

In [212]:
```
r_copy = r.copy()
```

We can add arrays together.

In [213]:
```
import numpy as np
np.arange(3) + np.arange(3)
```

Out[213]:
```
array([0, 2, 4])
```

Alternatively,

In [214]:
```
import numpy as np
np.add(np.arange(3),np.arange(3))
```

Out[214]:
```
array([0, 2, 4])
```

We can subtract arrays.

In [215]:
```
import numpy as np
np.arange(3) - np.arange(3)
```

Out[215]:
```
array([0, 0, 0])
```

Alternatively,

In [216]:
```
import numpy as np
np.subtract(np.arange(3),np.arange(3))
```

Out[216]:
```
array([0, 0, 0])
```

We can multiply arrays.

In [217]:
```
import numpy as np
np.arange(3) * np.arange(3)
```

Out[217]:
```
array([0, 1, 4])
```

Alternatively,

In [218]:
```
import numpy as np
np.multiply(np.arange(3),np.arange(3))
```

Out[218]:
```
array([0, 1, 4])
```

We can divide arrays.

In [219]:
```
import numpy as np
np.arange(3) / np.arange(3)
```

Out[219]:
```
array([nan,  1.,  1.])
```

Alternatively,

In [220]:
```
import numpy as np
np.divide(np.arange(3),np.arange(3))
```

Out[220]:
```
array([nan,  1.,  1.])
```

Notice that one of the values is "nan." This stands for "not a number." The reason we are getting this value is because we tried to divide by zero, which results in infinity. We can take the reciprocal of an array.

In [221]:
```
import numpy as np
1 / np.arange(3)
```

Out[221]:
```
array([inf, 1. , 0.5])
```

We can raise an array to the power of another array.

In [222]:
```
import numpy as np
base_array=np.array([5,10,15])
raise_to_power=np.array([2,3,4])
np.power(base_array,raise_to_power)
```

Out[222]:
```
array([   25,  1000, 50625])
```

We can square all elements of an array.

In [223]:
```
import numpy as np
np.arange(3)**2
```

Out[223]:
```
array([0, 1, 4])
```

We can take the square root of all elements of an array.

In [224]:
```
import numpy as np
np.sqrt(np.arange(3))
```

Out[224]:
```
array([0.        , 1.        , 1.41421356])
```

We can take the exponential of all elements of an array.

In [225]:
```
import numpy as np
np.exp(np.arange(3))
```

Out[225]:
```
array([1.        , 2.71828183, 7.3890561 ])
```

We can take the logarithm of all elements of an array.

In [226]:
```
import numpy as np
np.log(np.arange(3))
```

Out[226]:
```
array([      -inf, 0.        , 0.69314718])
```

We can confirm that taking the exponential of the logarithm of an array is the same as taking the logarithm of the exponential of an array.

In [227]:
```
import numpy as np
np.exp(np.log(np.arange(3))) == np.log(np.exp(np.arange(3)))
```

Out[227]:
```
array([ True,  True,  True])
```

We can take the absolute value of all elements of an array.

In [228]:
```
import numpy as np
np.abs(np.linspace(-1,1,10))
```

Out[228]:
```
array([1.        , 0.77777778, 0.55555556, 0.33333333, 0.11111111,
       0.11111111, 0.33333333, 0.55555556, 0.77777778, 1.        ])
```

We can take the ceiling of all elements of an array.

In [229]:
```
import numpy as np
np.ceil(np.linspace(-10,10,10))
```

Out[229]:
```
array([-10.,  -7.,  -5.,  -3.,  -1.,   2.,   4.,   6.,   8.,  10.])
```

We can take the floor of all elements of an array.

In [230]:
```
import numpy as np
np.floor(np.linspace(-10,10,10))
```

Out[230]:
```
array([-10.,  -8.,  -6.,  -4.,  -2.,   1.,   3.,   5.,   7.,  10.])
```

We can take the sin of all elements of an array.

In [231]:
```
import numpy as np
np.sin(np.arange(3))
```

Out[231]:
```
array([0.        , 0.84147098, 0.90929743])
```

We can take the cosine of all elements of an array.

In [232]:
```
import numpy as np
np.cos(np.arange(3))
```

Out[232]:
```
array([ 1.        ,  0.54030231, -0.41614684])
```

We can find the percentile of an array.

In [233]:
```
import numpy as np
l=np.arange(10)
np.percentile(l,25)
```

Out[233]:
```
2.25
```

We can check if a number is imaginary.

In [234]:
```
z=(-1)**.5
z.imag
```

Out[234]:
```
1.0
```

We see that an imaginary number correspondings to an imaginary value of 1. Conversely,

In [235]:
```
z=(1)**.5
z.imag
```

Out[235]:
```
0.0
```

Section: Scipy

SciPy is a collection of mathematical algorithms built on top of NumPy. We can import the SciPy library.

In [236]:
```
import scipy as sp
```

We can use the SciPy library to work with the cumulative distribution function (cdf).

In [237]:
```
import scipy.stats as stats
print(stats.norm.cdf(1))
```

```
0.8413447460685429
```

We can show what proportion of the popular would fall in various confidence intervals.

In [238]:
```
import scipy.stats as stats
for i in list(range(1,6)):
    print('Confidence Interval with', i,'standard deviations: ' ,stats.norm
```

```
Confidence Interval with 1 standard deviations:  0.6826894921370859
Confidence Interval with 2 standard deviations:  0.9544997361036416
Confidence Interval with 3 standard deviations:  0.9973002039367398
Confidence Interval with 4 standard deviations:  0.9999366575163338
Confidence Interval with 5 standard deviations:  0.9999994266968562
```

We can find skewness of a normal random variable.

In [239]:
```
import numpy as np
from scipy.stats import skew
x_random = np.random.normal(0, 1, 10000)
print ('Skewness =', skew(x_random))
```

```
Skewness = -0.02397258741240306
```

We can find kurtosis of a normal random variable.

In [240]:
```
import numpy as np
from scipy.stats import kurtosis
x_random = np.random.normal(0, 1, 10000)
print ('Kurtosis =', kurtosis(x_random))
```

```
Kurtosis = -0.07032861118043243
```

Section: Pandas

Pandas is a popular library for working with tables of data, similar to a table in excel. We can import the "pandas" library.

In [241]:
```
import pandas as pd
```

We can turn a list into a series without labels.

In [242]:
```
import pandas as pd
pd.Series(list(range(1,4)))
```

Out[242]:
```
0    1
1    2
2    3
dtype: int64
```

We can turn a list into a series with labels.

In [243]:
```
import pandas as pd
pd.Series(data=list(range(1,4)),index=['a','b','c'])
```

Out[243]:
```
a    1
b    2
c    3
dtype: int64
```

We can add two series together.

In [244]:
```
import pandas as pd
pd.Series(data=list(range(1,4)),index=['a','b','c']) + pd.Series(data=list(
```

Out[244]:
```
a    NaN
b    3.0
c    5.0
d    NaN
dtype: float64
```

We can confirm the type of a series.

In [245]:
```
import pandas as pd
type(pd.Series(list(range(1,4))))
```

Out[245]:
```
pandas.core.series.Series
```

We can convert a series into a dataframe.

In [246]:
```
import pandas as pd
pd.Series(list(range(1,4))).to_frame()
```

Out[246]:

	0
0	1
1	2
2	3

We can confirm the type of our dataframe.

In [247]:
```
import pandas as pd
type(pd.Series(list(range(1,4))).to_frame())
```

Out[247]:
```
pandas.core.frame.DataFrame
```

We can create a dataframe by inputting a list.

In [248]:
```
import pandas as pd
df = pd.DataFrame({'Column A': [1,2,3],
                   'Column B':[4,5,6],
                 'Year':[2018,2019,2020]})
df
```

Out[248]:

	Column A	Column B	Year
0	1	4	2018
1	2	5	2019
2	3	6	2020

It is standard to save a dataframe as "df", although it isn't required. We can return the names of the columns of a dataframe.

In [249]:
```
df.columns
```

Out[249]:
```
Index(['Column A', 'Column B', 'Year'], dtype='object')
```

We can display a column of a dataframe.

In [250]:
```
df['Year']
```

Out[250]:
```
0    2018
1    2019
2    2020
Name: Year, dtype: int64
```

Alternatively,

In [251]:
```
df.Year
```
Out[251]:
```
0    2018
1    2019
2    2020
Name: Year, dtype: int64
```

We can perform an operation on a column of a dataframe.

In [252]:
```
df['Column A'] = df['Column A'] ** 2
df
```
Out[252]:

	Column A	Column B	Year
0	1	4	2018
1	4	5	2019
2	9	6	2020

We could have also created a new column instead of writing over column A.

In [253]:
```
df['Column C'] = df['Column A'] ** 2
df
```
Out[253]:

	Column A	Column B	Year	Column C
0	1	4	2018	1
1	4	5	2019	16
2	9	6	2020	81

We can rank elements of a dataframe in descending order.

In [254]:
```
df.rank(ascending=False)
```
Out[254]:

	Column A	Column B	Year	Column C
0	3.0	3.0	3.0	3.0
1	2.0	2.0	2.0	2.0
2	1.0	1.0	1.0	1.0

We can rank elements of a dataframe in ascending order.

In [255]:
```
df.rank()
```

Out[255]:

	Column A	Column B	Year	Column C
0	1.0	1.0	1.0	1.0
1	2.0	2.0	2.0	2.0
2	3.0	3.0	3.0	3.0

Alternatively,

In [256]:
```
df.rank(ascending=True)
```

Out[256]:

	Column A	Column B	Year	Column C
0	1.0	1.0	1.0	1.0
1	2.0	2.0	2.0	2.0
2	3.0	3.0	3.0	3.0

We can calculate the mean of a specific column of a dataframe.

In [257]:
```
df['Column A'].mean()
```

Out[257]:
```
4.666666666666667
```

We can also calculate the mean of all columns.

In [258]:
```
df.mean()
```

Out[258]:
```
Column A       4.666667
Column B       5.000000
Year        2019.000000
Column C      32.666667
dtype: float64
```

Similarly, we can calculate the standard deviation of a column of a dataframe.

In [259]:
```
df['Column A'].std()
```

Out[259]:
```
4.041451884327381
```

We can also calculate the standard deviation of all columns.

In [260]:
```
df.std()
```

Out[260]:
```
Column A     4.041452
Column B     1.000000
Year         1.000000
Column C    42.524503
dtype: float64
```

We can take the cumulative sum within columns of a dataframe.

In [261]:
```
df.cumsum()
```

Out[261]:

	Column A	Column B	Year	Column C
0	1	4	2018	1
1	5	9	4037	17
2	14	15	6057	98

We can rename the columns of a dataframe.

In [262]:
```
df.rename(columns={'Column A':'Column_A','Column B':'Column_B',
                   'Column C':'Column_C'},inplace=True)
df
```

Out[262]:

	Column_A	Column_B	Year	Column_C
0	1	4	2018	1
1	4	5	2019	16
2	9	6	2020	81

We can rename the columns of a dataframe using a list comprehension.

In [263]:
```
df.columns=[i.lower() for i in df.columns]
df
```

Out[263]:

	column_a	column_b	year	column_c
0	1	4	2018	1
1	4	5	2019	16
2	9	6	2020	81

We can drop a column of a dataframe.

In [264]:
```
df.drop('column_c',axis=1,inplace=True)
df
```

Out[264]:

	column_a	column_b	year
0	1	4	2018
1	4	5	2019
2	9	6	2020

Alternatively, we can drop a column without using "inplace".

In [265]:
```
df = df.drop('column_b',axis=1)
df
```

Out[265]:

	column_a	year
0	1	2018
1	4	2019
2	9	2020

We can display the index of a dataframe.

In [266]:
```
df.index
```

Out[266]:
```
RangeIndex(start=0, stop=3, step=1)
```

We can set a new column to be the index of a dataframe.

In [267]:
```
df.set_index('year',inplace=True)
df
```

Out[267]:

	column_a
year	
2018	1
2019	4
2020	9

We can reset the index.

In [268]:
```
df.reset_index(inplace=True)
df
```

Out[268]:

	year	column_a
0	2018	1
1	2019	4
2	2020	9

We can slice columns of a dataframe.

In [269]:
```
df.set_index('year',inplace=True)
df['column_b'] = [6,8,8]
df[['column_a','column_b']]
```

Out[269]:

	column_a	column_b
year		
2018	1	6
2019	4	8
2020	9	8

We can return the data types of all columns of a dataframe.

In [270]:
```
df.dtypes
```

Out[270]:
```
column_a    int64
column_b    int64
dtype: object
```

We can find the number of unique entries of a column of a dataframe.

In [271]:
```
df['column_a'].nunique()
```

Out[271]:
```
3
```

We can find the number of unique entries of all columns of a dataframe.

In [272]:
```
df.nunique()
```

Out[272]:
```
column_a    3
column_b    2
dtype: int64
```

We can find the set of unique values of a column of a dataframe.

In [273]:
```
df['column_a'].unique()
```
Out[273]:
```
array([1, 4, 9])
```

We can find the value counts of a column of a dataframe.

In [274]:
```
df['column_b'].value_counts()
```
Out[274]:
```
8    2
6    1
Name: column_b, dtype: int64
```

Notice default is set to descending order. Alternatively,

In [275]:
```
df['column_b'].value_counts(ascending=False)
```
Out[275]:
```
8    2
6    1
Name: column_b, dtype: int64
```

We can find the value counts of a column of a dataframe in ascending order.

In [276]:
```
df['column_b'].value_counts(ascending=True)
```
Out[276]:
```
6    1
8    2
Name: column_b, dtype: int64
```

We can also find normalized value counts of a column of a dataframe.

In [277]:
```
df['column_b'].value_counts(normalize=True)
```
Out[277]:
```
8    0.666667
6    0.333333
Name: column_b, dtype: float64
```

We can find the top value counts of a column of a dataframe.

In [278]:
```
df['column_b'].value_counts().nlargest(2)
```
Out[278]:
```
8    2
6    1
Name: column_b, dtype: int64
```

We can find the corresponding indices.

In [279]:
```
df['column_b'].value_counts().nlargest(2).index
```
Out[279]:
```
Int64Index([8, 6], dtype='int64')
```

Similarly, we can find the bottom value counts of a column of a dataframe.

In [280]:
```
df['column_b'].value_counts().nsmallest(2)
```

Out[280]:
```
6    1
8    2
Name: column_b, dtype: int64
```

We can find the corresponding indices.

In [281]:
```
df['column_b'].value_counts().nsmallest(2).index
```

Out[281]:
```
Int64Index([6, 8], dtype='int64')
```

Often we have big datasets and we will want to analyze them in pandas. For example, one of the most popular starting datasets for aspiring datasets is the "Titanic" dataset, which can be found on Kaggle.com (source - [https://www.kaggle.com/c/titanic) (https://www.kaggle.com/c/titanic)](https://www.kaggle.com/c/titanic).

We can read in a csv file and create a dataframe from the data.

In [282]:
```
import pandas as pd
titanic = pd.read_csv('titanic.csv')
```

We can find number of rows of a dataframe.

In [283]:
```
len(titanic)
```

Out[283]:
```
891
```

Alternatively,

In [284]:
```
titanic.shape[0]
```

Out[284]:
```
891
```

We can find the number of columns of a dataframe.

In [285]:
```
len(titanic.columns)
```

Out[285]:
```
13
```

Alternatively,

In [286]:
```
titanic.shape[1]
```

Out[286]:
```
13
```

We can preview the data.

In [287]:
```
titanic.head()
```

Out[287]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	
0	0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.
1	1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.
2	2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.
3	3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.
4	4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.

Notice that it defaults to showing us five rows of data. Alternatively,

In [288]:
```
titanic.head(5)
```

Out[288]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	
0	0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.
1	1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.
2	2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.
3	3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.
4	4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.

We can preview the data with a specified number of rows.

In [289]:
```
titanic.head(3)
```

Out[289]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	
0	0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.
1	1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.
2	2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.

We can preview the bottom of the data.

In [290]:
```
titanic.tail()
```

Out[290]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fa
886	886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.(
887	887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.(
888	888	889	0	?	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4
889	889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.(
890	890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7

Alternatively,

In [291]:
```
titanic.tail(5)
```

Out[291]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fa
886	886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.(
887	887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.(
888	888	889	0	?	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4
889	889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.(
890	890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7

We can also pass in a specified number of rows to observe.

In [292]:
```
titanic.tail(2)
```

Out[292]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
889	889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.00
890	890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.75

We can remove duplicate rows from our dataframe, if there are any.

In [293]:
```
titanic[titanic.duplicated()]
```

Out[293]:

Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin

We can find the number of null values by column of a dataframe.

In [294]:
```
titanic.isna().sum()
```

Out[294]:
```
Unnamed: 0       0
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

Alternatively,

In [295]:
```
titanic.isnull().sum()
```

Out[295]:
```
Unnamed: 0       0
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

We can drop all rows of a dataframe that contain null values.

In [296]:
```
titanic.dropna(inplace=True)
```

We can find descriptive statistics by column of a dataframe.

In [297]:
```
titanic.describe()
```

Out[297]:

	Unnamed: 0	PassengerId	Survived	Age	SibSp	Parch	Fare
count	183.000000	183.000000	183.000000	183.000000	183.000000	183.000000	183.000000
mean	454.366120	455.366120	0.672131	35.674426	0.464481	0.475410	78.682469
std	247.052476	247.052476	0.470725	15.643866	0.644159	0.754617	76.347843
min	1.000000	2.000000	0.000000	0.920000	0.000000	0.000000	0.000000
25%	262.500000	263.500000	0.000000	24.000000	0.000000	0.000000	29.700000
50%	456.000000	457.000000	1.000000	36.000000	0.000000	0.000000	57.000000
75%	675.000000	676.000000	1.000000	47.500000	1.000000	1.000000	90.000000
max	889.000000	890.000000	1.000000	80.000000	3.000000	4.000000	512.329200

We can find column-wide information on the data types.

In [298]:
```
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 183 entries, 1 to 889
Data columns (total 13 columns):
Unnamed: 0     183 non-null int64
PassengerId    183 non-null int64
Survived       183 non-null int64
Pclass         183 non-null object
Name           183 non-null object
Sex            183 non-null object
Age            183 non-null float64
SibSp          183 non-null int64
Parch          183 non-null int64
Ticket         183 non-null object
Fare           183 non-null float64
Cabin          183 non-null object
Embarked       183 non-null object
dtypes: float64(2), int64(5), object(6)
memory usage: 20.0+ KB
```

We can find the types of each column of a dataframe.

In [299]:
```
titanic.dtypes
```

Out[299]:
```
Unnamed: 0       int64
PassengerId      int64
Survived         int64
Pclass          object
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
```

We can find the quantile of a column of a dataframe.

In [300]:
```
titanic['Age'].quantile(.65)
```

Out[300]:
```
41.3
```

We can also display several quantiles of a dataframe at the same time.

In [301]:
```
titanic['Age'].quantile([.25,.75])
```

Out[301]:
```
0.25    24.0
0.75    47.5
Name: Age, dtype: float64
```

We can use numpy to create conditionals of a dataframe. For instance, we can create a new variable based on a specified condition.

In [302]:
```
import numpy as np
titanic['IsAdult']=np.where(titanic['Age']>18,1,0)
titanic['IsAdult'].head(4)
```

Out[302]:
```
1     1
3     1
6     1
10    0
Name: IsAdult, dtype: int64
```

We can take a random sample of a dataframe by choosing the proportion.

In [303]:
```
titanic.sample(frac=1/100)
```

Out[303]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
23	23	24	1	1	Sloper, Mr. William Thompson	male	28.0	0	0	113788	35.5
3	3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1

Alternatively, we can take a random sample of a dataframe by passing in a sample size.

In [304]:
```
titanic.sample(n=2)
```

Out[304]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
853	853	854	1	1	Lines, Miss. Mary Conover	female	16.0	0	1	PC 17592	39.4
487	487	488	0	1	Kent, Mr. Edward Austin	male	58.0	0	0	11771	29.7

We could create a column of random integers for each index.

In [305]:
```
import numpy as np
df['randomize'] = np.random.randint(0,100,size=(len(df)))
df['randomize'].head(3)
```

Out[305]:
```
year
2018    66
2019    17
2020    91
Name: randomize, dtype: int64
```

We can move a column to the front of a dataframe.

In [306]:
```
randomize = df['randomize']
df.drop(labels=['randomize'], axis=1,inplace = True)
df.insert(0, 'randomize', randomize)
df.head(3)
```

Out[306]:

	randomize	column_a	column_b
year			
2018	66	1	6
2019	17	4	8
2020	91	9	8

We can take the transpose of a dataframe.

In [307]:
```
df.T.head(3)
```

Out[307]:

year	2018	2019	2020
randomize	66	17	91
column_a	1	4	9
column_b	6	8	8

We can write a lambda function on a dataframe.

In [308]:
```
df['column_a_sq'] = df['column_a'].map(lambda x: x**2)
df
```

Out[308]:

	randomize	column_a	column_b	column_a_sq
year				
2018	66	1	6	1
2019	17	4	8	16
2020	91	9	8	81

We can work with dates in pandas as well. For example, we can extract the days from a string of dates of a dataframe using lambda functions.

In [309]:
```
import pandas as pd
dates = pd.Series(['12-01-2020', '12-02-2020', '12-03-2020', '12-04-2020'])
dates.map(lambda x: x.split('-')[1])
```

Out[309]:
```
0    01
1    02
2    03
3    04
dtype: object
```

We can turn a string into a datetime object.

In [310]:
```
import datetime as dt
df.reset_index()
df['dates']=['2017-03-24','2018-04-01','2019-07-28']
df['dates']=pd.to_datetime(df['dates'])
```

We can extract the year from our datetime index.

In [311]:
```
import datetime as dt
df['Year']=df['dates'].dt.year
df
```

Out[311]:

	randomize	column_a	column_b	column_a_sq	dates	Year
year						
2018	66	1	6	1	2017-03-24	2017
2019	17	4	8	16	2018-04-01	2018
2020	91	9	8	81	2019-07-28	2019

We can extract the month from our datetime index.

In [312]:
```
import datetime as dt
df['Month']=df['dates'].dt.month
df
```

Out[312]:

	randomize	column_a	column_b	column_a_sq	dates	Year	Month
year							
2018	66	1	6	1	2017-03-24	2017	3
2019	17	4	8	16	2018-04-01	2018	4
2020	91	9	8	81	2019-07-28	2019	7

We can extract the day from our datetime index.

In [313]:
```
import datetime as dt
df['Day']=df['dates'].dt.day
df
```

Out[313]:

	randomize	column_a	column_b	column_a_sq	dates	Year	Month	Day
year								
2018	66	1	6	1	2017-03-24	2017	3	24
2019	17	4	8	16	2018-04-01	2018	4	1
2020	91	9	8	81	2019-07-28	2019	7	28

We can sort values of a dataframe.

In [314]:
```
df.sort_values('Day')
```

Out[314]:

	randomize	column_a	column_b	column_a_sq	dates	Year	Month	Day
year								
2019	17	4	8	16	2018-04-01	2018	4	1
2018	66	1	6	1	2017-03-24	2017	3	24
2020	91	9	8	81	2019-07-28	2019	7	28

Alternatively,

In [315]:
```
df.sort_values(by='Day')
```

Out[315]:

	randomize	column_a	column_b	column_a_sq	dates	Year	Month	Day
year								
2019	17	4	8	16	2018-04-01	2018	4	1
2018	66	1	6	1	2017-03-24	2017	3	24
2020	91	9	8	81	2019-07-28	2019	7	28

We can save a dataframe to a csv file.

In [316]:
```
df.to_csv('saved_csv_file.csv')
```

We can save a dataframe to a csv file without including the index.

In [317]:
```
df.to_csv('saved_csv_file.csv',index=False)
```

We can save a dataframe to an excel file.

In [318]:
```
df.to_excel('saved_excel_file.xlsx')
```

We can save a dataframe to an excel file without including the index.

In [319]:
```
df.to_excel('saved_excel_file.xlsx',index=False)
```

We can read in a csv file with label-encoding.

In [320]:
```
import pandas as pd
df=pd.read_csv('saved_csv_file.csv',encoding='latin-1')
df
```

Out[320]:

	randomize	column_a	column_b	column_a_sq	dates	Year	Month	Day
0	66	1	6	1	2017-03-24	2017	3	24
1	17	4	8	16	2018-04-01	2018	4	1
2	91	9	8	81	2019-07-28	2019	7	28

We can read in an excel file.

In [321]:
```
import pandas as pd
df=pd.read_excel('saved_excel_file.xlsx')
df
```

Out[321]:

	randomize	column_a	column_b	column_a_sq	dates	Year	Month	Day
0	66	1	6	1	2017-03-24	2017	3	24
1	17	4	8	16	2018-04-01	2018	4	1
2	91	9	8	81	2019-07-28	2019	7	28

We can read in an excel file and specify the sheet name to display.

In [322]:
```
import pandas as pd
df=pd.read_excel('excel_multiple_sheets.xlsx',sheet_name='SecondSheet')
df.head(3)
```

Out[322]:

	year	column aa	column bb	column d	Percent Change Column B
0	2017	1	24	0.333333	NaN
1	2018	2	30	0.333333	0.25
2	2019	3	36	0.333333	0.20

We can view all sheet names within an excel file.

In [323]:
```
import pandas as pd
workbook=pd.ExcelFile('excel_multiple_sheets.xlsx')
workbook.sheet_names
```

Out[323]:
```
['FirstSheet', 'SecondSheet']
```

We can import in a specific worksheet.

In [324]:
```
df = workbook.parse(sheet_name='FirstSheet')
```

We can skip the first several rows of a dataframe.

In [325]:
```
import pandas as pd
df=pd.read_excel('saved_excel_file.xlsx',skiprows=1)
df
```

Out[325]:

	66	1	6	1.1	2017-03-24 00:00:00	2017	3	24
0	17	4	8	16	2018-04-01	2018	4	1
1	91	9	8	81	2019-07-28	2019	7	28

We can skip the last several rows of a dataframe.

In [326]:
```
import pandas as pd
df=pd.read_excel('saved_excel_file.xlsx',skipfooter=2)
df
```

Out[326]:

	randomize	column_a	column_b	column_a_sq	dates	Year	Month	Day
0	66	1	6	1	2017-03-24	2017	3	24

We can read in specific columns of a dataframe.

