Iterating through list of numbers for URL parameters

```
import requests
```

```
url = 'http://domain.com/product/'
# Generate list of numbers from 1 to 100
Product_numbers = list(range(1, 100))
# Loop to make the request with each number
for i in Product_numbers:
    r = requests.get(url+str(i))
    # Print response in text
    prinnt (r.text)
```

Pass By Reference

When two variables point to the same section of memory hence any modification on one is passed to the other one.



Checking the type of experssion

int for integers, str for strings and float for floats

type(12) type("Hello, Python 101!")

Typecasting: converting from type to another

type (2) # checking object type

float (2) # Converting the integer to string

type(float(2)) # checking the object type of yielded output of float(2)

int ('1') # Converting from strings to an integer

str (1) # Converting integer to string

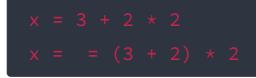
int(True) # Converting from boolean true into integer. Result is 1

bool(1) # Converting from integer to boolean. Result is 1

/ : one slash stands for float division

//: double slash stands for integer division

Assigning variables to mathematical operations



String indexes

Because indexing starts at 0, it means the first index is on the index 0

name = "Michael Jackson" # assigning string to a variable print(name[0]) # printing the first letter, that is'M'.

print(name[7]) # prints the space between the name and surname.

Negative index can help us to count the element from the end of the string

print(name[-1]) # prints the last letter starting from the end of the string. 'n'

print(name[-15]) # prints the first letter starting from the end of the string. 'n'

String Slicing: Obtaining multiple characters from a string

print (name[0:3]) # prints M,i,c,h.

print (name[8:12]) # prints J,a,c,k

String Stride: used to define a set step to jump between the string letters

print (name[::2]) # prints M,c,a,l,J,c,s,n
print (name[0:4:2]) # prints M,c

String concatenation

statement = name + ' hello' # prints Michael Jackson hello

String replication

statement = name * 3 # prints Michael Jackson three
times

times

Escape sequences.

print(" Michael Jackson \n is the best") # prints 'is
the best' to a new line
print(" Michael Jackson \t is the best") # prints tab
between Michael Jackson and is the best
print(" Michael Jackson \\ is the best") # prints
backslash between them
print(r" Michael Jackson \ is the best") # prints

String operations

| a = "Thriller is the sixth studio album" | | | | | |
|---|--|--|--|--|--|
| <pre>b = a.upper() # will convert the string a into upper</pre> | | | | | |
| case letters | | | | | |
| <pre>b = a.replace('Michael', 'Janet') # replaces michael</pre> | | | | | |
| with janet. | | | | | |
| <pre>name.find('el') # finds 'el' within the string and</pre> | | | | | |
| returns the first index of it. In that case, its '5' | | | | | |

Lists

L = ["Michael Jackson", 10.1,1982,"MJ",1] L[3:5] # slicing same as slicing a string L.extend(['pop', 10]) # extending a list by adding two elements; pop and 10 L = ["Michael Jackson", 10.1,1982,"MJ",1,"pop",10] L.append(['a','b']) # appending a list inside the first list to become nested list L = ["Michael Jackson", 10.1,1982,"MJ",1,"pop",10,['a','b']] L[0] = 'hard rock' # changing specific element L = ["hard rock", 10.1,1982,"MJ",1,"pop",10,['a','b']] del(L[0]) # Deleting a specific element L = [10.1,1982,"MJ",1,"pop",10,['a','b']] H= 'hard rock'.split() # Converting a string into a list H= ['hard','rock']

List cloning

B = H # B and H are referencing the same list in memory. Any change on H will reflect on B and not vice versa.
B = H[:] # B now is referencing H so any change on H will not reflect on B.

defining a tuple

tuple1 = ("disco",10,1.2]

accessing tuples elements, printing, negative indexing, slicing and concatenation works same as lists

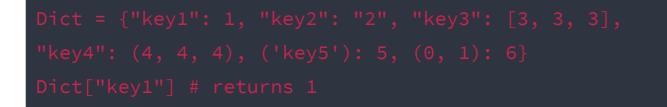
len(tuple1) #returns the number of elements
sorted(tuple1) #returns sorted elements in the tuple
NestedT =(1, 2, ("pop", "rock") ,(3,4),("disco",(1,2)))
Nested tuple
NestedT[2][0] #pop
NestedT[2][1] #rock
NestedT[2][1] #rock
NestedT[3][0] #3
NestedT[3][1] #4
NestedT[4][0] #disco
NestedT[4][1] #(1,2)
NestedT[4][1][0] #1
NestedT[4][1][1] #2

Dictionaries

A dictionary consists of keys and values. It is helpful to compare a dictionary to a list.

Instead of the numerical indexes such as a list, dictionaries have keys.

These keys are the keys that are used to access values within a dictionary



Each key is separated from its value by a colon ":". Commas separate the items, and the whole dictionary is enclosed in curly braces.

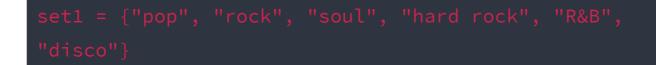
An empty dictionary without any items is written with just two curly braces, like this "{}".

```
Dict.keys() # Get all the keys in dictionary
Dict.values() # Get all the values in dictionary
Dict['key1'] = '2007' # Append value with key into
dictionary
del(Dict['key1']) # Delete entries by key
```

sets

A set is a unique collection of objects in Python. You can denote a set with a curly bracket {}.

Python will automatically remove duplicate items



Conditions and Branching

For Loops

for i,x in enumerate(\['A','B','C'\]):
 print(i+1,x)

Result:

1 A

2 B

3 C

Functions

- Functions blocks begin def followed by the function name and parentheses ().
- There are input parameters or arguments that should be placed within these parentheses.
- You can also define parameters inside these parentheses.
- There is a body within every function that starts with a colon () and is indented.
- You can also place documentation before the body.
- The statement **return** exits a function, optionally passing back a value.

Define a function for multiple two numbers

```
def Mult(a, b):
    c = a * b
    return(c)
    print('This is not printed')
result = Mult(12,2)
print(result)
```

Variables

- The input to a function is called a formal parameter.
- A variable that is declared inside a function is called a local variable. The parameter only exists within the function (i.e. the point where the function starts and stops).
- A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout the program. We will discuss more about global variables at the end of the lab

Pre-defined functions

album_ratings = [10.0, 8.5, 9.5, 7.0, 7.0, 9.5, 9.0, 9.5] print(album_ratings) sum(album_ratings) #Use sum() to add every element in a list or tuple together len(album_ratings) #Show the length of the list or tuple

Setting default argument values in your custom functions

#Example for setting param with default value

```
def isGoodRating(rating=4):
    if(rating < 7):
        print("this album sucks it's rating is",rating)</pre>
```

else:

print("this album is good its rating is", rating)

Global variables

#Example
artist = "Michael Jackson"
def printer(artist):
 global internal_var
 internal_var= "Whitney Houston"
 print(artist,"is an artist")
printer(artist)
printer(internal_var)

When the number of arguments are unknown for a function, They can all be packed into a tuple as shown:

```
def printAll(*args): #All the arguments are 'packed'
into args which can be treated like a tuple
    print("No of arguments:", len(args))
    for argument in args:
        print(argument)

printAll('Horsefeather','Adonis','Bone') #printAll with
3 arguments
printAll('Sidecar','Long Island','Mudslide','Carriage')
#printAll with 4 arguments
```

Similarly, The arguments can also be packed into a dictionary as shown

```
def printDictionary(**args):
    for key in args:
        print(key + " : " + args[key])
printDictionary(Country='Canada',Province='Ontario',City
='Toronto')
```

Exception Handling

An exception is an error that occurs during the execution of code. This error causes the code to raise an exception and if not prepared to handle it will halt the execution of the code

Try Except

A try except will allow you to execute code that might raise an exception and in the case of any exception or a specific one we can handle or catch the exception and execute specific code. This will allow us to continue the execution of our program even if there is an exception.

Python tries to execute the code in the **try** block. In this case if there is any exception raised by the code in the **try** block it will be caught and the code block in the **except** block will be executed.

