Python Introductory Workshop by University of San Diego (USD)

Leonid Shpaner
January 21, 2022

Whereas JupyterLab and Jupyter Notebook are the two most commonly used interactive computing platforms warehoused within the Anaconda distribution, data scientists can also leverage the cloud-based coding environment of Google Colab.

https://colab.research.google.com/?utm_source=scs-index

JupyterLab Basics

https://jupyterlab.readthedocs.io/en/stable/user/interface.html

Cells

[1]:
```python
# This is a code cell/block!

# basic for loop
for x in [1,2,3]:
    print(x)
```

```
1
2
3
```

This is a markdown cell!
Level 3 Heading
italics characters are surrounded by one asterisk
bold characters are surrounded by two asterisks
* one asterisk in front of an item of text can serve as a bullet point.
* to move the text to the next line, ensure to enter two spaces after the line.
1. Simply number the items on a list using normal numering schema.
2. If you have numbers in the same markdown cell as bullet points (i.e., below), 3. skip 4 spaces after the last bullet point and then begin numbering.

Here's a guide for syntax: https://www.markdownguide.org/basic-syntax/

Python Basics

We can use a code cell to conduct basic math operations as follows.

[2]:
```python
2+2
```

[2]: 4

Order of operations in Python is just as important.

[3]:
```python
1+3*10 == (1+3)*10
```

[3]: False

For more basic operations including but not limited to taking the square root, log, and generally more advanced mathematical operations, we will have to import our first library (math) as follows:

[4]:
```python
import math
```

However, if we run two consecutive lines below without assigning to a variable or parsing in a print statement ahead of the function calls, only the latest (last) function will print the output of what is in the cell block.

[5]:
```python
math.sqrt(9)
math.log10(100)
math.factorial(10)
```

[5]:
```
3628800
```

Let us try this again, by adding `print()` statements in front of the three functions, respectively.

[6]:
```python
# print the output on multiple lines
print(math.sqrt(9))
print(math.log10(100))
print(math.factorial(10))
```

```
3.0
2.0
3628800
```

What is a string?

A string is simply any open alpha/alphanumeric text characters surrounded by either single quotation marks or double quotation marks, with no preference assigned for either single or double quotation marks, with a print statement being called prior to the strings. For example,

[7]:
```python
print('This is a string.')
print( "This is also a string123.")
```

```
This is a string.
This is also a string123.
```

Strings in Python are arrays of bytes that represent "Unicode characters" (GeeksforGeeks, 2021).

Creating Objects

Unlike R, Python uses the '=' symbol for making assignment statements. we can `print()` what is contained in the assignment or just call the assignment to the string as follows:

[8]:
```python
cheese = 'pepper jack'
cheese
```

[8]:
```
'pepper jack'
```

Determining and setting the current working directory

The importance of determining and setting the working directory cannot be stressed enough. 1. Run `import os` to import operating system module. 2. then assign `os.getcwd()` to `cwd`. 3. You may choose to print the working directory using the print function as shown below. 4. Or change the working directory by running `os.chdir('')`.

[9]:
```python
import os # import the operating system module
cwd = os.getcwd()
# Print the current working directory
print("Current working directory: {0}".format(cwd))

# Change the current working directory
# os.chdir('')

# Print the current working directory
print("Current working directory: {0}".format(os.getcwd()))
```

```
Current working directory: C:\Users\lshpaner\Desktop\Accidents Dataset\Github
Repository\dse\Python Session
Current working directory: C:\Users\lshpaner\Desktop\Accidents Dataset\Github
Repository\dse\Python Session
```

Installing Libraries

To install most common libraries, simply type in the command `pip install followed by library name` into an empty code cell and run it.

[10]:
```python
# pip install pandas
```

Loading Libraries

For data science applications, we most commonly use pandas for "data structures and data analysis tools" (Pandas, 2021) and NumPy for "scientific computing with Python" (Numpy.org, n.d.).

Ideally, at the beginning of a project, we will create an empty cell block that loads all of the libraries that we will be using. However, as we progress throughout this tutorial, we will load the necessary libraries separately.

Let us now load these two libraries into an empty code cell block using `import name of the library as abbreviated form`.

[11]:
```python
import pandas as pd
import numpy as np
```

Sometimes, `Python` throws warning messages in pink on the output of code cell blocks that otherwise run correctly.

[12]:
```python
import warnings
warnings.warn('This is an example warning message.') # displaying warning
```

```
<ipython-input-12-300454090b32>:2: UserWarning: This is an example warning message.
  warnings.warn('This is an example warning message.') # displaying warning
```

Sometimes, it is helpful to see what these warnings are saying as added layers of de-bugging. However, they may also create unsightly output. For this reason, we will suppress any and all warning messages for the remainder of this tutorial.

To disable/suppress warning messages, let us write the following:

[13]:
```python
import warnings
warnings.filterwarnings("ignore")
```

Data Types

Text Type (string): `str`
Numeric Types: `int`, `float`, `complex`
Sequence Types: `list`, `tuple`, `range`
Mapping Type: `dict` - dictionary (used to store key:value pairs)
Logical: `bool` - boolean (True or False)
Binary Types: `bytes`, `bytearray`, `memoryview`

Let us convert an integer to a string. We do this using the `str()` function. Recall, how in R, this same function call is designated for something completely different - inspecting the structure of the dataframe.

We can also examine `floats` and `bools` as follows:

[14]:
```python
# assign the variable to an int
int_numb  = 2356
print('Integer:', int_numb)

# assign the variable to a float
float_number = 2356.0
print('Float:', float_number)

# convert the variable to a string
str_numb = str(int_numb)
print('String:',str_numb)

# convert variable from float to int
int_number = int(float_number)

# boolean
bool1 = 2356 > 235
bool2 = 2356 == 235
print(bool1)
print(bool2)
```

```
Integer: 2356
Float: 2356.0
String: 2356
True
False
```

Data Structures

What is a variable? A variable is a container for storing a data value, exhibited as a reference to "to an object in memory which means that whenever a variable is assigned to an instance, it gets mapped to that instance. A variable in R can store a vector, a group of vectors or a combination of many R objects" (GeeksforGeeks, 2020).

There are 3 most important data structures in Python: vector, matrix, and dataframe.

Vector: the most basic type of data structure within R; contains a series of values of the same data class. It is a "sequence of data elements" (Thakur, 2018).

Matrix: a 2-dimensional version of a vector. Instead of only having a single row/list of data, we have rows and columns of data of the same data class.

Dataframe: the most important data structure for data science. Think of dataframe as loads of vectors pasted together as columns. Columns in a dataframe can be of different data class, but values within the same column must be the same data class.

Creating Objects

We can make a one-dimensional horizontal list as follows:

[15]:
```
list1 = [0, 1, 2, 3]
list1
```

[15]:
```
[0, 1, 2, 3]
```

or a one-dimensional vertical list as follows:

[16]:
```
list2 = [[1],
         [2],
         [3],
         [4]]
list2
```

[16]:
```
[[1], [2], [3], [4]]
```

Vectors and Their Operations

Now, to vectorize these lists, we simply assign it to the `np.array()` function call:

[17]:
```
vector1 = np.array(list1)
print(vector1)
print('\n') # for printing an empty new line
            # between outputs
vector2 = np.array(list2)
print(vector2)
```

```
[0 1 2 3]



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

Running the following basic between vector arithmetic operations (addition, subtraction, and division, respectively) changes the resulting data structures from one-dimensional arrays to two-dimensional matrices.

[18]:
```
# adding vector 1 and vector 2
addition = vector1 + vector2

# subtracting vector 1 and vector 2
subtraction = vector1 - vector2

# multiplying vector 1 and vector 2
multiplication = vector1 * vector2
```

```
# divifing vector 1 by vector 2
division = vector1 / vector2

# Now let's print the results of these operations
print('Vector Addition: ', '\n', addition, '\n')
print('Vector Subtraction:', '\n', subtraction, '\n')
print('Vector Multiplication:', '\n', multiplication, '\n')
print('Vector Division:', '\n', division)
```

```
Vector Addition:
 [[1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]
 [4 5 6 7]]

Vector Subtraction:
 [[-1  0  1  2]
 [-2 -1  0  1]
 [-3 -2 -1  0]
 [-4 -3 -2 -1]]

Vector Multiplication:
 [[ 0  1  2  3]
 [ 0  2  4  6]
 [ 0  3  6  9]
 [ 0  4  8 12]]

Vector Division:
 [[0.         1.         2.         3.        ]
 [0.         0.5        1.         1.5       ]
 [0.         0.33333333 0.66666667 1.        ]
 [0.         0.25       0.5        0.75      ]]
```

Similarly, a vector of logical strings will contain

[19]:
```
vector3 = np.array([True, False, True, False, True])
vector3
```

[19]:
```
array([ True, False,  True, False,  True])
```

Whereas in R, we use the `length()` function to measure the length of an object (i.e., vector, variable, or dataframe), we apply the `len()` function in `Python` to determine the number of members inside this object.

[20]:
```
len(vector3)
```

[20]:
```
5
```

Let us say for example, that we want to access the third element of `vector1` from what we defined above. In this case, the syntax is the same as in R. We can do so as follows:

[21]:
```
vector1[3]
```

[21]:
```
3
```

Let us now say we want to access the first, fifth, and ninth elements of this dataframe. To this end, we do the following:

[22]:
```
vector4 = np.array([1,3,5,7,9,20,2,8,10,35,76,89,207])
vector4_index = vector4[1], vector4[5], vector4[9]
vector4_index
```

[22]:
```
(3, 20, 35)
```

What if we wish to access the third element on the first row of this matrix?

[23]:
```
# create (define) new matrix
matrix1 = np.array([[1,2,3,4,5], [6,7,8,9,10],
                   [11,12,13,14,15]])
print(matrix1)

print('\n','3rd element on 1st row:', matrix1[0,2])
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]

 3rd element on 1st row: 3
```

Counting Numbers and Accessing Elements in Python

Whereas it would make sense to start counting items in an array with the number `1` like we do in R, this is not true in Python. We `ALWAYS` start counting items with the number `0` as the first number of any given array in Python.

What if we want to access certain elements within the dataframe? For example:

[24]:
```
# find the length of vector 1
print(len(vector1))

# get all elements
print(vector1[0:4])

# get all elements except last one
print(vector1[0:3])
```

```
4
[0 1 2 3]
[0 1 2]
```

Mock Dataframe Examples

Unlike R, creating a dataframe in Python involves a little bit more work. For example, we will be using the pandas library to create what is called a pandas dataframe using the `pd.DataFrame()` function and map our variables to a dictionary. Like we previously discussed, a dictionary is used to index key:value pairs and to store these mapped values. Dictionaries are always started (created) using the `{` symbol, followed by the `name` in `quotation marks`, a `:`, and an opening `[`. They are ended using the opposite closing symbols.

Let us create a mock dataframe for five fictitious individuals representing different ages, and departments at a research facility.

[25]:
```python
df = pd.DataFrame({'Name': ['Jack', 'Kathy', 'Latesha',
                    'Brandon', 'Alexa',
                    'Jonathan', 'Joshua', 'Emily',
                    'Matthew', 'Anthony', 'Margaret',
                    'Natalie'],

                   'Age':[47, 41, 23, 55, 36, 54, 48,
                          23, 22, 27, 37, 43],

                   'Experience':[7,5,9,3,11,6,8,9,5,2,1,4],
                   'Position': ['Economist',
                    'Director of Operations',
                    'Human Resources', 'Admin. Assistant',
                    'Data Scientist', 'Admin. Assistant',
                    'Account Manager', 'Account Manager',
                    'Attorney', 'Paralegal','Data Analyst',
                    'Research Assistant']})
df
```

[25]:

	Name	Age	Experience	Position
0	Jack	47	7	Economist
1	Kathy	41	5	Director of Operations
2	Latesha	23	9	Human Resources
3	Brandon	55	3	Admin. Assistant
4	Alexa	36	11	Data Scientist
5	Jonathan	54	6	Admin. Assistant
6	Joshua	48	8	Account Manager
7	Emily	23	9	Account Manager
8	Matthew	22	5	Attorney
9	Anthony	27	2	Paralegal
10	Margaret	37	1	Data Analyst
11	Natalie	43	4	Research Assistant

Examining the Structure of a Dataframe

Let us examine the structure of the dataframe. Once again, recall that whereas in R we would use `str()` to look at the structure of a dataframe, in Python, `str()` refers to string. Thus, we will use the `df.dtypes`, `df.info()`, `len(df)`, and `df.shape` operations/functions, respectively to examine the dataframe's structure.

[26]:
```python
print(df.dtypes, '\n') # data types
print(df.info(), '\n') # more info on dataframe