In [327]:
```
import pandas as pd
df=pd.read_excel('saved_excel_file.xlsx',usecols=[0,1,3])
df
```

Out[327]:

	randomize	column_a	column_a_sq
0	66	1	1
1	17	4	16
2	91	9	81

We can set the index when reading in a dataframe.

In [328]:
```
import pandas as pd
data=pd.read_excel('saved_excel_file.xlsx',index_col=0)
data
```

Out[328]:

	column_a	column_b	column_a_sq	dates	Year	Month	Day
randomize							
66	1	6	1	2017-03-24	2017	3	24
17	4	8	16	2018-04-01	2018	4	1
91	9	8	81	2019-07-28	2019	7	28

We can read in an html file.

In [329]:
```
import pandas as pd
data=pd.read_html('https://www.fdic.gov/bank/individual/failed/banklist.htm
data[0].head()
```

Out[329]:

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date
0	The First State Bank	Barboursville	WV	14361	MVB Bank, Inc.	April 3, 2020
1	Ericson State Bank	Ericson	NE	18265	Farmers and Merchants Bank	February 14, 2020
2	City National Bank of New Jersey	Newark	NJ	21111	Industrial Bank	November 1, 2019
3	Resolute Bank	Maumee	OH	58317	Buckeye State Bank	October 25, 2019
4	Louisa Community Bank	Louisa	KY	58112	Kentucky Farmers Bank Corporation	October 25, 2019

We can read in a txt file.

In [330]:
```
import pandas as pd
df_txt=pd.read_csv('bp.txt',delimiter='\t')
df_txt.head(3)
```

Out[330]:

	Pt	BP	Age	Weight	BSA	Dur	Pulse	Stress
0	1	105	47	85.4	1.75	5.1	63	33
1	2	115	49	94.2	2.10	3.8	70	14
2	3	116	49	95.3	1.98	8.2	72	10

We can slice a dataframe based on the index location, using "iloc."

In [331]:
```
df_txt.iloc[2]
```

Out[331]:
```
Pt           3.00
BP         116.00
Age         49.00
Weight      95.30
BSA          1.98
Dur          8.20
Pulse       72.00
Stress      10.00
Name: 2, dtype: float64
```

We can see that this displayed the third row, which makes sense, because indexing starts at row 0. Similarly, we can slice both rows and columns together.

In [332]:
```
df_txt.iloc[0:,:2].head(3)
```

Out[332]:

	Pt	BP
0	1	105
1	2	115
2	3	116

We can also use "loc" for slicing which is based on criteria that are not indices.

In [333]:
```
df.loc[0:,['column_a','column_a_sq']]
```

Out[333]:

	column_a	column_a_sq
0	1	1
1	4	16
2	9	81

We can slice based on several criteria with an and statement.

In [334]:
```
df.loc[(df['column_a']>1.5) & (df['column_a_sq']<100)]
```

Out[334]:

	randomize	column_a	column_a_sq
1	17	4	16
2	91	9	81

We can slice based on several criteria with an or statement.

In [335]:
```
df.loc[(df['column_a']>1.5) | (df['column_a_sq']<100)]
```

Out[335]:

	randomize	column_a	column_a_sq
0	66	1	1
1	17	4	16
2	91	9	81

We can slice based on a specified criteria for the value of a certain column.

In [336]:
```
df.loc[df['column_a']==4,:]
```

Out[336]:

	randomize	column_a	column_a_sq
1	17	4	16

We can slice based on criteria included in a list of values.

In [337]:
```
list_to_pull=[1,4]
df.loc[df['column_a'].isin(list_to_pull),:]
```

Out[337]:

	randomize	column_a	column_a_sq
0	66	1	1
1	17	4	16

We can slice based on string conditionals.

In [338]:
```
df['column_b']=['hello','good morning','good night']
df.loc[df['column_b'].str.contains('good'),:]
```

Out[338]:

	randomize	column_a	column_a_sq	column_b
1	17	4	16	good morning
2	91	9	81	good night

We can change the type of a column of a dataframe.

In [339]:
```
df['prices']=['$3','$4','$5']
df['prices']=df['prices'].str.replace('$','').astype(float)
df
```

Out[339]:

	randomize	column_a	column_a_sq	column_b	prices
0	66	1	1	hello	3.0
1	17	4	16	good morning	4.0
2	91	9	81	good night	5.0

We can replace a value in a dataframe.

In [340]:
```
df=df.replace(1,1000)
df
```

Out[340]:

	randomize	column_a	column_a_sq	column_b	prices
0	66	1000	1000	hello	3.0
1	17	4	16	good morning	4.0
2	91	9	81	good night	5.0

We can replace null values in a dataframe with a string.

In [341]:
```
import numpy as np
df['column_c']=np.nan
df.fillna(value='FILL VALUE')
```

Out[341]:

	randomize	column_a	column_a_sq	column_b	prices	column_c
0	66	1000	1000	hello	3.0	FILL VALUE
1	17	4	16	good morning	4.0	FILL VALUE
2	91	9	81	good night	5.0	FILL VALUE

Alternatively, we can replace null values in a dataframe with numeric values.

In [342]:
```
df['column_c'].fillna(value=df['randomize'].mean())
```

Out[342]:
```
0    58.0
1    58.0
2    58.0
Name: column_c, dtype: float64
```

We can create a dataframe filled entirely with random numbers.

In [343]:
```
import numpy as np
df=pd.DataFrame(np.random.randn(5,4),['A','B','C','D','E'],['W','X','Y','Z'
df
```

Out[343]:

	W	X	Y	Z
A	1.664322	0.726047	-0.192614	-1.319056
B	2.342958	1.005876	1.001243	-0.280369
C	-1.156685	-0.873723	-1.032086	0.471776
D	0.173310	-1.410176	-0.228439	-0.879408
E	-0.694403	1.431919	-0.287195	-1.561931

We can drop a specific row of a dataframe.

In [344]:
```
df.drop('E',axis=0,inplace=True)
```

We can drop a specific column of a dataframe.

In [345]:
```
df.drop('X',axis=1,inplace=True)
```

We can use booleans to find which values of a dataframe are positive.

In [346]:
```
booldf = df>0
booldf
```

Out[346]:

	W	Y	Z
A	True	False	False
B	True	True	False
C	False	False	True
D	True	False	False

We can display positive values of a dataframe with negative values rounded to zero.

In [347]:
```
import numpy as np
round(df[booldf].replace(np.nan,0),2)
```

Out[347]:

	W	Y	Z
A	1.66	0.0	0.00
B	2.34	1.0	0.00
C	0.00	0.0	0.47
D	0.17	0.0	0.00

We can find the index corresponding to the maximum value of a dataframe.

In [348]:
```
df['W'].idxmax()
```

Out[348]:
```
'B'
```

We can find the index corresponding to the minimum value of a dataframe

In [349]:
```
df['W'].idxmin()
```

Out[349]:
```
'C'
```

We can display a matrix of correlations of values.

In [350]:
```
df.corr()
```

Out[350]:

	W	Y	Z
W	1.000000	0.895514	-0.558128
Y	0.895514	1.000000	-0.276490
Z	-0.558128	-0.276490	1.000000

We can display a heatmap of a matrix of correlations of values.

In [351]:
```
import seaborn as sns
sns.heatmap(df.corr(),annot=True,cmap='coolwarm')
```

Out[351]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12a8e6c18>
```

We can perform a groupby on the mean of numeric columns.

In [352]:
```
titanic.groupby('Sex').mean()
```

Out[352]:

Sex	Unnamed: 0	PassengerId	Survived	Age	SibSp	Parch	Fare	IsAdult
female	460.818182	461.818182	0.931818	32.676136	0.534091	0.545455	89.000900	0.840909
male	448.389474	449.389474	0.431579	38.451789	0.400000	0.410526	69.124343	0.905263

We notice that certain columns were left off, because groupby only displays numeric columns. We can perform a groupby on the median of numeric columns.

In [353]:
```
titanic.groupby('Sex').median()
```

Out[353]:

Sex	Unnamed: 0	PassengerId	Survived	Age	SibSp	Parch	Fare	IsAdult
female	454	455	1	32.25	0	0	77.9583	1
male	456	457	0	37.00	0	0	51.8625	1

We can perform a groupby on the sum of numeric columns.

In [354]:
```
titanic.groupby('Survived').sum()
```

Out[354]:

Survived	Unnamed: 0	PassengerId	Age	SibSp	Parch	Fare	IsAdult
0	24119	24179	2481.00	22	27	3842.8957	57
1	59030	59153	4047.42	63	60	10555.9961	103

We can perform a groupby to describe a dataframe.

In [355]:
```
titanic.groupby('Sex').describe().T.head(3)
```

Out[355]:

	Sex	female	male
Unnamed: 0	count	88.000000	95.000000
	mean	460.818182	448.389474
	std	247.666236	247.645177

We can perform a groupby with several criteria.

In [356]:
```
titanic.groupby(['Sex','IsAdult']).mean()
```

Out[356]:

Sex	IsAdult	Unnamed: 0	PassengerId	Survived	Age	SibSp	Parch	Fare
female	0	473.142857	474.142857	0.857143	12.642857	0.714286	1.000000	104.884229
	1	458.486486	459.486486	0.945946	36.466216	0.500000	0.459459	85.995946
male	0	452.666667	453.666667	0.888889	6.991111	0.777778	1.333333	75.185178
	1	447.941860	448.941860	0.383721	41.744186	0.360465	0.313953	68.490070

We can concatenate dataframes onto each other.

In [357]:
```
import pandas as pd
df1=pd.DataFrame({'A':['A0','A1'],
                  'B':['B0','B1']},
                 index=[0,1])
df2=pd.DataFrame({'A':['A2','A3'],
                  'B':['B2','B3']},
                 index=[2,3])
pd.concat([df1,df2])
```

Out[357]:

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

We can merge dataframes with the inner method.

In [358]:
```
import pandas as pd
left=pd.DataFrame({'key':['K0','K1','K2','K3'],
                  'A':['A0','A1','A2','A3'],
                  'B':['B0','B1','B2','B3']})
right=pd.DataFrame({'key':['K0','K1','K2','K3'],
                   'C':['C0','C1','C2','C3'],
                   'D':['D0','D1','D2','D3']})
pd.merge(left,right,how='inner',on='key')
```

Out[358]:

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

We can merge dataframes with the outer method.

In [359]:
```
import pandas as pd
pd.merge(left,right,how='outer',on='key')
```

Out[359]:

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

We can create a pivot table.

In [360]:
```
titanic.pivot_table(index='IsAdult',columns='Sex',values='Fare')
```

Out[360]:

Sex	female	male
IsAdult		
0	104.884229	75.185178
1	85.995946	68.490070

We might want to run statistical tests to test our theories. For example, we might want to explore whether or not technology stocks outperform health care stocks, at a statistically significant level, based on historical data. We can create a hypothesis test. The null hypothesis (H0) is that there is no significant difference in returns between technology stocks and health care stocks. The alternative hypothesis (Ha) is that there is a significant difference in returns between technology stocks and health care stocks. We can run the hypothesis test.

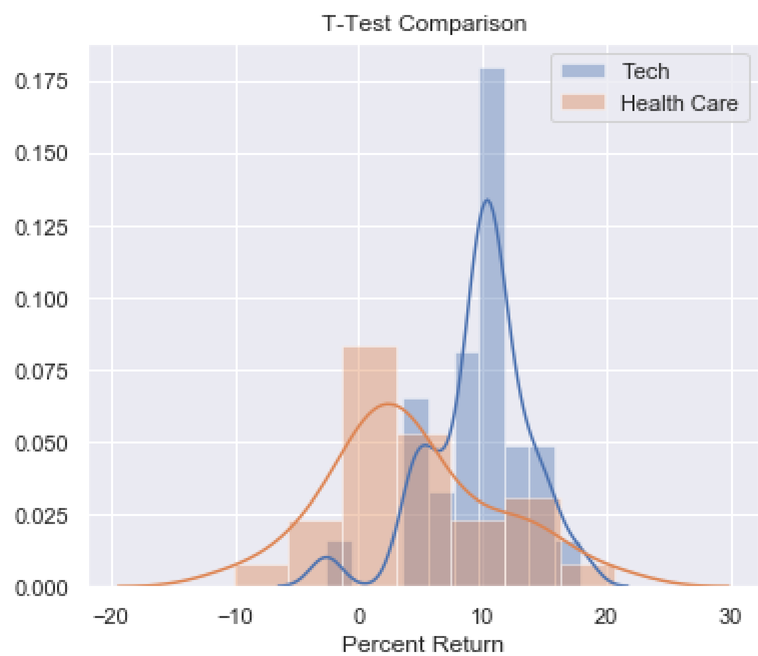
In [361]:
```
import pandas as pd
from scipy import stats
tech_returns=pd.read_excel('Hypothesis_Test.xlsx',sheetname='Tech')
health_care_returns=pd.read_excel('Hypothesis_Test.xlsx',
                                  sheetname='Health_Care')
control=tech_returns['Percent Return']
experimental=health_care_returns['Percent Return']
stats.ttest_ind(experimental, control,equal_var=False)
```

Out[361]:
```
Ttest_indResult(statistic=-3.7585177072659626, pvalue=0.00047066769250311
336)
```

We see that the p-value is extremely small, far less than a typical threshold (usually 5%, but up to the discretion of the tester. We therefore reject the null hypothesis. Results are statistically significant with p-value nearly 0. Tech stocks have a significantly different returns profile than health care stocks, based on this data. We can create a plot of this (we will come back to plotting in a later section).

In [362]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(color_codes=True)
sns.set(rc={'figure.figsize':(6,5)})
sns.distplot(control)
sns.distplot(experimental)
plt.title('T-Test Comparison',fontsize=12)
plt.legend(['Tech','Health Care'])
```

Out[362]: <matplotlib.legend.Legend at 0x12aa8fb38>



We can run an ANOVA test.

In [363]:
```
'''
import pandas as pd
import statsmodels.api as sm
from statsmodels.formula.api import ols
df = pd.read_csv('IT_salaries.csv')
formula = 'S ~ C(E) + C(M) + X'
lm = ols(formula, df).fit()
table = sm.stats.anova_lm(lm, typ=2)
print(table)
'''
```

Out[363]:
```
"\nimport pandas as pd\nimport statsmodels.api as sm\nfrom statsmodels.fo
rmula.api import ols\ndf = pd.read_csv('IT_salaries.csv')\nformula = 'S ~
C(E) + C(M) + X'\nlm = ols(formula, df).fit()\ntable = sm.stats.anova_lm
(lm, typ=2)\nprint(table)\n"
```

The rightmost column shows the probability that the factor is influential. All values above are far less than an alpha of .05 indicating rejection of the null hypothesis. All factors appear influential, and management appears the most influential.

Section: Visualization

The Matplotlib library is popular for visualizations. We can import the Matplotlib library.

In [364]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

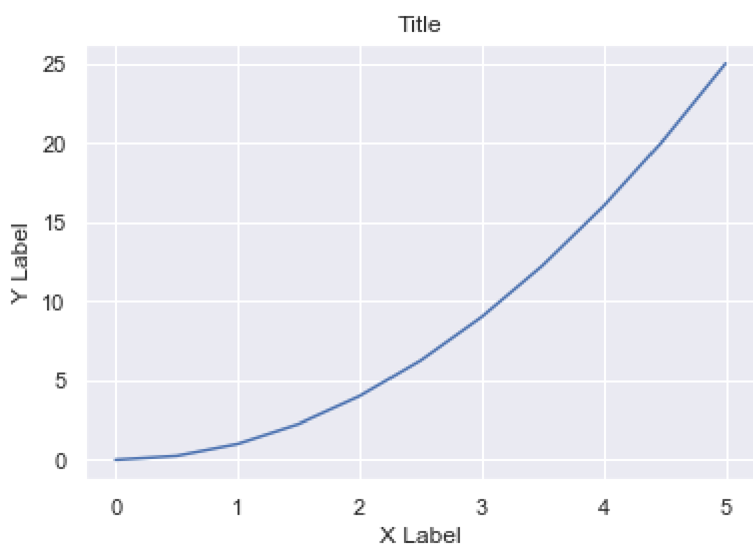
Notice the "%matplotlib inline" command, which outputs visualizations within this jupyter notebook.

The Seaborn library is often used in conjunction with Matplotlib to make visualizations more appealing. We can import the Seaborn library.

In [365]:
```
import seaborn as sns
```

We can plot a line graph with a solid line.

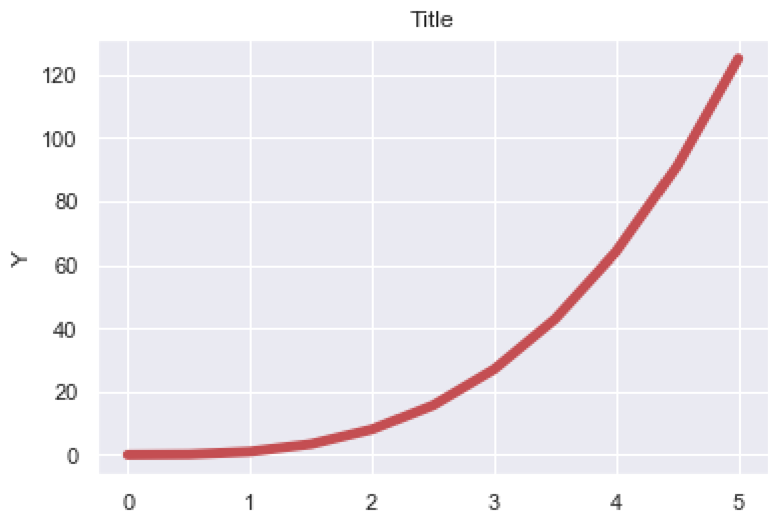
In [366]:
```
import matplotlib.pyplot as plt
%matplotlib inline
x=np.linspace(0,5,11)
y=x**2
plt.plot(x,y)
plt.xlabel('X Label')
plt.ylabel('Y Label')
plt.title('Title')
plt.show()
```



We can plot a line graph with a heavier line.

In [367]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
x=np.linspace(0,5,11)
y=x**3
plt.plot(x,y,color='r',linewidth=5)
plt.title('Title')
plt.ylabel('Y')
```

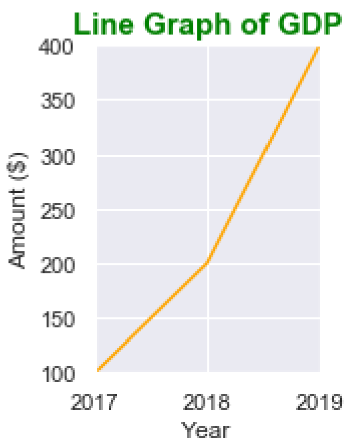
Out[367]: Text(0, 0.5, 'Y')



We can plot a line graph by taking in lists of values.

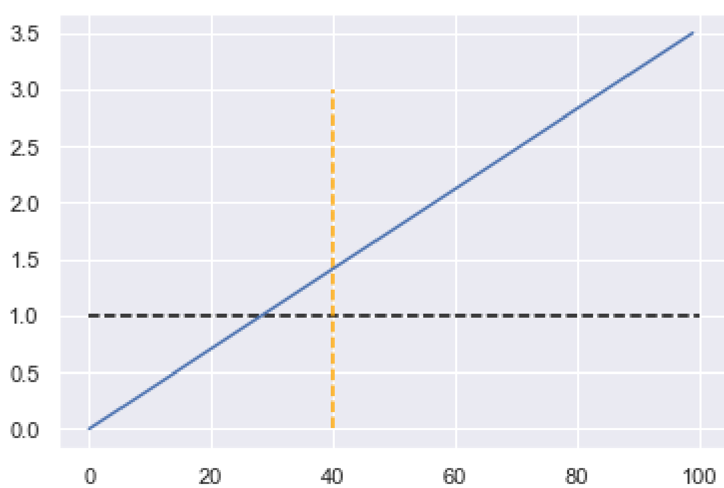
In [368]:
```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
gdp=[100,200,400]
year=list(range(2017,2020))
line_g=pd.DataFrame({'gdp':gdp},index=year)
line_g.plot(figsize=(2,3),color='orange',legend=False)
plt.title('Line Graph of GDP',fontsize=15,fontweight='bold',color='green')
plt.xlabel('Year')
plt.ylabel('Amount ($)')
plt.ylim(100,400)
```

Out[368]:
```
(100, 400)
```



We can plot a line graph with dashed lines.

In [369]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
x = np.linspace(0, 3.5 , 100)
plt.plot(x)
plt.hlines(y=1, xmin=0, xmax=100, linestyle = "dashed", color= 'black')
plt.vlines(x=40, ymin=0, ymax=3, linestyle = "dashed", color= 'orange')
plt.show()
```



We can plot a bar graph.

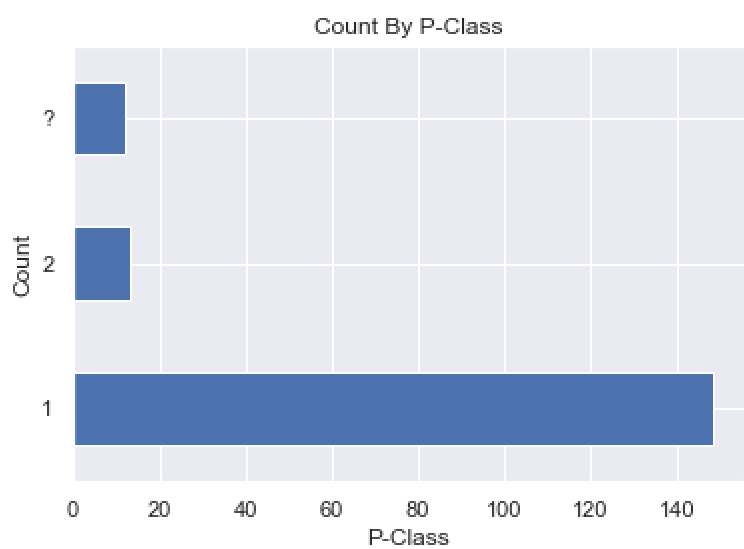
In [370]:
```
import matplotlib.pyplot as plt
%matplotlib inline
titanic['Pclass'].value_counts().head(3).plot.bar(title='Count By P-Class')
plt.xlabel('P-Class')
plt.ylabel('Count')
```

Out[370]: `Text(0, 0.5, 'Count')`

We can also plot a horizontal bar graph.

In [371]:
```
import matplotlib.pyplot as plt
%matplotlib inline
titanic['Pclass'].value_counts().head(3).plot.barh(title='Count By P-Class'
plt.xlabel('P-Class')
plt.ylabel('Count')
```

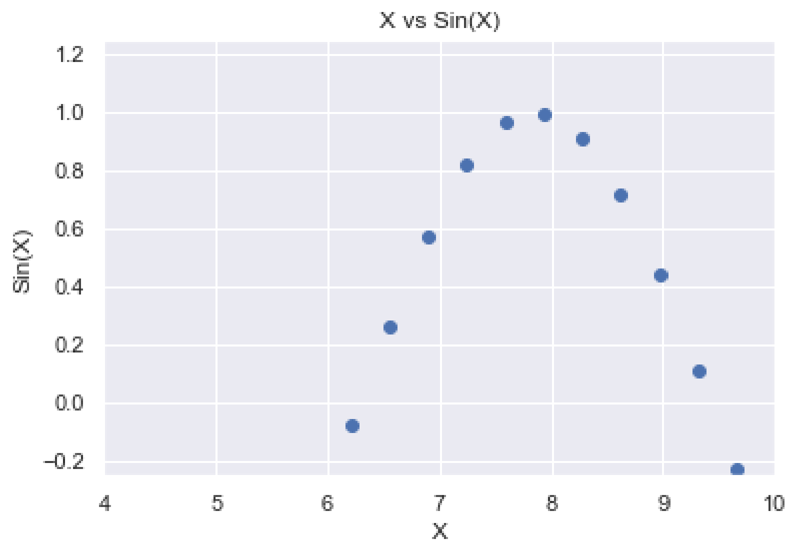
Out[371]: `Text(0, 0.5, 'Count')`



We can create a scatter plot.

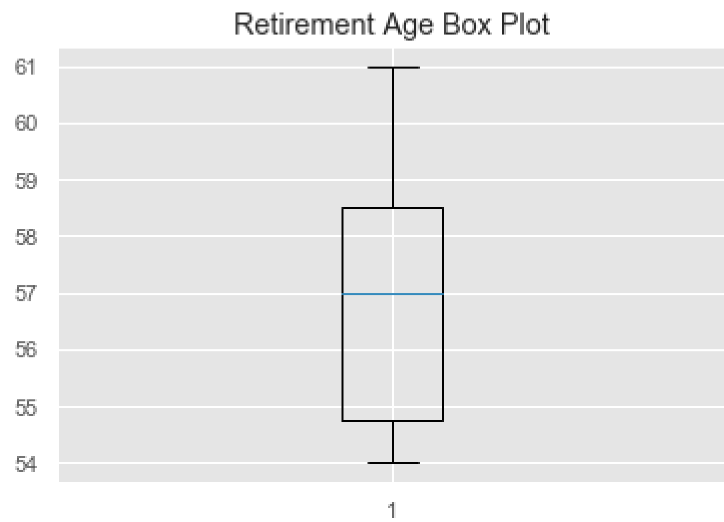
In [372]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
x=np.linspace(0,10,30)
y=np.sin(x)
plt.scatter(x,y)
plt.ylim(-0.25,1.25)
plt.xlim(4,10)
plt.title('X vs Sin(X)')
plt.xlabel('X')
plt.ylabel('Sin(X)')
```

Out[372]:
```
Text(0, 0.5, 'Sin(X)')
```



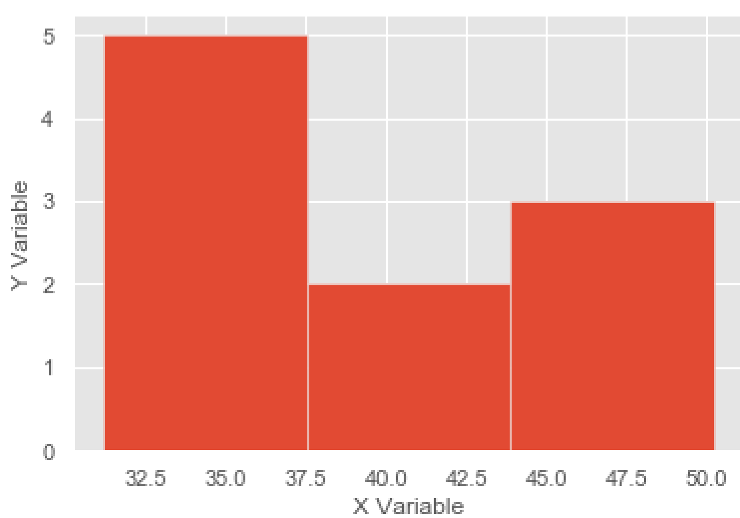
We can create a GG-Plot.

In [373]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
x=[54,54,54,55,56,57,57,58,58,60,61,88]
plt.boxplot(x,showfliers=False)
plt.title('Retirement Age Box Plot')
plt.show()
```



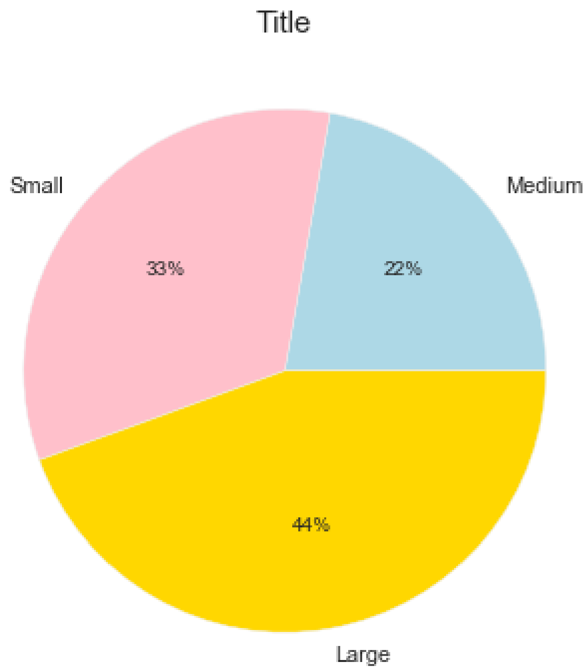
We can plot a histogram by taking in a list.