Example [1]

a = 1 try: b = int(input("Please enter a number to divide a")) a = a/b print("Success a=",a) except: print("There was an error")

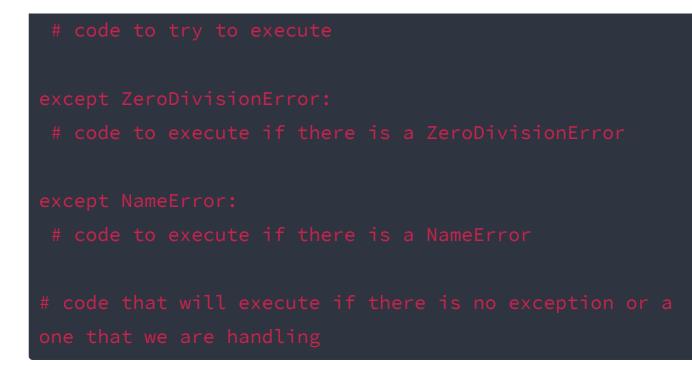
Example [2]

| <pre># potential code before try catch</pre> |
|--|
| try: |
| # code to try to execute |
| except (ZeroDivisionError, NameError): |
| <pre># code to execute if there is an exception of the given types</pre> |
| <pre># code that will execute if there is no exception or a one that we are handling</pre> |

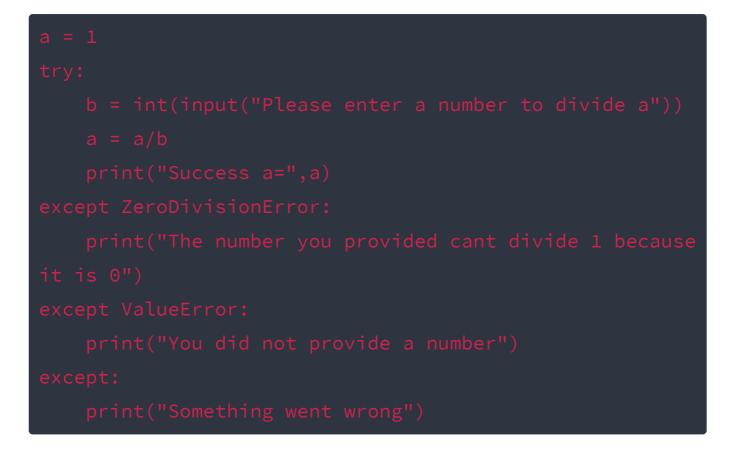
Example [3]

potential code before try catch

try:



Example [4]



Try Except Else and Finally

else allows one to check if there was no exception when executing the try block. This is useful when we want to execute something only if there were no errors.

finally allows us to always execute something even if there is an exception or not. This is usually used to signify the end of the try except.

Example [1]

Example [2]

trv:

b = int(input("Please enter a number to divide a")

| a = a/b |
|--|
| except ZeroDivisionError: |
| print("The number you provided cant divide 1 because |
| it is 0") |
| except ValueError: |
| <pre>print("You did not provide a number")</pre> |
| except: |
| <pre>print("Something went wrong")</pre> |
| else: |
| print("success a=",a) |
| finally: |
| <pre>print("Processing Complete")</pre> |

Classes, Objects and Methods

#Class is marked by having attributes (variables) that contain data which make up the class. Example is a rectangle class which has height, width and color as attributes
#object can be considered as a subset of the class with certain attribute. Example is red rectangle which is an object of the class rectangle.

#Methods are functions used to change and interact with objects.

For example, lets say we would like to increase the radius by a specified amount of a circle.

We can create a method called **add_radius(r)** that increases the radius by **r**.

After applying the method to the "orange circle object", the radius of the object increases accordingly. The "dot" notation means to apply the method to the object, which is essentially applying a function to the information in the object.

The first step in creating your own class is to use the **class** keyword, then the name of the class

The next step is a special method called a constructor ****init****, which is used to initialize the object

The input are data attributes. The term **self** contains all the attributes in the set. For example the **self.color** gives the value of the attribute color and **self.radius** will give you the radius of the object. We also have the method **add_radius()** with the parameter **r**, the method adds the value of **r** to the attribute radius. To access the radius we use the syntax

self.radius

Example [1]

```
import matplotlib.pyplot as plt
%matplotlib inline
# Create a class Circle
class Circle(object):
# Constructor
def __init__(self, radius=3, color='blue'):
    self.radius = radius
    self.color = color
```

print('Radius of object of after applying the method
add_radius(5):',RedCircle.radius)

Example [2]

SkinnyBlueRectangle.color
Use the drawRectangle method to draw the shape
SkinnyBlueRectangle.drawRectangle()

Example [3]

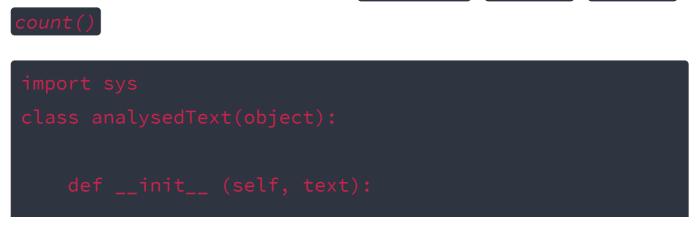
Text Analysis

You have been recruited by your friend, a linguistics enthusiast, to create a utility tool that can perform analysis on a given piece of text. Complete the class 'analysedText' with the following methods -

- Constructor Takes argument 'text',makes it lower case and removes all punctuation. Assume only the following punctuation is used - period (.), exclamation mark (!), comma (,) and question mark (?). Store the argument in "fmtText"
- freqAll returns a dictionary of all unique words in the text along with the number of their occurences.
- freqOf returns the frequency of the word passed in argument.

The skeleton code has been given to you. Docstrings can be ignored for the purpose of the exercise.

Hint: Some useful functions are replace(), lower(), split()



```
sampleMap = {'eirmod': 1,'sed': 1, 'amet': 2, 'diam': 5;
'consetetur': 1, 'labore': 1, 'tempor': 1, 'dolor': 1,
'magna': 2, 'et': 3, 'nonumy': 1, 'ipsum': 1, 'lorem':
2}
```

print("Constructor: ")

try:

samplePassage = analysedText("Lorem ipsum dolor! diam amet, consetetur Lorem magna. sed diam nonumy eirmod tempor. diam et labore? et diam magna. et diam amet.")

#samplePassage.fmtText == "lorem ipsum dolor diam amet consetetur lorem magna sed diam nonumy eirmod tempor diam et labore et diam magna et diam amet"

print(samplePassage.fmtText)

```
except:
    print("Error detected. Recheck your function " )
print("freqAll: ")
try:
    wordMap = samplePassage.freqAll()
    print(wordMap) #{'ipsum': 1, 'diam': 5, 'amet': 2,
'et': 3, 'consetetur': 1, 'labore': 1, 'dolor': 1,
'eirmod': 1, 'sed': 1, 'tempor': 1, 'magna': 2,
'nonumy': 1, 'lorem': 2}
    print(samplePassage.fmtText)
```

```
wordMap==sampleMap
```

print(wordMap) # lorem ipsum dolor diam amet consetetur lorem magna sed diam nonumy eirmod tempor diam et labore et diam magna et diam amet except:

print("Error detected. Recheck your function ")

```
print("freqOf: ")
try:
    passed = True
    for word in sampleMap:
        if samplePassage.freqOf(word) !=
sampleMap[word]:
        passed = False
        break
```

except: print("Error detected. Recheck your function "

Example [4]

```
class Points(object):
```

```
def \_\_\(self,x,y):
```

```
self.x=x
```

```
self.y=y
def print\_point(self):
    print('x=',self.x,' y=',self.y)
pl=Points("A","B")
pl.print\_point()
# output: x= A y= B
```

Example [5]

```
class Points(object):
def \_\_init\_\_(self,x,y):
self.x=x
self.y=y
def print\_point(self):
print('x=',self.x,' y=',self.y)
```

| I | p2=Points(1,2) |
|---|----------------|
| | |
| | |
| | |

Downloading files ove the network

```
import urllib.request
url = 'https://cf-courses-data.s3.us.cloud-obj
storage.appdomain.cloud/IBMDeveloperSkillsNetw
PY0101EN-
SkillsNetwork/labs/Module%204/data/example1.tx
filename = 'Example1.txt'
```

reading and writing files

One way to read or write a file in Python is to use the built-in open function. The open function provides a File object that contains the methods and attributes you need in order to read, save, and manipulate the file. In this notebook, we will only cover .txt files. The first parameter you need is the file path and the file name

- r Read mode for reading files
- w Write mode for writing files

Example [1]

```
``
example1 = "Example1.txt"
file1 = open(example1, "r")
```

Print the path of file

file1.name

Print the mode of file, either 'r' or 'w'

file1.mode

Read the file

FileContent = file1.read() FileContent

Print the file with '\n' as a new line

print(FileContent)

Type of file content

type(FileContent)

Close file after finish

Example [2]

Using the with statement is better practice, it automatically closes the file even if the code encounters an exception. The code will run everything in the indent block then close the file object.