# print length of df (rows)
print('Length of Dataframe:', len(df),
      '\n')

# number of rows of dataframe
print('Number of Rows:', df.shape[0])

# number of columns of dataframe
print('Number of Columns:', df.shape[1])
```

```
Name          object
Age            int64
Experience     int64
Position      object
dtype: object

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12 entries, 0 to 11
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Name        12 non-null     object
 1   Age         12 non-null     int64
 2   Experience  12 non-null     int64
 3   Position    12 non-null     object
dtypes: int64(2), object(2)
memory usage: 512.0+ bytes
None

Length of Dataframe: 12

Number of Rows: 12
Number of Columns: 4
```

Sorting Data

Let us say that now we want to sort this dataframe in order of age (youngest to oldest).

[27]:
```python
# pandas sorts values in ascending order
# by default, so there is no need to parse
# in ascending=True as a parameter

df_age = df.sort_values(by=['Age'])
df_age
```

[27]:

	Name	Age	Experience	Position
8	Matthew	22	5	Attorney
2	Latesha	23	9	Human Resources
7	Emily	23	9	Account Manager
9	Anthony	27	2	Paralegal
4	Alexa	36	11	Data Scientist
10	Margaret	37	1	Data Analyst
1	Kathy	41	5	Director of Operations
11	Natalie	43	4	Research Assistant
0	Jack	47	7	Economist
6	Joshua	48	8	Account Manager
5	Jonathan	54	6	Admin. Assistant
3	Brandon	55	3	Admin. Assistant

Now, if we want to also sort by experience while keeping age sorted according to previous specifications, we can do the following:

[28]:
```python
df_age_exp = df.sort_values(by = ['Age', 'Experience'])
df_age_exp
```

[28]:

	Name	Age	Experience	Position
8	Matthew	22	5	Attorney
2	Latesha	23	9	Human Resources
7	Emily	23	9	Account Manager
9	Anthony	27	2	Paralegal
4	Alexa	36	11	Data Scientist
10	Margaret	37	1	Data Analyst
1	Kathy	41	5	Director of Operations
11	Natalie	43	4	Research Assistant
0	Jack	47	7	Economist
6	Joshua	48	8	Account Manager
5	Jonathan	54	6	Admin. Assistant
3	Brandon	55	3	Admin. Assistant

Handling $\#NA$ values

$\#NA$ (not available) refers to missing values. What if our dataset has missing values? How should we handle this scenario? For example, age has some missing values.

However, in our particular case, we have introduced `NaNs`. NaN simply refers to Not a Number. Since we are looking at age as numeric values, let us observe when two of those numbers appear as missing values of the NaN form.

[29]:
```python
df_2 = pd.DataFrame({'Name': ['Jack', 'Kathy', 'Latesha',
                     'Brandon', 'Alexa',
                     'Jonathan', 'Joshua', 'Emily',
                     'Matthew', 'Anthony', 'Margaret',
                     'Natalie'],

                    'Age':[47, np.nan , 23, 55, 36, 54, 48,
                           np.nan, 22, 27, 37, 43],

                    'Experience':[7,5,9,3,11,6,8,9,5,2,1,4],
                    'Position': ['Economist',
                     'Director of Operations',
                     'Human Resources', 'Admin. Assistant',
                     'Data Scientist', 'Admin. Assistant',
                     'Account Manager', 'Account Manager',
                     'Attorney', 'Paralegal','Data Analyst',
                     'Research Assistant']})
df_2
```

[29]:

	Name	Age	Experience	Position
0	Jack	47.0	7	Economist
1	Kathy	NaN	5	Director of Operations
2	Latesha	23.0	9	Human Resources
3	Brandon	55.0	3	Admin. Assistant
4	Alexa	36.0	11	Data Scientist
5	Jonathan	54.0	6	Admin. Assistant

```
6     Joshua  48.0           8        Account Manager
7      Emily   NaN           9        Account Manager
8    Matthew  22.0           5               Attorney
9    Anthony  27.0           2              Paralegal
10  Margaret  37.0           1           Data Analyst
11   Natalie  43.0           4     Research Assistant
```

Inspecting *#NA* values

[30]:
```python
# inspect dataset for missing values
# with logical (bool) returns
print(df_2.isnull(), '\n')

# sum up all of the missing values in
# each row (if there are any)
print(df_2.isnull().sum())
```

```
     Name    Age  Experience  Position
0   False  False       False     False
1   False   True       False     False
2   False  False       False     False
3   False  False       False     False
4   False  False       False     False
5   False  False       False     False
6   False  False       False     False
7   False   True       False     False
8   False  False       False     False
9   False  False       False     False
10  False  False       False     False
11  False  False       False     False

Name          0
Age           2
Experience    0
Position      0
dtype: int64
```

We can delete the rows with missing values by making an `dropna()` function call in the following manner:

[31]:
```python
# drop missing values
df_2.dropna(subset=['Age'], inplace=True)

# inspect the dataframe; there are no
# more missing values, since we dropped them
df_2
```

[31]:
```
        Name   Age  Experience            Position
0       Jack  47.0           7           Economist
2    Latesha  23.0           9     Human Resources
3    Brandon  55.0           3    Admin. Assistant
4      Alexa  36.0          11      Data Scientist
5   Jonathan  54.0           6    Admin. Assistant
```

```
6     Joshua  48.0           8     Account Manager
8    Matthew  22.0           5            Attorney
9    Anthony  27.0           2           Paralegal
10  Margaret  37.0           1        Data Analyst
11   Natalie  43.0           4  Research Assistant
```

What if we receive a dataframe that, at a cursory glance, warehouses numerical values where we see numbers, but when running additional operations on the dataframe, we discover that we cannot conduct numerical exercises with columns that appear to have numbers. This is exactly why it is of utmost importance for us to always inspect the structure of the dataframe using the `df.dtypes` function call. Here is an example of the same dataframe with altered data types.

[32]:
```python
df_3 = pd.DataFrame({
                    'Name': ['Jack', 'Kathy', 'Latesha',
                    'Brandon', 'Alexa',
                    'Jonathan', 'Joshua', 'Emily',
                    'Matthew', 'Anthony', 'Margaret',
                    'Natalie'],

                    'Age':['47', '41', '23', '55', '36', '54',
                          '48', '23', '22', '27', '37', '43'],

                    'Experience':[7,5,9,3,11,6,8,9,5,2,1,4],
                    'Position': ['Economist',
                    'Director of Operations',
                    'Human Resources', 'Admin. Assistant',
                    'Data Scientist', 'Admin. Assistant',
                    'Account Manager', 'Account Manager',
                    'Attorney', 'Paralegal','Data Analyst',
                    'Research Assistant']
                })
df_3
```

[32]:
```
        Name Age  Experience                Position
0       Jack  47           7               Economist
1      Kathy  41           5  Director of Operations
2    Latesha  23           9         Human Resources
3    Brandon  55           3        Admin. Assistant
4      Alexa  36          11          Data Scientist
5   Jonathan  54           6        Admin. Assistant
6     Joshua  48           8         Account Manager
7      Emily  23           9         Account Manager
8    Matthew  22           5                Attorney
9    Anthony  27           2               Paralegal
10  Margaret  37           1            Data Analyst
11   Natalie  43           4      Research Assistant
```

At a cursory glance, the data frame looks identical to the `df` we had originally. However, inspecting the data types yields unexpected information, that age is not an integer:

[33]:
```python
# age is now an object
df_3.dtypes
```

[33]:
```
Name          object
Age           object
Experience     int64
Position      object
dtype: object
```

Let us convert age back to an integer and re-inspect the dataframe. Notice how converting entire columns of dataframes from an objects to numeric data is more than just calling the `int()` function. We re-assign the variable with the specified column back to itself using the `pd.to_numeric()` function as follows:

[34]:
```
df_3['Age'] = pd.to_numeric(df_3['Age'])
df_3.dtypes
```

[34]:
```
Name          object
Age            int64
Experience     int64
Position      object
dtype: object
```

However, to cast a variable in a dataframe into an object (i.e., string), we can simply apply the `str()` function call before the dataframe name and specified column in the following manner:

[35]:
```
df_3['Experience'] = str(df_3['Age'])
df_3.dtypes
```

[35]:
```
Name          object
Age            int64
Experience    object
Position      object
dtype: object
```

Import data from flat .csv file

Assuming that your file is located in the same working directory that you have specified at the onset of this tutorial/workshop, make an assignment to a new variable (i.e., `ex_csv`) and call `pd.read_csv()` in the following generalized format:

Notice that the `pd` in front of `read_csv()` belongs to the pandas library which we imported as `pd` earlier.

```
ex_csv <- pd.read.csv(filename)
```

Specifying a Random State/Seed

Whereas in R, we use the `set.seed()` command to specify an arbitrary number for reproducibility of results, in `Python` we use the `random_state()`function. It is always best practice to use the same assigned random state throughout the entire experiment. Setting the random state to this arbitrary number (of any length) will guarantee exactly the same output across all Python notebooks, sessions and users, respectively.

When working with simulated numpy arrays, it is best practice to set a seed using the `np.random.seed()` function.

Basic Statistics

Let us create a new data frame of numbers 1 - 100 and go over the basic statistical functions.

[36]:
```python
mystats = pd.DataFrame(list(range(1,101)))
mystats
```

[36]:
```
      0
0     1
1     2
2     3
3     4
4     5
..  …
95   96
96   97
97   98
98   99
99  100

[100 rows x 1 columns]
```

[37]:
```python
mean = mystats.mean() # mean of the vector
median = mystats.median() # median of the vector
minimum = mystats.min() # minimum of the vector
maximum = mystats.max() # maximum of the vector
range_mystats = (mystats.min(),mystats.max())
sum_mystats = mystats.sum() # sum of the vector
stdev = mystats.std() # standard deviation of the vector
summary = mystats.describe() # summary of the dataset

# we put a '0' in brackets after each of the following
# variables so that we can access only the respective
# statistics of each function which are contained in
# the first element
print('Mean:', mean[0])
print('Median:', median[0])
print('Minimum:', minimum[0])
print('Maximum:', maximum[0])
print('Sum of values:', sum_mystats[0])
print('Standard Deviation:', stdev[0])
```

```
Mean: 50.5
Median: 50.5
Minimum: 1
Maximum: 100
Sum of values: 5050
Standard Deviation: 29.01149197588202
```

Now, we can simply this endeavor by using the `df.describe()` function which will output the summary statistics:

[38]:
```python
mystats.describe()
```

[38]:
```
                0
count  100.000000
```

```
mean     50.500000
std      29.011492
min       1.000000
25%      25.750000
50%      50.500000
75%      75.250000
max     100.000000
```

Transposing The Contents of a Dataframe

If we wish to transpose this dataframe, we can place a `.T` behind `.describe()` like so:

[39]:
```python
mystats.describe().T
```

[39]:
```
   count  mean        std  min    25%   50%    75%    max
0  100.0  50.5  29.011492  1.0  25.75  50.5  75.25  100.0
```

Simulating a Random Normal Distribution

[40]:
```python
# set seed for reproducibility
np.random.seed(222)

mu, sigma = 50, 10 # mean and standard deviation

# assign variable to a dataframe
norm_vals = pd.DataFrame(np.random.normal(mu, sigma, 100))

x = list(range(1, 101))
y = list(norm_vals[0])

norm_vals = pd.DataFrame(y,x)
norm_vals.reset_index(inplace=True)
norm_vals.rename(columns = {0:'Number',
                              'index': 'Index'},
                               inplace = True) # mod. w/out copy
norm_vals
```

[40]:
```
    Index     Number
0       1  69.634250
1       2  52.757697
2       3  54.586582
3       4  60.012647
4       5  42.361647
..    …          …
95     96  43.675536
96     97  51.745236
97     98  47.626588
98     99  55.307725
99    100  38.732611