In [374]:
```
import matplotlib.pyplot as plt
%matplotlib inline
x=[43.1,35.6,37.5,36.5,45.3, 40.3,50.2,47.3,31.2,36.5]
plt.hist(x,bins=3)
plt.xlabel('X Variable')
plt.ylabel('Y Variable')
plt.show()
```



We can create a pie chart.

In [375]:
```
import matplotlib.pyplot as plt
%matplotlib inline
labels=['Medium','Small','Large']
sizes=[100,150,200]
colors=['lightblue','pink','gold']
explode=[0,0,0,]
plt.style.use('seaborn-pastel')
plt.figure(figsize=(6,6))
plt.pie(sizes,explode=explode,labels=labels,colors=colors,autopct= '%1.0f%%
plt.title('Title',fontsize=15)
plt.show()
labels
```

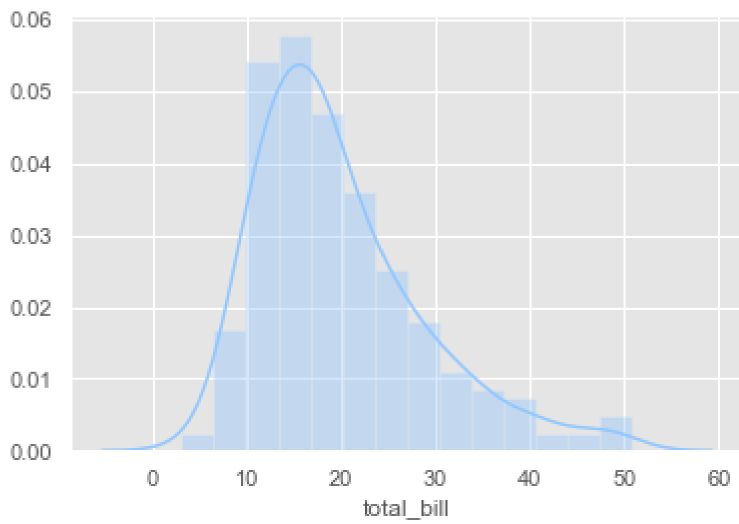


Out[375]:
```
['Medium', 'Small', 'Large']
```

We can plot a kernel density estimate (KDE).

In [376]:
```
import seaborn as sns
tips=sns.load_dataset('tips')
sns.distplot(tips['total_bill'])
```

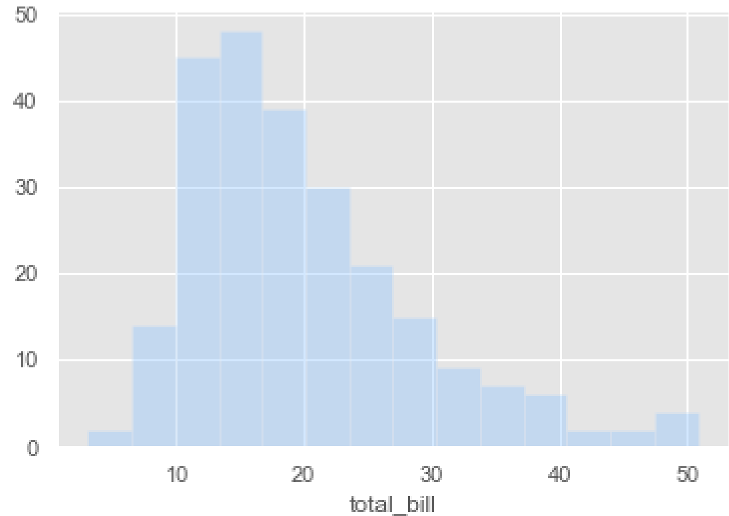
Out[376]: <matplotlib.axes._subplots.AxesSubplot at 0x12afd0d68>



We can plot a distribution without showing the KDE (essentially a histogram in seaborn).

In [377]:
```
import seaborn as sns
sns.distplot(tips['total_bill'],kde=False)
```

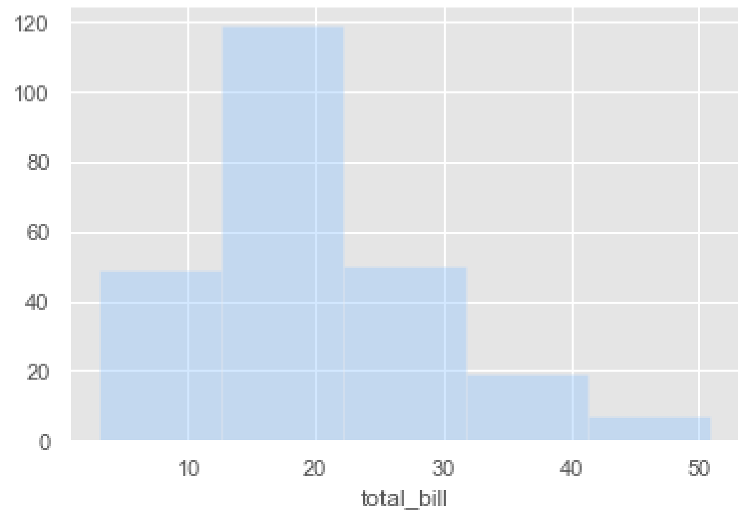
Out[377]: <matplotlib.axes._subplots.AxesSubplot at 0x12b2f4d68>



We can plot a distribution without showing the KDE (essentially a histogram in seaborn) and set the number of bins.

In [378]:
```
import seaborn as sns
sns.distplot(tips['total_bill'],kde=False,bins=5)
```

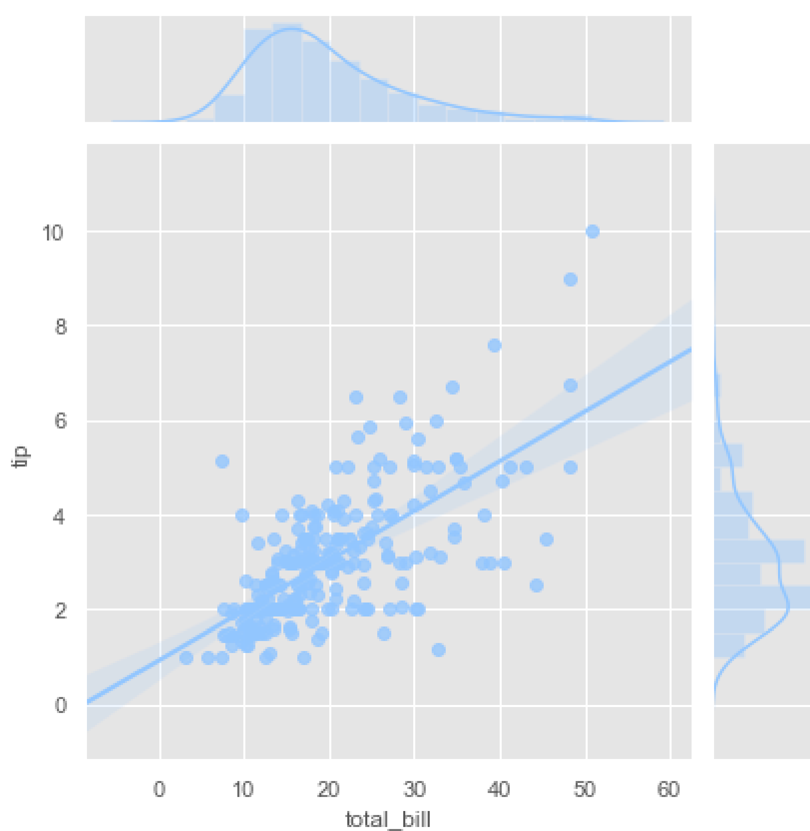
Out[378]: <matplotlib.axes._subplots.AxesSubplot at 0x12b3d5e80>



We can plot a joint-plot of two distributions.

In [379]:
```
import seaborn as sns
sns.jointplot(x='total_bill',y="tip",data=tips,kind='reg')
```

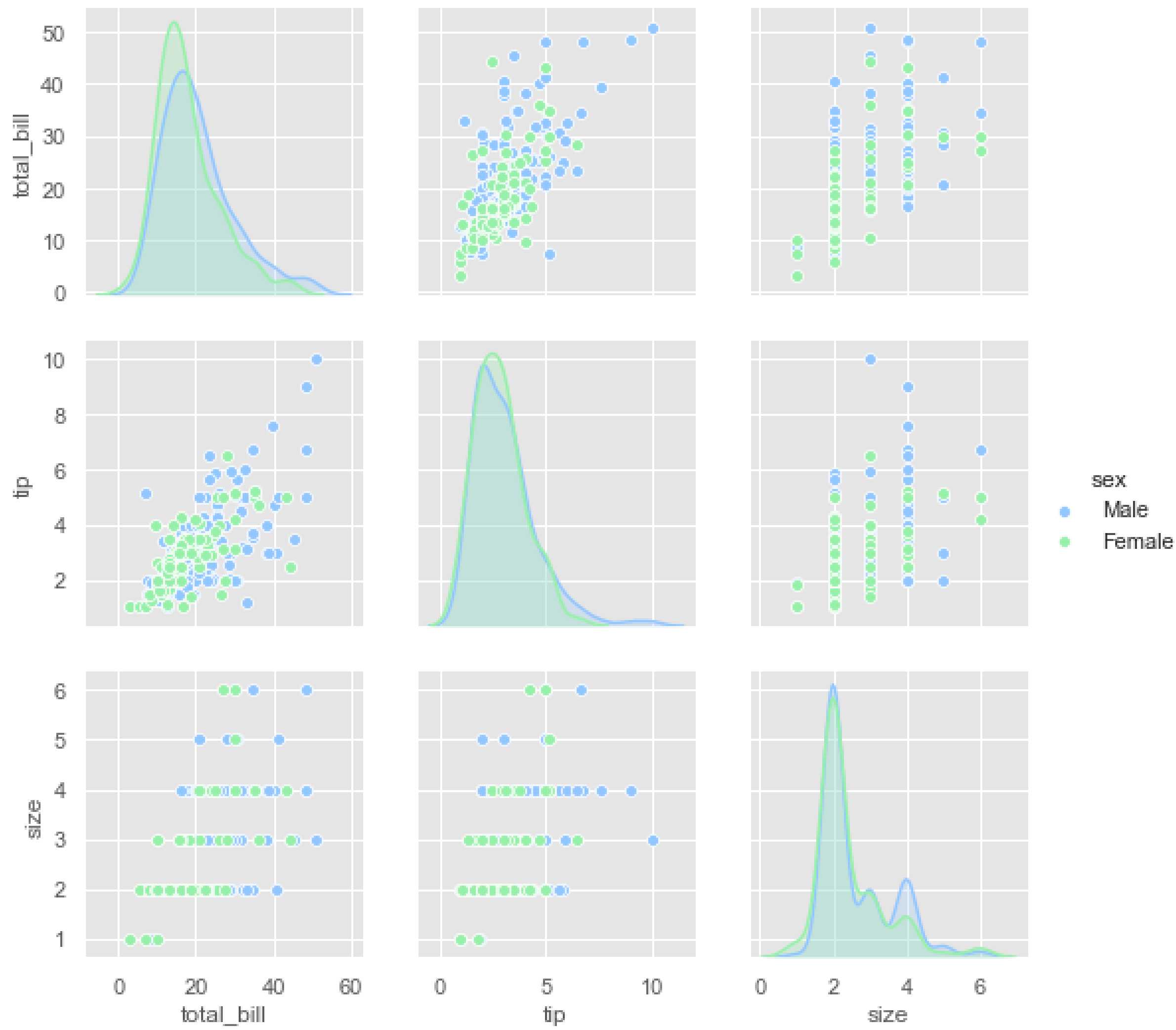
Out[379]: <seaborn.axisgrid.JointGrid at 0x12b4955f8>



We can create a pair plot.

In [380]:
```
import seaborn as sns
sns.pairplot(tips,hue='sex') #hue is optional
```

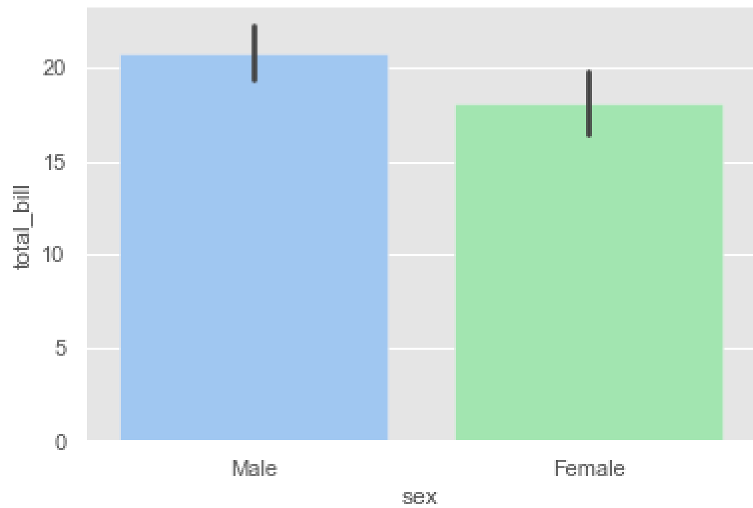
Out[380]: <seaborn.axisgrid.PairGrid at 0x12b61a4e0>



We can create a bar plot with seaborn.

In [381]:
```
import seaborn as sns
sns.barplot(x='sex',y='total_bill',data=tips) #mean unless specified
```

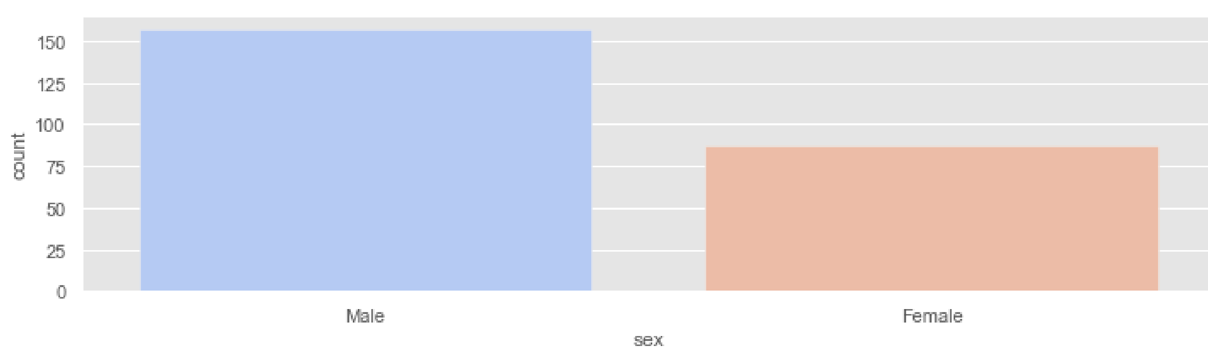
Out[381]: <matplotlib.axes._subplots.AxesSubplot at 0x12b8b7be0>



We can create a count-plot.

In [382]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
plt.figure(figsize=(12,3)) #optional
sns.countplot(x='sex',data=tips,palette='coolwarm')
```

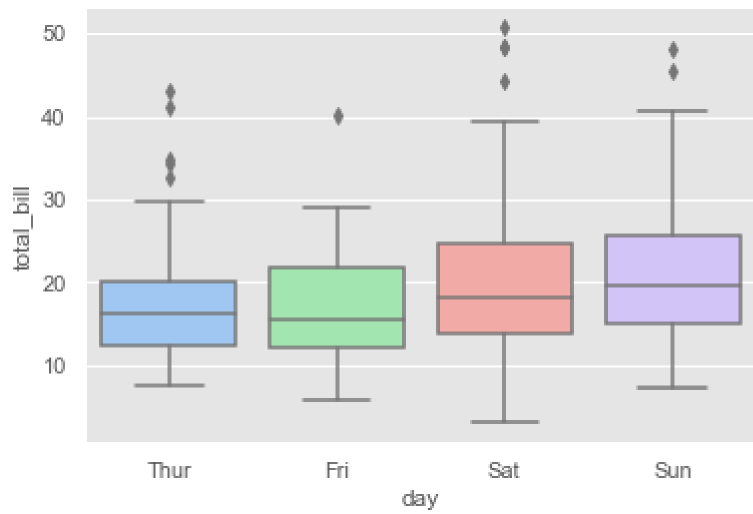
Out[382]: <matplotlib.axes._subplots.AxesSubplot at 0x12bc13198>



We can create a box-and-whisker plot.

In [383]:
```
import seaborn as sns
sns.boxplot(x='day',y='total_bill',data=tips)
```

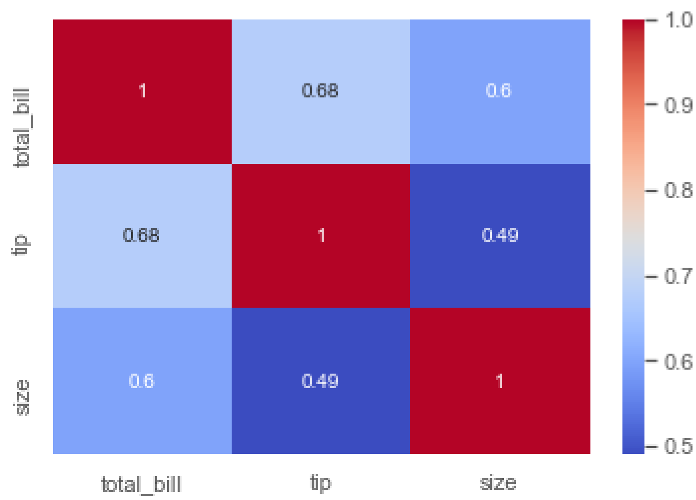
Out[383]: <matplotlib.axes._subplots.AxesSubplot at 0x12ba7db00>



We can create a heatmap of a correlation matrix.

In [384]:
```
import seaborn as sns
sns.heatmap(tips.corr(),annot=True,cmap='coolwarm')
```

Out[384]: <matplotlib.axes._subplots.AxesSubplot at 0x12bedcc18>



We can plot a heatmap of a pivot table.

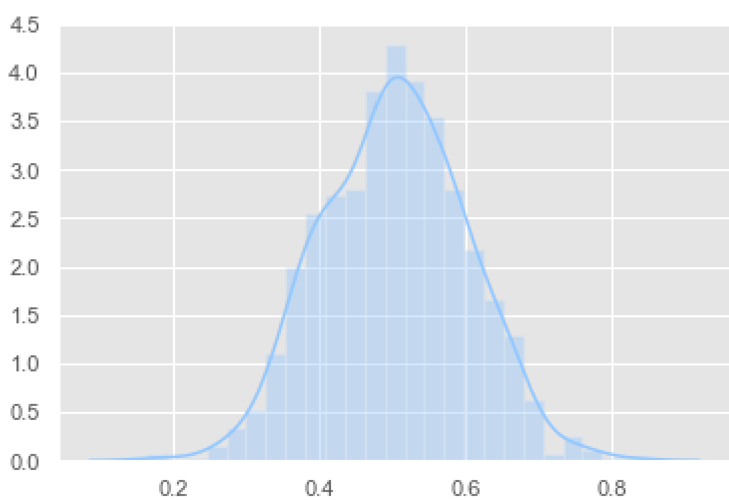
In [385]:
```
import seaborn as sns
sns.heatmap(titanic.pivot_table(index='IsAdult',columns='Sex',values='Fare'
```

Out[385]: <matplotlib.axes._subplots.AxesSubplot at 0x12bee6550>

We can plot a distribution of generated normal random variables.

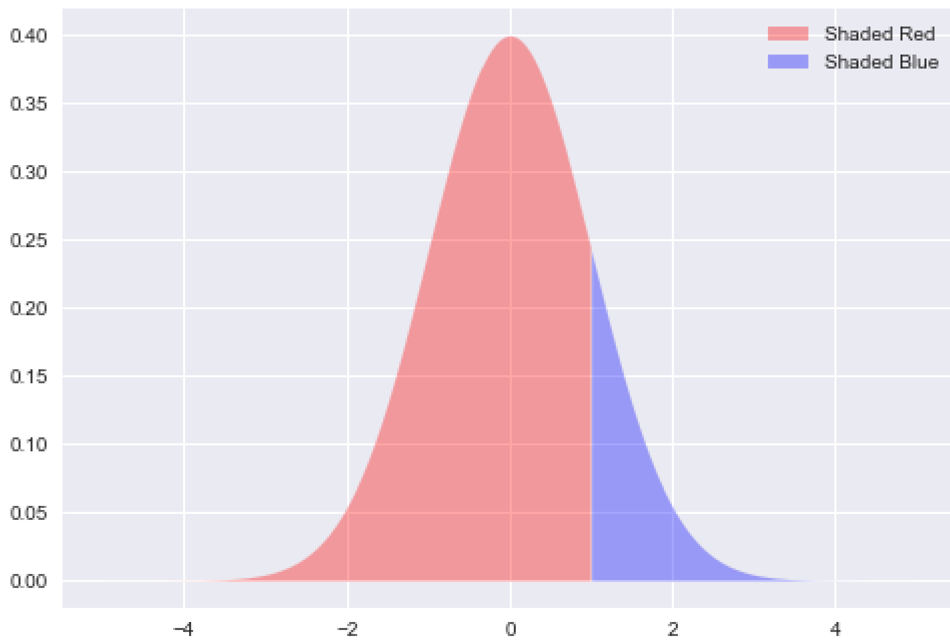
In [386]:
```
import numpy as np
import seaborn as sns
mu, sigma = 0.5, 0.1
n = 1000
s = np.random.normal(mu, sigma, n)
sns.distplot(s)
```

Out[386]: <matplotlib.axes._subplots.AxesSubplot at 0x12c0b3518>



We can plot a shaded probability density function (pdf).

In [387]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('seaborn')
plt.fill_between(x=np.arange(-5,1,0.01),
    y1= stats.norm.pdf(np.arange(-5,1,0.01)) ,
    facecolor='red',
    alpha=0.35,
    label= 'Shaded Red')
plt.fill_between(x=np.arange(1,5,0.01),
    y1= stats.norm.pdf(np.arange(1,5,0.01)) ,
    facecolor='blue',
    alpha=0.35,
    label= 'Shaded Blue')
plt.legend()
plt.show()
```



We can show a list of plotting styles.