• •

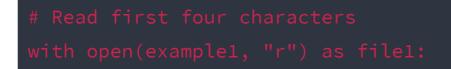
Open file using with

```
with open(example1, "r") as file1:
FileContent = file1.read()
print(FileContent)
```

Verify if the file is closed

file1.closed

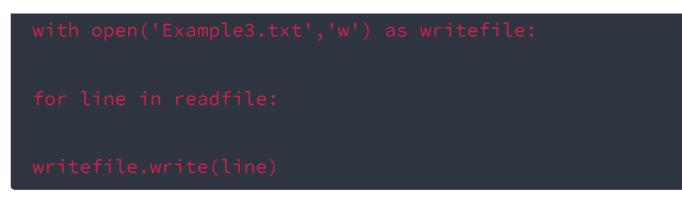
We don't have to read the entire file, for example, we can read the first 4 characters by entering three as a parameter to the method **.read()** Once the method **.read(4)** is called the first 4 characters are called. If we call the method again, the next 4 characters are called



Example [3]

copying from example.txt to example3.txt

with open('Example2.txt','r') as readfile:



REST APIs

Rest API's function by sending a request, the request is communicated via HTTP message. The HTTP message usually contains a JSON file. This contains instructions for what operation we would like the service or resource to perform. In a similar manner, API returns a response, via an HTTP message, this response is usually contained within a JSON

Example [1]

Create one candlestick graph for Bitcoin. Get the price data for 30 days with 24 observation per day, 1 per hour. We will find the max, min, open, and close price per day meaning we will have 30 candlesticks and use that to generate the candlestick graph. Although we are using the CoinGecko API we will use a Python client/wrapper for the API called PyCoinGecko. PyCoinGecko will make performing the requests easy and it will deal with the enpoint targeting.

Lets start off by getting the data we need. Using the get_coin_market_chart_by_id(id, vs_currency, days). id is
the name of the coin you want, vs_currency is the currency

you want the price in, and days is how many days back from today you want

```
data = pd.DataFrame(bitcoin_price_data, columns=
['TimeStamp', 'Price'])
```

print(data)

output

#

Now that we have the DataFrame we will convert the timestamp to datetime and save it as a column called `Date`. We will map our `unix_to_datetime` to each timestamp and convert it to a readable datetime.

data['Date'] = pd.to_datetime(data['TimeStamp'],

unit='ms')

print(data)

output

Using this modified dataset we can now group by the `Date` and find the min, max, open, and close for the candlesticks

```
candlestick_data = data.groupby(data.Date.dt.date,
as_index=False).agg({"Price": ['min', 'max', 'first',
'last']})
```

```
print (candlestick_data)
```

Finally we are now ready to use plotly to create our

```
Candlestick Chart
```

fig.show()

Panda Library for Data analysis and CSV processing

• •

import pandas import pandas as pd

Read data from CSV file

csv_path = 'https://cf-courses-data.s3.us.cloud-objectstorage.appdomain.cloud/IBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork/Iabs/Module%204/data/TopSellingAlbums.csv'

df = pd.read_csv(csv_path)

read_csv : built in function in pandas to read csv file

read_excel: built in function in pandas to read xlsx file

df: dataframe. Dataframes comprised of rows and columns.

df.head()

examine the first five rows of the dataframe

converting a dictionary into a dataframe. keys are mapped as column labels and values correspond to the rows.

```
dictionary = {key1:value, key2:value}
dic-frame =pd.DataFrame(dictionary)
```

creating data frame of one column. In this case it can be used to access the column 'Length' of the dataframe 'df'

```
x = df'Length'
x
```

Access to multiple columns

```
y = df'Artist','Length','Genre'
y
```

Accessing specific rows and columns in a dataframe

Access the value on the first row and the first column

df.iloc[0, 0]

Access the value on the first row and the third column

df.iloc[0,2]

Access the column using the name

df.loc[1, 'Artist']

Slicing the dataframe

df.iloc[0:2, 0:3]

Slicing the dataframe using name

```
df.loc[0:2, 'Artist':'Released']
```

Transform Function in Pandas

JSON file Format with Pandas

JSON is built on two structures:

 A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. 2. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

JSON is a language-independent data format. It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data. It is a very common data format, with a diverse range of applications.

The text in JSON is done through quoted string which contains the value in key-value mapping within { }. It is similar to the dictionary in Python.

Python supports JSON through a built-in package called **json**. To use this feature, we import the json package in Python script

Writing JSON to a File

This is usually called **serialization**. It is the process of converting an object into a special format which is suitable for transmitting over the network or storing in file or database

To handle the data flow in a file, the JSON library in Python uses **dump()** or **dumps()** function to convert the Python objects into their respective JSON object, so it makes easy to write data to files

serialization using dump() function

json.dump() method can be used for writing to JSON file.

Syntax: json.dump(dict, file_pointer)

Parameters:

- 1. **dictionary** name of dictionary which should be converted to JSON object.
- file pointer pointer of the file opened in write or append mod

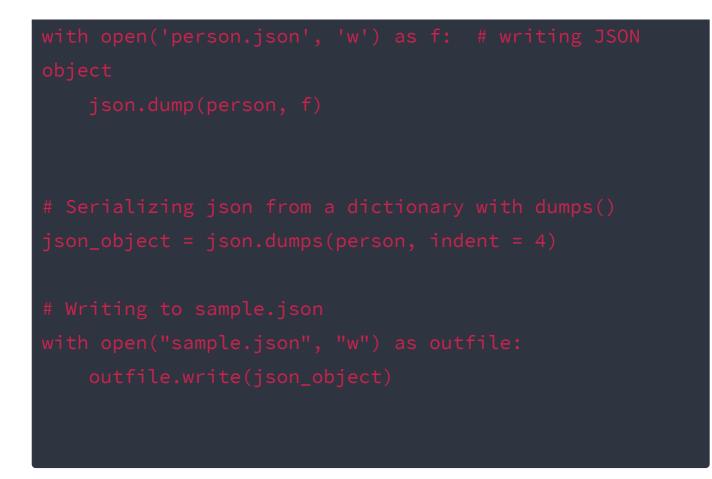
serialization using dumps() function

json.dumps() that helps in converting a dictionary to a JSON object.

It takes two parameters:

- 1. **dictionary** name of dictionary which should be converted to JSON object.
- 2. indent defines the number of units for indentation

```
import json
person = {
    'first_name' : 'Mark',
    'last_name' : 'abc',
    'age' : 27,
    'address': {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021-3100"
    }
# serializing Json object to file with dump(
```



Reading JSON to a File

This process is usually called **Deserialization**: It is the reverse of serialization. It converts the special format returned by the serialization back into a usable object.

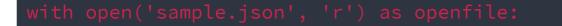
Using json.load()

The JSON package has json.load() function that loads the json content from a json file into a dictionary.

It takes one parameter:

File pointer: A file pointer that points to a JSON file





Reading from json file
json_object = json.load(openfile)

print(json_object)
print(type(json_object))

XLSX file Format with Pandas

```
import pandas as pd
import urllib.request
urllib.request.urlretrieve("https://cf-courses-
data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
PY0101EN-
SkillsNetwork/labs/Module%205/data/file_example_XLSX_10.
xlsx", "sample.xlsx")
df = pd.read_excel("sample.xlsx")
```

XML file Format with Pandas

XML is also known as Extensible Markup Language. As the name suggests, it is a markup language. It has certain rules for encoding data. XML file format is a human-readable and machine-readable file format. Pandas does not include any methods to read and write XML files. Here, we will take a look at how we can use other modules to read data from an XML file, and load it into a Pandas DataFrame

The **xml.etree.ElementTree** module comes built-in with Python. It provides functionality for parsing and creating XML documents. ElementTree represents the XML document as a tree. We can move across the document using nodes which are elements and sub-elements of the XML file

```
# create the file structure
employee = ET.Element('employee')
details = ET.SubElement(employee, 'details')
first = ET.SubElement(details, 'firstname')
second = ET.SubElement(details, 'lastname')
third = ET.SubElement(details, 'age')
first.text = 'Shiv'
second.text = 'Shiv'
second.text = 'Mishra'
third.text = '23'
# create a new XML file with the results
mydata1 = ET.ElementTree(employee)
# myfile = open("items2.xml", "wb")
# myfile.write(mydata)
with open("new_sample.xml", "wb") as files:
mydata1.write(files)
```

Let's have a look at a one ways to read XML data and put it in a Pandas DataFrame. You can see the XML file in the Notepad of your local machine

```
import pandas as pd
import xml.etree.ElementTree as etree
!wget https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
PY0101EN-SkillsNetwork/labs/Module%205/data/Sample-
employee-XML-file.xml
```

astname, title, division, building, room], index

columns), ignore_index = True)

Correspondingly, Pandas enables us to save the dataset to csv by using the **dataframe.to_csv()** method, you can add the file path and name along with quotation marks in the brackets.