[100 rows x 2 columns]
```

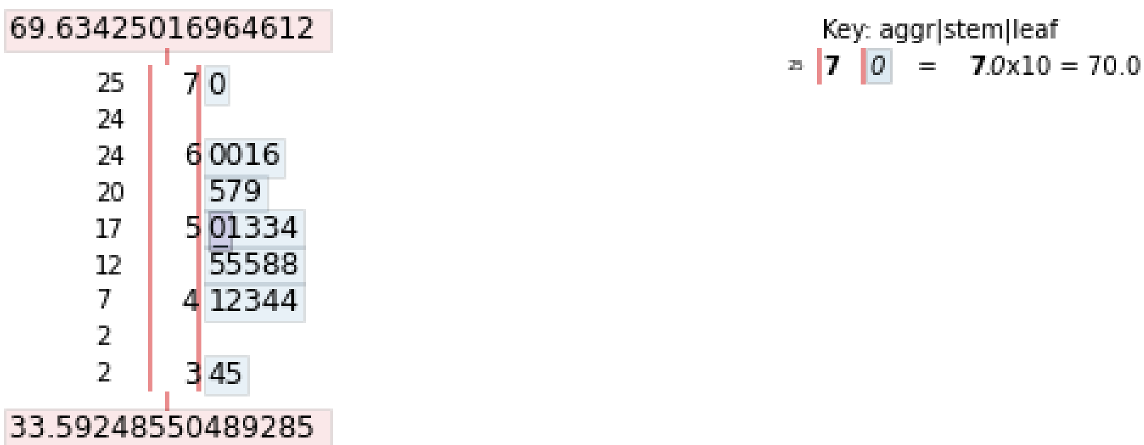
Creating Basic Plots

Unlike `R` where we can simply call the `stem()` function to create a stem-and-leaf plot, in `Python` we must first install and import the `stemgraphic` library.

[41]:
```
# pip install stemgraphic
```

We can limit the output of how many rows get printed in the resulting output by parsing in the `.loc()` function. So, if we want to print only the first 25 rows, we will access our dataframe, `norm_vals['Number']` followed by `.loc[0:25]`.

[42]:
```
import stemgraphic

# create stem-and-leaf plot
fig, ax = stemgraphic.stem_graphic(norm_vals['Number'].iloc[0:25])
```



Matplotlib Library

We must first import the `matplotlib` library, a most frequently used graphical library for making basic plots in Python. Let us import this library and plot the histogram.

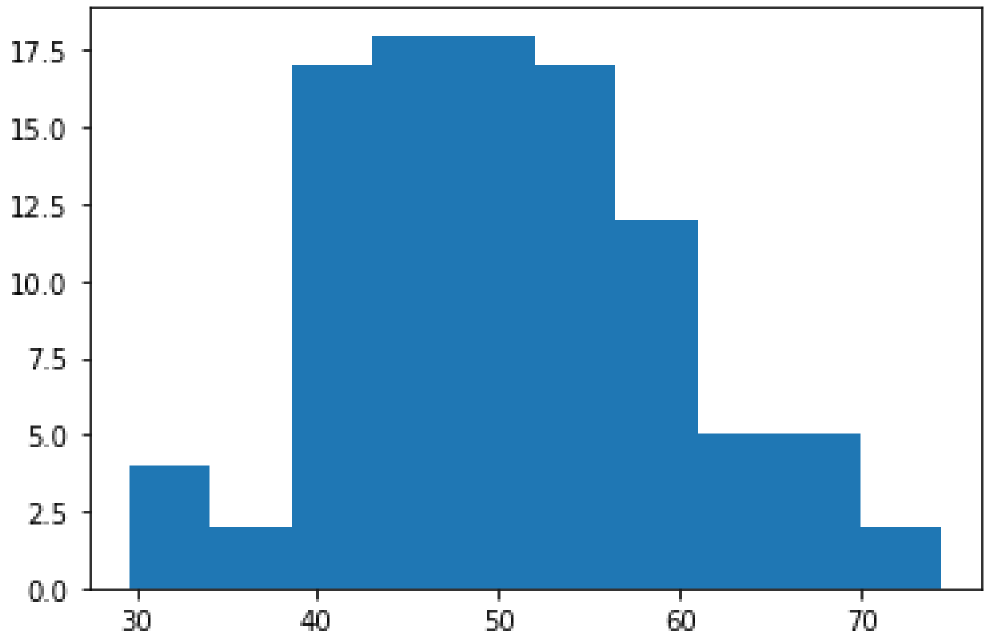
[43]:
```
import matplotlib.pyplot as plt
```

Histograms

We can proceed to plot a histogram of these `norm_vals` in order to inspect their distribution from a purely graphical standpoint.

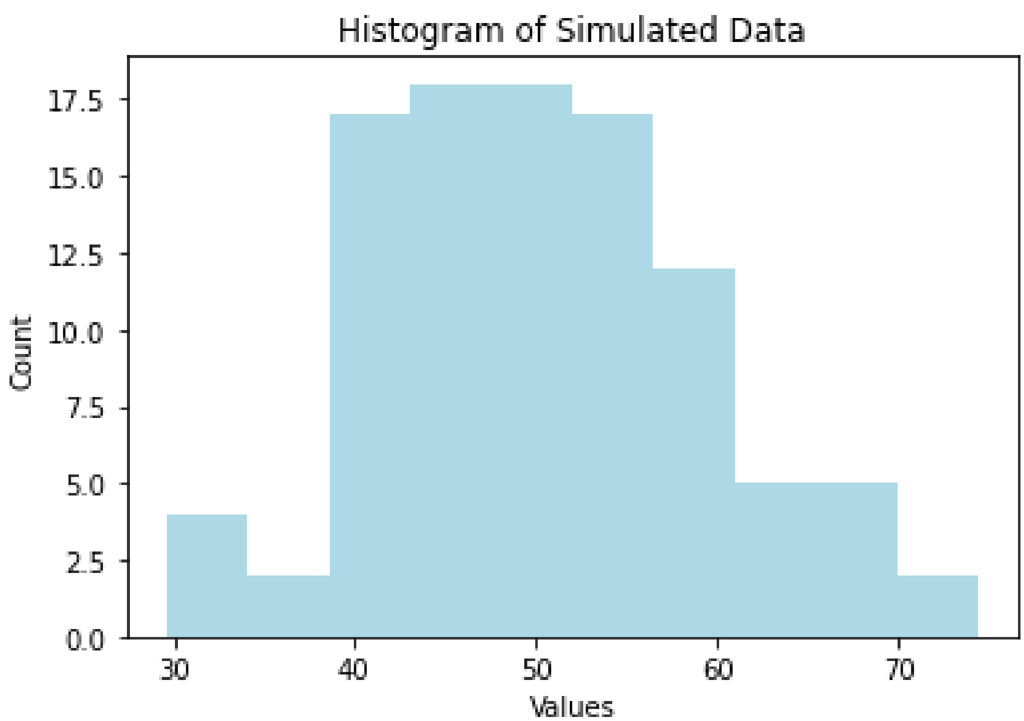
Unlike R, Python does not use a built-in `hist()` function to accomplish this task. To make the plot, we will parse in our `dataframe` follows by `.hist(grid = False)` where `grid = False` explicitly avoids plotting on a grid. Moreover, `plt.show()` expressly tells `matplotlib` to avoid extraneous output above the plot.

[44]:
```
# plot a basic histogram
norm_vals['Number'].hist(grid = False)
plt.show()
```



Our title, x-axis, and y-axis labels are given to us by default. However, let us say that we want to change all of them to our desired specifications. To this end, we can parse in and control the following parameters:

[45]:
```
norm_vals['Number'].hist(grid=False,
               color = "lightblue") # change color
plt.title ('Histogram of Simulated Data') # title
plt.xlabel('Values')                      # x-axis
plt.ylabel('Count')                       # y-axis
plt.show()
```



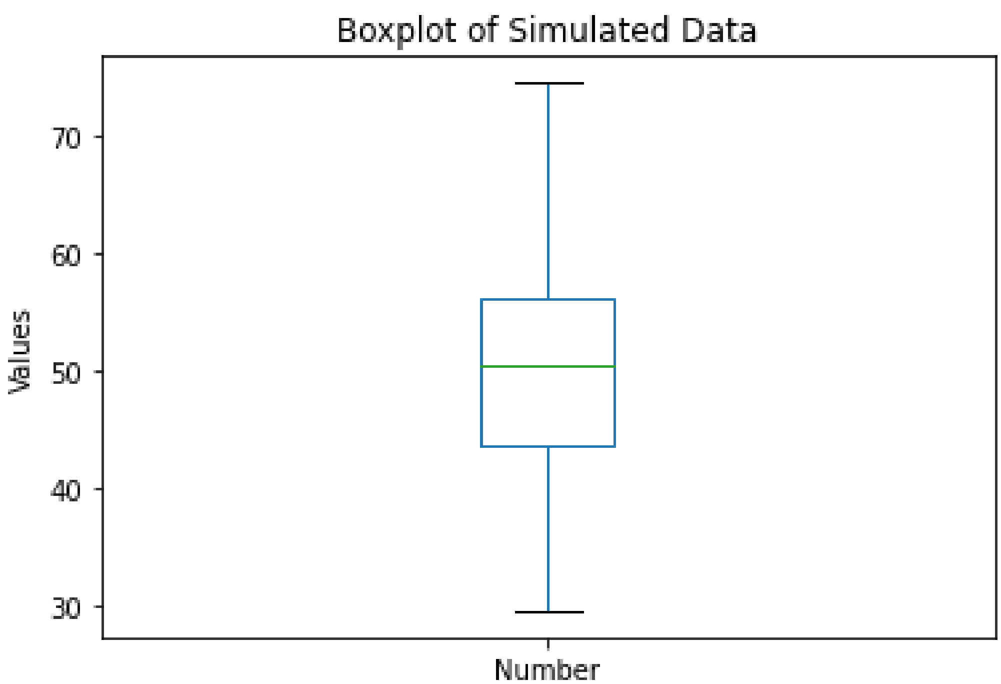
Boxplots

Similarly, we can make a boxplot using the `df.boxplot()` function call. However, the `norm_vals` dataframe has two columns. Let us only examine the randomly distributed 100 rows that we have contained in the `Number` column.

One way to do this is to create a new dataframe to only access the `Number` column:

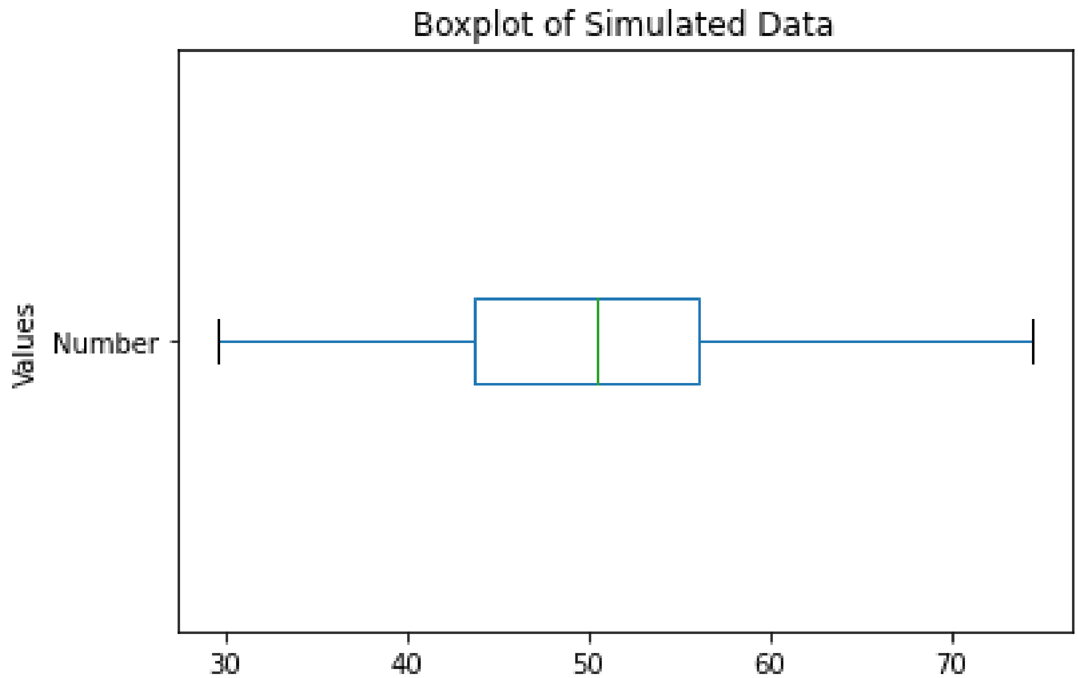
[46]:
```
norm_number = pd.DataFrame(norm_vals['Number'])
```

[47]:
```
norm_number.boxplot(grid = False)