In [388]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.available
```

Out[388]:
```
['_classic_test',
 'bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn-bright',
 'seaborn-colorblind',
 'seaborn-dark-palette',
 'seaborn-dark',
 'seaborn-darkgrid',
 'seaborn-deep',
 'seaborn-muted',
 'seaborn-notebook',
 'seaborn-paper',
 'seaborn-pastel',
 'seaborn-poster',
 'seaborn-talk',
 'seaborn-ticks',
 'seaborn-white',
 'seaborn-whitegrid',
 'seaborn',
 'Solarize_Light2',
 'tableau-colorblind10']
```

Section: Working With SQL

We can connect to a SQL database.

In [389]:
```
import sqlite3
conn = sqlite3.connect('sqlplanets.db')
cur = conn.cursor()
```

We can pull all columns from a SQL database.

In [390]:
```
import pandas as pd
cur.execute("""SELECT * FROM planets;""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df.head()
```

Out[390]:

	id	name	color	num_of_moons	mass	rings
0	1	Mercury	gray	0	0.55	0
1	2	Venus	yellow	0	0.82	0
2	3	Earth	blue	1	1.00	0
3	4	Mars	red	2	0.11	0
4	5	Jupiter	orange	68	317.90	0

We can pull specific columns from a SQL database.

In [391]:
```
import pandas as pd
cur.execute("""SELECT name,color FROM planets;""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df.head()
```

Out[391]:

	name	color
0	Mercury	gray
1	Venus	yellow
2	Earth	blue
3	Mars	red
4	Jupiter	orange

We can run a conditional on a SQL database.

In [392]:
```
import pandas as pd
cur.execute("""SELECT * FROM planets WHERE mass > 1;""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df.head()
```

Out[392]:

	id	name	color	num_of_moons	mass	rings
0	5	Jupiter	orange	68	317.90	0
1	6	Saturn	hazel	62	95.19	1
2	7	Uranus	light blue	27	14.54	1
3	8	Neptune	dark blue	14	17.15	1

We can run several conditionals on a SQL database.

In [393]:
```
import pandas as pd
cur.execute("""SELECT name, color
               FROM planets
               WHERE color == 'blue'
               OR color == 'light blue'
               OR color == 'dark blue';
               """)
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df.head()
```

Out[393]:

	name	color
0	Earth	blue
1	Uranus	light blue
2	Neptune	dark blue

We can run a query with descending order on a SQL database.

In [394]:
```
import pandas as pd
cur.execute("""SELECT name,color,mass FROM planets ORDER BY mass DESC;""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df.head()
```

Out[394]:

	name	color	mass
0	Jupiter	orange	317.90
1	Saturn	hazel	95.19
2	Neptune	dark blue	17.15
3	Uranus	light blue	14.54
4	Earth	blue	1.00

We can run a query with ascending order on a SQL database.

In [395]:
```
import pandas as pd
cur.execute("""SELECT name,color,mass FROM planets ORDER BY mass ASC;""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df.head()
```

Out[395]:

	name	color	mass
0	Mars	red	0.11
1	Mercury	gray	0.55
2	Venus	yellow	0.82
3	Earth	blue	1.00
4	Uranus	light blue	14.54

We can run a query with limited output on a SQL database.

In [396]:
```
import pandas as pd
cur.execute("""SELECT name,color,mass FROM planets LIMIT 4""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df
```

Out[396]:

	name	color	mass
0	Mercury	gray	0.55
1	Venus	yellow	0.82
2	Earth	blue	1.00
3	Mars	red	0.11

We can find the number of rows using a SQL query.

In [397]:
```
import pandas as pd
cur.execute("""SELECT COUNT(*) FROM planets""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df
```

Out[397]:

	COUNT(*)
0	8

We can find the average of a column using a SQL query.

In [398]:
```
import pandas as pd
cur.execute("""SELECT AVG(mass) FROM planets""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df
```

Out[398]:

	AVG(mass)
0	55.9075

We can use rounding on a column using a SQL query.

In [399]:
```
import pandas as pd
cur.execute("""SELECT ROUND(AVG(mass),2) FROM planets""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df
```

Out[399]:

	ROUND(AVG(mass),2)
0	55.91

We can find the minimum of a column using a SQL query.

In [400]:
```
import pandas as pd
cur.execute("""SELECT MIN(mass) FROM planets""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df
```

Out[400]:

	MIN(mass)
0	0.11

We can find the maximum of a column using a SQL query.

In [401]:
```
import pandas as pd
cur.execute("""SELECT MAX(mass) FROM planets""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df
```

Out[401]:

	MAX(mass)
0	317.9

We can find the sum of a column using a SQL query.

In [402]:
```
import pandas as pd
cur.execute("""SELECT SUM(mass) FROM planets""")
df = pd.DataFrame(cur.fetchall())
df.columns = [x[0] for x in cur.description]
df
```

Out[402]:

	SUM(mass)
0	447.26

Module 2: Predictive Modeling and Machine Learning

Section: General Machine Learning Set-up

When running machine learning models, we will want to take in certain predictor variables (which we generally refer to as our X variables) and one target variable (which we generally refer to as our y variable). We can split our data into our X and Y components.

In [403]:
```
import pandas as pd
df=pd.read_csv('USA_Housing.csv')
X=df[['Avg. Area Income',
 'Avg. Area House Age',
 'Avg. Area Number of Rooms',
 'Avg. Area Number of Bedrooms',
 'Area Population']]
y=df['Price']
```

Alternatively,

In [404]:
```
import pandas as pd
df=pd.read_csv('USA_Housing.csv')
X=df.drop(['Price','Address'],axis=1)
y=df['Price']
```

To prevent over-fitting our model, we will need to know how to split our data into a training set and a testing set. The idea is that we will run our model on our training set to teach our model, and then we will test out the model's accuracy on our testing set.

In [405]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,random_sta
```

Notice that we used a test size of 30%, which implies that we used a training set of 70%. There is an aspect of randomness used to create the training and test sets, and the random state parameter is optional (and helpful for result replication). Generally, if we have a large amount of data, we will want to test around 30%+, although if we have a small amount of data, we may want to use most of it to teach our model, and therefore our test size could be 20% or less. We can confirm that our test size is now 30% of the data.

In [406]:
```
len(X_test) / (len(X_test) + len(X_train))
```

Out[406]:
```
0.3
```

Often, we will need to scale our variables so that different sizes don't throw off our model. One way to scale our variables is to set them to a standard normal distribution, with mean 0. We can perform standard normal feature scaling.

In [407]:
```
import pandas as pd
import numpy as np
df=pd.read_csv('KMC_House_Price_Data.csv',usecols=[1,2,4])
sqft_lot_normalizing = df['sqft_lot']
df_scaled = pd.DataFrame([])
df_scaled['sqft_lot'] = (sqft_lot_normalizing-np.mean(sqft_lot_normalizing)
df['sqft_lot'] = df_scaled['sqft_lot']
df['sqft_lot'].describe()
```

Out[407]:
```
count    2.159700e+04
mean    -6.625508e-17
std      1.000023e+00
min     -3.520603e-01
25%     -2.429124e-01
50%     -1.806594e-01
75%     -1.065982e-01
max      3.951203e+01
Name: sqft_lot, dtype: float64
```

Another method is to use min-max feature scaling, where we scale our features to be between 0 and 1. We can perform min-max feature scaling.

In [408]:
```
minmax = df['sqft_lot']
df_scaled['sqft_lot'] = (minmax-min(minmax))/(max(minmax))
df['sqft_lot']=df_scaled['sqft_lot']
df['sqft_lot'].describe()
```

Out[408]:
```
count    21597.000000
mean         0.008910
std          0.025309
min          0.000000
25%          0.002762
50%          0.004338
75%          0.006212
max          1.008910
Name: sqft_lot, dtype: float64
```

Another method is to use use logarithmic scaling. We can perform logarithmic feature scaling.

In [409]:
```
import pandas as pd
import numpy as np
df=pd.read_csv('KMC_House_Price_Data.csv',usecols=[1,2,4])
df_scaled = pd.DataFrame([])
df_scaled["logsqft_lot"] = np.log(df["sqft_lot"])
df_scaled['logsqft_lot'].describe()
```

Out[409]:
```
count    21597.000000
mean         8.989805
std          0.902078
min          6.253829
25%          8.525161
50%          8.938269
75%          9.276596
max         14.317109
Name: logsqft_lot, dtype: float64
```

We can scale multiple variables at the same time.

In [410]:
```
import pandas as pd
df=pd.read_csv('KMC_House_Price_Data.csv')#,usecols=[1,2,4])
columns_to_normalize = ['bedrooms','bathrooms','floors']
df[columns_to_normalize] = df[columns_to_normalize].apply(lambda x: (x - x.
df[['bedrooms','bathrooms','floors']].head(3)
```

Out[410]:

	bedrooms	bathrooms	floors
0	0.2	0.066667	0.0
1	0.2	0.200000	0.4
2	0.1	0.066667	0.0

We will need to create dummy variables from categorical variables. For example, the number of bedrooms in a house is a categorical variable, because it can only take discrete, integer values, such as 1 bedroom, 2 bedrooms, etc. We can create dummy variables from categorical variables.

In [411]:
```
import pandas as pd
df=pd.read_csv('KMC_House_Price_Data.csv',usecols=[1,2,4])
bed_dummies=pd.get_dummies(df['bedrooms'],prefix='bed',drop_first=True)
df=pd.concat([df,bed_dummies],axis=1)
df.drop(['bedrooms'],axis=1,inplace=True)
df.head(3)
```

Out[411]:

	price	sqft_lot	bed_2	bed_3	bed_4	bed_5	bed_6	bed_7	bed_8	bed_9	bed_10	bed_11
0	221900.0	5650	0	1	0	0	0	0	0	0	0	0
1	538000.0	7242	0	1	0	0	0	0	0	0	0	0
2	180000.0	10000	1	0	0	0	0	0	0	0	0	0

Notice that we dropped the first dummy variable, bed_1. This is to prevent multi-collinearity ##

Outliers can be problematic because they can throw off our models. We can drop outliers.

In [412]:
```
upper_2_percent = df.quantile(0.98)
outliers_top = (df> upper_2_percent)
lower_2_percent = df.quantile(0.02)
outliers_bottom = (df< lower_2_percent)
df=df.mask(outliers_top,upper_2_percent,axis=1)
df=df.mask(outliers_bottom,lower_2_percent,axis=1)
df.describe()
```

Out[412]:

	price	sqft_lot	bed_2	bed_3	bed_4	bed_5	bed
count	2.159700e+04	21597.000000	21597.000000	21597.000000	21597.000000	21597.000000	2159
mean	5.280866e+05	12416.936797	0.127796	0.454924	0.318655	0.074131	(
std	2.960213e+05	17939.004784	0.333870	0.497976	0.465966	0.261989	(
min	1.750000e+05	1184.000000	0.000000	0.000000	0.000000	0.000000	(
25%	3.220000e+05	5040.000000	0.000000	0.000000	0.000000	0.000000	(
50%	4.500000e+05	7618.000000	0.000000	0.000000	0.000000	0.000000	(
75%	6.450000e+05	10685.000000	0.000000	1.000000	1.000000	0.000000	(
max	1.600000e+06	107157.000000	1.000000	1.000000	1.000000	1.000000	(

We can also drop outliers using another approach.

In [413]:
```
df = df[df.sqft_lot < 100000]
```

We might want to shuffle the dataframe.

In [414]:
```
df = df.sample(frac=1)
```

We can test normality using a Jacque-Bera test.

In [415]:
```
'''
import statsmodels.stats.api as sms
import statsmodels.formula.api as smf
f = 'price~bed_2'
model = smf.ols(formula=f, data=df).fit()
name = ['Jarque-Bera','Prob','Skew', 'Kurtosis']
test = sms.jarque_bera(model.resid)
list(zip(name, test))'''
```

Out[415]:
```
"\nimport statsmodels.stats.api as sms\nimport statsmodels.formula.api as
smf\nf = 'price~bed_2'\nmodel = smf.ols(formula=f, data=df).fit()\nname =
['Jarque-Bera','Prob','Skew', 'Kurtosis']\ntest = sms.jarque_bera(model.r
esid)\nlist(zip(name, test))"
```

High JB implies errors are not normally distributed. ##problematic for regression

Section: Linear Regression

If we want to predict a quantity ("how much") of something, linear regression is one way to do so. For example, we might want to see if we can predict house prices (our target, or Y, variable) based on several features, such as the number of bedrooms, square footage, and whether or not the home has a waterfront view. We start by reading in the data, and turning the number of bedrooms, which is a categorical variable, into dummy variables as we did previously.

In [416]:
```
import pandas as pd
df=pd.read_csv('KMC_House_Price_Data.csv',usecols=[1,2,4,6])
bed_dummies=pd.get_dummies(df['bedrooms'],prefix='bed',drop_first=True)
df=pd.concat([df,bed_dummies],axis=1)
df.drop(['bedrooms'],axis=1,inplace=True)
df.head(3)
```

Out[416]:

	price	sqft_lot	waterfront	bed_2	bed_3	bed_4	bed_5	bed_6	bed_7	bed_8	bed_9	bed_1
0	221900.0	5650	0.0	0	1	0	0	0	0	0	0	
1	538000.0	7242	0.0	0	1	0	0	0	0	0	0	
2	180000.0	10000	0.0	1	0	0	0	0	0	0	0	

Want to use these variables to predict the price. We then set the price to be the target variable and the others to be the dependent variables that are used to make the price prediction.

In [417]:
```
X=df.drop('price',axis=1)
y=df['price']
```

We can test to see if we have a lot of data.

In [418]:
```
len(X)
```

Out[418]:
```
21597
```

Since we have over 21,000 rows of data, we can probably get away with training only 70% of the data instead of 80% of the data. We can run a train-test split.

In [419]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,random_sta
```

We can fit a Linear Regression model using the Sklearn library.

In [420]:
```
from sklearn.linear_model import LinearRegression
lm=LinearRegression()
lm.fit(X_train,y_train)
```

Out[420]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

We might want to perform task, such as fit a Linear Regression model, but not display its output. We can use a ; at the end of a command to suppress output, which will be used throughout the rest of this text to suppress output.

In [421]:
```
lm.fit(X_train,y_train);
```

We can print the regression intercept.

In [422]:
```
print(lm.intercept_)
```

```
286233.30465356796
```

We can print the regression coefficients.

In [423]:
```
print(lm.coef_)
```

```
[6.47920864e-01 1.18881512e+06 9.84235651e+04 1.63493859e+05
 3.33278084e+05 4.77236012e+05 5.32567534e+05 7.06851315e+05
 6.97069942e+05 5.88886242e+05 6.06369383e+05 2.30553008e+05]
```

This is difficult to interpret, so we can create a table of our regression coefficients.

In [424]:
```
import pandas as pd
coef_df=pd.DataFrame(lm.coef_,X.columns,columns=['Coefficients'])
coef_df
```

Out[424]:

	Coefficients
sqft_lot	6.479209e-01
waterfront	1.188815e+06
bed_2	9.842357e+04
bed_3	1.634939e+05
bed_4	3.332781e+05
bed_5	4.772360e+05
bed_6	5.325675e+05
bed_7	7.068513e+05
bed_8	6.970699e+05
bed_9	5.888862e+05
bed_10	6.063694e+05
bed_11	2.305530e+05

We can create predictions based on these coefficients.

In [425]:
```
predictions=lm.predict(X_test)
predictions
```

Out[425]:
```
array([455281.78923054, 453668.4662782 , 646724.06453175, ...,
       456387.79014607, 765973.53039166, 395328.12640332])
```

We now have predictions that we can compare against actual values through our testing set (y_test). For example, we can analyze the predictions alongside our actual values.

In [426]:
```
X_test['actual_values']=y_test
X_test['predicted_values']=predictions
X_test.head(3)
```

Out[426]:

	sqft_lot	waterfront	bed_2	bed_3	bed_4	bed_5	bed_6	bed_7	bed_8	bed_9	bed_10	be
3686	8573	0.0	0	1	0	0	0	0	0	0	0	
10247	6083	0.0	0	1	0	0	0	0	0	0	0	
4037	42000	0.0	0	0	1	0	0	0	0	0	0	

So, for example, the first data point is a home that has 8,573 square feet, no waterfront view, and three bedrooms. The actual price of the home is 132,500 and the predicted price of the home is 455,282. We can multiply our regression coefficients by the data and see if we can obtain the same prediction, using principles.

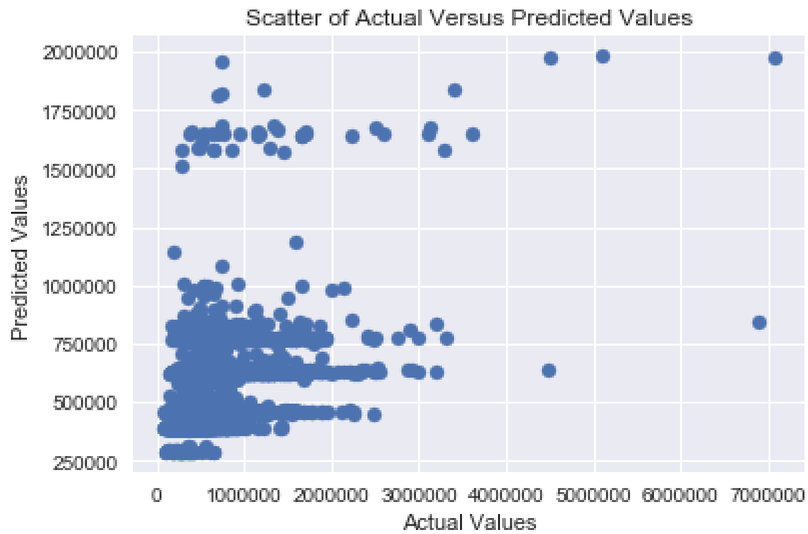
In [427]:
```
coef_df.loc['sqft_lot'][0] * 8573 + coef_df.loc['waterfront'][0] * 0 + coef
```

Out[427]:
```
455281.78923054435
```

We can plot a scatter of actual values against predicted values.

In [428]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.scatter(y_test,predictions)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Scatter of Actual Versus Predicted Values')
```

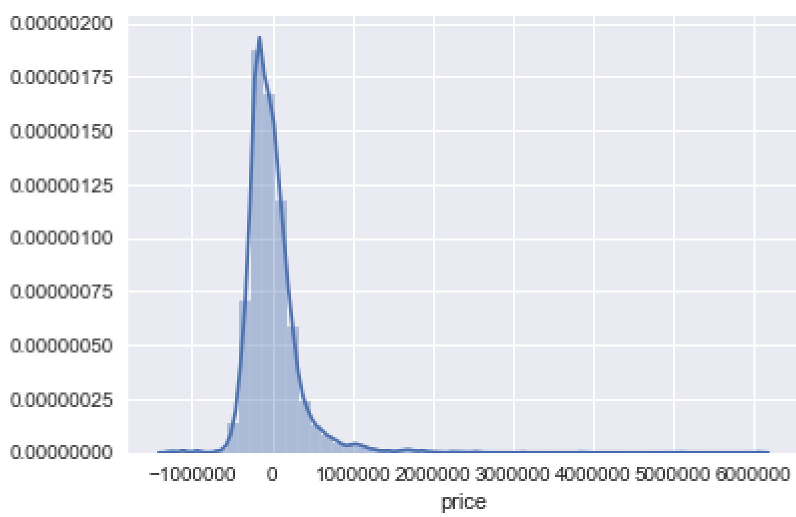
Out[428]:
```
Text(0.5, 1.0, 'Scatter of Actual Versus Predicted Values')
```



We can plot a histogram of residuals.

In [429]:
```
import seaborn as sns
sns.distplot((y_test-predictions))
```

Out[429]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12c1cb9e8>
```



We observe that residuals are approximately normally distributed, which is a good sign for our regression model. We can evaluate the Mean Absolute Error.

In [430]:
```
from sklearn import metrics
metrics.mean_absolute_error(y_test,predictions)
```

Out[430]: 214451.57798272194

We can evaluate the Mean Squared Error (MSE).

In [431]:
```
from sklearn import metrics
metrics.mean_squared_error(y_test,predictions)
```

Out[431]: 110618366256.33511

We can evaluate the Root Mean Squared Error (RMSE).

In [432]:
```
import numpy as np
from sklearn import metrics
np.sqrt(metrics.mean_squared_error(y_test,predictions))
```

Out[432]: 332593.3947875921

We can also run a Linear Regression using the "statsmodels" library.

In [433]:
```
'''
import pandas as pd
from statsmodels.formula.api import ols
data = pd.read_csv('Advertising.csv', index_col=0)
dep_vble = 'sales'
indep_vbles = ['TV', 'radio', 'newspaper']
predictors = '+'.join(indep_vbles)
formula = dep_vble + '~' + predictors
model = ols(formula=formula, data=data).fit()
model.summary()
'''
```

Out[433]:
```
"\nimport pandas as pd\nfrom statsmodels.formula.api import ols\ndata = p
d.read_csv('Advertising.csv', index_col=0)\ndep_vble = 'sales'\nindep_vbl
es = ['TV', 'radio', 'newspaper']\npredictors = '+'.join(indep_vbles)\nfo
rmula = dep_vble + '~' + predictors\nmodel = ols(formula=formula, data=da
ta).fit()\nmodel.summary()\n"
```

One of the benefits of the statsmodels library over the sklearn library for linear regressions is that statsmodels can display p-values. We can show the regression p-values.

In [434]:
```
#model.pvalues
```

Section: Logistic Regression

Whereas Linear Regression was good for predicting the magnitude of price moves, there are other types of prediction methods that are better for binary classification. For example, we might want to predict a yes or no outcome. We have already observed the Titanic dataset, and we can use

Logistic Regression, a classification model, to predict whether or not a passenger died based on a set of features. First, we can set up our data.

In [435]:
```
import pandas as pd
import numpy as np
titanic = pd.read_csv('titanic.csv',index_col=1)
gender_dummies=pd.get_dummies(titanic['Sex'],prefix='gender',drop_first=Tru
titanic=pd.concat([titanic,gender_dummies],axis=1)
titanic.drop(['Sex'],axis=1,inplace=True)
embarked_dummies=pd.get_dummies(titanic['Embarked'],prefix='embarked',drop_
titanic=pd.concat([titanic,embarked_dummies],axis=1)
titanic.drop(['Embarked'],axis=1,inplace=True)
titanic['IsRich']=np.where(titanic['Fare']>titanic['Fare'].quantile(.80),1,
titanic.drop(['Fare'],axis=1,inplace=True)
class_dummies=pd.get_dummies(titanic['Pclass'],prefix='class',drop_first=Tr
titanic.drop(['Pclass'],axis=1,inplace=True)
titanic.drop(['Unnamed: 0','Name','Ticket','Cabin'],axis=1,inplace=True)
titanic=titanic.dropna()
X=titanic.drop('Survived',axis=1)
y=titanic['Survived']
```

We can run a train-test split for Logistic Regression.

In [436]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_sta
```

We can fit a Logistic Regression using Sklearn.

In [437]:
```
from sklearn.linear_model import LogisticRegression
logmodel=LogisticRegression()
logmodel.fit(X_train,y_train);
```

We can create predictions for the testing set.

In [438]:
```
predictions=logmodel.predict(X_test)
```

We can create a classification report for a Logistic Regression.

In [439]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.77      0.79      0.78        87
           1       0.66      0.62      0.64        56

   micro avg       0.73      0.73      0.73       143
   macro avg       0.71      0.71      0.71       143
weighted avg       0.73      0.73      0.73       143
```

We can create a confusion matrix for a Logistic Regression.

In [440]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,predictions))
```

```
[[69 18]
 [21 35]]
```

We can display model accuracy for a Logistic Regression.

In [441]:
```
from sklearn.metrics import accuracy_score
logistic_accuracy_score=accuracy_score(y_test,predictions)
print(format(logistic_accuracy_score,".2%"))
```

```
72.73%
```

We can confirm this accuracy by using a principles taken from true and false positives and negatives from a confusion matrix.

In [442]:
```
a=confusion_matrix(y_test,predictions)[0,0]+confusion_matrix(y_test,predict
b=confusion_matrix(y_test,predictions)[0,1]+confusion_matrix(y_test,predict
accuracy=a/(a+b)
round(accuracy,4)
```

Out[442]:
```
0.7273
```

When we run a train-test split, we inherently have randomness in the split of the data into training and test sets. Therefore, it's important to run K-fold Cross Validation. We can run 10-fold Cross Validation for a Logistic Regression.

In [443]:
```
from sklearn.model_selection import cross_val_score
scores=cross_val_score(logmodel,X_train,y_train,scoring="accuracy",cv=10)
print('Ten-fold cross validation scores: ' , scores)
print('Average across cross validation scores: ' ,scores.mean())
```

```
Ten-fold cross validation scores:  [0.79310345 0.72413793 0.75862069 0.79
310345 0.73684211 0.78947368
 0.85964912 0.82142857 0.85714286 0.71428571]
Average across cross validation scores:  0.7847787572379223
```

We can find the AUC score for a Logistic Regression.

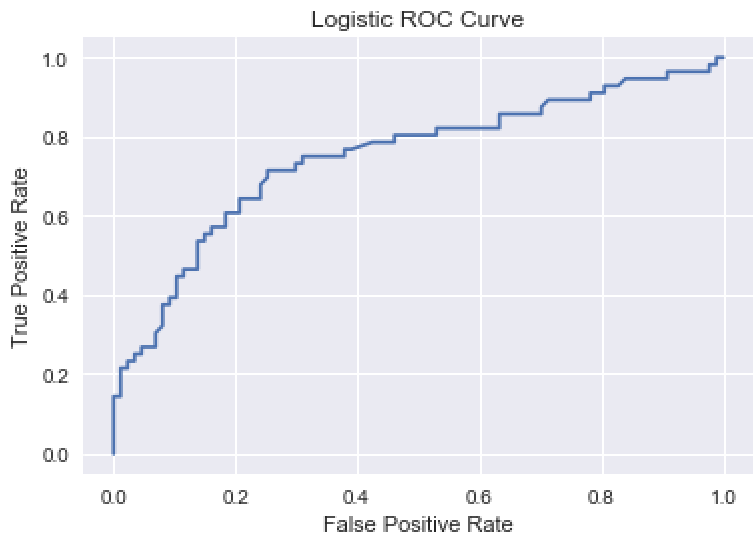
In [444]:
```
from sklearn.metrics import roc_auc_score
probabilities_logistic = logmodel.predict_proba(X_test)
auc_logistic = roc_auc_score(y_test, probabilities_logistic[:,1])
auc_logistic
```

Out[444]:
```
0.7482553366174056
```

We can plot the ROC Curve for a Logistic Regression.

In [445]:
```
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.metrics import roc_curve
fpr_log, tpr_log, thresholds_log = roc_curve(y_test, probabilities_logistic
plt.plot(fpr_log, tpr_log)
plt.title('Logistic ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

Out[445]:
```
Text(0, 0.5, 'True Positive Rate')
```



Section: K-Nearest Neighbors (KNN)

Another classification method is K-Nearest Neighbors (KNN). We can perform a train-test split for KNN.

In [446]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_sta
```

We can run a K-Nearest Neighbors to segment data with a specified number of "neighbors" using Sklearn.

In [447]:
```
from sklearn.neighbors import KNeighborsClassifier
knn=KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train,y_train);
```

We can create predictions for a K-Nearest Neighbors model.

In [448]:
```
pred=knn.predict(X_test)
```

We can create a classication report for a K-Nearest Neighbors model.

In [449]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.78      0.74      0.76        87
           1       0.62      0.68      0.65        56

   micro avg       0.71      0.71      0.71       143
   macro avg       0.70      0.71      0.70       143
weighted avg       0.72      0.71      0.72       143
```

We can create a confusion matrix for a K-Nearest Neighbors model.

In [450]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,pred))
```

```
[[64 23]
 [18 38]]
```

We can display the accuracy score for a K-Nearest Neighbors model.

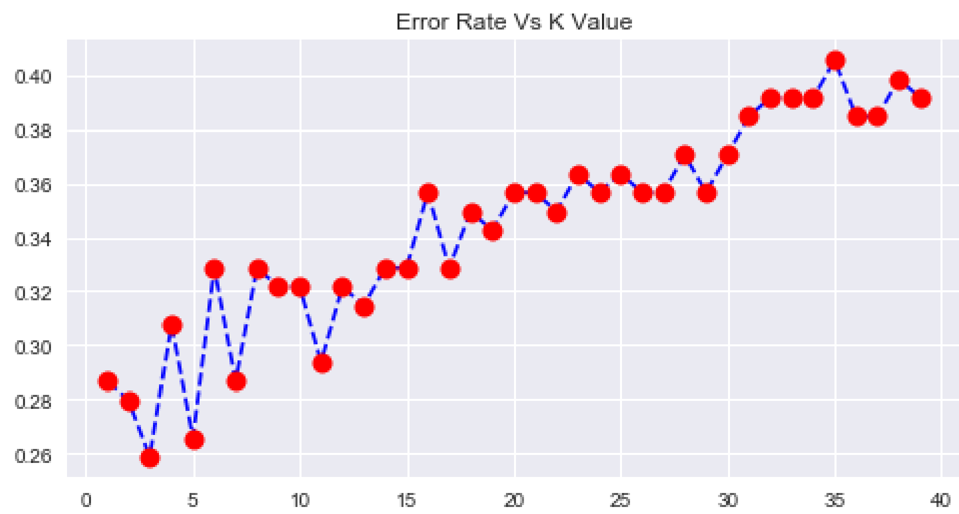
In [451]:
```
from sklearn.metrics import accuracy_score
KNN_accuracy_score=accuracy_score(y_test,pred)
print(format(KNN_accuracy_score,".2%"))
```

```
71.33%
```

We can plot error rates for various numbers of neighbors for a K-Nearest Neighbors model.