For example, if you would save the dataframe df as
employee.csv to your local machine

datatframe.to_csv("employee.csv", index=False)

Binary file Format with Pandas

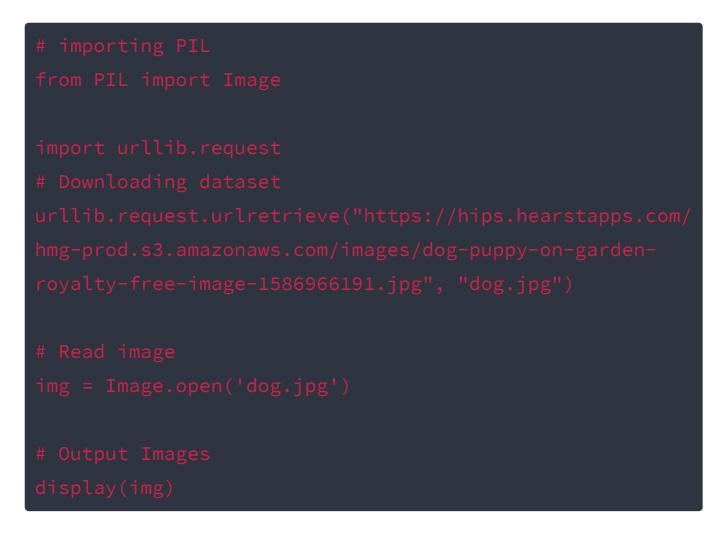
"Binary" files are any files where the format isn't made up of readable characters. It contain formatting information that only certain applications or processors can understand. While humans can read text files, binary files must be run on the appropriate software or processor before humans can read them.

Binary files can range from image files like JPEGs or GIFs, audio files like MP3s or binary document formats like Word or PDF

Reading the Image file

Python supports very powerful tools when comes to image processing. Let's see how to process the images using **PIL**library.

PIL is the Python Imaging Library which provides the python interpreter with image editing capabilities



Secnario For Analysing Data with Pandas

the **Diabetes Dataset** is an online source, and it is in CSV (comma separated value) format. Let's use this dataset as an example to practice data reading

Context This dataset is originally from the **National Institute** of **Diabetes and Digestive and Kidney Diseases**. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

Content The datasets consists of several medical predictor variables and one target variable, Outcome. Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on

We have 768 rows and 9 columns. The first 8 columns represent the features and the last column represent the target/label

df.head(5)

```
# view the dimensions of the dataframe
df.shape
```

df.info() # This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage

```
df.describe()
```

Pandas **describe()** is used to view some basic statistical details like percentile, mean, std etc. of a data frame or a series of numeric values. When this method is applied to a series of string, it returns a different output

We use Python's built-in functions to identify these missing values. There are two methods to detect missing data:

```
# **.isnull()**
```

```
# **.notnull()**
```

The output is a boolean value indicating whether the value that is passed into the argument is in fact missing data

```
missing_data = df.isnull()
```

missing_data.head(5)

Count missing values in each column
(http://localhost:8888/notebooks/Downloads/PY0101EN5.4_WorkingWithDifferentFileTypes.ipynb#Count-missingvalues-in-each-column)

Using a for loop in Python, we can quickly figure out the number of missing values in each column. As mentioned above, "True" represents a missing value, "False" means the value is present in the dataset. In the body of the for loop the method ".value_counts()" counts the number of "True" values

for column in missing_data.columns.values.tolist():
 print(column)
 print (missing_data[column].value_counts())
 print("")

Correct data
format(http://localhost:8888/notebooks/Downloads/PY0101E
N-5.4_WorkingWithDifferentFileTypes.ipynb#Correct-dataformat)

Check all data is in the correct format (int, float, text or other).

In Pandas, we use

.dtype() to check the data type

Numpy LLibrary for scientific computations

A numpy array or ND array is similar to a list. It's usually fixed in size and each element is of the same type, in this case integers

```
# Adding two tists together or vectors together.
b=2*z
# multiplying the array z with 2. Each elements will be
multiplied by 2.
c=np.dot(va,ua)
# np.dot multiplies corresponding elements and finally
adds them together: c= 1*0 + 0*1 = 0
e=z+1
# adds one to each elements of the array z
mean_e=e.mean()
# calculates the mean of e array elements.
```

we can access the data via an index. As with the list, we can access each element with an integer and a square bracket

plotting arays

```
x=np.linspace(0,2*np.pi,100)
# x is array of 100 elements starts at 0 and ends at
2*np.pi
y=np.sin(x)
# y is an array that corresponds to the sin function of
each element of x
```

import matplotlib.pyplot as plt
importing the plot library

%matplotlib inline
display the plot

plt.plot(x,y)
plot the arrays

2D Arrays

| a=[[11,12,13],[21,22,23],[31,32,33]] |
|--|
| A=np.array(a) |
| |
| A.shape |
| # returns the tuple (3,3) (number of rows, number of |
| columns) |
| |
| A.size |
| # returns 9 or 3*3 |
| |

It is helpful to visualize the numpy array as a rectangular array each

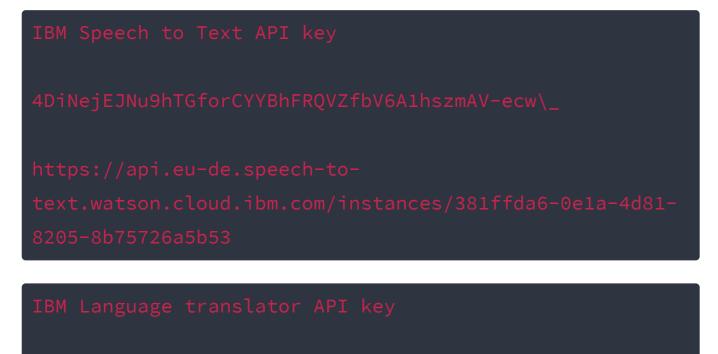
nested lists corresponds to a different row of the matrix.

$$A:\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

The indexing is illustrated below



Adding, subtracting, multiplication and multiplying by a number applies the same way as in 1D arrays.



ePlcBBRCH3ICSnDhXJqCS2XZlv9zWMTRlfnIwjHbXjQN

https://api.eu-gb.language-

translator.watson.cloud.ibm.com/instances/affe2127-f05d-43e4-a302-e6b30b0dfc11

Webscrabbing

Beautiful Soup is a Python library for pulling data out of HTML and XML files, we will focus on HTML files. This is accomplished by representing the HTML as a set of objects with methods used to parse the HTML. We can navigate the HTML as a tree and/or filter out what we are looking for

```
print(soup.prettify())
```

```
# display the HTML in the nested structure
```

tag_object=soup.title
print("tag object:",tag_object)
The `Tag` object corresponds to an HTML tag in the
original document, for example, the tag title. Prints
the page title
output: tag object: <title>Page Title</title>

tag_object=soup.n3
print("tag object:",tag_object)
print h3 tag and if there is more than one, print the
first occurrence
output: tag object: <h3><b id="boldest">Lebron
James</h3>

```
tag_child =tag_object.b
print("tag child:",tag_child)
# print the b tag
# output: tag child: <b id="boldest">Lebron James</b>
```

```
parent_tag=tag_child.parent
print("parent_tag:",parent_tag)
# print the tag parent to b which in this case is h3
# output:parent_tag: <h3><b id="boldest">Lebron
James</b></h3>
```

body_element=tag_object.parent
print("body_element:",body_element)
print the parent to h3
output:body_element: body_element: <body><h3><b
id="boldest">Lebron James</h3> Salary: \$
92,000,000 <h3> Stephen Curry</h3> Salary:
\$85,000, 000 <h3> Kevin Durant </h3> Salary:
\$73,200, 000</body>

sibling_1=tag_object.next_sibling
print("sibling_1:",sibling_1)
print the sibiling to the tag_object h3
output: sibling_1: Salary: \$ 92,000,000

sibling_2=sibling_1.next_sibling
print("sibling_2:",sibling_2)
output: sibling_2: <h3> Stephen Curry</h3>

sibiling_3=sibling_2.next_sibling
print("sibiling_3:",sibiling_3)
output: sibiling_3: Salary: \$85,000, 000

tag_child_id= tag_child['id']
print("tag_child_id:",tag_child_id)
output:tag_child_id: boldest
tag_child.get('id')

tag_string=tag_child.string

Filter with findall()

The **find_all()** method looks through a tag's descendants and retrieves all descendants that match your filters.