# plot title
plt.title ('Boxplot of Simulated Data')
# x-axis label
plt.xlabel('')
# y-axis label
plt.ylabel('Values')

plt.show()
```



Now, let us pivot the boxplot by parsing in the `vert=False` parameter:

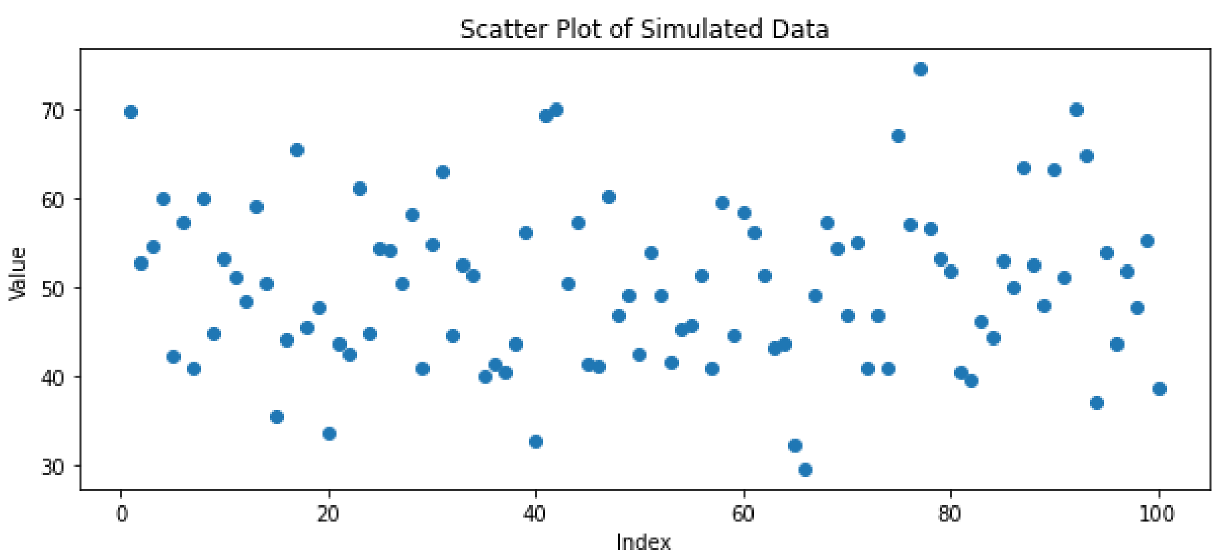
[48]:
```
norm_number.boxplot(grid = False, vert = False) # re-orient
plt.title ('Boxplot of Simulated Data') # title
plt.ylabel('Values')                    # y-axis
plt.show()
```



Scatter Plots

To make a simple scatter plot, we will call the `plot.scatter()` function on the dataframe as follows:

[49]:
```
x = norm_vals['Index'] # independent variable
y = norm_vals['Number'] # dependent variable

fig,ax = plt.subplots(figsize = (10,4))
plt.scatter(x, y) # scatter plot call
plt.title('Scatter Plot of Simulated Data')

# we can also separate lines by semicolon
plt.xlabel('Index'); plt.ylabel('Value'); plt.show()
```

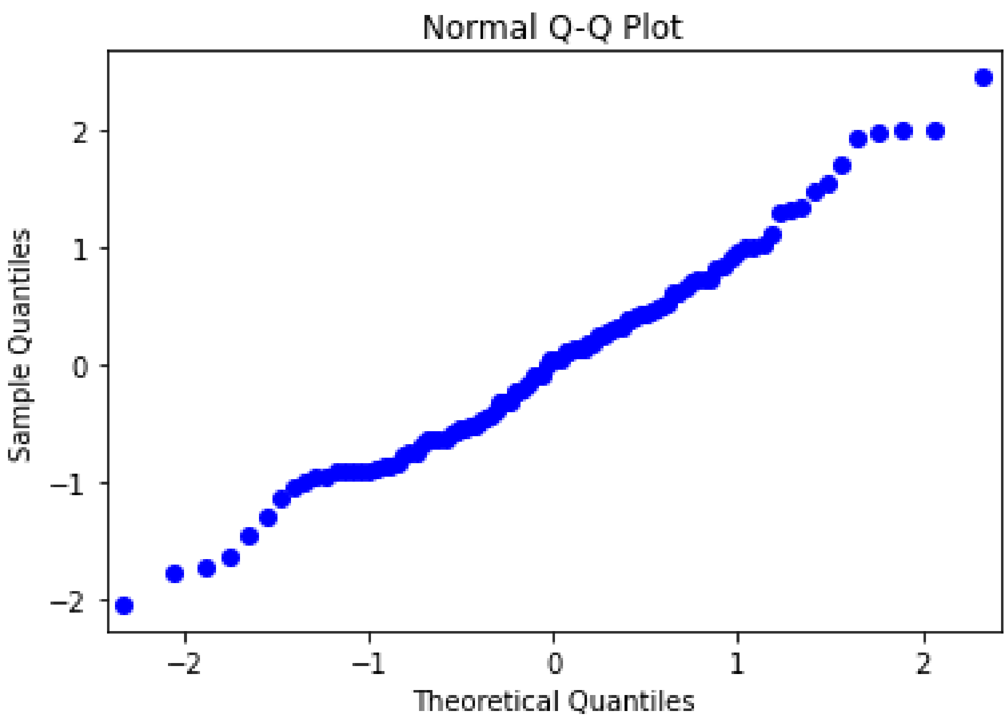


Quantile-Quantile Plot

Let us create a vector from simulated data for the next example and generate a normal quantile plot.

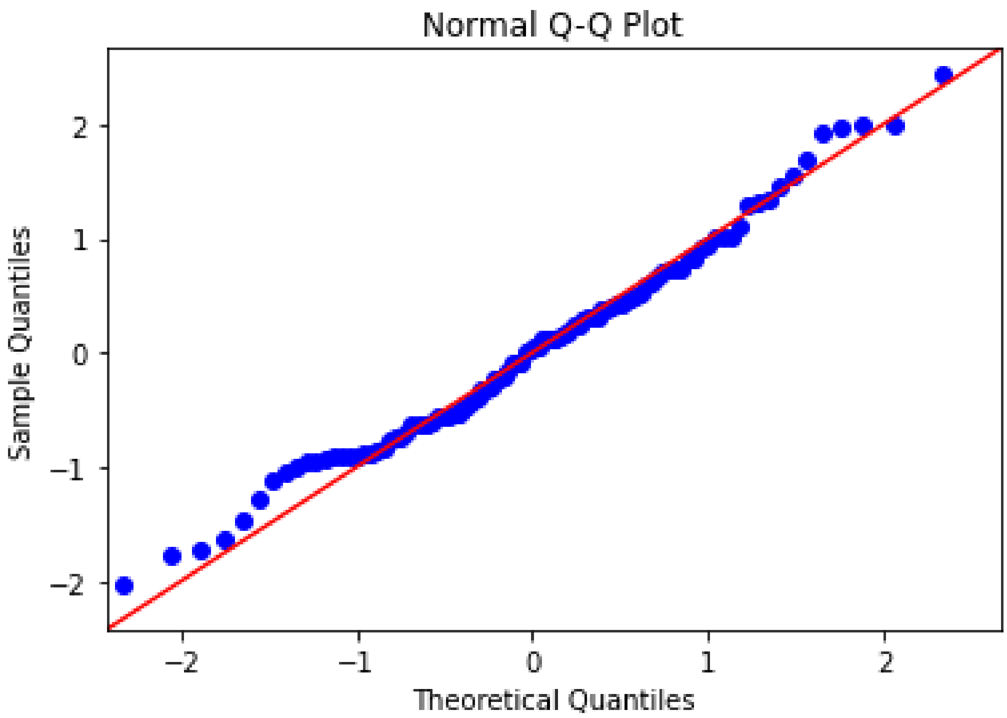
[50]:
```
# pip install statsmodels
```

[51]:
```
import statsmodels.api as sm

np.random.seed(222)
quant_ex = np.random.normal(0, 1, 100)

sm.qqplot(quant_ex)
plt.title('Normal Q-Q Plot')
plt.show()
```



Let us now add a theoretical Q-Q line at a 45 degree angle.

[52]:
```
import statsmodels.api as sm

np.random.seed(222)
quant_ex = np.random.normal(0, 1, 100)

sm.qqplot(quant_ex,
          line='45') # theoretical Q-Q line
plt.title('Normal Q-Q Plot')
plt.show()
```



Skewness and Box-Cox Transformation

From statistics, let us recall that if the mean is greater than the median, the distribution will be positively skewed. Conversely, if the median is greater than the mean, or the mean is less than the median, the distribution will be negatively skewed.

[53]:
```
mean_norm_vals = norm_vals['Number'].mean()
median_norm_vals = norm_vals['Number'].median()

print('Mean of norm_vals:', mean_norm_vals)
print('Median of norm_vals:', median_norm_vals)

print('Difference =', mean_norm_vals - median_norm_vals)
```

```
Mean of norm_vals: 50.2559224420852
Median of norm_vals: 50.401266487660806
Difference = -0.1453440455756052
```

Since both the mean and the median values are fairly close together, the data appears to be normally distributed, so we will simulate another example involving skewness.

Whereas in R, we use the all-encompassing `caret` machine learning library to handle multiple tasks, often we find ourselves loading more libraries in `Python` like the `scipy` library to handle Box-Cox transformations.

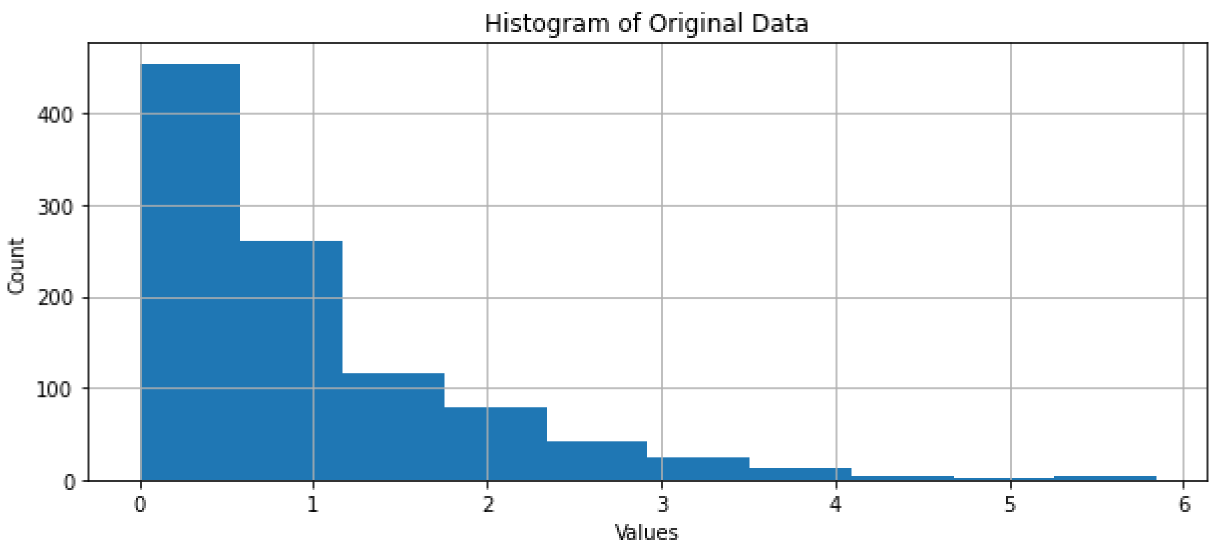
[54]:
```
# pip install scipy
```

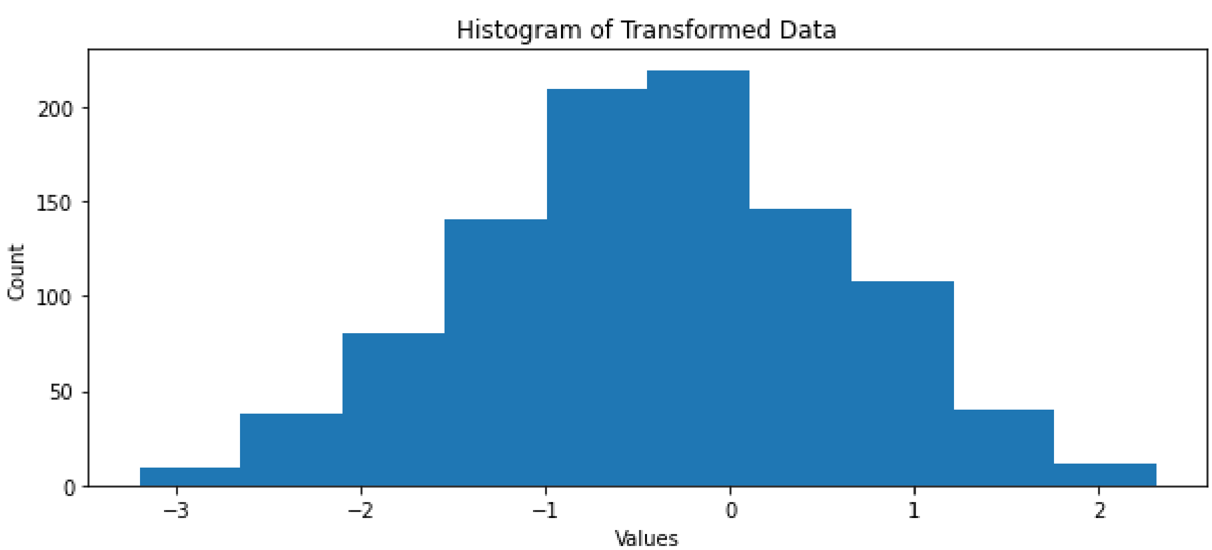
[55]:
```
from scipy import stats
original_data = np.random.exponential(size = 1000)
```