In [452]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
error_rate=[]
for i in range(1,40):
    knn=KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train,y_train)
    pred_i=knn.predict(X_test)
    error_rate.append(np.mean(pred_i!=y_test)) #avg error rate
plt.figure(figsize=(8,4))
plt.plot(range(1,40),error_rate,color='blue',linestyle='dashed',
         marker='o',markerfacecolor='red',markersize=10)
plt.title('Error Rate Vs K Value')
```

Out[452]:
```
Text(0.5, 1.0, 'Error Rate Vs K Value')
```



We can confirm that the minimum error rate is when K=3 neighbors (index=2).

In [453]:
```
error_rate[2]==min(error_rate)
```

Out[453]:
```
True
```

Section: Decision Trees

Decision Trees are popular for classification. We can fit a Decision Tree model in Sklearn.

In [454]:
```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=10)
model.fit(X_train,y_train);
```

We can generate predictions for Decision Trees.

In [455]:
```
predictions=model.predict(X_test)
```

We can create a classification report for Decision Trees.

In [456]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.76      0.78      0.77        87
           1       0.65      0.62      0.64        56

   micro avg       0.72      0.72      0.72       143
   macro avg       0.71      0.70      0.70       143
weighted avg       0.72      0.72      0.72       143
```

We can create a confusion matrix for Decision Trees.

In [457]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test,predictions)
```

Out[457]:
```
array([[68, 19],
       [21, 35]])
```

We can display the accuracy score for Decision Trees.

In [458]:
```
from sklearn.metrics import accuracy_score
DT_accuracy_score=accuracy_score(y_test,predictions)
print(format(DT_accuracy_score,".2%"))
```

```
72.03%
```

We can find the AUC Score for a Decision Tree.

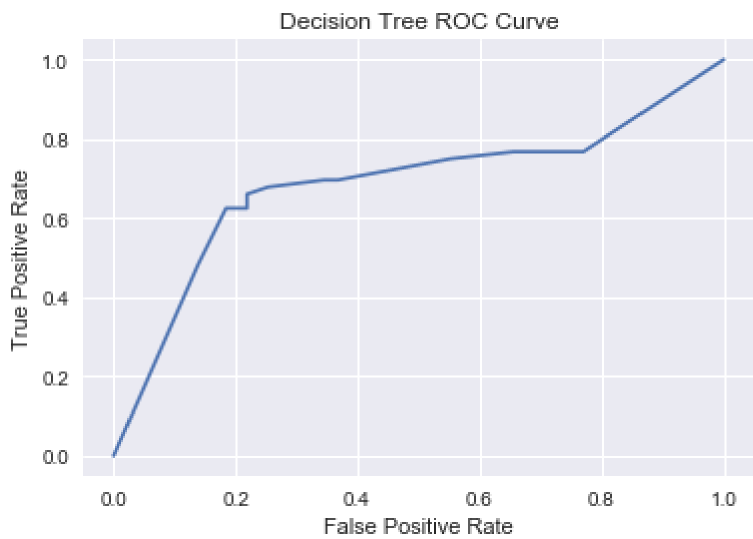
In [459]:
```
from sklearn.metrics import roc_auc_score
probabilities_tree = model.predict_proba(X_test)
auc_tree = roc_auc_score(y_test, probabilities_tree[:,1])
auc_tree
```

Out[459]:
```
0.6855500821018062
```

We can plot the ROC Curve for a Decision Tree.

In [460]:
```
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.metrics import roc_curve
fpr_tree, tpr_tree, thresholds_tree = roc_curve(y_test, probabilities_tree[
plt.plot(fpr_tree, tpr_tree)
plt.title('Decision Tree ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

Out[460]:
```
Text(0, 0.5, 'True Positive Rate')
```



Section: Random Forests

We can fit a Random Forest in Sklearn.

In [461]:
```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=200)
model.fit(X_train,y_train)
```

Out[461]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gin
i',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

We can run predictions for a Random Forest model.

In [462]:
```
predictions=model.predict(X_test)
```

We can create a classification report for a Random Forest model.

In [463]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.77      0.77      0.77        87
           1       0.64      0.64      0.64        56

   micro avg       0.72      0.72      0.72       143
   macro avg       0.71      0.71      0.71       143
weighted avg       0.72      0.72      0.72       143
```

We can create a confusion matrix for a Random Forest model.

In [464]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,predictions))
```

```
[[67 20]
 [20 36]]
```

We can display model accuracy for a Random Forest model.

In [465]:
```
from sklearn.metrics import accuracy_score
RF_accuracy_score = accuracy_score(y_test,predictions)
print(format(RF_accuracy_score,".2%"))
```

```
72.03%
```

We can calculate the AUC Score for a Random Forest model.

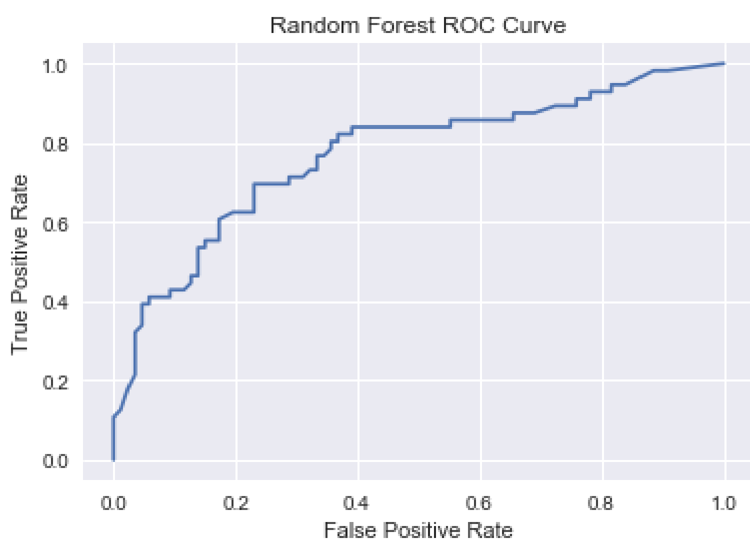
In [466]:
```
from sklearn.metrics import roc_auc_score
probabilities_forest = model.predict_proba(X_test)
auc_forest = roc_auc_score(y_test, probabilities_forest[:,1])
auc_forest
```

Out[466]:
```
0.7696018062397374
```

We can plot the ROC Curve for a Random Forest model.

In [467]:
```
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.metrics import roc_curve
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_test, probabilities
plt.plot(fpr_forest, tpr_forest)
plt.title('Random Forest ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

Out[467]:
```
Text(0, 0.5, 'True Positive Rate')
```



We can run a GridSearchCV to optimize the parameters for a Random Forest model.

In [468]:
```
import numpy as np
from sklearn.model_selection import GridSearchCV
n_estimators = [int(x) for x in np.linspace(start=1000,stop=2000,num=2)]
max_depth=[int(x) for x in np.linspace(30,60,num=2)]
min_samples_leaf = [5,10]
param_grid = {'n_estimators': n_estimators,
              'max_depth': max_depth,
              'min_samples_leaf': min_samples_leaf}
print(param_grid)
```

```
{'n_estimators': [1000, 2000], 'max_depth': [30, 60], 'min_samples_leaf':
[5, 10]}
```

We can fit a model using GridSearchCV.

In [469]:
```
%%capture
grid = GridSearchCV(RandomForestClassifier(),param_grid,verbose=5)
grid.fit(X_train,y_train);
```

We can find optimal parameters from GridSearchCV.

In [470]:
```
grid.best_params_
```

Out[470]:
```
{'max_depth': 30, 'min_samples_leaf': 5, 'n_estimators': 1000}
```

We can find the best estimator from GridSearchCV.

In [471]:
```
grid.best_estimator_
```

Out[471]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gin
i',
            max_depth=30, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=5, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=1000, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

We can now run predictions with this optimized estimator.

In [472]:
```
grid_predictions=grid.predict(X_test)
```

We can create a classification report for this optimized estimator.

In [473]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,grid_predictions))
```

```
              precision    recall  f1-score   support

           0       0.79      0.82      0.80        87
           1       0.70      0.66      0.68        56

   micro avg       0.76      0.76      0.76       143
   macro avg       0.74      0.74      0.74       143
weighted avg       0.75      0.76      0.75       143
```

We can create a confusion matrix of this estimator.

In [474]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,grid_predictions))
```

```
              precision    recall  f1-score   support

           0       0.79      0.82      0.80        87
           1       0.70      0.66      0.68        56

   micro avg       0.76      0.76      0.76       143
   macro avg       0.74      0.74      0.74       143
weighted avg       0.75      0.76      0.75       143
```

We can display model accuracy of this estimator.

In [475]:
```
from sklearn.metrics import accuracy_score
RF_accuracy_score = accuracy_score(y_test,grid_predictions)
print(format(RF_accuracy_score,".2%"))
```

```
75.52%
```

Section: Support Vector Machines (SVM)

Another popular classification algorithm is SVM. We can fit in SVM in Sklearn.

In [476]:
```
from sklearn.svm import SVC
model=SVC()
model.fit(X_train,y_train)
```

Out[476]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

We can create predictions for a Support Vector Machine model.

In [477]:
```
predictions=model.predict(X_test)
```

We can create a classification report for a Support Vector Machine model.

In [478]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.78      0.87      0.83        87
           1       0.76      0.62      0.69        56

   micro avg       0.78      0.78      0.78       143
   macro avg       0.77      0.75      0.76       143
weighted avg       0.77      0.78      0.77       143
```

We can create a confusion matrix for a Support Vector Machine model.

In [479]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,predictions))
```

```
[[76 11]
 [21 35]]
```

We can display model accuracy for a Support Vector Machine model.

In [480]:
```
from sklearn.metrics import accuracy_score
SVM_accuracy_score=accuracy_score(y_test,grid_predictions)
print(format(SVM_accuracy_score,".2%"))
```

```
75.52%
```

We can run a GridSearchCV to optimize hyper-parameters for a Support Vector Machine.

In [481]:
```
%%capture
from sklearn.model_selection import GridSearchCV
param_grid={'C':[1,100],
            'gamma':[0.001,0.1]}
grid=GridSearchCV(SVC(),param_grid,verbose=3)
grid.fit(X_train,y_train);
```

We can display the best parameters.

In [482]:
```
grid.best_params_
```

Out[482]:
```
{'C': 100, 'gamma': 0.001}
```

We can display the best estimator of a Support Vector Machine.

In [483]:
```
grid.best_estimator_
```

Out[483]:
```
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

We can create predictions from our optimal estimator.

In [484]:
```
grid_predictions=grid.predict(X_test)
```

We can create a classification report for our optimal estimator.

In [485]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,grid_predictions))
```

```
              precision    recall  f1-score   support

           0       0.79      0.84      0.82        87
           1       0.73      0.66      0.69        56

   micro avg       0.77      0.77      0.77       143
   macro avg       0.76      0.75      0.75       143
weighted avg       0.77      0.77      0.77       143
```

We can create a confusion matrix for our optimal estimator.

In [486]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,grid_predictions))
```

```
[[73 14]
 [19 37]]
```

We can display model accuracy for our optimal estimator.

In [487]:
```
from sklearn.metrics import accuracy_score
SVM_accuracy_score=accuracy_score(y_test,grid_predictions)
print(format(SVM_accuracy_score,".2%"))
```

```
76.92%
```

Section: K-Means Clustering

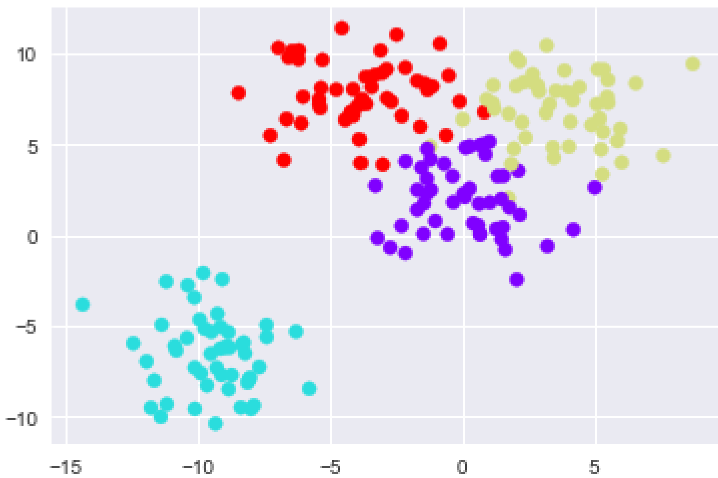
We can create blobs of clusters using Sklearn.

In [488]:
```
from sklearn.datasets import make_blobs
data = make_blobs(n_samples=200,n_features=2,centers=4,
                  cluster_std=1.8,random_state=101)
```

We can create a scatter plot of the blobs of clusters using Sklearn.

In [489]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.scatter(data[0][:,0],data[0][:,1],c=data[1],cmap='rainbow')
```

Out[489]: <matplotlib.collections.PathCollection at 0x1302c55f8>



We can fit a K-Means Clustering using Sklearn.

In [490]:
```
from sklearn.cluster import KMeans
kmeans=KMeans(n_clusters=4)
kmeans.fit(data[0])
```

Out[490]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=4, n_init=10, n_jobs=None, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

We can display the K-Means Clustering cluster centers.

In [491]:
```
kmeans.cluster_centers_
```

Out[491]:
```
array([[-4.13591321,  7.95389851],
       [-9.46941837, -6.56081545],
       [-0.0123077 ,  2.13407664],
       [ 3.71749226,  7.01388735]])
```

We can display the K-Means Clustering cluster labels.

In [492]:
```
kmeans.labels_
```

Out[492]:
```
array([0, 3, 2, 3, 3, 1, 3, 2, 3, 2, 0, 2, 3, 3, 0, 2, 3, 2, 1, 0, 1, 2,
       2, 1, 0, 1, 1, 2, 3, 3, 0, 1, 3, 2, 2, 0, 1, 1, 1, 2, 1, 0, 0, 0,
       2, 3, 0, 2, 1, 2, 2, 0, 3, 2, 1, 0, 2, 2, 0, 3, 1, 3, 1, 0, 3, 2,
       1, 3, 3, 1, 3, 2, 1, 2, 1, 3, 3, 2, 0, 2, 2, 1, 3, 1, 2, 2, 2, 0,
       2, 1, 1, 1, 1, 2, 2, 1, 3, 0, 1, 3, 2, 1, 2, 2, 3, 2, 1, 3, 1, 1,
       3, 0, 0, 3, 1, 3, 0, 0, 3, 0, 2, 0, 2, 0, 2, 3, 0, 2, 1, 0, 0, 0,
       2, 1, 1, 0, 3, 0, 3, 2, 1, 3, 1, 0, 0, 3, 2, 1, 0, 0, 0, 0, 2, 3,
       2, 0, 3, 3, 3, 2, 3, 2, 2, 0, 1, 0, 2, 3, 0, 2, 3, 2, 0, 3, 2, 0,
       3, 3, 1, 3, 0, 1, 1, 0, 1, 1, 1, 1, 1, 2, 1, 3, 3, 0, 1, 2, 3, 3,
       1, 2], dtype=int32)
```

Section: Principal Component Analysis (PCA)

We can scale a dataset for PCA.

In [493]:
```
import pandas as pd
from sklearn.datasets import load_breast_cancer
cancer=load_breast_cancer()
df = pd.DataFrame(cancer['data'],columns=cancer['feature_names'])
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(df)
scaled_data=scaler.transform(df)
```

We can run PCA on scaled data.

In [494]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(scaled_data)
x_pca=pca.transform(scaled_data)
```

We can confirm the shape of scaled PCA dataset.

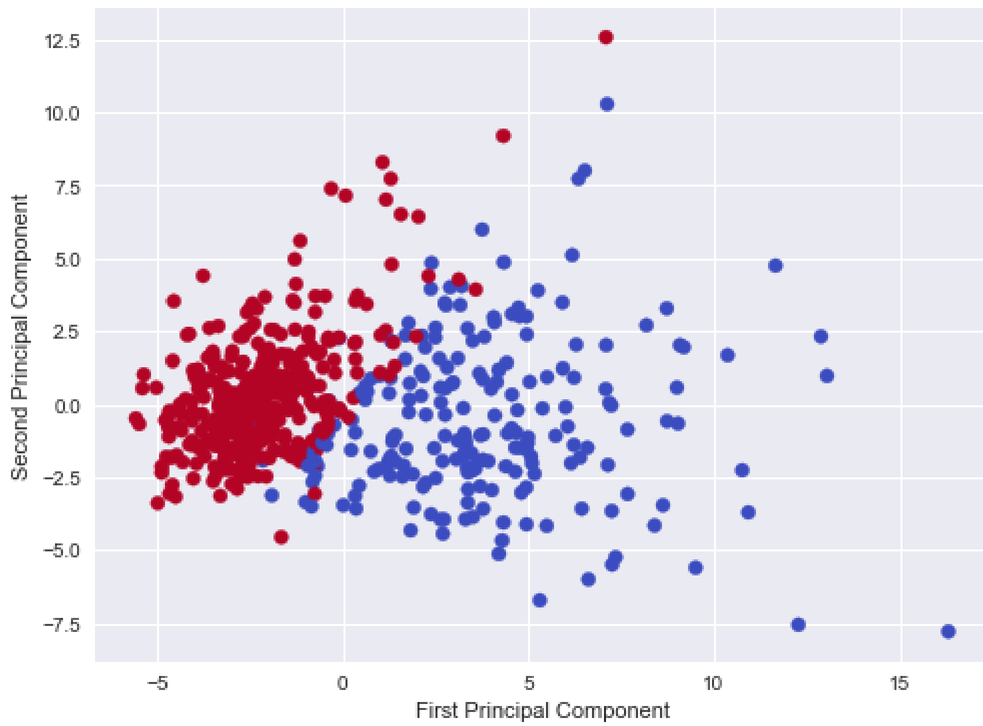
In [495]:
```
scaled_data.shape
x_pca.shape
```

Out[495]:
```
(569, 2)
```

We can plot a figure of principal components.

In [496]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure(figsize=(8,6))
plt.scatter(x_pca[:,0],x_pca[:,1],c=cancer['target'],cmap='coolwarm')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
```

Out[496]: Text(0, 0.5, 'Second Principal Component')



We can return an array of PCA components.

In [497]:
```
pca.components_
```

Out[497]:
```
array([[ 0.21890244,  0.10372458,  0.22753729,  0.22099499,  0.14258969,
         0.23928535,  0.25840048,  0.26085376,  0.13816696,  0.06436335,
         0.20597878,  0.01742803,  0.21132592,  0.20286964,  0.01453145,
         0.17039345,  0.15358979,  0.1834174 ,  0.04249842,  0.10256832,
         0.22799663,  0.10446933,  0.23663968,  0.22487053,  0.12795256,
         0.21009588,  0.22876753,  0.25088597,  0.12290456,  0.13178394],
       [-0.23385713, -0.05970609, -0.21518136, -0.23107671,  0.18611302,
         0.15189161,  0.06016536, -0.0347675 ,  0.19034877,  0.36657547,
        -0.10555215,  0.08997968, -0.08945723, -0.15229263,  0.20443045,
         0.2327159 ,  0.19720728,  0.13032156,  0.183848  ,  0.28009203,
        -0.21986638, -0.0454673 , -0.19987843, -0.21935186,  0.17230435,
         0.14359317,  0.09796411, -0.00825724,  0.14188335,  0.2753394
7]])
```

We can create a dataframe of PCA components.

In [498]:
```
import pandas as pd
df_comp=pd.DataFrame(pca.components_,columns=cancer['feature_names'])
df_comp.head(3)
```

Out[498]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	sym
0	0.218902	0.103725	0.227537	0.220995	0.142590	0.239285	0.258400	0.260854	0.1
1	-0.233857	-0.059706	-0.215181	-0.231077	0.186113	0.151892	0.060165	-0.034768	0.1

2 rows × 30 columns

Section: Comparing Machine Learning Models

We can create a dataframe comparing machine learning accuracy results.

In [499]:
```
import pandas as pd
df=pd.DataFrame({"Machine Learning Algorithm":
                ['Logistic Regression',
                'K-Nearest Neighbors',
                'Decision Trees','Random Forest',
                'Support Vector Machine'],
                "Accuracy":[logistic_accuracy_score,KNN_accuracy_score,DT_a
                           RF_accuracy_score,SVM_accuracy_score]})
df.set_index('Machine Learning Algorithm',inplace=True)
pd.options.display.float_format = '{:.2f}%'.format
df['Accuracy']=round(df['Accuracy'],3)
df
```

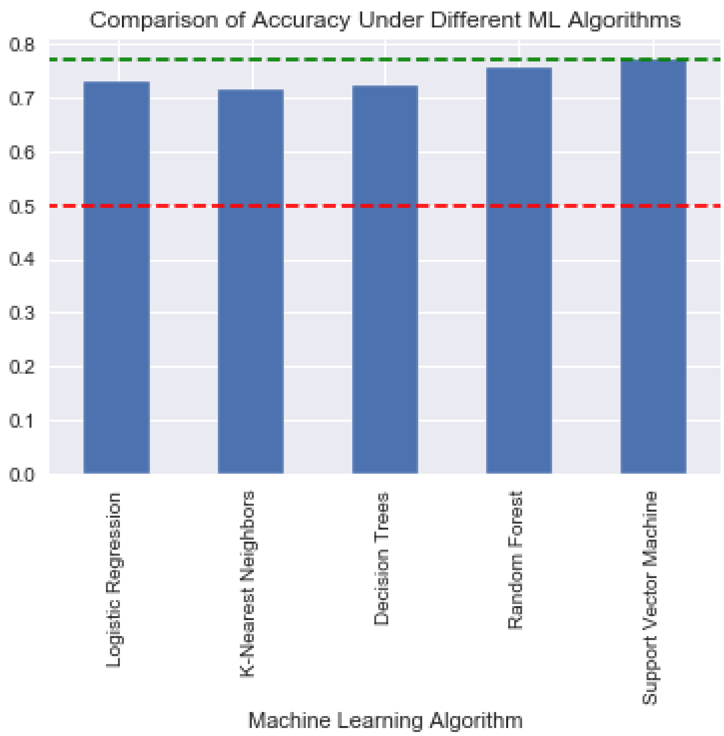
Out[499]:

Machine Learning Algorithm	Accuracy
Logistic Regression	0.73%
K-Nearest Neighbors	0.71%
Decision Trees	0.72%
Random Forest	0.76%
Support Vector Machine	0.77%

We can create a plot to compare accuracy across machine learning models.

In [500]:
```
import matplotlib.pyplot as plt
%matplotlib inline
df.plot(kind='bar',legend=False)
plt.hlines(y=0.5,xmin=-100,xmax=100,linestyle='dashed',color='red')
plt.hlines(y=max(df['Accuracy']),xmin=-100,xmax=100,
          linestyle='dashed',color='green')
plt.ylabel('Accuracy (%)', color='white')
plt.xlabel('Machine Learning Algorithm')
plt.title('Comparison of Accuracy Under Different ML Algorithms')
```

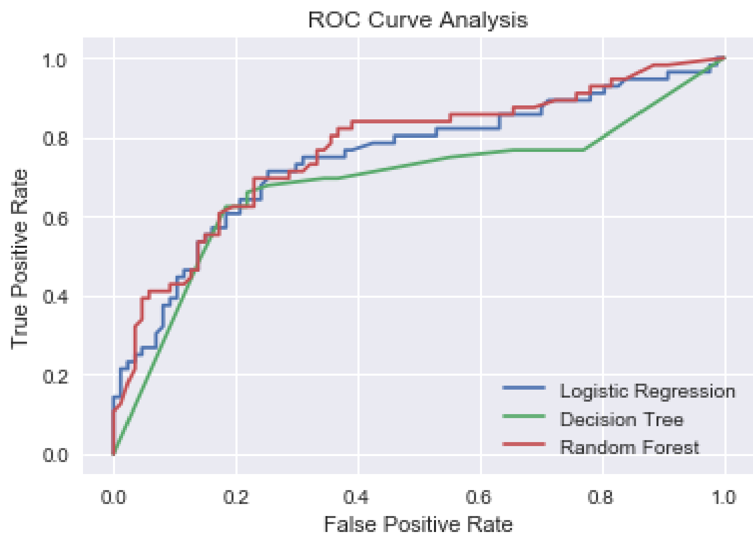
Out[500]:
```
Text(0.5, 1.0, 'Comparison of Accuracy Under Different ML Algorithms')
```



We can plot an ROC Curve to compare machine learning models.

In [501]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(fpr_log, tpr_log, label="Logistic Regression")
plt.plot(fpr_tree, tpr_tree, label="Decision Tree")
plt.plot(fpr_forest, tpr_forest, label="Random Forest")
plt.legend()
plt.title('ROC Curve Analysis')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

Out[501]:
```
Text(0, 0.5, 'True Positive Rate')
```



Section: Deep Learning

Another type of machine learning is known as deep learning, in which artificial neural networks are used. We can use neural networks to solve either regression-type or classification-type problems. We can perform feature engineering on a dataset to prepare for our deep learning model.

In [502]:
```
import pandas as pd
df = pd.read_csv('kc_house_data.csv')
df.drop('id',axis=1,inplace=True)
df['date'] = pd.to_datetime(df['date'])
df['year'] = df['date'].apply(lambda date: date.year)
df['month'] = df['date'].apply(lambda date: date.month)
df.drop('date',axis=1,inplace=True)
df.drop('zipcode',axis=1,inplace=True)
df.drop('sqft_basement',axis=1,inplace=True)
df = df.dropna()
```

We can split the data into our features and our target variable.

In [503]:
```
X = df.drop('price',axis=1)
y= df['price']
```

We can set up a train-test split.

In [504]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_s
```

We need to scale the data in order to run a neural network. We can scale our data in Sklearn.