The Method signature for find_all(name, attrs, recursive,
string, limit, **kwargs)

When we set the name parameter to a tag name, the method will extract all the tags with that name and its children.

based on tags

```
from bs4 import BeautifulSoup
# this module helps in web scrapping.
import requests
# this module helps us to download a web page
table="Flight No

table="
table><ta
```

```
80 kg"
```

```
table_bs = BeautifulSoup(table, 'html5lib')
```

```
table_rows=table_bs.find_all('tr')
#extracting the tag 'tr' and all its children
# output is the below list where each element is a tag
object:
```

```
first_row =table_rows[0]
first_row
# output : Flight NoLaunch
site Payload mass
```

```
colunm 2 cell Payload mass
```

row 1

colunm 0 cell 1

As row is a cell object, we can apply the method find_all to it and extract table cells in the object cells using the tag td, this is all the children with the name td. The result is a list, each element corresponds to a cell and is a Tag object, we can iterate through this list as well. We can extract the content using the string attribute.

based on strings

With string you can search for strings instead of tags, where we find all the elments with Florida

table_bs.find_all(string="Florida")

The find_all() method scans the entire document looking for results, it's if you are looking for one element you can use the find() method to find the first element in the document. Consider the following two table

two_tables="<h3>Rocket Launch </h3>Flight NoLaunch
sitePayload massLaunch

Payload mass1Florida300 kgFlorida300 kg>Texas94 kg>333>Florida 80 kg>>Florida 80 kg>Florida 80 kg>Florida 80 kg>Florida 10>Florida 1010>>Florida 1010>>Little Caesars12144 >Texa<</td>12144 >Texa10165

two_tables_bs= BeautifulSoup(two_tables, 'html.parser')
two_tables_bs.find("table")

output

Flight NoLaunch site Payload mass1 Florida300 kg2 Florida300 kg3Kg3Kg4Kg</t

two_tables_bs.find("table",class_='pizza')

output

Pizza Place Orders Slices /tr>Domino's Pizza 10 /td> 100 /tr>Little Caesars 12 144 /tr>Papa John's 15 /td> 165 /tr>#

Downloading And Scraping The Contents Of A Web Page

url = "http://www.ibm.com"
data = requests.get(url).text
soup = BeautifulSoup(data,"html5lib") # create a soup
object using the variable 'data'
scrape links
for link in soup.find_all('a',href=True):
in html anchorlink is represented by the tag <a>
 print(link.get('href'))

Get all columns in each row.

```
cols = row.find_all('td')
# in html a column is represented by the tag
```

```
color_name = cols[2].string
# store the value in column 3 as color_name
```

```
color_code = cols[3].string
# store the value in column 4 as color_code
```

```
print("{}--->{}".format(color_name,color_code))
```

Scrape data from HTML tables into a DataFrame using BeautifulSoup and Pandas



```
for row in tables[table_index].tbody.find_all("tr"):
    col = row.find_all("td")
    if (col != []):
        rank = col[0].text
        country = col[1].text
        population = col[2].text.strip()
        area = col[3].text.strip()
        density = col[4].text.strip()
        population_data =
population_data.append({"Rank":rank, "Country":country,
"Population":population, "Area":area,
"Density":density}, ignore_index=True)
```

HTTP Requests in Python

Requests is a python Library that allows you to send HTTP/1.1 requests easily

```
import requests
import os
from PIL import Image
from IPython.display import IFrame
# making a GET request and storing the response in 'r'
url='https://www.ibm.com/'
r=requests.get(url)
```

url='https://gitlab.com/ibm/skills-

Get Request with URL Parameters

```
# performing get request to the specified url
url_get='http://httpbin.org/get'
# storing url parameters in payload dict variable.
payload={"name":"Joseph","ID":"123"}
# passing the dictionary `payload` to the `params`
parameter of the `get()` function
```

POST Request with URL Parameters

```
url_post='http://httpbin.org/post'
r_post=requests.post(url_post,data=payload)
print("POST request URL:",r_post.url )
print("GET request URL:",r.url)
print("POST request body:",r_post.request.body)
print("GET request body:",r.request.body)
r_post.json()['form']
```

Data Engineering Process

There are several steps in Data Engineering process.

- Extract :- Data extraction is getting data from multiple sources. Ex. Data extraction from a website using Web scraping or gathering information from the data that are stored in different formats(JSON, CSV, XLSX etc.).
- Transform :- Tarnsforming the data means removing the data that we don't need for further analysis and converting the data in the format that all the data from the multiple sources is in the same format.
- 3. Load :- Loading the data inside a data warehouse. Data warehouse essentially contains large volumes of data that are accessed to gather insights

Project: extracting stock data with finance library

For this project, you will assume the role of a Data Scientist / Data Analyst working for a new startup investment firm that helps customers invest their money in stocks. Your job is to extract financial data like historical share price and quarterly revenue reportings from various sources using Python libraries and webscraping on popular stocks. After collecting this data you will visualize it in a dashboard to identify patterns or trends. The stocks we will work with are Tesla, Amazon, AMD, and GameStop A company's [stock] share is a piece of the company; more precisely:

A stock (also known as equity) is a security that represents the ownership of a fraction of a [corporation]. This entitles the owner of the stock to a proportion of the corporation's [assets] _and profits equal to how much stock they own. Units of stock are called "shares."

An investor can buy a stock and sell it later. If the stock price increases, the investor profits, If it decreases, the investor with incur a loss. Determining the stock price is complex; it depends on the number of outstanding shares, the size of the company's future profits, and much more. People trade stocks throughout the day. The **stock ticker** is a report of the price of a certain stock, updated continuously throughout the trading session by the various **stock** market exchanges. In this lab, you will use the y-finance API to obtain the stock ticker and extract information about the stock. You will then be asked questions about your results

Code: Apple stocks analysis

1- import the yfinance library

2- Using the **Ticker** module we can create an object that will allow us to access functions to extract data. To do this we need to provide the ticker symbol for the stock, here the company is Apple and the ticker symbol is **AAPL**. Now we can access functions and variables to extract the type of data we need. You can view them and what they represent here

3- Using the attribute info we can extract information about the stock as a Python dictionary.

4- We can get the **'country'** using the key country

5- A share is the single smallest part of a company's stock that you can buy, the prices of these shares fluctuate over time. Using the history() method we can get the share price of the stock over a certain period of time. Using the period parameter we can set how far back from the present to get data. The options for period are 1 day (1d), 5d, 1 month (1mo) , 3mo, 6mo, 1 year (1y), 2y, 5y, 10y, ytd, and max

6- Extracting Dividends : Dividends are the distribution of a companys profits to shareholders. In this case they are defined as an amount of money returned per share an investor owns. Using the variable **dividends** we can get a dataframe of the data. The period of the data is given by the period defined in the 'history` function

```
import yfinance as yf
import pandas as pd
apple = yf.Ticker("AAPL")
apple_info=apple.info
apple_info
```

The format that the data is returned in is a Pandas DataFrame. With the Date as the index the share Open, High, Low, Close, Volume, and Stock Splits are given for each day

apple_share_price_data.head()

apple_share_price_data.reset_index(inplace=True)

We can reset the index of the DataFrame with the **reset_index** function. We also set the **inplace** paramter to **True** so the change takes place to the DataFrame itself.

apple_share_price_data.plot(x="Date", y="Open")

We can plot the **Open** price against the **Date**

print (apple.dividends)

output: Name: Dividends, Length: 70, dtype: float64

```
apple.dividends.plot()
```

Code: AMD (Advanced Micro Devices) stocks analysis

```
import yfinance as yf
import pandas as pd
amd = yf.Ticker("AMD")
amd info=amd.info
amd_country=amd_info['country']
print(amd_country)
amd_sector=amd_info['sector']
print(amd_sector)
amd_share_price_data = amd.history(period="max")
amd_share_price_data.reset_index(inplace=True)
volcolumn=amd_share_price_data'Volume'
volmax=volcolumn.max()
print('maximum value of volume columns is:', volmax)
```

• •

• •

Project: extracting stock data with webscrabing

Code: Extracting stock data from a webpage

import pandas as pd import requests from bs4 import BeautifulSoup import urllib.request

url = 'https://finance.yahoo.com/quote/AMZN/history? period1=1451606400&period2=1612137600&interval=1mo&filte r=history&frequency=1mo&includeAdjustedClose=true&cm_mmc =Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-PY0220EN-SkillsNetwork-23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-

23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-PY0220EN-SkillsNetwork-

23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-PY0220EN-SkillsNetwork-

23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ'