```
# transform training data & save lambda value
fitted_data, fitted_lambda = stats.boxcox(original_data)
```

[56]:
```
original_data1 = pd.DataFrame(original_data)
fitted_data1 = pd.DataFrame(fitted_data)
```

[57]:
```
# original hist(0)
original_data1.hist(figsize=(10,4))
plt.title ('Histogram of Original Data')
plt.xlabel('Values'); plt.ylabel('Count')

# transformed hist()
fitted_data1.hist(grid = False,
                  figsize=(10,4))
plt.title ('Histogram of Transformed Data')
plt.xlabel('Values'); plt.ylabel('Count')
plt.show()
```





Basic Modeling

Simple Linear Regression

Let us set up an example dataset for the following modeling endeavors.

We will be accessing Python's most commonly used machine learning library, `scikit-learn` to build the ensuing algorithms, though there are others like `pycaret` and `SciPy`, to name a few.

So, let us go ahead and import `sklearn` for linear regression into our environment.

[58]:
```
# pip install sklearn
```

Notice a more refined importing syntax, atypical of the standard `import library name`. We are telling `Python` to import the Linear Regression module from the `scikit-learn` library as follows:

[59]:
```
from sklearn.linear_model import LinearRegression
```

[60]:
```
lin_mod = pd.DataFrame({
                # X1
                'Hydrogen':[.18,.20,.21,.21,.21,.22,.23,
                            .23,.24,.24,.25,.28,.30,.37,.31,
                            .90,.81,.41,.74,.42,.37,.49,.07,
                            .94,.47,.35,.83,.61,.30,.61,.54],
                # X2
                'Oxygen':[.55,.77,.40,.45,.62,.78,.24,.47,
                          .15,.70,.99,.62,.55,.88,.49,.36,
                          .55,.42,.39,.74,.50,.17,.18,.94,
                          .97,.29,.85,.17,.33,.29,.85],
                # X3
                'Nitrogen':[.35,.48,.31,.75,.32,.56,.06,.46,
                            .79,.88,.66,.04,.44,.61,.15,.48,
                            .23,.90,.26,.41,.76,.30,.56,.73,
                            .10,.01,.05,.34,.27,.42,.83],
                # y
                'Gas Porosity':[.46,.70,.41,.45,.55,
                                .44,.24,.47,.22,.80,.88,.70,
                                .72,.75,.16,.15,.08,.47,.59,
                                .21,.37,.96,.06,.17,.10,.92,
                                .80,.06,.52,.01,.37]})
lin_mod
```

[60]:

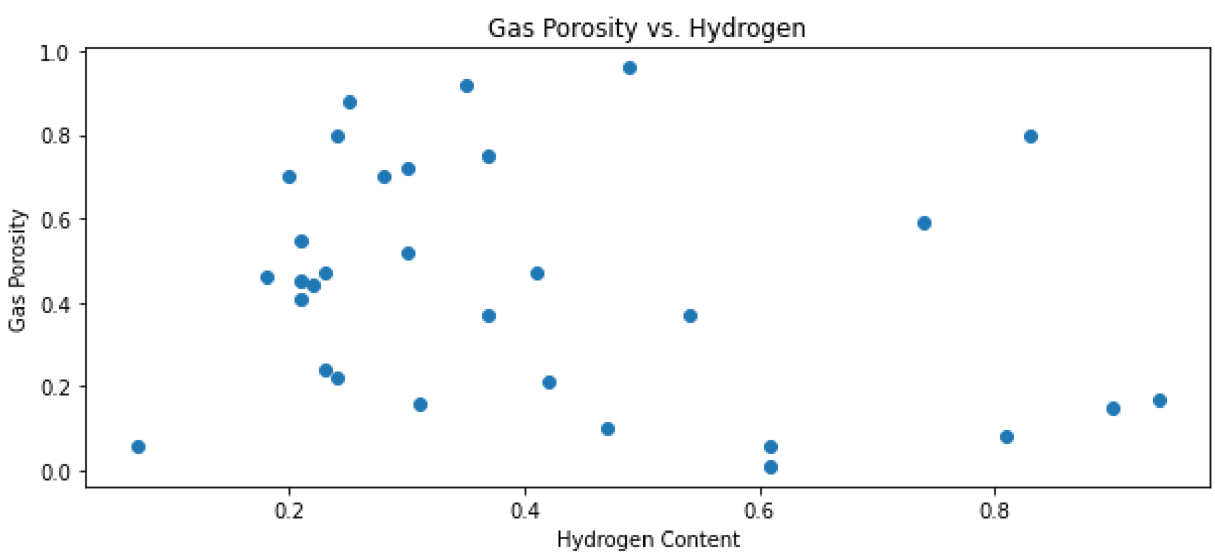
	Hydrogen	Oxygen	Nitrogen	Gas Porosity
0	0.18	0.55	0.35	0.46
1	0.20	0.77	0.48	0.70
2	0.21	0.40	0.31	0.41
3	0.21	0.45	0.75	0.45
4	0.21	0.62	0.32	0.55
5	0.22	0.78	0.56	0.44
6	0.23	0.24	0.06	0.24

7	0.23	0.47	0.46	0.47
8	0.24	0.15	0.79	0.22
9	0.24	0.70	0.88	0.80
10	0.25	0.99	0.66	0.88
11	0.28	0.62	0.04	0.70
12	0.30	0.55	0.44	0.72
13	0.37	0.88	0.61	0.75
14	0.31	0.49	0.15	0.16
15	0.90	0.36	0.48	0.15
16	0.81	0.55	0.23	0.08
17	0.41	0.42	0.90	0.47
18	0.74	0.39	0.26	0.59
19	0.42	0.74	0.41	0.21
20	0.37	0.50	0.76	0.37
21	0.49	0.17	0.30	0.96
22	0.07	0.18	0.56	0.06
23	0.94	0.94	0.73	0.17
24	0.47	0.97	0.10	0.10
25	0.35	0.29	0.01	0.92
26	0.83	0.85	0.05	0.80
27	0.61	0.17	0.34	0.06
28	0.30	0.33	0.27	0.52
29	0.61	0.29	0.42	0.01
30	0.54	0.85	0.83	0.37

[61]:
```
x1 = lin_mod['Hydrogen']; x2 = lin_mod['Oxygen']
x3 = lin_mod['Nitrogen']; y = lin_mod['Gas Porosity']
```

Prior to modeling, it is best practice to examine correlation visa vie visual scatterplot analysis as follows:

[62]:
```
fig,ax = plt.subplots(figsize = (10,4)) # resize plot
plt.scatter(x1, y)
plt.title('Gas Porosity vs. Hydrogen')
plt.xlabel('Hydrogen Content'); plt.ylabel('Gas Porosity')
plt.show()
```



Now let us calculate our correlation coefficient for the first variable relationship.

[63]:
```
corr1 = np.corrcoef(x1, y)
r1 = corr1[0,1]
r1
```

[63]:
```
-0.23843715627655337
```

By the correlation coefficient r you will see that there exists a relatively moderate (positive) relationship. Let us now build a simple linear model from this dataframe.

[64]:
```
# notice the additional brackets
# we do this to specify columns
# within our dataframe of interest
X1 = lin_mod[['Hydrogen']]
y = lin_mod[['Gas Porosity']]

# set-up the linear regression
lm_model1 = LinearRegression().fit(X1, y)
```

Next, we will rely on the stats model package to obtain a summary output table. Here, it is important to note that unlike in `R`, the `statsmodels` package in Python does not add a constant to the summary output, so for reproducible results, we will add it by the `sm.add_constant()` function.

[65]:
```
from statsmodels.api import OLS
X1 = sm.add_constant(X1)
X1_results = OLS(y,X1).fit()
X1_results.summary()
```

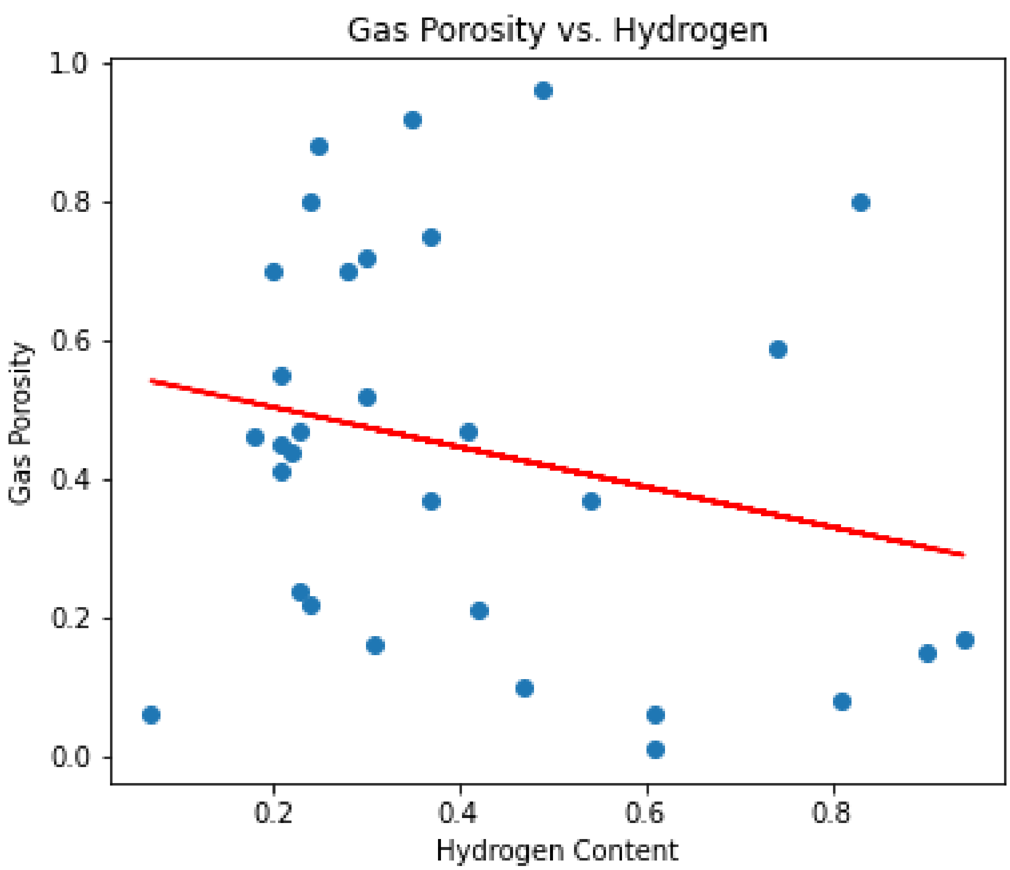
[65]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:           Gas Porosity   R-squared:                       0.057
Model:                            OLS   Adj. R-squared:                  0.024
Method:                 Least Squares   F-statistic:                     1.748
Date:                Thu, 27 Jan 2022   Prob (F-statistic):              0.196
Time:                        21:47:27   Log-Likelihood:                -3.5012
No. Observations:                  31   AIC:                             11.00
Df Residuals:                      29   BIC:                             13.87
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.5616      0.102      5.527      0.000       0.354       0.769
Hydrogen      -0.2885      0.218     -1.322      0.196      -0.735       0.158
==============================================================================
Omnibus:                        2.121   Durbin-Watson:                   2.093
Prob(Omnibus):                  0.346   Jarque-Bera (JB):                1.474
```

```
Skew:                           0.308   Prob(JB):                   0.479
Kurtosis:                       2.127   Cond. No.                    5.08
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""
```

Notice how the p-value for hydrogen content is 0.196, which lacks statistical significance when compared to the alpha value of 0.05 (at the 95% confidence level). Moreover, the R-Squared value of 0.057 suggests that roughly 6% of the variance for gas propensity is explained by hydrogen content.