In [505]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train); #can transform and fit in one step
X_test = scaler.transform(X_test); #don't fit to test set
```

We can confirm that our training data is now scaled between 0 and 1.

In [506]:
```
print('Minimum of Scaled Data: ' ,X_train.min())
print('Maximum of Scaled Data : ' ,X_train.max())
```

```
Minimum of Scaled Data:  0.0
Maximum of Scaled Data :  1.0000000000000002
```

We can import Tensorflow models.

In [507]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

We will need to set the number of "neurons" for our model. A general rule of thumb is that we use one neuron for every column of our dataset. Therefore, we can find the number of features of our dataset.

In [508]:
```
X.shape[1]
```

Out[508]: 18

We can create a Sequential Tensorflow model.

In [509]:
```
model = Sequential()

model.add(Dense(18))
model.add(Dense(18))
model.add(Dense(18))
model.add(Dense(18))

model.add(Dense(1))

model.compile(optimizer='adam',loss='mse')
```

Notice how we "added" in four layers of input. A network is considered to be deep learning if it contains at least two hidden networks, and this one has four. For the output, because we are trying to output just a predicted price, we use one neuron corresponding to a target variable, price. We can fit our model.

In [510]:
```
%%capture
model.fit(x=X_train,y=y_train,validation_data=(X_test,y_test),
        batch_size=128,epochs=400); #batch size typical in powers of 2
```

Notice how we used a batch size of 128. Batch sizes are typically in powers of two. Training will take longer if there is a smaller batch size. We can create a dataframe for the loss function of the validation data.

In [511]:
```
losses = pd.DataFrame(model.history.history)
losses.head(3)
```

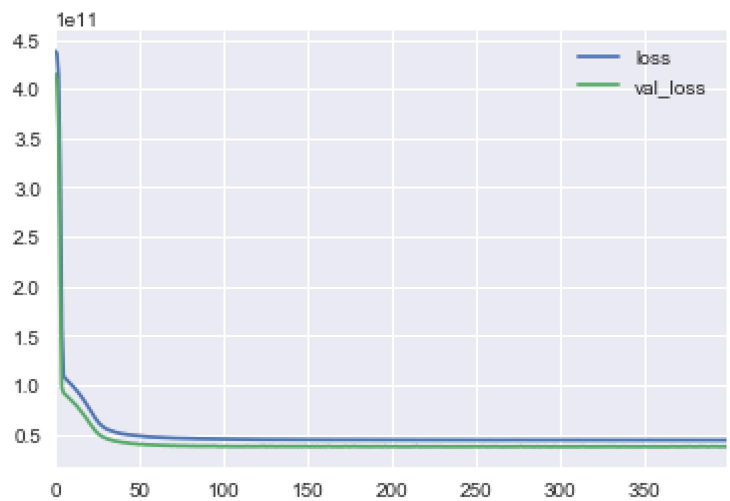
Out[511]:

	loss	val_loss
0	438367617024.00%	415382044672.00%
1	436324532224.00%	408951062528.00%
2	413836967936.00%	359511523328.00%

We can plot the loss function of the validation data.

In [512]:
```
losses.plot()
```

Out[512]: <matplotlib.axes._subplots.AxesSubplot at 0x131394d30>



In this case, we can see that validation loss is below loss, which is what we want to see. We can create predictions for our testing set.

In [513]:
```
predictions = model.predict(X_test)
```

We can calculate the Mean Squared Error (MSE) for our neural network.

In [514]:
```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test,predictions)
```

Out[514]: 37231140940.34892

We can calculate the Root Mean Squared Error (RMSE) for our neural network.

In [515]:
```
import numpy as np
np.sqrt(mean_squared_error(y_test,predictions))
```

Out[515]: 192953.72745906963

We can calculate the Mean Absolute Error (MAE) for our neural network.

In [516]:
```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test,predictions)
```

Out[516]: 127342.83779027643

We can calculate the Explained Variance Score for our neural network.

In [517]:
```
from sklearn.metrics import explained_variance_score
explained_variance_score(y_test,predictions)
```

Out[517]: 0.7052720244378192

We can show what our prediction is for a single datapoint.

In [518]:
```
single_house = df.drop('price',axis=1).iloc[0]
single_house = scaler.transform(single_house.values.reshape(-1,18))
model.predict(single_house)
```

Out[518]: array([[748378.6]], dtype=float32)

In addition to using deep learning for regression, we can use deep learning for classifcation. We can set up the data for our classification model.

In [519]:
```
import pandas as pd
df = pd.read_csv('cancer_data.csv')
df.drop(['diagnosis','id'],axis=1,inplace=True)
df.head(3)
```

Out[519]:

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
0	17.99%	10.38%	122.80%	1001.00%	0.12%	0.28%
1	20.57%	17.77%	132.90%	1326.00%	0.08%	0.08%
2	19.69%	21.25%	130.00%	1203.00%	0.11%	0.16%

3 rows × 31 columns

We can set up a train-test split.

In [520]:
```
from sklearn.model_selection import train_test_split
X = df.drop('malignant_ind',axis=1).values
y = df['malignant_ind'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra
```

We can scale our data using Sklearn.

In [521]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test);
```

We can import our Tensorflow Models.

In [522]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Dropout
```

We can find the number of features of our model.

In [523]:
```
X_train.shape[1]
```

Out[523]: 30

We can create our Sequential Deep Learning model.

In [524]:
```
model = Sequential()
model.add(Dense(30,activation = 'relu')) #30 neurons
model.add(Dense(15,activation = 'relu'))
model.add(Dense(1,activation = 'sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
```

Because this is a binary classification, we used the sigmoid function above. We can fit our Neural Network model.

In [525]:
```
%%capture
model.fit(x=X_train,y=y_train,epochs=600,validation_data=(X_test,y_test));
```

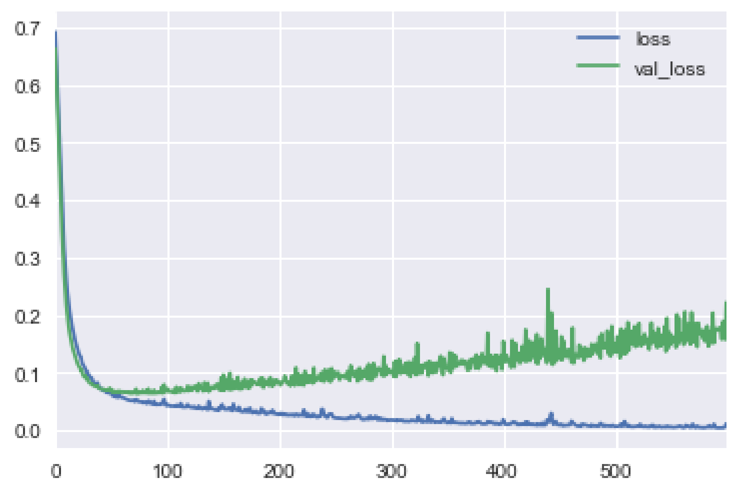
We can create a dataframe of our validation losses.

In [526]:
```
import pandas as pd
losses = pd.DataFrame(model.history.history)
```

We can create a plot of our validation losses.

In [527]:
```
losses.plot()
```

Out[527]: <matplotlib.axes._subplots.AxesSubplot at 0x151239470>



We can see that validation loss decreases over time but eventually increases. This tells us that we are over-fitting to our training data set (we are training for too many epochs). Therefore, we'll rebuild our model with early stopping to account for over-fitting. We can create our Neural Network model.

In [528]:
```
model = Sequential()
model.add(Dense(30,activation = 'relu')) #30 neurons
model.add(Dense(15,activation = 'relu'))
model.add(Dense(1,activation = 'sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
```

We can import Early Stopping.

In [529]:
```
from tensorflow.keras.callbacks import EarlyStopping
```

We can set up Early Stopping to track validation loss.

In [530]:
```
early_stop = (EarlyStopping(monitor='val_loss',mode='min',
            verbose=1,patience=25)) #trying to minimize what we are monitor
```

The mode is set to minimum because we are trying to minimize validation losses. If we were using a figure such as accuracy, we would set the mode to maximum. We can fit our Neural Network model with Early Stopping.

In [531]:
```
%%capture
model.fit(x=X_train,y=y_train,epochs=600,validation_data=(X_test,y_test),ca
```

We can create a dataframe of validation losses.

In [532]:
```
import pandas as pd
model_loss = pd.DataFrame(model.history.history)
model_loss.head(3)
```

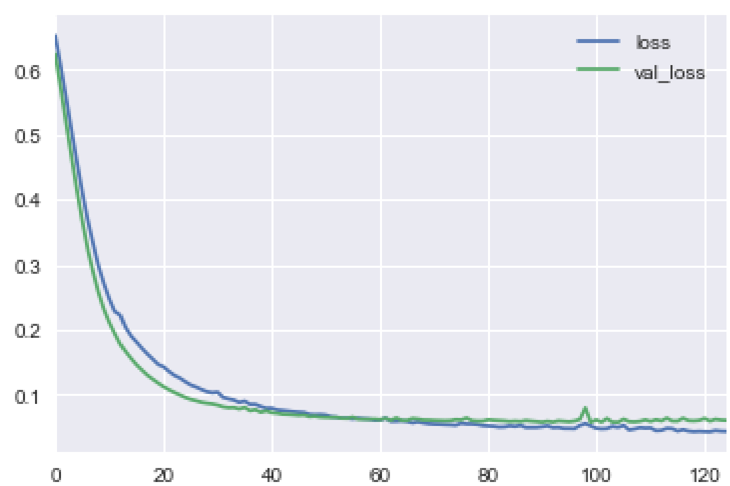
Out[532]:

	loss	val_loss
0	0.65%	0.62%
1	0.61%	0.57%
2	0.56%	0.52%

We can plot our validation losses.

In [533]:
```
model_loss.plot()
```

Out[533]: <matplotlib.axes._subplots.AxesSubplot at 0x131182b00>



We can import Dropout.

In [534]:
```
from tensorflow.keras.layers import Dropout
```

We can create our Neural Network model with Dropout.

In [535]:
```
model = Sequential()
model.add(Dense(30,activation='relu'))
model.add(Dropout(0.5)) #each neuron has 50% prob of being turned off, each
model.add(Dense(15,activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
```

We can fit our Neural Network model.

In [536]:
```
%%capture
model.fit(x=X_train,y=y_train,epochs=600,validation_data=(X_test,y_test),ca
```

We can create a dataframe of our validation losses.

In [537]:
```
import pandas as pd
model_loss = pd.DataFrame(model.history.history)
model_loss.head(3)
```

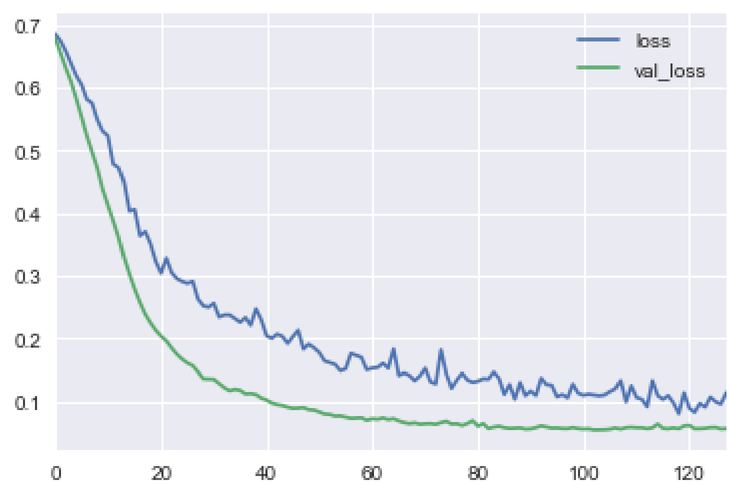
Out[537]:

	loss	val_loss
0	0.69%	0.68%
1	0.67%	0.65%
2	0.66%	0.63%

We can plot our validation losses.

In [538]:
```
model_loss.plot()
```

Out[538]: <matplotlib.axes._subplots.AxesSubplot at 0x151072c18>



We can create predictions for our Neural Network model.

In [539]:
```
predictions = model.predict_classes(X_test)
```

```
WARNING:tensorflow:From <ipython-input-539-bc83193b8b59>:1: Sequential.pr
edict_classes (from tensorflow.python.keras.engine.sequential) is depreca
ted and will be removed after 2021-01-01.
Instructions for updating:
Please use instead:* `np.argmax(model.predict(x), axis=-1)`,   if your mo
del does multi-class classification   (e.g. if it uses a `softmax` last-l
ayer activation).* `(model.predict(x) > 0.5).astype("int32")`,   if your
model does binary classification   (e.g. if it uses a `sigmoid` last-laye
r activation).
```

We can create a classification report for our Neural Network model.

In [540]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99        71
           1       0.98      0.98      0.98        43

   micro avg       0.98      0.98      0.98       114
   macro avg       0.98      0.98      0.98       114
weighted avg       0.98      0.98      0.98       114
```

We can create a confusion matrix for our Neural Network model.

In [541]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,predictions))
```

```
[[70  1]
 [ 1 42]]
```

We evaluated several machine learning algorithms with the Titanic dataset. We can also evaluate that dataset using Deep Learning. We start by setting up the data for classification.

In [542]:
```
import pandas as pd
import numpy as np
titanic = pd.read_csv('titanic.csv',index_col=1)
gender_dummies=pd.get_dummies(titanic['Sex'],prefix='gender',drop_first=Tru
titanic=pd.concat([titanic,gender_dummies],axis=1)
titanic.drop(['Sex'],axis=1,inplace=True)
embarked_dummies=pd.get_dummies(titanic['Embarked'],prefix='embarked',drop_
titanic=pd.concat([titanic,embarked_dummies],axis=1)
titanic.drop(['Embarked'],axis=1,inplace=True)
titanic['IsRich']=np.where(titanic['Fare']>titanic['Fare'].quantile(.80),1,
titanic.drop(['Fare'],axis=1,inplace=True)
class_dummies=pd.get_dummies(titanic['Pclass'],prefix='class',drop_first=Tr
titanic.drop(['Pclass'],axis=1,inplace=True)
titanic.drop(['Unnamed: 0','Name','Ticket','Cabin'],axis=1,inplace=True)
titanic=titanic.dropna()
X=titanic.drop('Survived',axis=1)
y=titanic['Survived']
```

We create a train-test split for our Neural Network, using the same random number as earlier.

In [543]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra
```

We can scale our data using Sklearn.

In [544]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train);
X_test = scaler.transform(X_test);
```

We can import our Tensorflow Models.

In [545]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Dropout
```

We can find the number of features of our model.

In [546]:
```
X_train.shape[1]
```

Out[546]: 7

We can create our Neural Network model.

In [547]:
```
model = Sequential()
model.add(Dense(7,activation = 'relu')) #7 neurons
model.add(Dense(7,activation = 'relu'))
model.add(Dense(1,activation = 'sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
```

We can import Early Stopping.

In [548]:
```
from tensorflow.keras.callbacks import EarlyStopping
```

We can use Early Stopping.

In [549]:
```
early_stop = (EarlyStopping(monitor='val_loss',mode='min',
             verbose=1,patience=25))
```

We can fit our Neural Network model.

In [550]:
```
%%capture
model.fit(x=X_train,y=y_train,epochs=600,validation_data=(X_test,y_test),ca
```

We can create a dataframe of validation losses.

In [551]:
```
import pandas as pd
model_loss = pd.DataFrame(model.history.history)
model_loss.head(3)
```

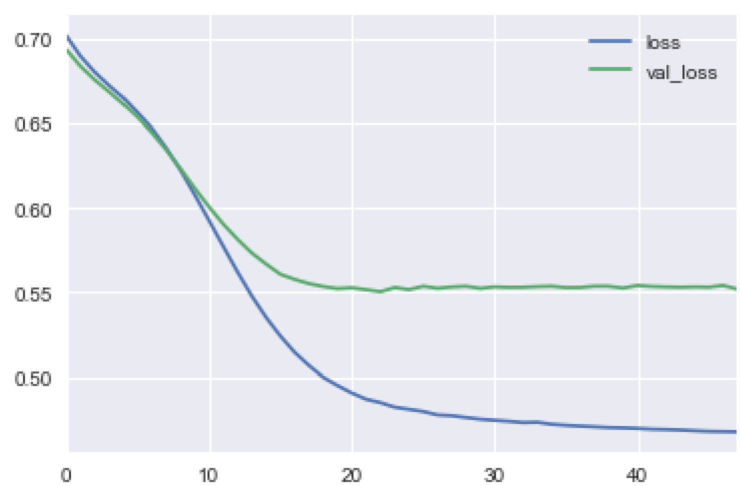
Out[551]:

	loss	val_loss
0	0.70%	0.69%
1	0.69%	0.68%
2	0.68%	0.68%

We can plot our validation losses.

In [552]:
```
model_loss.plot()
```

Out[552]: <matplotlib.axes._subplots.AxesSubplot at 0x12cbeae10>



We can create predictions for our Neural Network model.

In [553]:
```
predictions = model.predict_classes(X_test)
```

We can create a classification report for our Neural Network model.

In [554]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.78      0.82      0.80        87
           1       0.69      0.64      0.67        56

   micro avg       0.75      0.75      0.75       143
   macro avg       0.74      0.73      0.73       143
weighted avg       0.75      0.75      0.75       143
```

We can create a confusion matrix for our Neural Network model.

In [555]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,predictions))
```

```
[[71 16]
 [20 36]]
```

We can display accuracy score for our Neural Network model.

In [556]:
```
from sklearn.metrics import accuracy_score
NN_accuracy_score=accuracy_score(y_test,predictions)
print(format(NN_accuracy_score,".2%"))
```

```
74.83%
```

Section: Recommender Systems

We can load in the data to use for our recommender system.

In [557]:
```
import pandas as pd
column_names = ['user_id','item_id','rating','timestamp']
df = pd.read_csv('u.data',sep='\t',names = column_names)
df.head(3)
```

Out[557]:

	user_id	item_id	rating	timestamp
0	0	50	5	881250949
1	0	172	5	881250949
2	0	133	1	881250949

We can load in the data to use for our recommender system.

In [558]:
```
import pandas as pd
movie_titles = pd.read_csv('Movie_Id_Titles')
movie_titles.head(3)
```

Out[558]:

	item_id	title
0	1	Toy Story (1995)
1	2	GoldenEye (1995)
2	3	Four Rooms (1995)

We can merge two tables together.

In [559]:
```
import pandas as pd
df = pd.merge(df,movie_titles,on='item_id')
df.head(3)
```

Out[559]:

	user_id	item_id	rating	timestamp	title
0	0	50	5	881250949	Star Wars (1977)
1	290	50	5	880473582	Star Wars (1977)
2	79	50	4	891271545	Star Wars (1977)

We can create a dataframe of ratings using groupby.

In [560]:
```
import pandas as pd
ratings = pd.DataFrame(df.groupby('title')['rating'].mean())
ratings.head(3)
```

Out[560]:

	rating
title	
'Til There Was You (1997)	2.33%
1-900 (1994)	2.60%
101 Dalmatians (1996)	2.91%

We can add a column of the number of ratings to our dataframe.

In [561]:
```
import pandas as pd
ratings['num of ratings'] = pd.DataFrame(df.groupby('title')['rating'].coun
ratings.head(3)
```

Out[561]:

title	rating	num of ratings
'Til There Was You (1997)	2.33%	9
1-900 (1994)	2.60%	5
101 Dalmatians (1996)	2.91%	109

We can create a pivot table of all users and the ratings they gave to all movies.

In [562]:
```
moviemat = df.pivot_table(index='user_id',columns='title',values='rating')
moviemat.head(3)
```

Out[562]:

user_id \ title	'Til There Was You (1997)	1-900 (1994)	101 Dalmatians (1996)	12 Angry Men (1957)	187 (1997)	2 Days in the Valley (1996)	20,000 Leagues Under the Sea (1954)	2001: A Space Odyssey (1968)	3 Ninjas: High Noon At Mega Mountain (1998)	39 Steps, The (1935)	...
0	nan%	nan%	nan%	nan%	nan%	nan%	nan%	nan%	nan%	nan%	...
1	nan%	nan%	2.00%	5.00%	nan%	nan%	3.00%	4.00%	nan%	nan%	...
2	nan%	nan%	nan%	nan%	nan%	nan%	nan%	nan%	1.00%	nan%	...

3 rows × 1664 columns

We can find ratings of Star Wars for all users.

In [563]:
```
starwars_user_ratings = moviemat['Star Wars (1977)']
starwars_user_ratings.head(3)
```

Out[563]:
```
user_id
0    5.00%
1    5.00%
2    5.00%
Name: Star Wars (1977), dtype: float64
```

We can find which movies have high correlation in user ratings with Star Wars.

In [564]:
```
similar_to_starwars = moviemat.corrwith(starwars_user_ratings)
similar_to_starwars.head(3)
```

Out[564]:
```
title
'Til There Was You (1997)    0.87%
1-900 (1994)                -0.65%
101 Dalmatians (1996)        0.21%
dtype: float64
```

In [565]:
```
import pandas as pd
corr_starwars = pd.DataFrame(similar_to_starwars,columns=['Correlation'])
corr_starwars.dropna(inplace=True)
corr_starwars.head(3)
```

Out[565]:

title	Correlation
'Til There Was You (1997)	0.87%
1-900 (1994)	-0.65%
101 Dalmatians (1996)	0.21%

We can add number of ratings to our dataframe.

In [566]:
```
corr_starwars = corr_starwars.join(ratings['num of ratings'])
corr_starwars.head(3)
```

Out[566]:

title	Correlation	num of ratings
'Til There Was You (1997)	0.87%	9
1-900 (1994)	-0.65%	5
101 Dalmatians (1996)	0.21%	109

We can show the top movies recommended for a user who enjoyed Star Wars, with a set criterion such as the number of ratings.

In [567]:
```
corr_starwars[corr_starwars['num of ratings']>100].sort_values('Correlation
```

Out[567]:

title	Correlation	num of ratings
Star Wars (1977)	1.00%	584
Empire Strikes Back, The (1980)	0.75%	368
Return of the Jedi (1983)	0.67%	507
Raiders of the Lost Ark (1981)	0.54%	420
Austin Powers: International Man of Mystery (1997)	0.38%	130

Section: Natural Language Processing (NLP)

Often, we will want to deal with more unstructured data, such as text/words. This is where Natural Language Processing (NLP) comes in. The two most popular libraries are NLTK and Spacy. Spacy is the more up-and-coming choice and is generally more efficient, although a downside of Spacy is that you can't choose which algorithm to run behind the scenes. We can import Spacy.

In [568]:
```
import spacy
```

We can load in a pre-loaded small library for a model called nlp.

In [569]:
```
import spacy
nlp = spacy.load('en_core_web_sm')
```

We can load in a pre-loaded large library for a model called nlp.

In [570]:
```
import spacy
nlp = spacy.load('en_core_web_lg')
```

We can create a document object and applying text to our model.

In [571]:
```
doc = nlp(u'Tesla wants to buy a startup.')
```

We can find the part of speech and syntactic dependency of our text.

In [572]:
```
for i in doc:
    print(i.text, i.pos_, i.dep_)
```

```
Tesla PROPN nsubj
wants VERB ROOT
to PART aux
buy VERB xcomp
a DET det
startup NOUN dobj
. PUNCT punct
```

We can return tokens.

In [573]:
```
doc[0]
```

Out[573]:
```
Tesla
```

In [574]:
```
doc[0:2]
```

Out[574]:
```
Tesla wants
```

We can find the base form of a word.

In [575]:
```
doc[1].lemma_
```

Out[575]:
```
'want'
```

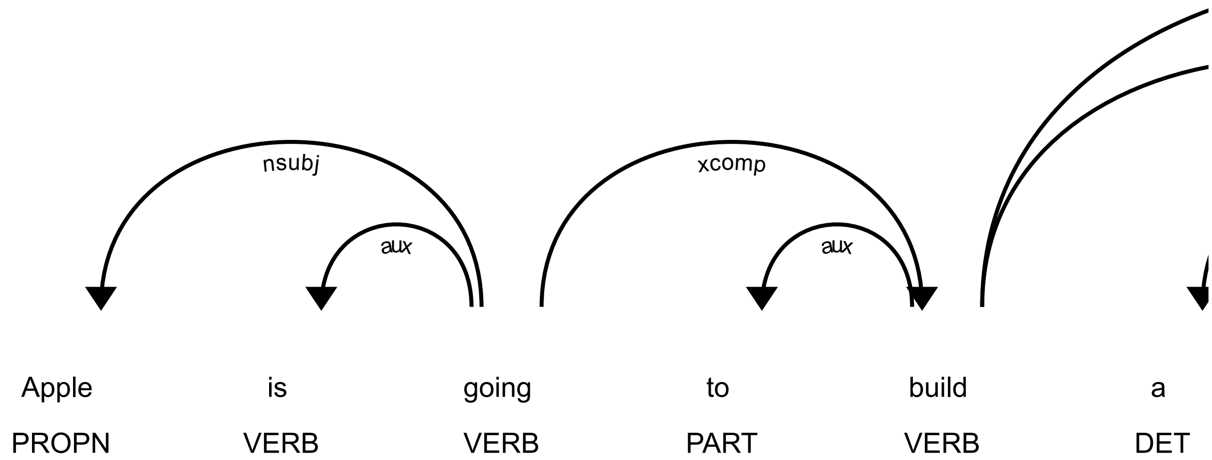
We can split sentences of a document.

In [576]:
```
doc = nlp(u"This is the first sentence. This is another sentence. This is t
for sentence in doc.sents:
    print(sentence)
```

```
This is the first sentence.
This is another sentence.
This is the last sentence.
```

We can visualize our text.

In [577]:
```
from spacy import displacy
doc = nlp(u"Apple is going to build a U.K. factory for $6 million.")
displacy.render(doc,style='dep',jupyter=True,options={'distance':110})
```



We might want to count certain similar words, such as boat, boats, and boating, as the same word. In this case we can perform stemming to chop off letters until the stem is reached. We can use PorterStemmer.

In [578]:
```
import nltk
from nltk.stem.porter import PorterStemmer
p_stemmer = PorterStemmer()
words = ['run','runner','ran','runs']
for word in words:
    print(word + ' Stem: ' + p_stemmer.stem(word))
```

```
run Stem: run
runner Stem: runner
ran Stem: ran
runs Stem: run
```

We can use SnowballStemmer.