Project: extracting stock data with webscrabing and finance API

1- we define the function make_graph. You don't have to know how the function works, you should only care about the inputs. It takes a dataframe with stock data (dataframe must contain Date and Close columns), a dataframe with revenue data (dataframe must contain Date and Revenue columns), and the name of the stock

Tesla Stock Data Extraction and Cleaning

```
import yfinance as yf
import pandas as pd
import requests
from bs4 import BeautifulSoup
import plotly.graph_objects as go
from plotly.subplots import make_subplots
def make_graph(stock_data, revenue_data, stock):
    fig = make_subplots(rows=2, cols=1,
    shared_xaxes=True, subplot_titles=("Historical Share
Price", "Historical Revenue"), vertical_spacing = .3)
```

fig.add_trace(go.Scatter(x=pd.to_datetime(stock_data.Dat

PY0220EN-SkillsNetwork-

23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-PY0220EN-SkillsNetwork-

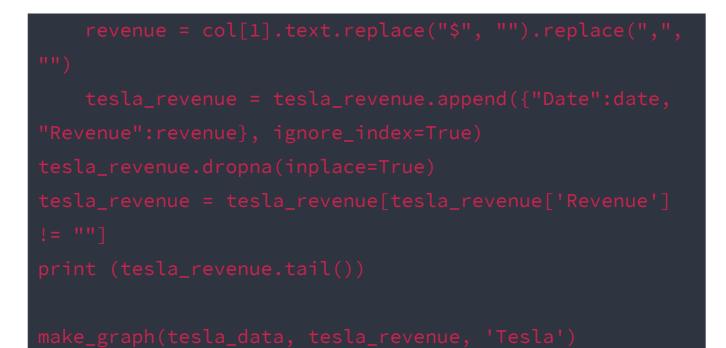
23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-PY0220EN-SkillsNetwork-

23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-PY0220EN-SkillsNetwork-

23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ'

```
html_data = requests.get(url).text
soup = BeautifulSoup(html_data, 'html5lib')
tesla_revenue = pd.DataFrame(columns=["Date",
"Revenue"])
tesla_table = soup_find_all('table')[1]
```

```
for row in tesla_table.find("tbody").find_all('tr'):
    col = row.find_all("td")
    date =col[0].string
    revenue =col[1].string
```



GameStop Stock Data Extraction and Cleaning

import yfinance as yf import pandas as pd import requests from bs4 import BeautifulSoup import plotly.graph_objects as go from plotly.subplots import make_subplots def make_graph(stock_data, revenue_data, stock): fig = make_subplots(rows=2, cols=1, shared_xaxes=True, subplot_titles=("Historical Share Price", "Historical Revenue"), vertical_spacing = .3) fic add trace(colocation(upped to detetion(upped))

```
fig.add_trace(go.Scatter(x=pd.to_datetime(stock_data.Dat
e, infer_datetime_format=True),
```

```
y=stock_data.Close.astype("float"), name="Share Price");
row=1, col=1)
```

```
url =
```

```
'https://www.macrotrends.net/stocks/charts/GME/gamestop/
revenue?cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-
WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-
PY0220EN-SkillsNetwork-
23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1
2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0
```

00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-PY0220EN-SkillsNetwork-

23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-PY0220EN-SkillsNetwork-

```
23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1
2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0
00026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-
WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-
PY0220EN-SkillsNetwork-
```

23455606&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M1 2345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=0 00026UJ'

```
html_data = requests.get(url).text
soup = BeautifulSoup(html_data, 'html5lib')
gme_revenue = pd.DataFrame(columns=["Date", "Revenue"])
gme_table = soup.find_all('table')[1]
```

```
for row in gme_table.find("tbody").find_all('tr'):
    col = row.find_all("td")
    date =col[0].string
    revenue =col[1].string
    revenue = col[1].text.replace("$", "").replace(",",
"")
    gme_revenue = gme_revenue.append({"Date":date,
```

```
'Revenue":revenue}, ignore_index=True)
```

```
gme_revenue.dropna(inplace=True)
gme_revenue = gme_revenue[gme_revenue['Revenue'] != ""]
print (gme_revenue.tail())
```

make_graph(gme_data, gme_revenue, 'GameStop')

Python with SQL

Connecting to databases

The ibm_db provides a variety of useful Python functions for accessing and manipulating data in an IBM® data server database, including functions for connecting to a database, preparing and issuing SQL statements, fetching rows from result sets, calling stored procedures, committing and rolling back transactions, handling errors, and retrieving metadata.

import ibm_db

```
#Replace the placeholder values with your actual IBM Db2
hostname, username, and password:
dsn_hostname = "dashdb-txn-sbox-yp-lon02-15.services.eu-
gb.bluemix.net"
dsn_uid = "tcc02603"
dsn_pwd = "43360z^w0llrh6rm"
dsn_driver = "DATABASE=BLUDB;HOSTNAME=dashdb-txn-sbox-
yp-lon02-15.services.eu-
interestioned.eu/
```

```
conn = ibm db.connect(dsn. "". "'
```

```
print ("Connected to database: ", dsn_database, "as
user: ", dsn_uid, "on host: ", dsn_hostname)
```

cept:
 print ("Unable to connect: ". ibm db.conn errorm

ibm_db.close(conn)

Creating tables and queries

Table definition

INSTRUCTOR

| COLUMN NAME | DATA TYPE | NULLABLE |
|-------------|-----------|----------|
| ID | INTEGER | Ν |
| FNAME | VARCHAR | Υ |
| LNAME | VARCHAR | Y |
| CITY | VARCHAR | Y |
| CCODE | CHARACTER | Υ |

connect to the database as the code above shows
#Lets first drop the table INSTRUCTOR in case it exists
from a previous attempt

dropQuery = "drop table INSTRUCTOR"

```
#Now execute the drop statment
dropStmt = ibm_db.exec_immediate(conn, dropQuery)
```

#Construct the Create Table DDL statement

```
createQuery = "create table INSTRUCTOR(ID INTEGER
PRIMARY KEY NOT NULL, FNAME VARCHAR(20), LNAME
VARCHAR(20), CITY VARCHAR(20), CCODE CHAR(2))"
```

```
createStmt = ibm_db.exec_immediate(conn,createQuery)
```

Insert data into the table

#Construct the query - replace ... with the insert

```
insertQuery = "insert into INSTRUCTOR values (1, 'Rav',
'Ahuja', 'TORONTO', 'CA')"
```

insertStmt = ibm_db.exec_immediate(conn, insertQuery)

```
insertQuery2 = "insert into INSTRUCTOR values (2,
'Raul', 'Chong', 'Markham', 'CA'), (3, 'Hima',
'Vasudevan', 'Chicago', 'US')"
```

insertStmt2 = ibm_db.exec_immediate(conn, insertQuery2)

Query data in the table

```
#Construct the query that retrieves all rows from the
INSTRUCTOR table
selectQuery = "select * from INSTRUCTOR"
```

#Execute the statement
selectStmt = ibm_db.exec_immediate(conn, selectQuery)

#Fetch the Dictionary (for the first row only)
ibm_db.fetch_both(selectStmt)

```
#Fetch the rest of the rows and print the ID and FNAME
for those rows
```

```
while ibm_db.fetch_row(selectStmt) != False:
    print (" ID:", ibm_db.result(selectStmt, 0), "
FNAME:", ibm_db.result(selectStmt, "FNAME"))
# write and execute an update statement that changes the
Rav's CITY to MOOSETOWN
updateQuery = "update INSTRUCTOR set CITY='MOOSETOWN'
where FNAME='Rav'"
updateStmt = ibm_db.exec_immediate(conn, updateQuery))
```

Retrieve data into Pandas

```
# retrieve the contents of the INSTRUCTOR table into a
Pandas dataframe
import pandas
import ibm_db_dbi
#connection for pandas
pconn = ibm_db_dbi.Connection(conn)
#query statement to retrieve all rows in INSTRUCTOR
table
selectQuery = "select * from INSTRUCTOR"
#retrieve the query results into a pandas dataframe
pdf = pandas.read_sql(selectQuery, pconn)
```

```
#print just the LNAME for first row in the pandas data
frame
pdf.LNAME[0]
#print the entire data frame
pdf
#use the shape method to see how many rows and columns
are in the dataframe
pdf.shape
ibm_db.close(conn)
```

Accessing Databases with SQL Magic

To communicate with SQL Databases from within a JupyterLab notebook, we can use the SQL "magic" provided by the [ipython-sql] extension. "Magic" is JupyterLab's term for special commands that start with "%". Below, we'll use the [load_ext] magic to load the ipython-sql extension

```
!pip install sqlalchemy==1.3.9
!pip install ibm_db_sa
%load_ext sql
# Enter your Db2 credentials in the connection string
below
```