We can make the same scatter plot, but this time with a best fit line.

[66]:
```
fig, ax = plt.subplots(figsize=(6,5))
plt.scatter(x1, y)
plt.title('Gas Porosity vs. Hydrogen')
plt.xlabel('Hydrogen Content')
plt.ylabel('Gas Porosity')

# create best-fit line based on slope-intercept form
m, b = np.polyfit(x1, y, 1)
plt.plot(x1, m*x1 + b,
         color = 'red')

plt.show()
```



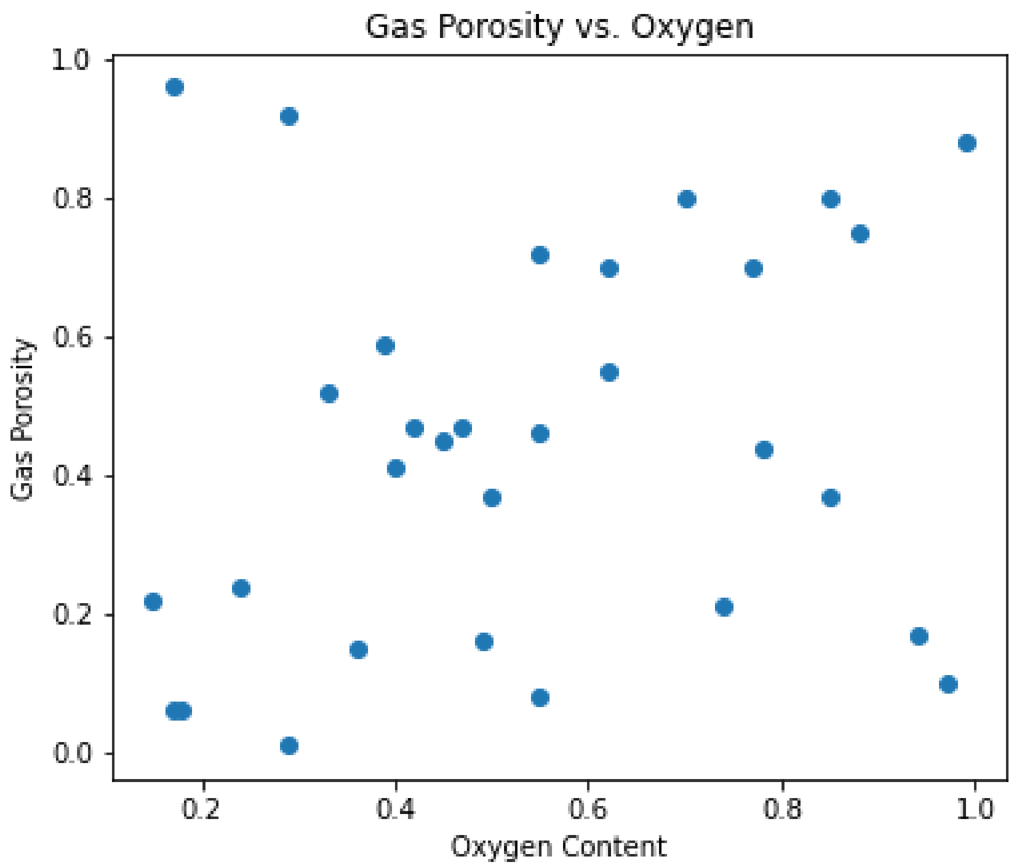
Multiple Linear Regression

To account for all independent (x) variables in the model, let us set up the model in a dataframe:

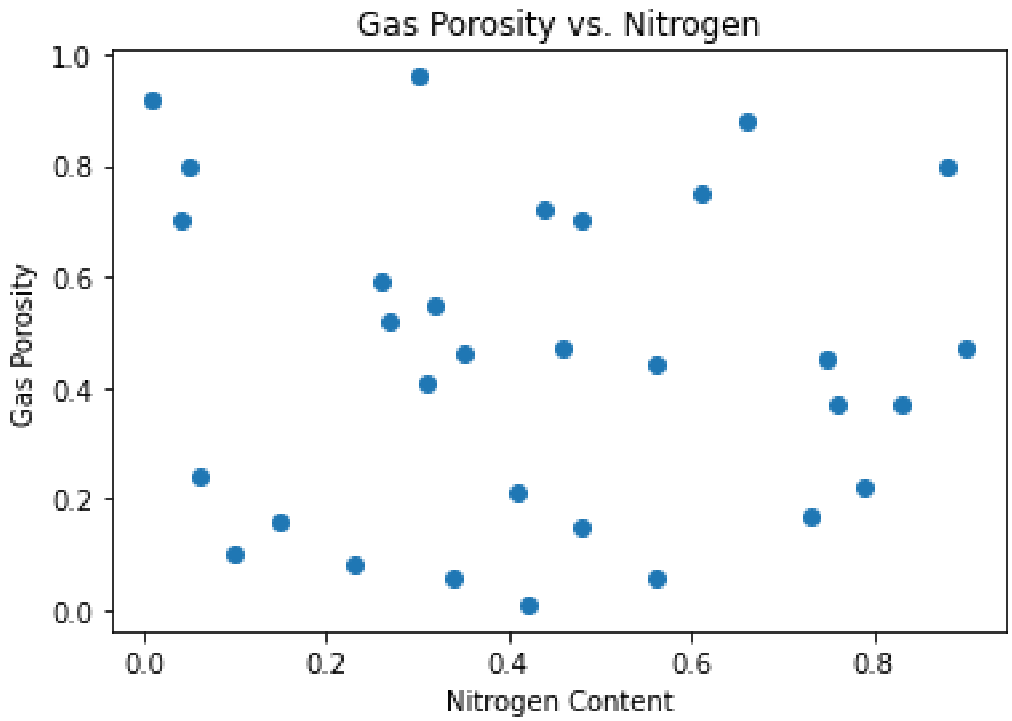
[67]:
```
X = lin_mod[['Hydrogen', 'Oxygen', 'Nitrogen']]
y = lin_mod[['Gas Porosity']]
```

Let us plot the remaining variable relationships.

[68]:
```
fig, ax = plt.subplots(figsize=(6,5))
plt.scatter(x2,y)
plt.title('Gas Porosity vs. Oxygen')
plt.xlabel('Oxygen Content')
plt.ylabel('Gas Porosity')
plt.show()
```



[69]:
```
x3_plot =plt.scatter(x3,y) # create scatter plot
plt.title('Gas Porosity vs. Nitrogen') # title
plt.xlabel('Nitrogen Content')       # x-axis label
plt.ylabel('Gas Porosity')           # y-axis label
x3_plot
plt.show()
```



[70]:
```
X = sm.add_constant(X)
lin_model_results = OLS(y,X).fit()
lin_model_results.summary()
```

[70]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:          Gas Porosity   R-squared:                       0.136
Model:                           OLS   Adj. R-squared:                  0.040
Method:                Least Squares   F-statistic:                     1.413
Date:               Thu, 27 Jan 2022   Prob (F-statistic):              0.260
Time:                       21:47:28   Log-Likelihood:                -2.1477
No. Observations:                 31   AIC:                             12.30
Df Residuals:                     27   BIC:                             18.03
Df Model:                          3
Covariance Type:           nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.4751      0.160      2.970      0.006       0.147       0.803
Hydrogen      -0.3475      0.220     -1.578      0.126      -0.799       0.104
Oxygen         0.3082      0.203      1.519      0.140      -0.108       0.724
Nitrogen      -0.1269      0.196     -0.647      0.523      -0.530       0.276
==============================================================================
Omnibus:                       0.990   Durbin-Watson:                   2.184
Prob(Omnibus):                 0.610   Jarque-Bera (JB):                0.839
Skew:                          0.380   Prob(JB):                        0.657
```

```
Kurtosis:                       2.732   Cond. No.                         6.62
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""
```

Logistic Regression

Whereas in linear regression, it is necessary to have a quantitative and continuous target variable, logistic regression is part of the generalized linear model series that has a categorical (often binary) target (outcome) variable. For example, let us say we want to predict grades for mathematics courses taught at a university. So, we have the following example dataset:

[71]:
```
math_df = pd.DataFrame(
    {'Calculus1':[56,80,10,8,20,90,38,42,57,58,90,2,
                  34,84,19,74,13,67,84,31,82,67,99,
                  76,96,59,37,24,3,57,62],
     'Calculus2':[83,98,50,16,70,31,90,48,67,78,55,
                  75,20,80,74,86,12,100,63,36,91,
                  19,69,58,85,77,5,31,57,72,89],
     'linear_alg':[87,90,85,57,30,78,75,69,83,85,90,
                   85,99,97, 38,95,10,99,62,47,17,
                   31,77,92,13,44,3,83,21,38,70],
     'pass_fail':['P','F','P','F','P','P','P','P',
                  'F','P','P','P','P','P','P','F',
                  'P','P','P','F','F','F','P','P',
                  'P','P','P','P','P','P','P']})
math_df
```

[71]:

	Calculus1	Calculus2	linear_alg	pass_fail
0	56	83	87	P
1	80	98	90	F
2	10	50	85	P
3	8	16	57	F
4	20	70	30	P
5	90	31	78	P
6	38	90	75	P
7	42	48	69	P
8	57	67	83	F
9	58	78	85	P
10	90	55	90	P
11	2	75	85	P
12	34	20	99	P
13	84	80	97	P
14	19	74	38	P
15	74	86	95	F
16	13	12	10	P
17	67	100	99	P
18	84	63	62	P

19	31	36	47	F
20	82	91	17	F
21	67	19	31	F
22	99	69	77	P
23	76	58	92	P
24	96	85	13	P
25	59	77	44	P
26	37	5	3	P
27	24	31	83	P
28	3	57	21	P
29	57	72	38	P
30	62	89	70	P

At this juncture, we cannot build a model with categorical values until and unless they are binarized using a dictionary mapping. A passing score will be designated by a 1, and failing score with a 0, respectively.

[72]:
```python
# binarize pass fail to 1 = pass, 0=fail
# into new column
math_df['math_outcome'] = math_df['pass_fail'].map({'P':1,'F':0})
math_df
```

[72]:

	Calculus1	Calculus2	linear_alg	pass_fail	math_outcome
0	56	83	87	P	1
1	80	98	90	F	0
2	10	50	85	P	1
3	8	16	57	F	0
4	20	70	30	P	1
5	90	31	78	P	1
6	38	90	75	P	1
7	42	48	69	P	1
8	57	67	83	F	0
9	58	78	85	P	1
10	90	55	90	P	1
11	2	75	85	P	1
12	34	20	99	P	1
13	84	80	97	P	1
14	19	74	38	P	1
15	74	86	95	F	0
16	13	12	10	P	1
17	67	100	99	P	1
18	84	63	62	P	1
19	31	36	47	F	0
20	82	91	17	F	0
21	67	19	31	F	0
22	99	69	77	P	1
23	76	58	92	P	1
24	96	85	13	P	1
25	59	77	44	P	1
26	37	5	3	P	1
27	24	31	83	P	1
28	3	57	21	P	1
29	57	72	38	P	1

30 62 89 70 P 1

Let us import the Linear Regression module from the scikit-learn library as follows:

[73]:
```python
from sklearn.linear_model import LogisticRegression
```

Instead of `OLS.fit()` like we did for linear regression, we will be using the `sm.Logit()` function call to pass in our y and x, respectively.

[74]:
```python
# we can also drop the columns that we
# will not be using
logit_X = math_df.drop(columns=['pass_fail',
                                'math_outcome'])
logit_X = sm.add_constant(logit_X)
logit_y = math_df['math_outcome']

# notice the sm.Logit() function call
log_results = sm.Logit(logit_y,logit_X).fit()
log_results.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.525623
         Iterations 5
```

[74]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                           Logit Regression Results
==============================================================================
Dep. Variable:           math_outcome   No. Observations:                   31
Model:                          Logit   Df Residuals:                       27
Method:                           MLE   Df Model:                            3
Date:                Thu, 27 Jan 2022   Pseudo R-squ.:                 0.01598
Time:                        21:47:28   Log-Likelihood:                -16.294
converged:                       True   LL-Null:                       -16.559
Covariance Type:            nonrobust   LLR p-value:                    0.9124
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          1.1325      1.278      0.886      0.376      -1.373       3.638
Calculus1     -0.0106      0.016     -0.647      0.518      -0.043       0.022
Calculus2      0.0061      0.018      0.347      0.728      -0.028       0.041
linear_alg     0.0049      0.015      0.328      0.743      -0.024       0.034
==============================================================================
"""
```

Decision Trees

Let us import the Decision Tree Classifier from `scikit-learn`.