In [579]:
```
import nltk
from nltk.stem.snowball import SnowballStemmer #diff set of rules to get tc
s_stemmer = SnowballStemmer(language='english')
words = ['run','runner','ran','runs']
for word in words:
    print(word + ' Stem: ' + s_stemmer.stem(word))
```

```
run Stem: run
runner Stem: runner
ran Stem: ran
runs Stem: run
```

Spacy doesn't have stemming, but it does have lemmatization, which arguably provides more information.

In [580]:
```
import spacy
np = spacy.load('en_core_web_sm')
doc = nlp(u"I am a runner running in a race because I love to run since I r
for token in doc:
    print(token.text,'\t',token.pos_,'\t',token.lemma,'\t',token.lemma_)
```

```
I        PRON    561228191312463089      -PRON-
am       VERB    10382539506755952630    be
a        DET     11901859001352538922    a
runner   NOUN    12640964157389618806    runner
running          VERB    12767647472892411841    run
in       ADP     3002984154512732771     in
a        DET     11901859001352538922    a
race     NOUN    8048469955494714898     race
because          ADP     16950148841647037698    because
I        PRON    561228191312463089      -PRON-
love     VERB    3702023516439754181     love
to       PART    3791531372978436496     to
run      VERB    12767647472892411841    run
since    ADP     10066841407251338481    since
I        PRON    561228191312463089      -PRON-
ran      VERB    12767647472892411841    run
today    NOUN    11042482332948150395    today
.        PUNCT   12646065887601541794    .
```

Each number points to a specific lemma inside the language library. We can see that running, run, and ran all get reduced to the lemma "run." We might also want to remove stop words, like "a" or "the" which might not add value to our models. We can print default stop words.

In [581]:
```
print(nlp.Defaults.stop_words)
```

```
{'hers', 'amount', 'several', 'it', 'doing', 'six', 'amongst', 'as', 'bu
t', 'were', 'myself', 'above', 'her', 'whence', 'thereby', 'within', 'a',
'hence', 'four', 'by', 'something', 'top', 'throughout', 'sometime', 'for
ty', 'somehow', 'get', 'least', 'at', 'my', 'had', 'we', 'would', 're',
'thru', 'most', 'please', 'some', 'thereafter', 'quite', 'more', 'onto',
'yet', 'namely', ''ll', 'out', 'then', 'former', 'various', 'she', 'canno
t', 'nine', '‘ll', '‘ve', 'our', 'did', 'everything', 'hereby', 'whateve
r', 'part', 'becomes', 'after', 'seem', 'being', 'and', 'against', 'unti
l', 'during', 'over', 'hereupon', 'whereas', 'across', 'none', 'whereafte
r', 'ever', 'could', 'was', 'still', 'n‘t', 'call', 'sometimes', 'using',
'down', 'anyway', 'than', 'so', 'whoever', 'such', 'he', 'whom', 'are',
'whether', 'while', 'however', 'although', 'besides', 'those', 'noone',
'ourselves', 'elsewhere', 'because', 'wherein', 'fifteen', 'herein', 'als
o', 'eleven', 'everyone', 'others', 'when', 'the', 'either', 'off', ''v
e', 'go', 'else', 'say', 'regarding', 'another', 'too', 'just', 'even',
'might', 'an', 'am', 'done', 'whereby', 'yourself', 'where', 'everywher
e', 'be', 'hundred', 'ours', 'up', 'eight', 'twelve', 'much', 'around',
'unless', 'almost', 'not', ''d', "'s", 'somewhere', 'no', 'formerly', 'mo
reover', 'about', 'someone', 'what', 'will', 'below', 'again', 'seemed',
'beyond', 'there', 'who', 'do', 'really', 'on', ''s', 'nothing', 'one',
'seeming', 'first', 'them', 'thence', 'beforehand', 'meanwhile', 'indee
d', 'make', 'they', 'hereafter', 'together', 'empty', 'in', 'though', 'ne
xt', 'toward', 'before', 'name', 'perhaps', 'five', 'often', 'therefore',
'herself', 'its', 'enough', 'latter', 'itself', 'or', 'is', 'further',
"'ll", '‘d', 'alone', 'their', 'otherwise', 'except', 'along', 'ca', 'wit
hout', 'see', 'fifty', 'take', 'nowhere', 'used', 'whenever', 'themselve
s', 'through', 'n't', 'rather', 'seems', 'via', 'has', 'towards', 'ever
y', '‘s', 'i', 'three', 'if', 'these', ''m', "n't", '‘re', 'sixty', 'las
t', 'wherever', 'this', 'show', 'under', 'you', 'own', 'mine', 'nobody',
'two', 'became', 'from', 'should', 'very', "'d", 'all', 'latterly', 'besi
de', 'made', 'into', 'mostly', 'any', 'same', 'between', 'whose', 'your
s', 'few', 'many', ''re', 'always', "'m", 'anything', 'twenty', 'us', 'wi
th', 'becoming', 'each', 'does', 'among', 'back', 'why', 'himself', 'beco
me', 'give', 'have', 'put', 'must', 'since', 'anyhow', 'may', 'upon', 'we
ll', 'neither', 'move', 'already', 'thereupon', 'per', 'anywhere', 'after
wards', 'full', 'ten', 'yourselves', 'his', 'me', 'been', 'for', 'which',
'behind', 'never', 'other', 'only', 'once', 'now', 'whither', 'due', 'you
r', 'whole', '‘m', 'third', 'of', 'can', "'ve", 'keep', 'that', 'both',
'therein', "'re", 'whereupon', 'thus', 'how', 'serious', 'side', 'to', 'h
im', 'nevertheless', 'bottom', 'here', 'anyone', 'front', 'nor', 'less'}
```

We can confirm the number of stop words.

In [582]:
```
stop_word_length = len(nlp.Defaults.stop_words)
print(stop_word_length)
```

```
326
```

We can check if a word is a stop word.

In [583]:
```
nlp.vocab['is'].is_stop
```
Out[583]:
```
True
```

We can add a word to a list of stop words.

In [584]:
```
nlp.Defaults.stop_words.add('btw')
nlp.vocab['btw'].is_stop = True
```

We can confirm the addition of our word to the length of our list of stop words.

In [585]:
```
len(nlp.Defaults.stop_words) == stop_word_length+1
```
Out[585]:
```
True
```

We can remove a word from a list of stop words.

In [586]:
```
nlp.Defaults.stop_words.remove('beyond')
nlp.vocab['beyond'].is_stop = False
```

We can pull in our data for Text Classification.

In [587]:
```
import pandas as pd
df=pd.read_csv('moviereviews2.tsv',sep='\t')
df.dropna(inplace=True)
df['label'].value_counts()
```
Out[587]:
```
pos    2990
neg    2990
Name: label, dtype: int64
```

As we can see, our dataset is perfectly balanced between positive and negative reviews. Therefore, we would expect a randomly guessing accuracy of 50%.

In [588]:
```
from sklearn.model_selection import train_test_split
X = df['review']
y = df['label']
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.33,ranc
```

We can fit our NLP model.

In [589]:
```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
text_clf = Pipeline([('tfidf',TfidfVectorizer()),
                     ('clf',LinearSVC())])
text_clf.fit(X_train,y_train);
```

We can create predictions for text classification.

In [590]:
```
predictions = text_clf.predict(X_test)
```

We can create a classification report for text classification.

In [591]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

         neg       0.93      0.91      0.92       991
         pos       0.91      0.94      0.92       983

   micro avg       0.92      0.92      0.92      1974
   macro avg       0.92      0.92      0.92      1974
weighted avg       0.92      0.92      0.92      1974
```

We can create a confusion matrix for text classification.

In [592]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,predictions))
```

```
[[900  91]
 [ 63 920]]
```

We can display accuracy score for text classification.

In [593]:
```
from sklearn.metrics import accuracy_score
print(accuracy_score(y_test,predictions))
```

```
0.9219858156028369
```

We can import VADER to test sentiment.

In [594]:
```
import nltk
nltk.download('vader_lexicon')
from nltk.sentiment.vader import SentimentIntensityAnalyzer
sid = SentimentIntensityAnalyzer()
```

```
[nltk_data] Downloading package vader_lexicon to
[nltk_data]     /Users/evanokin/nltk_data...
[nltk_data]   Package vader_lexicon is already up-to-date!
```

We can test a string to see its sentiment.

In [595]:
```
a = "This was a fantastic, great movie"
sid.polarity_scores(a)
```

Out[595]:
```
{'neg': 0.0, 'neu': 0.28, 'pos': 0.72, 'compound': 0.8271}
```

We can confirm that capitalizaiton factors into sentiment.

```
In [596]: a = "This was a FANTASTIC, GREAT movie"
          sid.polarity_scores(a)
```

```
Out[596]: {'neg': 0.0, 'neu': 0.247, 'pos': 0.753, 'compound': 0.8797}
```

We can confirm that punctuation factors into sentiment.

```
In [597]: a = "This was a FANTASTIC, GREAT movie!!!"
          sid.polarity_scores(a)
```

```
Out[597]: {'neg': 0.0, 'neu': 0.23, 'pos': 0.77, 'compound': 0.901}
```

We can read in a dataframe to test sentiment.

```
In [598]: import pandas as pd
          df = pd.read_csv('amazonreviews.tsv',sep='\t')
          df.head(3)
```

Out[598]:

	label	review
0	pos	Stuning even for the non-gamer: This sound tra...
1	pos	The best soundtrack ever to anything.: I'm rea...
2	pos	Amazing!: This soundtrack is my favorite music...

We can drop null values from our dataframe.

```
In [599]: df.dropna(inplace=True)
          df['label'].value_counts()
```

```
Out[599]: neg    5097
          pos    4903
          Name: label, dtype: int64
```

We can print the sentiment score for the first review, along with the review itself.

```
In [600]: print(sid.polarity_scores(df.iloc[0]['review']))
          df.iloc[0]['review']
```

```
          {'neg': 0.088, 'neu': 0.669, 'pos': 0.243, 'compound': 0.9454}
```

```
Out[600]: 'Stuning even for the non-gamer: This sound track was beautiful! It paint
          s the senery in your mind so well I would recomend it even to people who
          hate vid. game music! I have played the game Chrono Cross but out of all
          of the games I have ever played it has the best music! It backs away from
          crude keyboarding and takes a fresher step with grate guitars and soulful
          orchestras. It would impress anyone who cares to listen! ^_^'
```

VADER seems accurate on this review. We can use a lambda function to pull in sentiment analysis for all elements of a dataframe.

In [601]:
```
df['scores'] = df['review'].apply(lambda review: sid.polarity_scores(review
df.head(3)
```

Out[601]:

	label	review	scores
0	pos	Stuning even for the non-gamer: This sound tra...	{'neg': 0.088, 'neu': 0.669, 'pos': 0.243, 'co...
1	pos	The best soundtrack ever to anything.: I'm rea...	{'neg': 0.018, 'neu': 0.837, 'pos': 0.145, 'co...
2	pos	Amazing!: This soundtrack is my favorite music...	{'neg': 0.04, 'neu': 0.692, 'pos': 0.268, 'com...

We can pull out just the overall sentiment.

In [602]:
```
df['compound'] = df['scores'].apply(lambda d: d['compound'])
df.head()
```

Out[602]:

	label	review	scores	compound
0	pos	Stuning even for the non-gamer: This sound tra...	{'neg': 0.088, 'neu': 0.669, 'pos': 0.243, 'co...	0.95%
1	pos	The best soundtrack ever to anything.: I'm rea...	{'neg': 0.018, 'neu': 0.837, 'pos': 0.145, 'co...	0.90%
2	pos	Amazing!: This soundtrack is my favorite music...	{'neg': 0.04, 'neu': 0.692, 'pos': 0.268, 'com...	0.99%
3	pos	Excellent Soundtrack: I truly like this soundt...	{'neg': 0.09, 'neu': 0.615, 'pos': 0.295, 'com...	0.98%
4	pos	Remember, Pull Your Jaw Off The Floor After He...	{'neg': 0.0, 'neu': 0.746, 'pos': 0.254, 'comp...	0.98%

We can test if our sentiment exceeds a threshold.

In [603]:
```
df['comp_score'] = df['compound'].apply(lambda score: 'pos' if score >=0 el
df.head()
```

Out[603]:

	label	review	scores	compound	comp_score
0	pos	Stuning even for the non-gamer: This sound tra...	{'neg': 0.088, 'neu': 0.669, 'pos': 0.243, 'co...	0.95%	pos
1	pos	The best soundtrack ever to anything.: I'm rea...	{'neg': 0.018, 'neu': 0.837, 'pos': 0.145, 'co...	0.90%	pos
2	pos	Amazing!: This soundtrack is my favorite music...	{'neg': 0.04, 'neu': 0.692, 'pos': 0.268, 'com...	0.99%	pos
3	pos	Excellent Soundtrack: I truly like this soundt...	{'neg': 0.09, 'neu': 0.615, 'pos': 0.295, 'com...	0.98%	pos
4	pos	Remember, Pull Your Jaw Off The Floor After He...	{'neg': 0.0, 'neu': 0.746, 'pos': 0.254, 'comp...	0.98%	pos

We can create a classification report for our NLP model.

In [604]:
```
from sklearn.metrics import classification_report
print(classification_report(df['label'],df['comp_score']))
```

```
              precision    recall  f1-score   support

         neg       0.86      0.51      0.64      5097
         pos       0.64      0.91      0.75      4903

   micro avg       0.71      0.71      0.71     10000
   macro avg       0.75      0.71      0.70     10000
weighted avg       0.75      0.71      0.70     10000
```

We can create a confusion matrix for our NLP model.

In [605]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(df['label'],df['comp_score']))
```

```
[[2623 2474]
 [ 435 4468]]
```

We can display the accuracy score for our NLP model.

In [606]:
```
from sklearn.metrics import accuracy_score
accuracy_score(df['label'],df['comp_score'])
```

Out[606]:
```
0.7091
```

Clearly, VADER is better than randomly guessing, as we would expect 50% accuracy with random guessing.

Module 3: Finance Applications and Portfolio Management

Section: Time Value of Money

We can find the present value from a list of cash flows.

In [607]:
```
cash_flow_list_of_lists = [[1, 200], [4, 500], [7, 1600]]
rate=0.1
pv=0
for i in cash_flow_list_of_lists:
    pv+=i[1]/((1+rate)**i[0])
print(f"Present value: ${pv:9,.2f}")
```

```
Present value: $ 1,344.38
```

We can find the present value from a tuple of cash flows.

In [608]:
```
cash_flow_list_of_tuples = [(1, 200), (4, 500), (7, 1600)]
rate=0.1
pv=0
for i in cash_flow_list_of_tuples:
    pv+=i[1]/((1+rate)**i[0])
print(f"Present value: ${pv:9,.2f}")
```

```
Present value: $ 1,344.38
```

We can find the present value from a dictionary of cash flows.

In [609]:
```
cash_flow_dictionary = {1: 200, 4: 500, 7: 1600}
cash_list_keys=list(cash_flow_dictionary.keys())
cash_list_values=list(cash_flow_dictionary.values())
rate=0.1
pv=0
for i in range(1,4):
   pv+=cash_list_values[i-1]/((1+rate)**cash_list_keys[i-1])
print(f"Present value: ${pv:9,.2f}")
```

```
Present value: $ 1,344.38
```

We can find the future value from a list of cash flows.

In [610]:
```
cash_flow_list_of_lists = [[1, 200], [4, 500], [7, 1729]]
rate=0.1
end_year = 10
fv=0
for i in cash_flow_list_of_lists:
   fv+=i[1]*((1+rate)**(end_year-i[0]))
print(f"Future value: ${fv:9,.2f}")
```

```
Future value: $ 3,658.67
```

We can find the future value from a tuple of cash flows.

In [611]:
```
cash_flow_list_of_tuples = [(1, 200), (4, 500), (7, 1729)]
rate=0.1
end_year = 10
fv=0
for i in cash_flow_list_of_tuples:
   fv+=i[1]*((1+rate)**(end_year-i[0]))
print(f"Future value: ${fv:9,.2f}")
```

```
Future value: $ 3,658.67
```

We can find the future value from a dictionary of cash flows.

In [612]:
```
cash_flow_dictionary = {1: 200, 4: 500, 7: 1729}
cash_list_keys=list(cash_flow_dictionary.keys())
cash_list_values=list(cash_flow_dictionary.values())
rate=0.1
end_year = 10
fv=0
for i in range(1,4):
   fv+=cash_list_values[i-1]*((1+rate)**(end_year-cash_list_keys[i-1]))
print(f"Future value: ${fv:9,.2f}")
```

```
Future value: $ 3,658.67
```

Section: Yahoo Finance! API

We will need a way to import in the data from the web to python in real time. One way to do this is using the Yahoo Finance! API. We can import pandas datareader to establish the API.

In [613]:
```
import pandas_datareader.data as web
```

We can import daily stock returns.

In [614]:
```
import datetime
import pandas_datareader.data as web
start=datetime.datetime(2019,1,1)
end=datetime.datetime.now()
pull_list=['AAPL','FB']
df_api = web.get_data_yahoo(pull_list, start, end)
df_api.head(3)
```

Out[614]:

Attributes	Adj Close		Close		High		Low		Open
Symbols	AAPL	FB	AAPL	FB	AAPL	FB	AAPL	FB	AAPL
Date									
2019-01-02	154.79%	135.68%	157.92%	135.68%	158.85%	137.51%	154.23%	128.56%	154.89%
2019-01-03	139.38%	131.74%	142.19%	131.74%	145.72%	137.17%	142.00%	131.12%	143.98%
2019-01-04	145.33%	137.95%	148.26%	137.95%	148.55%	138.00%	143.80%	133.75%	144.53%

Alternatively,

In [615]:
```
import pandas_datareader.data as web
df_api = web.get_data_yahoo(pull_list, start, end,interval='d')
df_api.head(3)
```

Out[615]:

Attributes	Adj Close		Close		High		Low		Open
Symbols	AAPL	FB	AAPL	FB	AAPL	FB	AAPL	FB	AAPL
Date									
2019-01-02	154.79%	135.68%	157.92%	135.68%	158.85%	137.51%	154.23%	128.56%	154.89%
2019-01-03	139.38%	131.74%	142.19%	131.74%	145.72%	137.17%	142.00%	131.12%	143.98%
2019-01-04	145.33%	137.95%	148.26%	137.95%	148.55%	138.00%	143.80%	133.75%	144.53%

We can import weekly stock returns.

In [616]:
```
import pandas_datareader.data as web
df_api = web.get_data_yahoo(pull_list, start, end,interval='w')
df_api.head(3)
```

Out[616]:

Attributes	Adj Close		Close		High		Low		Open
Symbols	AAPL	FB	AAPL	FB	AAPL	FB	AAPL	FB	AAPL
Date									
2019-01-01	145.00%	138.05%	147.93%	138.05%	158.85%	138.87%	142.00%	128.56%	154.89%
2019-01-08	147.03%	145.39%	150.00%	145.39%	154.53%	146.57%	148.52%	139.54%	149.56%
2019-01-15	153.72%	150.04%	156.82%	150.04%	157.88%	152.43%	150.05%	145.99%	150.27%

We can import monthly stock returns.

In [617]:
```
import pandas_datareader.data as web
df_api = web.get_data_yahoo(pull_list, start, end,interval='m')
df_api.head(3)
```

Out[617]:

Attributes	Adj Close		Close		High		Low		Open
Symbols	AAPL	FB	AAPL	FB	AAPL	FB	AAPL	FB	AAPL
Date									
2019-01-01	163.15%	166.69%	166.44%	166.69%	169.00%	171.68%	142.00%	128.56%	154.89%
2019-02-01	169.72%	161.45%	173.15%	161.45%	175.87%	172.47%	165.93%	159.59%	166.96%
2019-03-01	186.99%	166.69%	189.95%	166.69%	197.69%	174.30%	169.50%	159.28%	174.28%

Section: Time Series Analysis

We can import stock prices with parsed dates.

In [618]:
```
import pandas as pd
df=pd.read_csv('walmart_stock.csv',index_col='Date',parse_dates=True)
df.head(3)
```

Out[618]:

	Open	High	Low	Close	Volume	Adj Close
Date						
2012-01-03	59.97%	61.06%	59.87%	60.33%	12668800	52.62%
2012-01-04	60.21%	60.35%	59.47%	59.71%	9593300	52.08%
2012-01-05	59.35%	59.62%	58.37%	59.42%	12768200	51.83%

We can resample a dataframe using annual rules and by taking the mean.

In [619]:
```
df.resample(rule='A').mean()
```

Out[619]:

Date	Open	High	Low	Close	Volume	Adj Close
2012-12-31	67.16%	67.60%	66.79%	67.22%	9239015.20%	59.39%
2013-12-31	75.26%	75.73%	74.84%	75.32%	6951496.03%	68.15%
2014-12-31	77.27%	77.74%	76.86%	77.33%	6515612.30%	71.71%
2015-12-31	72.57%	73.06%	72.03%	72.49%	9040769.44%	68.83%
2016-12-31	69.48%	70.02%	69.02%	69.55%	9371645.24%	68.05%

We can resample a dataframe using quarterly rules and by taking the mean.

In [620]:
```
df.resample(rule='Q').mean().head(3)
```

Out[620]:

Date	Open	High	Low	Close	Volume	Adj Close
2012-03-31	60.46%	60.81%	60.16%	60.52%	8850220.97%	52.88%
2012-06-30	62.89%	63.40%	62.59%	63.06%	11557947.62%	55.59%
2012-09-30	73.08%	73.55%	72.72%	73.17%	7871587.30%	64.89%

We can resample a dataframe using annual rule and by applying a customized function.

In [621]:
```
def first_day(entry):
    return entry[0]
df.resample('A').apply(first_day)
```

Out[621]:

Date	Open	High	Low	Close	Volume	Adj Close
2012-12-31	59.97%	61.06%	59.87%	60.33%	12668800	52.62%
2013-12-31	68.93%	69.24%	68.45%	69.24%	10390800	61.88%
2014-12-31	78.72%	79.47%	78.50%	78.91%	6878000	72.25%
2015-12-31	86.27%	86.72%	85.55%	85.90%	4501800	80.62%
2016-12-31	60.50%	61.49%	60.36%	61.46%	11989200	59.29%

We can display rolling averages of dataframe.

In [622]:
```
df.rolling(7).mean().head(10)
```

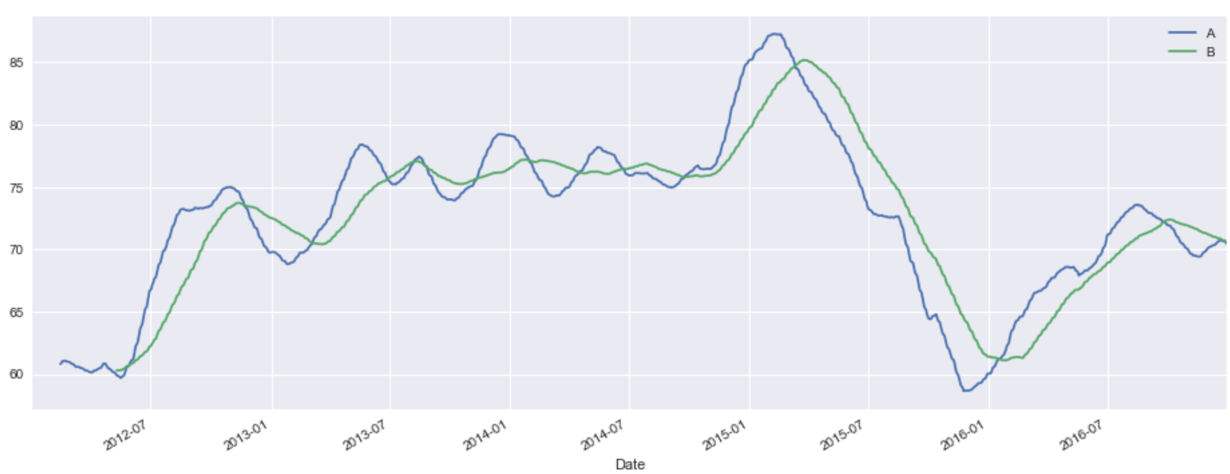
Out[622]:

Date	Open	High	Low	Close	Volume	Adj Close
2012-01-03	nan%	nan%	nan%	nan%	nan%	nan%
2012-01-04	nan%	nan%	nan%	nan%	nan%	nan%
2012-01-05	nan%	nan%	nan%	nan%	nan%	nan%
2012-01-06	nan%	nan%	nan%	nan%	nan%	nan%
2012-01-09	nan%	nan%	nan%	nan%	nan%	nan%
2012-01-10	nan%	nan%	nan%	nan%	nan%	nan%
2012-01-11	59.50%	59.90%	59.07%	59.44%	9007414.29%	51.84%
2012-01-12	59.47%	59.74%	59.01%	59.32%	8231357.14%	51.74%
2012-01-13	59.32%	59.64%	58.94%	59.30%	7965071.43%	51.72%
2012-01-17	59.40%	59.71%	59.11%	59.36%	7355328.57%	51.77%

We can create a plot of rolling averages.

In [623]:
```
df['A']=df['Close'].rolling(window=30).mean()
df['B']=df['Close'].rolling(window=90).mean()
df[['A','B']].plot(figsize=(16,6))
```

Out[623]: <matplotlib.axes._subplots.AxesSubplot at 0x179201a20>



We can pull stock prices into a dataframe.

In [624]:
```
import datetime
import pandas_datareader.data as web
start=datetime.datetime(2017,1,1)
end=datetime.datetime(2017,12,30)
aapl = web.get_data_yahoo('AAPL', start, end,interval='d')['Adj Close'].to_
fb = web.get_data_yahoo('FB', start, end,interval='d')['Adj Close'].to_fram
ibm = web.get_data_yahoo('IBM', start, end,interval='d')['Adj Close'].to_fr
amzn = web.get_data_yahoo('AMZN', start, end,interval='d')['Adj Close'].to_
```

We can calculate returns for each stock in a dataframe.

In [625]:
```
for stock_df in (aapl,fb,ibm,amzn):
    stock_df['Normed Return']=stock_df['Adj Close']/stock_df.iloc[0]['Adj C
```

We can assign allocations to stock and see how a portfolio of stocks moves over time.

In [626]:
```
for stock_df, alloc in zip((aapl,fb,ibm,amzn),[.25,.25,.25,.25]):
    stock_df['Allocation']=stock_df['Normed Return']*alloc
aapl.head(3)
```

Out[626]:

	Adj Close	Normed Return	Allocation
Date			
2017-01-03	110.39%	1.00%	0.25%
2017-01-04	110.27%	1.00%	0.25%
2017-01-05	110.83%	1.00%	0.25%

We can see how a portfolio account value moves over time.

In [627]:
```
start_amount=100
for stock_df in (aapl,fb,ibm,amzn):
    stock_df['Account_Value'] = stock_df['Allocation']*start_amount
aapl.head(3)
```

Out[627]:

	Adj Close	Normed Return	Allocation	Account_Value
Date				
2017-01-03	110.39%	1.00%	0.25%	25.00%
2017-01-04	110.27%	1.00%	0.25%	24.97%
2017-01-05	110.83%	1.00%	0.25%	25.10%

We can concatenate a dataframe of account positions.

In [628]:
```
all_positions=[aapl['Account_Value'],fb['Account_Value'],
               ibm['Account_Value'],amzn['Account_Value']]
portfolio_values = pd.concat(all_positions,axis=1)
portfolio_values.columns = ['AAPL Position','FB Position','IBM Position','A
portfolio_values['Total Position'] = portfolio_values.sum(axis=1)
portfolio_values.head(3)
```

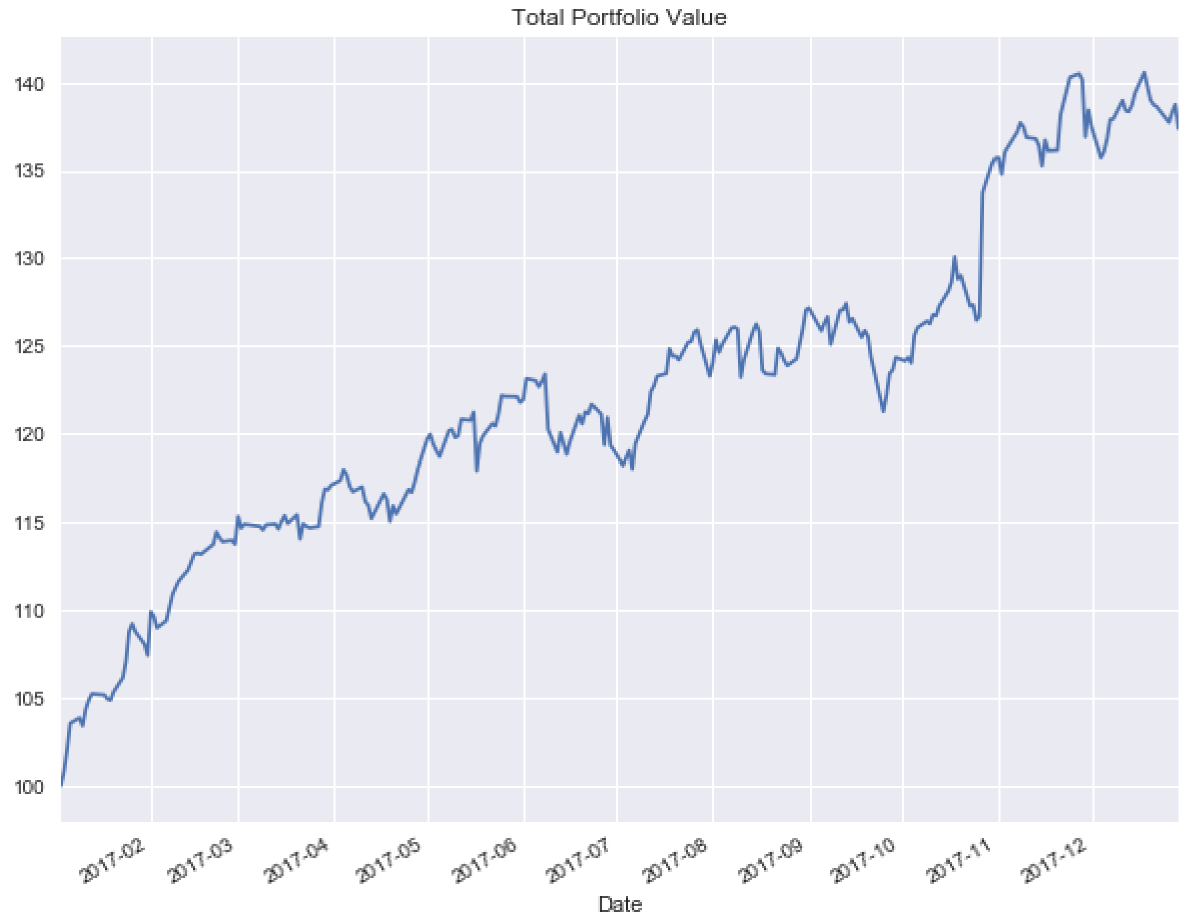
Out[628]:

Date	AAPL Position	FB Position	IBM Position	AMAZON Position	Total Position
2017-01-03	25.00%	25.00%	25.00%	25.00%	100.00%
2017-01-04	24.97%	25.39%	25.31%	25.12%	100.79%
2017-01-05	25.10%	25.82%	25.23%	25.89%	102.03%

We can plot how a total position of stocks moves over time.

In [629]:
```
import matplotlib.pyplot as plt
%matplotlib inline
portfolio_values['Total Position'].plot(figsize=(10,8))
plt.title('Total Portfolio Value')
```

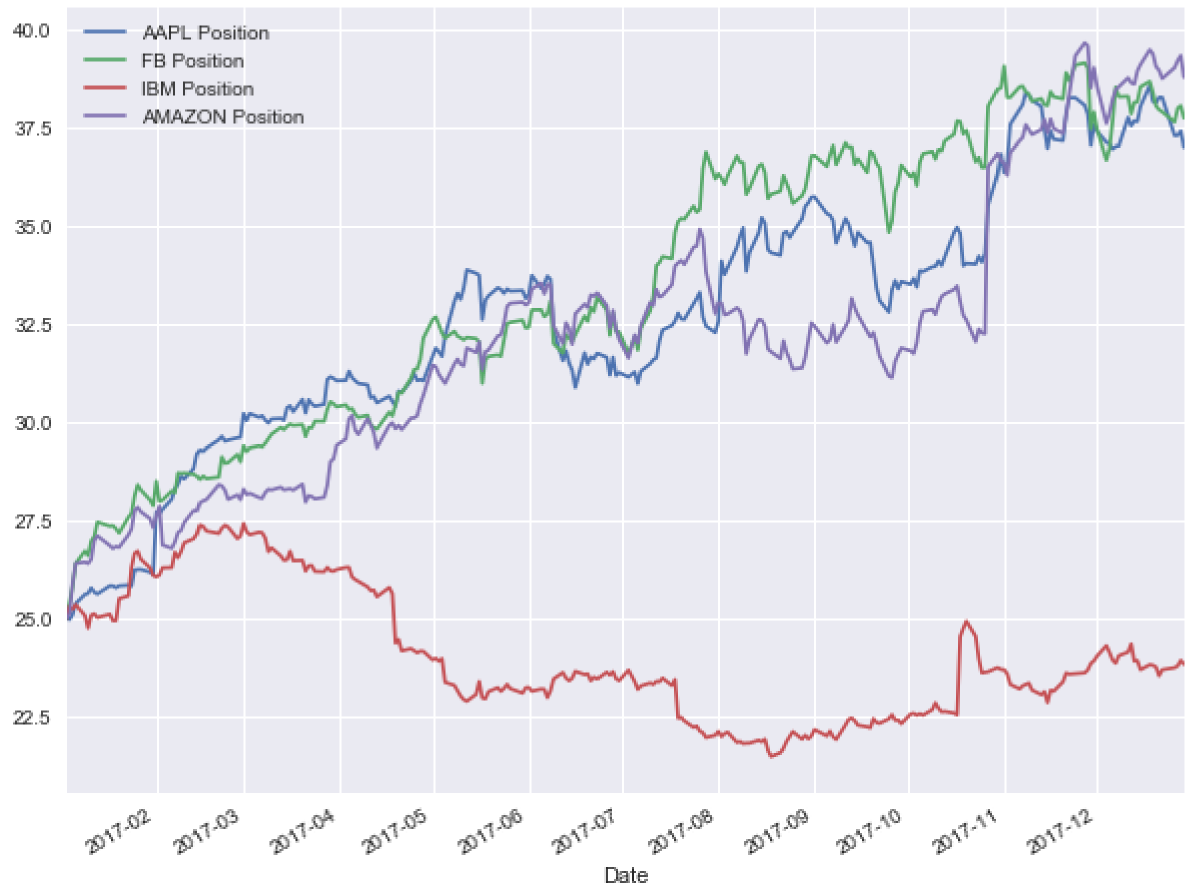
Out[629]: Text(0.5, 1.0, 'Total Portfolio Value')



We can plot how individual stocks in a portfolio move over time.

In [630]:
```
portfolio_values.drop('Total Position',axis=1).plot(figsize=(10,8))
```

Out[630]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x16d0dee48>
```