Recall you created Service Credentials in Part III of the first lab of the course in Week 1 # i.e. from the uri field in the Service Credentials copy everything after db2:// (but remove the double quote at the end) # for example, if your credentials are as in the screenshot above, you would write: # %sql ibm_db_sa://my-username:my-password@dashdb-txnsbox-yp-dal09-03.services.dal.bluemix.net:50000/BLUDB # Note the ibm_db_sa:// prefix instead of db2:// # This is because JupyterLab's ipython-sql extension uses sqlalchemy (a python SQL toolkit) # which in turn uses IBM's sqlalchemy dialect: ibm_db_sa

%sql ibm_db_sa://tcc02603:43360z%5Ew0llrh6rm@dashdb-txnsbox-yp-lon02-15.services.eu-gb.bluemix.net:50000/BLUDB

#For convenience, we can use %%sql (two %'s instead of one) at the top of a cell to indicate we want the entire cell to be treated as SQL. Let's use this to create a table and fill it with some test data for experimenting

%%sql

CREATE TABLE INTERNATIONAL_STUDENT_TEST_SCORES (country VARCHAR(50), first_name VARCHAR(50), last_name VARCHAR(50), test_score INT

('China', 'Berni', 'Daintier', 55), ('Poland', 'Cynthia', 'Hassell', 49), ('Canada', 'Carma', 'Schule', 49), ('Indonesia', 'Malia', 'Blight', 48), ('China', 'Paulo', 'Seivertsen', 47), ('Niger', 'Kaylee', 'Hearley', 54), ('Japan', 'Maure', 'Jandak', 46), ('Japan', 'Maure', 'Jandak', 46), ('Argentina', 'Foss', 'Feavers', 45), ('Argentina', 'Foss', 'Feavers', 45), ('Venezuela', 'Ron', 'Leggitt', 60), ('Russia', 'Flint', 'Gokes', 40), ('China', 'Linet', 'Conelly', 52), ('Philippines', 'Nikolas', 'Birtwell', 57) ('Australia', 'Eduard', 'Leipelt', 53)

##You can use python variables in your SQL statements by adding a ":" prefix to your python variable names.

#For example, if I have a python variable `country` with a value of `"Canada"`, I can use this variable in a SQL query to find all the rows of students from Canada

```
country = "Canada"
%sql select * from INTERNATIONAL_STUDENT_TEST_SCORES
where country = :country
```

You can use the normal python assignment syntax to
assign the results of your queries to python variables.
[]

For example, I have a SQL query to retrieve the distribution of test scores (i.e. how many students got each score). I can assign the result of this query to the variable `test_score_distribution` using the `=` operator

test_score_distribution = %sql SELECT test_score as "Test Score", count(*) as "Frequency" from INTERNATIONAL_STUDENT_TEST_SCORES GROUP BY test_score; test_score_distribution

You can easily convert a SQL query result to a pandas dataframe using the `DataFrame()` method. Dataframe objects are much more versatile than SQL query result objects. For example, we can easily graph our test score distribution after converting to a dataframe

dataframe = test_score_distribution.DataFrame()

%matplotlib inline
uncomment the following line if you get an module
error saying seaborn not found
!pip install seaborn
import seaborn

plot = seaborn.barplot(x='Test Score',y='Frequency', data=dataframe)

Selected Socioeconomic Indicators in Chicago

The city of Chicago released a dataset of socioeconomic data to the Chicago City Portal. This dataset contains a selection of six socioeconomic indicators of public health significance and a "hardship index," for each Chicago community area, for the years 2008 – 2012.

Scores on the hardship index can range from 1 to 100, with a higher index number representing a greater level of hardship.

A detailed description of the dataset can be found on the city of Chicago's website, but to summarize, the dataset has the following variables:

- **Community Area Number** (ca): Used to uniquely identify each row of the dataset
- **Community Area Name** (community_area_name): The name of the region in the city of Chicago
- Percent of Housing Crowded
 (percent_of_housing_crowded): Percent of occupied
 housing units with more than one person per room
- Percent Households Below Poverty
 (percent_households_below_poverty): Percent of
 households living below the federal poverty line
- Percent Aged 16+ Unemployed
 (percent_aged_16_unemployed): Percent of persons over the age of 16 years that are unemployed

Percent Aged 25+ without High School Diploma

(percent_aged_25_without_high_school_diploma): Percent of persons over the age of 25 years without a high school education

• **Percent Aged Under** 18 or Over 64:Percent of population under 18 or over 64 years of age

(percent_aged_under_18_or_over_64): (ie. dependents)

- Per Capita Income (per_capita_income_): Community Area per capita income is estimated as the sum of tractlevel aggragate incomes divided by the total population
- Hardship Index (hardship_index): Score that incorporates each of the six selected socioeconomic indicators

%load_ext sql

%sql ibm_db_sa://tcc02603:43360z%5Ew0llrh6rm@dashdb-txnsbox-yp-lon02-15.services.eu-gb.bluemix.net:50000/BLUDB

In many cases the dataset to be analyzed is available as a .CSV (comma separated values) file, perhaps on the internet. To analyze the data using SQL, it first needs to be stored in the database.

#We will first read the dataset source .CSV from the internet into pandas dataframe[]

#Then we need to create a table in our Db2 database to store the dataset. The PERSIST command in SQL "magic" simplifies the process of table creation and writing the data from a `pandas` dataframe into the table

```
import pandas
chicago_socioeconomic_data =
pandas.read_csv('https://data.cityofchicago.org/resource
/jcxq-k9xf.csv')
%sql PERSIST chicago_socioeconomic_data
```

#How many rows are in the dataset

%sql SELECT COUNT(*) FROM chicago_socioeconomic_data;

#How many community areas in Chicago have a hardship index greater than 50.0?

%sql SELECT COUNT(*) FROM chicago_socioeconomic_data
WHERE hardship_index > 50.0;

#What is the maximum value of hardship index in this dataset?

%sql SELECT MAX(hardship_index) FROM
chicago_socioeconomic_data;

```
#We can use the result of the last query to as an input
to this query:
%sql SELECT community_area_name FROM
chicago_socioeconomic_data where hardship_index=98.0
```

```
#or another option:
```

%sql SELECT community_area_name FROM chicago_socioeconomic_data ORDER BY hardship_index DESC NULLS LAST FETCH FIRST ROW ONLY;

#or you can use a sub-query to determine the max hardship index: %sql select community_area_name from chicago_socioeconomic_data where hardship_index = (select max(hardship_index) from chicago_socioeconomic_data)

Which Chicago community areas have per-capita incomes greater than \$60,000?

%sql SELECT community_area_name FROM chicago_socioeconomic_data WHERE per_capita_income_ > 60000;

Create a scatter plot using the variables `per_capita_income_` and `hardship_index`. Explain the correlation between the two variables # if the import command gives ModuleNotFoundError: No module named 'seaborn' # then uncomment the following line i.e. delete the # to install the seaborn package # !pip install seaborn

import matplotlib.pyplot as plt
%matplotlib inline

```
income_vs_hardship = %sql SELECT per_capita_income_,
hardship_index FROM chicago_socioeconomic_data;
plot =
sns.jointplot(x='per_capita_income_',y='hardship_index',
data=income_vs_hardship.DataFrame())
```

- You can access a database from a language like Python by using the appropriate API. Examples include ibm_db API for IBM DB2, psycopg2 for ProstgreSQL, and dblib API for SQL Server.
- DB-API is Python's standard API for accessing relational databases. It allows you to write a single program that works with multiple kinds of relational databases instead of writing a separate program for each one.
- The DB_API connect constructor creates a connection to the database and returns a Connection Object, which is then used by the various connection methods.
- The connection methods are:
- The cursor() method, which returns a new cursor object using the connection.
- The commit() method, which is used to commit any pending transaction to the database.
- The rollback() method, which causes the database to rollback to the start of any pending transaction. The close() method, which is used to close a database connection.
- You can use **SQL Magic** commands to execute queries more easily from Jupyter Notebooks. Magic commands have the general format **%sql select * from tablename**.

Cell magics start with a double %% (percent) sign and apply to the entire cell. **Line magics** start with a single % (percent) sign and apply to a particular line in a cell.