[75]:
```python
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
```

We will be using the `mtcars` dataset from `R`, and will have to import from `statsmodels` into `Python` first.

[76]:
```python
mtcars = sm.datasets.get_rdataset("mtcars", "datasets", cache=True).data
mtcars = pd.DataFrame(mtcars)
mtcars
```

[76]:
```
                      mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  \
Mazda RX4            21.0    6  160.0  110  3.90  2.620  16.46   0   1     4
Mazda RX4 Wag        21.0    6  160.0  110  3.90  2.875  17.02   0   1     4
Datsun 710           22.8    4  108.0   93  3.85  2.320  18.61   1   1     4
Hornet 4 Drive       21.4    6  258.0  110  3.08  3.215  19.44   1   0     3
Hornet Sportabout    18.7    8  360.0  175  3.15  3.440  17.02   0   0     3
Valiant              18.1    6  225.0  105  2.76  3.460  20.22   1   0     3
Duster 360           14.3    8  360.0  245  3.21  3.570  15.84   0   0     3
Merc 240D            24.4    4  146.7   62  3.69  3.190  20.00   1   0     4
Merc 230             22.8    4  140.8   95  3.92  3.150  22.90   1   0     4
Merc 280             19.2    6  167.6  123  3.92  3.440  18.30   1   0     4
Merc 280C            17.8    6  167.6  123  3.92  3.440  18.90   1   0     4
Merc 450SE           16.4    8  275.8  180  3.07  4.070  17.40   0   0     3
Merc 450SL           17.3    8  275.8  180  3.07  3.730  17.60   0   0     3
Merc 450SLC          15.2    8  275.8  180  3.07  3.780  18.00   0   0     3
Cadillac Fleetwood   10.4    8  472.0  205  2.93  5.250  17.98   0   0     3
Lincoln Continental  10.4    8  460.0  215  3.00  5.424  17.82   0   0     3
Chrysler Imperial    14.7    8  440.0  230  3.23  5.345  17.42   0   0     3
Fiat 128             32.4    4   78.7   66  4.08  2.200  19.47   1   1     4
Honda Civic          30.4    4   75.7   52  4.93  1.615  18.52   1   1     4
Toyota Corolla       33.9    4   71.1   65  4.22  1.835  19.90   1   1     4
Toyota Corona        21.5    4  120.1   97  3.70  2.465  20.01   1   0     3
Dodge Challenger     15.5    8  318.0  150  2.76  3.520  16.87   0   0     3
AMC Javelin          15.2    8  304.0  150  3.15  3.435  17.30   0   0     3
Camaro Z28           13.3    8  350.0  245  3.73  3.840  15.41   0   0     3
Pontiac Firebird     19.2    8  400.0  175  3.08  3.845  17.05   0   0     3
Fiat X1-9            27.3    4   79.0   66  4.08  1.935  18.90   1   1     4
Porsche 914-2        26.0    4  120.3   91  4.43  2.140  16.70   0   1     5
Lotus Europa         30.4    4   95.1  113  3.77  1.513  16.90   1   1     5
Ford Pantera L       15.8    8  351.0  264  4.22  3.170  14.50   0   1     5
Ferrari Dino         19.7    6  145.0  175  3.62  2.770  15.50   0   1     5
Maserati Bora        15.0    8  301.0  335  3.54  3.570  14.60   0   1     5
Volvo 142E           21.4    4  121.0  109  4.11  2.780  18.60   1   1     4

                     carb
Mazda RX4               4
Mazda RX4 Wag           4
Datsun 710              1
Hornet 4 Drive          1
Hornet Sportabout       2
Valiant                 1
Duster 360              4
Merc 240D               2
Merc 230                2
Merc 280                4
Merc 280C               4
Merc 450SE              3
```

```
Merc 450SL             3
Merc 450SLC            3
Cadillac Fleetwood     4
Lincoln Continental    4
Chrysler Imperial      4
Fiat 128               1
Honda Civic            2
Toyota Corolla         1
Toyota Corona          1
Dodge Challenger       2
AMC Javelin            2
Camaro Z28             4
Pontiac Firebird       2
Fiat X1-9              1
Porsche 914-2          2
Lotus Europa           2
Ford Pantera L         4
Ferrari Dino           6
Maserati Bora          8
Volvo 142E             2
```

[77]:

```
print(mtcars.dtypes, '\n')
print('Number of Rows:',mtcars.shape[0])
print('Number of Columns:',mtcars.shape[1])
```

```
mpg     float64
cyl       int64
disp    float64
hp        int64
drat    float64
wt      float64
qsec    float64
vs        int64
am        int64
gear      int64
carb      int64
dtype: object

Number of Rows: 32
Number of Columns: 11
```

[78]:

```
# convert from float to int
# otherwise DT won't run properly
mtcars = mtcars.astype(int)
print(mtcars.dtypes, '\n')
print('Number of Rows:',mtcars.shape[0])
print('Number of Columns:',mtcars.shape[1])
```

```
mpg     int32
cyl     int32
disp    int32
hp      int32
```

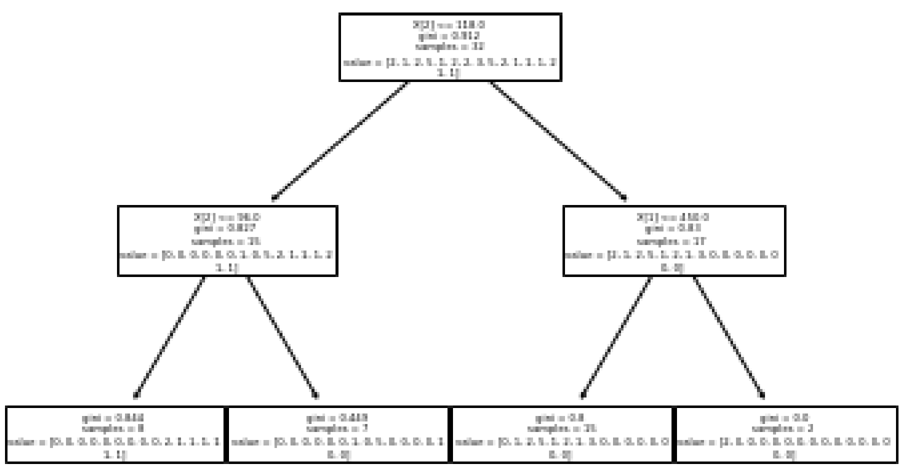
```
drat    int32
wt      int32
qsec    int32
vs      int32
am      int32
gear    int32
carb    int32
dtype: object

Number of Rows: 32
Number of Columns: 11
```

Similar to what we did for the logistic regression example, let us now create our x and y variables from this `mtcars` dataset.

[79]:
```
mtcars_X = mtcars.drop(columns=['mpg'])
mtcars_y = mtcars[['mpg']]
```

Without importing any special graphical libraries, the decision tree output plot will look condensed, small, and virtually unreadable. We can comment out the figure size dimensions, but that still will not produce anything sophisticated in nature.

[80]:
```
# fig,ax = plt.subplots(figsize = (30,30))
tree_model = DecisionTreeClassifier(max_depth=2)
tree_model = tree_model.fit(mtcars_X, mtcars_y)
tree_plot = tree.plot_tree(tree_model)
```



Another way to label our `x` and `y` variables is to assign the `x` variables to a list and use the `.remove()` function to remove our target variable from that list.

[81]:
```
X_var = list(mtcars.columns)
target = 'mpg'
X_var.remove(target)
```

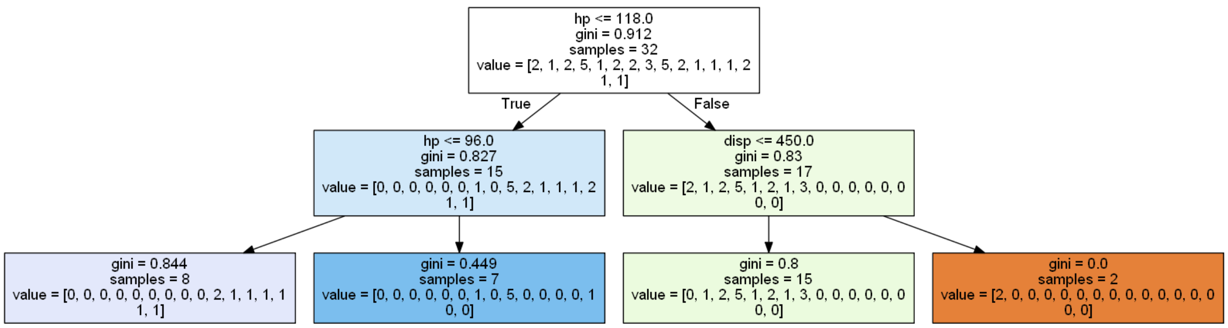
For a more sophisticated graphical output, we can tap into scikit-learn's `export_graphviz` package in conjunction with another library called `pydotplus` which we will need to install separately.

[82]:
```
# pip install pydotplus
```

[83]:
```
from sklearn.tree import export_graphviz
import pydotplus
from IPython.display import Image
```

[84]:
```
dot_data = export_graphviz(tree_model,
                           feature_names = X_var,
                           filled = True,
                           out_file = None)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```

[84]:



Basic Modeling and Cross-Validation in Python

[85]:
```
from sklearn.model_selection import train_test_split
```

[86]:
```
X_train, X_test, y_train, \
y_test = train_test_split(mtcars_X, mtcars_y,
                          test_size = 0.25,
                          random_state = 222)
```

[87]:
```
# get the shape of train and test sets
train_shape = X_train.shape[0]
test_shape = X_test.shape[0]

# calculate the proportions of each, respectively
train_percent = train_shape/(train_shape + test_shape)
test_percent = test_shape/(train_shape + test_shape)

print('Train Size:', train_percent)
print('Test Size:', test_percent)
```

```
Train Size: 0.75
Test Size: 0.25
```

Let us bring in a generalized linear model for this illustration.

[88]:
```
mtcars_X = mtcars.drop(columns=['mpg'])
mtcars_y = mtcars[['mpg']]

# notice the sm.Logit() function call
mtcars_model = sm.add_constant(X_train)

# back to the linear model since target
# variable is quantitative and continuous
mtcars_model_results = OLS(y_train, X_train).fit()
mtcars_model_results.summary()
```

[88]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                                 OLS Regression Results
=======================================================================================
Dep. Variable:                    mpg   R-squared (uncentered):                   0.995
Model:                            OLS   Adj. R-squared (uncentered):              0.991
Method:                 Least Squares   F-statistic:                              268.4
Date:                Thu, 27 Jan 2022   Prob (F-statistic):                    3.28e-14
Time:                        21:47:29   Log-Likelihood:                         -42.923
No. Observations:                  24   AIC:                                      105.8
Df Residuals:                      14   BIC:                                      117.6
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
cyl            0.4404      0.632      0.697      0.497      -0.915       1.796
disp           0.0128      0.012      1.055      0.309      -0.013       0.039
hp            -0.0260      0.018     -1.451      0.169      -0.064       0.012
drat           3.0707      0.967      3.177      0.007       0.998       5.144
wt            -2.2023      1.029     -2.140      0.050      -4.410       0.005
qsec           0.2337      0.329      0.710      0.489      -0.472       0.940
vs             2.8995      1.755      1.652      0.121      -0.864       6.663
am             1.0988      1.853      0.593      0.563      -2.875       5.073
gear           2.9577      1.068      2.770      0.015       0.668       5.248
carb          -0.6670      0.623     -1.071      0.302      -2.003       0.669
==============================================================================
Omnibus:                        0.090   Durbin-Watson:                   2.175
Prob(Omnibus):                  0.956   Jarque-Bera (JB):                0.288
Skew:                          -0.099   Prob(JB):                        0.866
Kurtosis:                       2.501   Cond. No.                     1.72e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.72e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

Before we can cross-validate, let us run our predictions of miles per gallon (mpg) on our holdout (test-set).

[89]:
```
mtcars_mod = LinearRegression()
mtcars_mod.fit(X_train, y_train)
y_pred = mtcars_mod.predict(X_test)
y_pred
```

[89]:
```
array([[28.6272386 ],
       [20.42470718],
       [27.03178327],
       [17.84957047],
       [23.85546714],
       [20.41546445],
       [30.53854242],
       [20.01004787]])
```

While our predictions remain nested in an array, we will bring in a baseline measure from the scikit-learn library as follows:

[90]:
```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_pred)
```

[90]:
```
3.3710342937771465
```

[91]:
```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from numpy import mean
from numpy import absolute
```

[92]:
```
# define cross-validation method to use
cv = KFold(n_splits=5, random_state=222, shuffle=True)