We can calculate daily returns for a portfolio of stocks.

In [631]:
```
portfolio_values['Daily Return']=portfolio_values['Total Position'].pct_cha
portfolio_values.head()
```

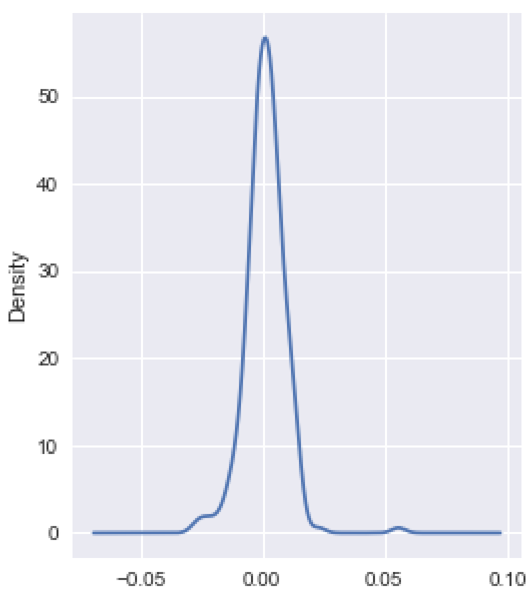
Out[631]:

Date	AAPL Position	FB Position	IBM Position	AMAZON Position	Total Position	Daily Return
2017-01-03	25.00%	25.00%	25.00%	25.00%	100.00%	nan%
2017-01-04	24.97%	25.39%	25.31%	25.12%	100.79%	0.01%
2017-01-05	25.10%	25.82%	25.23%	25.89%	102.03%	0.01%
2017-01-06	25.38%	26.40%	25.35%	26.40%	103.53%	0.01%
2017-01-09	25.61%	26.72%	25.07%	26.43%	103.83%	0.00%

We can display a KDE of daily returns of stocks.

In [632]:
```
portfolio_values['Daily Return'].plot(kind='kde',figsize=(4,5))
```

Out[632]: <matplotlib.axes._subplots.AxesSubplot at 0x179201320>



We can calculate the overall cumulative return on a portfolio.

In [633]:
```
cumulative_return = 100 * (portfolio_values['Total Position'][-1]/portfolic
cumulative_return
```

Out[633]: 37.38420011510728

We can calculate the daily Sharpe Ratio of a portfolio.

In [634]:
```
Sharpe_Ratio = portfolio_values['Daily Return'].mean() / portfolio_values['
Sharpe_Ratio
```

Out[634]: 0.15953830268691177

We can calculate the annualized Sharpe Ratio of a portfolio.

In [635]:
```
Annualized_Sharpe_Ratio = (252**0.5) * Sharpe_Ratio
Annualized_Sharpe_Ratio
```

Out[635]: 2.532592040993498

We can concatenate stock returns into a single dataframe.

In [636]:
```
import datetime
import pandas as pd
import pandas_datareader.data as web
start=datetime.datetime(2017,1,1)
end=datetime.datetime(2017,12,30)
aapl = web.get_data_yahoo('AAPL', start, end,interval='d')['Adj Close'].to_
fb = web.get_data_yahoo('FB', start, end,interval='d')['Adj Close'].to_fram
ibm = web.get_data_yahoo('IBM', start, end,interval='d')['Adj Close'].to_fr
amzn = web.get_data_yahoo('AMZN', start, end,interval='d')['Adj Close'].to_
stocks = pd.concat([aapl['Adj Close'],fb['Adj Close'],ibm['Adj Close'],amzn
stocks.columns = ['aapl','fb','ibm','amzn']
stocks.head(3)
```

Out[636]:

	aapl	fb	ibm	amzn
Date				
2017-01-03	110.39%	116.86%	143.49%	753.67%
2017-01-04	110.27%	118.69%	145.27%	757.18%
2017-01-05	110.83%	120.67%	144.79%	780.45%

We can find the average daily percent change of all stocks in a dataframe.

In [637]:
```
stocks.pct_change(1).mean()
```

Out[637]:
```
aapl    0.00%
fb      0.00%
ibm    -0.00%
amzn    0.00%
dtype: float64
```

We can observe a correlation matrix of all stocks in a portfolio.

In [638]:
```
stocks.pct_change(1).corr()
```

Out[638]:

	aapl	fb	ibm	amzn
aapl	1.00%	0.54%	-0.01%	0.51%
fb	0.54%	1.00%	-0.01%	0.65%
ibm	-0.01%	-0.01%	1.00%	0.00%
amzn	0.51%	0.65%	0.00%	1.00%

We can find the log returns of all stocks in a portfolio.

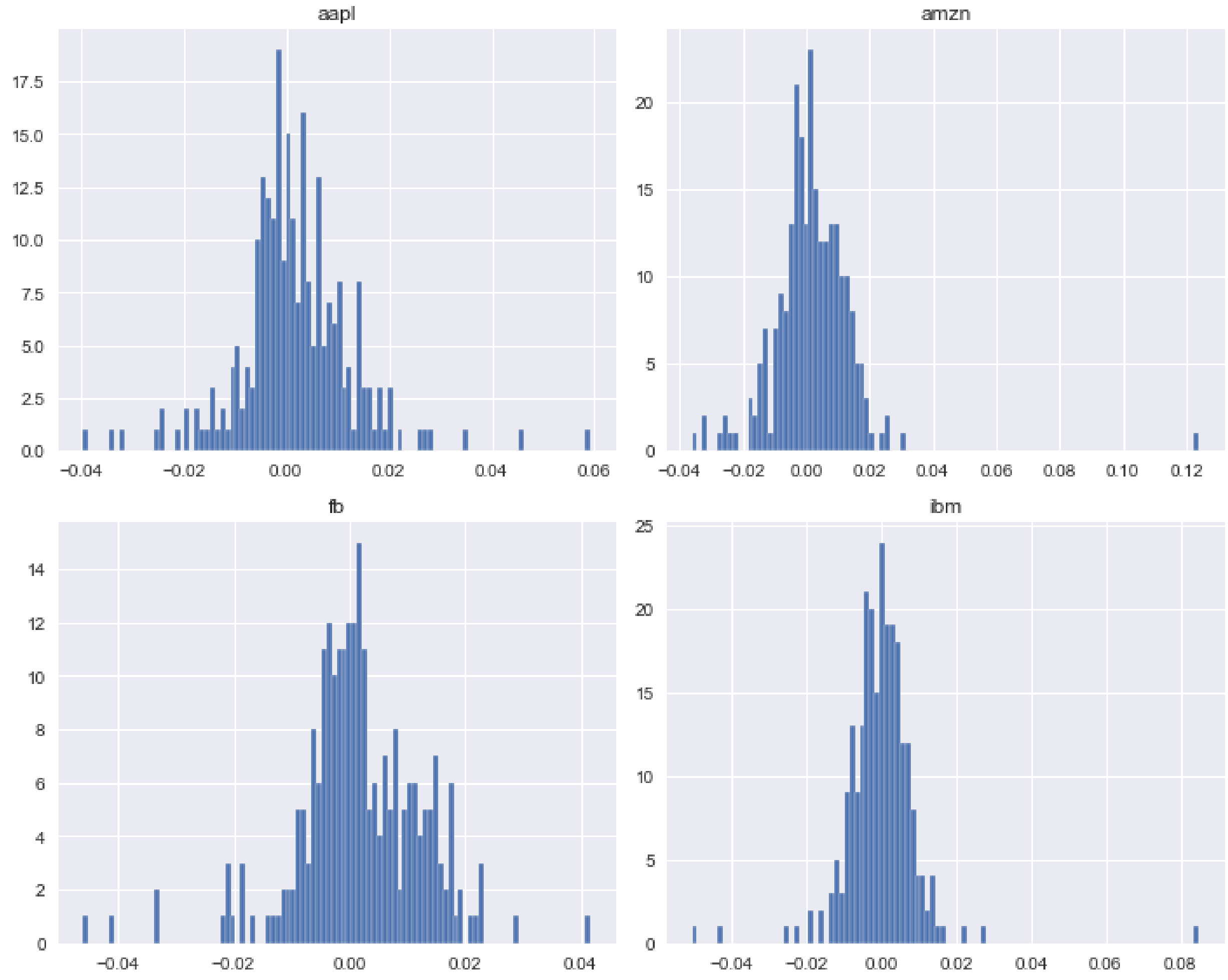
In [639]:
```
import numpy as np
log_returns = np.log(stocks/stocks.shift(1))
log_returns.head(3)
```

Out[639]:

	aapl	fb	ibm	amzn
Date				
2017-01-03	nan%	nan%	nan%	nan%
2017-01-04	-0.00%	0.02%	0.01%	0.00%
2017-01-05	0.01%	0.02%	-0.00%	0.03%

We can plot histograms of daily log returns for stocks in a dataframe using tight layout.

In [640]:
```
import matplotlib.pyplot as plt
%matplotlib inline
log_returns.hist(bins=100,figsize=(10,8))
plt.tight_layout()
```



We can calculate the mean of daily log returns for all stocks in a dataframe.

In [641]:
```
log_returns.mean()
```

Out[641]:
```
aapl     0.00%
fb       0.00%
ibm     -0.00%
amzn     0.00%
dtype: float64
```

We can calculate covariances of log returns.

In [642]:
```
log_returns.cov() *252
```

Out[642]:

	aapl	fb	ibm	amzn
aapl	0.03%	0.02%	-0.00%	0.02%
fb	0.02%	0.03%	-0.00%	0.02%
ibm	-0.00%	-0.00%	0.02%	0.00%
amzn	0.02%	0.02%	0.00%	0.04%

We can generate random weights to allocate towards stocks in a portfolio.

In [643]:
```
import numpy as np
np.random.seed(88)
print(stocks.columns)
weights=np.array(np.random.random(4))
print('Random Weights:')
print(weights)
print('Rebalanced Weights:')
weights = weights/np.sum(weights) #normalize
print(weights)
```

```
Index(['aapl', 'fb', 'ibm', 'amzn'], dtype='object')
Random Weights:
[0.64755105 0.50714969 0.52834138 0.8962852 ]
Rebalanced Weights:
[0.25105424 0.19662091 0.20483689 0.34748797]
```

We can calculate the expected return of a portfolio with random weights.

In [644]:
```
import numpy as np
print('Expected Portfolio Return')
expected_return = np.sum(log_returns.mean() * weights * 252)
print(expected_return)
```

```
Expected Portfolio Return
0.324963858036653
```

We can calculate the expected volatility of a portfolio.

In [645]:
```
import numpy as np
print('Expected Volatility')
expected_volatility = np.sqrt(np.dot(weights.T,np.dot(log_returns.cov()*252
print(expected_volatility)
```

```
Expected Volatility
0.1308450446858187
```

We can calculate the Sharpe Ratio for a portfolio.

In [646]:
```
print('Sharpe Ratio')
SR = expected_return / expected_volatility
print(SR)
```

```
Sharpe Ratio
2.483577875011979
```

We can generate simulations to calculate portfolio returns, volatility, and Sharpe Ratio.

In [647]:
```
import numpy as np
number_portfolios = 5000
all_weights = np.zeros((number_portfolios,len(stocks.columns)))
returns_array = np.zeros(number_portfolios)
volatility_array = np.zeros(number_portfolios)
sharpe_array = np.zeros(number_portfolios)
for i in range(number_portfolios):
    weights=np.array(np.random.random(4))
    weights=weights/np.sum(weights)
    all_weights[i,:] = weights #save weights
    returns_array[i] = np.sum ((log_returns.mean() * weights) * 252)
    volatility_array[i] = np.sqrt(np.dot(weights.T,np.dot(log_returns.cov()
    sharpe_array[i] = returns_array[i]/volatility_array[i]
```

We can find the maximum Sharpe Ratio across simulations.

In [648]:
```
sharpe_array.max()
```

Out[648]:
```
2.715836703186296
```

We can find the corresponding index of the maximum Sharpe Ratio.

In [649]:
```
sharpe_array.argmax()
```

Out[649]:
```
698
```

We can find optimal weights corresponding to the maximum Sharpe Ratio.

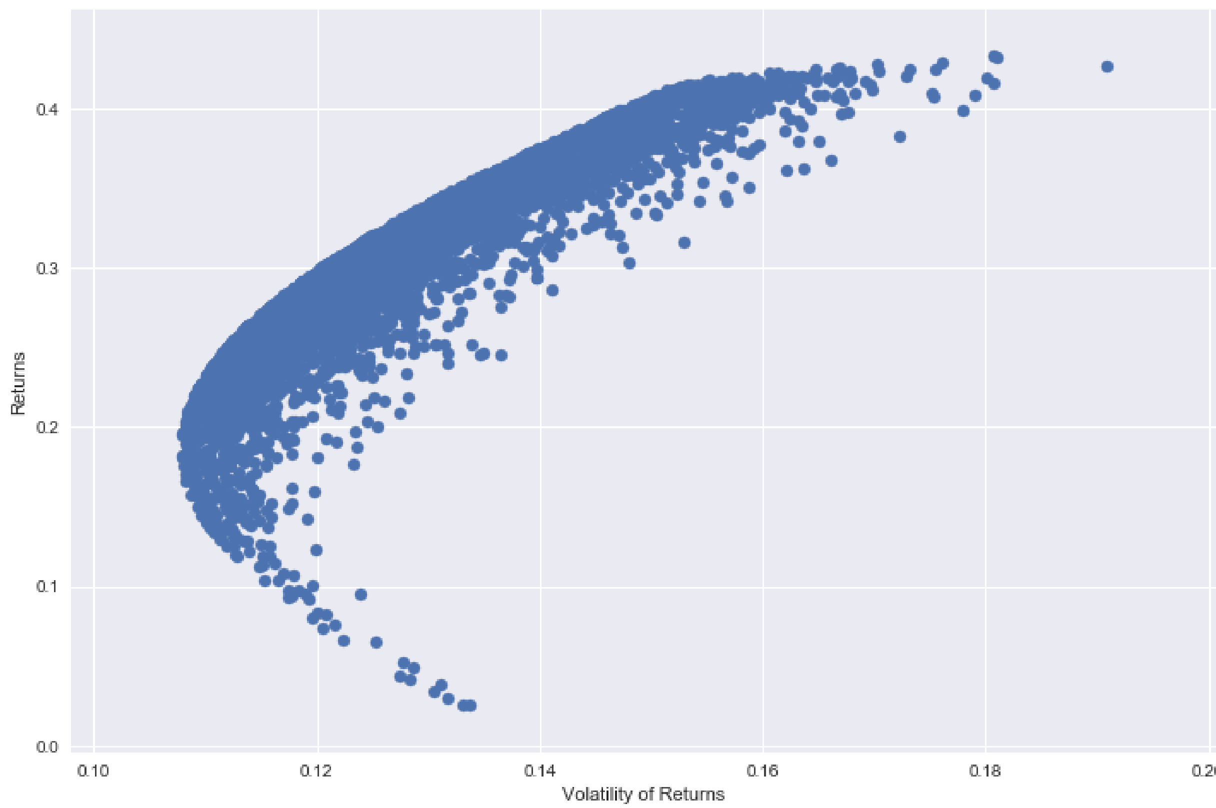
In [650]:
```
all_weights[698,:]
```

Out[650]:
```
array([0.40663288, 0.42528697, 0.00164522, 0.16643493])
```

We can plot an Efficient Frontier across simulations.

In [651]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure(figsize=(12,8))
plt.scatter(volatility_array,returns_array)
plt.xlabel('Volatility of Returns')
plt.ylabel('Returns')
```

Out[651]:
```
Text(0, 0.5, 'Returns')
```



We can create a function to calculate return, volatility and Sharpe Ratio.

In [652]:
```
import numpy as np
def get_returns_volatility_sharpe(weights):
    weights=np.array(weights)
    returns=np.sum(log_returns.mean()*weights*252)
    volatility=np.sqrt(np.dot(weights.T,np.dot(log_returns.cov()*252,weight
    sharpe=returns/volatility
    return np.array([returns,volatility,sharpe])
```

We can import the scipy.optimize library.

In [653]:
```
from scipy.optimize import minimize
```

We can write a function for negative Sharpe Ratio.

In [654]:
```
def neg_sharpe(weights):
    return get_returns_volatility_sharpe(weights)[2]*-1 #minimize negative
```

We can write a function to check generated weights of a portfolio.

In [655]:
```
def check_sum(weights):
    return np.sum(weights) - 1 #return 0 if sum of weights is 1
```

We can set up constraints for portfolio optimization.

In [656]:
```
constraints = ({'type':'eq','fun':check_sum})
```

We can set up bounds for portfolio optimization.

In [657]:
```
bounds = ((0,1),(0,1),(0,1),(0,1))
```

We can set up an initial guess for portfolio optimization.

In [658]:
```
initial_guess = [0.25,0.25,0.25,0.25]
```

We can display optimal results for portfolio optimization.

In [659]:
```
optimal_results = minimize(neg_sharpe,initial_guess,
                           method='SLSQP',bounds=bounds,constraints=constra
optimal_results
```

Out[659]:
```
     fun: -2.718308068017512
     jac: array([-9.44733620e-06,  2.47359276e-06,  2.88586408e-01,  1.37
090683e-05])
 message: 'Optimization terminated successfully.'
    nfev: 31
     nit: 5
    njev: 5
  status: 0
 success: True
       x: array([3.70203256e-01, 4.53471546e-01, 1.59095359e-17, 1.763251
99e-01])
```

We can display the optimal weights of a portfolio optimization.

In [660]:
```
optimal_results.x
```

Out[660]:
```
array([3.70203256e-01, 4.53471546e-01, 1.59095359e-17, 1.76325199e-01])
```

We can utilize a function on optimal weights of portfolio optimization.

In [661]:
```
get_returns_volatility_sharpe(optimal_results.x)
```

Out[661]:
```
array([0.41287068, 0.15188517, 2.71830807])
```