# use k-fold CV to evaluate model
scores = cross_val_score(mtcars_mod, X_train, y_train,
                         scoring='neg_mean_absolute_error',
                         cv=cv, n_jobs=-1)

# view mean absolute error
ma_scores = mean(absolute(scores))
ma_scores
```

[92]:
```
2.175723736388353
```

K-Means Clustering

A cluster is a collection of observations. We want to group these observations based on the most similar attributes. We use distance measures to measure similarity between clusters. This is one of the most widely-used unsupervised learning techniques that groups ``similar data points together and discover underlying patterns. To achieve this objective, K-means looks for a fixed number (k) of clusters in a dataset'' (Garbade, 2018).

The k in k-means is the fixed number of centroids (center of cluster) for which the algorithm will take the mean for based on the number of clusters (collection of data points).

[93]:
```
# import the necessary library
from sklearn.cluster import KMeans
```

Let us split the mtcars dataset into 3 clusters.

[94]:
```
kmeans = KMeans(n_clusters=3).fit(mtcars)
centroids = kmeans.cluster_centers_
print(centroids)
```

```
[[2.40625000e+01 4.62500000e+00 1.22000000e+02 9.68750000e+01
  3.37500000e+00 2.00000000e+00 1.80000000e+01 7.50000000e-01
  6.87500000e-01 4.12500000e+00 2.43750000e+00]
 [1.42222222e+01 8.00000000e+00 3.88222222e+02 2.32111111e+02
  3.00000000e+00 3.66666667e+00 1.58888889e+01 0.00000000e+00
  2.22222222e-01 3.44444444e+00 4.00000000e+00]
 [1.67142857e+01 7.42857143e+00 2.75714286e+02 1.50714286e+02
  2.71428571e+00 3.14285714e+00 1.77142857e+01 2.85714286e-01
  0.00000000e+00 3.00000000e+00 2.14285714e+00]]
```

[95]:
```
kmeans1 = KMeans(n_clusters=3,
                 random_state=222).fit(mtcars)
centroids1 = pd.DataFrame(kmeans1.cluster_centers_,
columns = mtcars.columns)
pd.set_option('precision', 3)
centroids1
```

[95]:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	\
0	14.222	8.000	388.222	232.111	3.000	3.667	15.889	0.000	0.222	3.444	
1	24.062	4.625	122.000	96.875	3.375	2.000	18.000	0.750	0.688	4.125	
2	16.714	7.429	275.714	150.714	2.714	3.143	17.714	0.286	0.000	3.000	

	carb
0	4.000
1	2.438
2	2.143

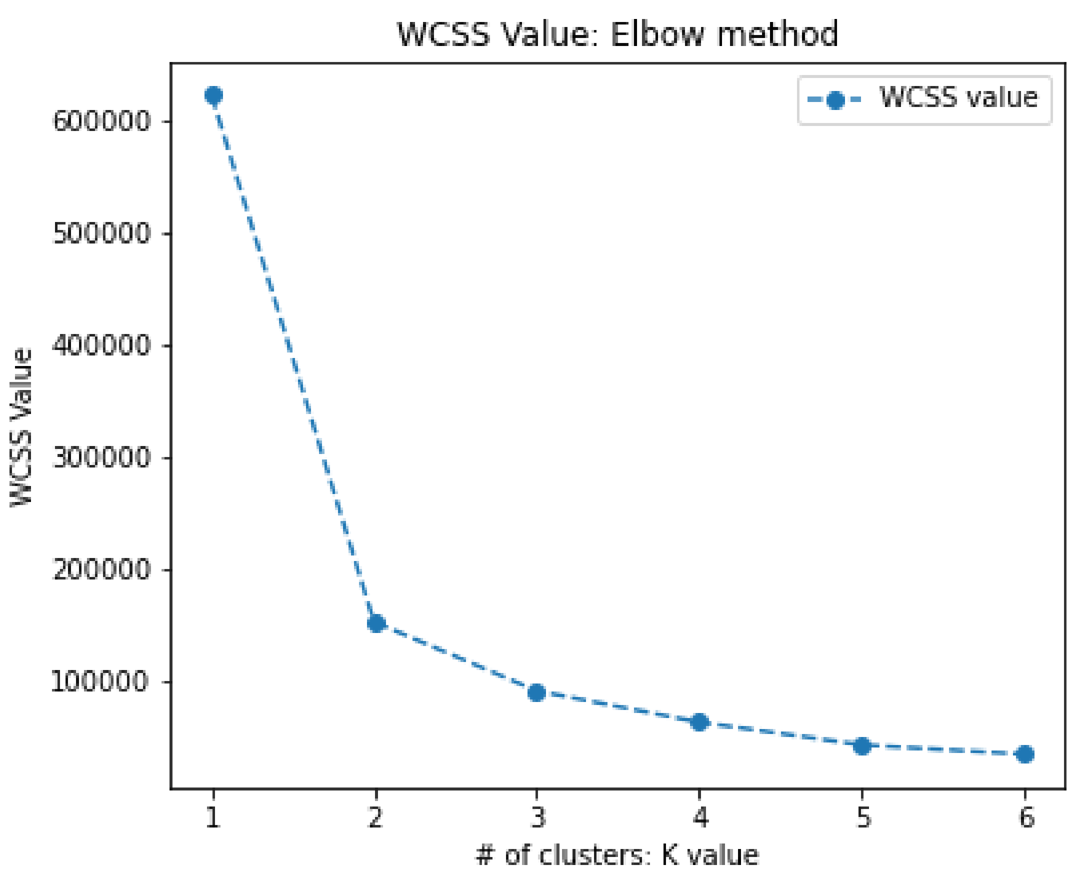
[96]:
```
withinClusterSS = [0] * 3
clusterCount = [0] * 3
for cluster, distance in zip(kmeans1.labels_,
                              kmeans1.transform(mtcars)):
    withinClusterSS[cluster] += distance[cluster]**2
    clusterCount[cluster] += 1
for cluster, withClustSS in enumerate(withinClusterSS):
    print('Cluster {} ({} members): {:5.2f} within cluster'.format(cluster,
clusterCount[cluster], withinClusterSS[cluster]))
```

```
Cluster 0 (9 members): 46660.67 within cluster
Cluster 1 (16 members): 32812.31 within cluster
Cluster 2 (7 members): 11852.00 within cluster
```

But what is the appropriate number of clusters that we should generate? Can we do better with more clusters?

[97]:
```
# let's create segments using K-means clustering
# using elbow method to find no of clusters
wcss=[]
for i in range(1,7):
    kmeans= KMeans(n_clusters = i,
                   init = 'k-means++',
                   random_state = 222)
    kmeans.fit(mtcars_X)
    wcss.append(kmeans.inertia_)
print(wcss)

fig, ax = plt.subplots(figsize=(6,5))
plt.plot(range(1,7),
         wcss, linestyle='--',
         marker='o',
         label='WCSS value')
plt.title('WCSS Value: Elbow method')
plt.xlabel('# of clusters: K value')
plt.ylabel('WCSS Value')
plt.legend()
plt.show()
```

```
[623046.28125, 152539.1984126984, 90847.05753968253, 62822.046428571426,
42672.45238095238, 34301.433333333334]
```



Hierarchical Clustering

This is another form of unsupervised learning type of cluster analysis, which takes on a more visual method, working particularly well with smaller samples (i.e., n < 500), such as this mtcars dataset. We start out with as many clusters as observations, and we go through a procedure of combining observations into clusters, culminating with combining clusters together as a reduction method for the total number of clusters that are present.

Moreover, the premise for combining clusters together is a direct result of:

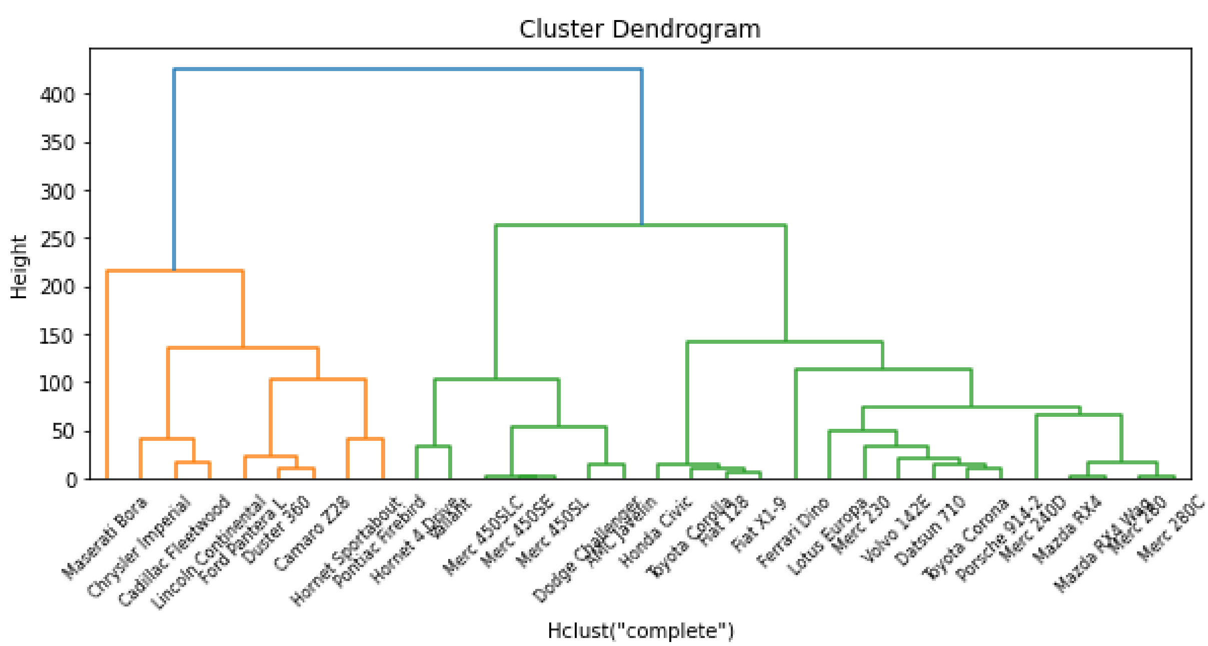
complete linkage - largest Euclidean distance between clusters.
single linkage - conversely, we look at the observations which are closest together (proximity).
centroid linkage - we can the distance between the centroid of each cluster.
group average (mean) linkage - taking the mean between the pairwise distances of the observations.

Complete linkage is the most traditional approach, so we parse in the method='complete' hyperparameter.

The tree structure that examines this hierarchical structure is called a dendogram.

[98]:
```python
import scipy.cluster.hierarchy as shc
```

[99]:
```python
plt.figure(figsize=(10, 4))
dend = shc.dendrogram(shc.linkage(mtcars, method='complete'),
                                    labels=list(mtcars.index))
plt.title("Cluster Dendrogram"); plt.xlabel('Hclust("complete")')
plt.ylabel('Height');plt.show()
```



Sources

finnstats. (2021, October 31). What Does Cross Validation Mean? R-bloggers. https://www.r-bloggers.com/2021/10/cross-validation-in-r-with-example/

Garbade, Michael. (2018, September 12). Understanding K-means Clustering in Machine Learning. Towards Data Science. https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1

GeeksforGeeks. (2020, April 22). Scope of Variable in R. GeeksforGeeks. https://www.geeksforgeeks.org/scope-of-variable-in-r/

Shmueli, G., Bruce, P. C., Gedeck, P., & Patel, N. R. (2020). *Data mining for business analytics: Concepts, techniques and applications in Python.* Wiley.