Chicago Public Schools - Progress Report Cards (2011-2012)

```
import pandas
import ibm_db_dbi
%load_ext sql
# Enter the connection string for your Db2 on Cloud
database instance below
# %sql ibm_db_sa://my-username:my-password@my-
hostname:my-port/my-db-name
```

%sql ibm_db_sa://nsk05922:834d52l077cnkd%2Bd@dashdb-txnsbox-yp-dal09-08.services.dal.bluemix.net:50000/BLUDB

type in your query to retrieve list of all tables in the database for your db2 schema (username)

```
#In Db2 the system catalog table called SYSCAT.TABLES contains the table metadata
```

%sql select TABSCHEMA, TABNAME, CREATE_TIME from SYSCAT.TABLES where TABSCHEMA=nsk05922

#or, just query for a specifc table that you want to verify exists in the database

```
%sql select * from SYSCAT.TABLES where TABNAME =
'SCHOOLS'
```

type in your query to retrieve the number of columns in the SCHOOLS table

%sql select COLNAME, TYPENAME, LENGTH from SYSCAT.COLUMNS where TABNAME = 'SCHOOLS'

#How many Elementary Schools are in the dataset?

%sql select count(*) from SCHOOLS where "Elementary, Middle, or High School" = 'ES'

#What is the highest Safety Score?

%sql select MAX(Safety_Score) AS MAX_SAFETY_SCORE from SCHOOLS

#Which schools have highest Safety Score?

%sql select Name_of_School, Safety_Score from SCHOOLS where #What are the top 10 schools with the highest "Average Student Attendance"?

%sql select Name_of_School, Average_Student_Attendance from SCHOOLS order by Average_Student_Attendance desc nulls last limit 10

#Retrieve the list of 5 Schools with the lowest Average Student Attendance sorted in ascending order based on attendance

%sql SELECT Name_of_School, Average_Student_Attendance
 from SCHOOLS
 order by Average_Student_Attendance
 fetch first 5 rows only

#Now remove the '%' sign from the above result set for Average Student Attendance column

%sql SELECT Name_of_School, REPLACE(Average_Student_Attendance, '%', '') from SCHOOLS order by Average_Student_Attendance fetch first 5 rows only # Which Schools have Average Student Attendance lower than 70%?

```
%sql SELECT Name_of_School, Average_Student_Attendance
from SCHOOLS
where CAST ( REPLACE(Average Student Attendance,
```

```
'%', '') AS DOUBLE ) < 70
```

order by Average_Student_Attendance

#or

```
%sql SELECT Name_of_School, Average_Student_Attendance
from SCHOOLS
where DECIMAL ( REPLACE(Average_Student_Attendance,
'%', '') ) < 70</pre>
```

#Get the total College Enrollment for each Community Area

```
%sql select Community_Area_Name, sum(College_Enrollment)
AS TOTAL_ENROLLMENT
from SCHOOLS
group by Community_Area_Name
```

#Get the 5 Community Areas with the least total College Enrollment sorted in ascending order

```
where ca ir
```

(select community_area_number from schools order by college_enrollment desc limit 1)

```
import pandas
```

import ibm_db_dbi
%load_ext sql
%sql ibm_db_sa://nsk05922:834d52l077cnkdA%2Bd@dashdbtxn-sbox-yp-dal0908.services.dal.bluemix.net:50000/BLUDB

Find the total number of crimes recorded in the CRIME table. %sal SELECT count(*) FROM CHICAGO CRIME DATA

List community areas with per capita income less than 11000

%sql SELECT COMMUNITY_AREA_NAME FROM CENSUS_DATA WHERE PER_CAPITA_INCOME < 11000

List all case numbers for crimes involving minors? (children are not considered minors for the purposes of crime analysis)

%sql SELECT CASE_NUMBER FROM CHICAGO_CRIME_DATA WHERE DESCRIPTION LIKE '%MINOR'

What kinds of crimes were recorded at schools?

%sql SELECT DISTINCT(PRIMARY_TYPE), DESCRIPTION, LOCATION_DESCRIPTION FROM CHICAGO_CRIME_DATA WHERE LOCATION_DESCRIPTION LIKE 'SCHOOL%' OR LOCATION_DESCRIPTION LIKE '%SCHOOL' ##### List the average safety score for all types of schools.

%sql SELECT avg(SAFETY_SCORE) FROM CHICAGO_PUBLIC_SCHOOLS

List 5 community areas with highest % of households below poverty line

%sql SELECT COMMUNITY_AREA_NAME FROM CENSUS_DATA ORDER BY PERCENT_HOUSEHOLDS_BELOW_POVERTY DESC NULLS LAST LIMIT 5

Which community area is most crime prone?

%sql SELECT COMMUNITY_AREA_NUMBER, COUNT(COMMUNITY_AREA_NUMBER) AS mostcrimeprone FROM CHICAGO_CRIME_DATA GROUP BY COMMUNITY_AREA_NUMBER ORDER BY mostcrimeprone DESC LIMIT 5

Use a sub-query to determine the Community Area Name with most number of crimes?

%sql SELECT COMMUNITY_AREA_NAME FROM CENSUS_DATA
WHERE COMMUNITY_AREA_NUMBER =
(SELECT COMMUNITY_AREA_NUMBER FROM CHICAGO_CRIME_DATA
GROUP BY COMMUNITY_AREA_NUMBER ORDER BY COUNT(*) DESC
LIMIT 1)

Adding headers to a dataset and playing with the values

```
# Replacing columns with the wanted headers.
df.columns = headers
df.head(10)
```

```
# We need to replace the "?" symbol with NaN so the
dropna() can remove the missing values
df1=df.replace('?',np.NaN)
```

```
# We can drop missing values along the column "price" as
follows:
df=df1.dropna(subset=["price"], axis=0)
df.head(20)
```

Print the name of the columns of the dataframe
print(df.columns)

Save the dataframe **df** as **automobile.csv** to
your local machine, you may use the syntax below, where
`index = False` means the row names will not be written

```
df.to_csv("automobile.csv", index=False)
```

```
# check the data type of data frame "df" by .dtypes
print(df.dtypes
```

#If we would like to get a statistical summary of each column e.g. count, column mean value, column standard deviation, etc., we use the describe method (describe

Data wrangling

Data wrangling is the process of converting data from the initial format to a format that may be better for analysis

Fuel consumption (L/100k) rate for the diesel car

```
import pandas as pd
import matplotlib.pylab as plt
import numpy as np
```

```
filename = "https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
DA0101EN-SkillsNetwork/labs/Data%20files/auto.csv"
```

```
df = pd.read_csv(filename, names = headers)
```

```
df.head()
```

Convert "?" to NaN (Not a Number)
replace "?" to NaN

```
df.replace("?", np.nan, inplace = True)
df.head(5)
```

The missing values are converted by default. We use the following functions to identify these missing values. There are two methods to detect missing data:

```
1. **.isnull()**
```

2. **.notnull()**

in the column are empty. In our dataset, none of the

columns are empty enough to drop entirely. We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns

Replace by mean

"normalized-losses": 41 missing data, replace them with mean

- "stroke": 4 missing data, replace them with mean

- "bore": 4 missing data, replace them with mean

- "horsepower": 2 missing data, replace them with mean

- "peak-rpm": 2 missing data, replace them with mean

Replace by frequency

"num-of-doors": 2 missing data, replace them with
 "four".

- Reason: 84% sedans is four doors. Since four doors is most frequent, it is most likely to occur

Drop the whole row

"price": 4 missing data, simply delete the whole row
 Reason: price is what we want to predict. Any
 data entry without price data cannot be used for
 prediction; therefore any row now without price data is
 not useful to us

Calculate the mean value for the "normalizedlosses" column

```
avg_norm_loss = df["normalized-
losses"].astype("float").mean(axis=0)
```

```
print("Average of normalized-losses:", avg_norm_loss)
```

Replace "NaN" with mean value in "normalizedlosses" column

df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)

Calculate the mean value for the "bore" column

avg_bore=df['bore'].astype('float').mean(axis=0)

print("Average of bore:", avg_bore)

```
df["bore"].replace(np.nan, avg_bore, inplace=True)
```

avg_stroke=df["stroke"].astype("float").mean(axis=0)

print("Average of stroke:", avg_stroke)

df["stroke"].replace(np.nan, avg_stroke, inplace=True)

Calculate the mean value for the "horsepower" column

```
avg_horsepower =
df['horsepower'].astype('float').mean(axis=0)
```

print("Average horsepower:", avg_horsepower)

df['horsepower'].replace(np.nan, avg_horsepower, inplace=True)

Calculate the mean value for "peak-rpm" column

```
avg_peakrpm=df['peak-rpm'].astype('float').mean(axis=0)
```

print("Average peak rpm:", avg_peakrpm)

df['peak-rpm'].replace(np.nan, avg_peakrpm,
inplace=True)

To see which values are present in a particular column, we can use the ".value_counts()" method

df['num-of-doors'].value_counts()

```
#### We can also use the ".idxmax()" method to
calculate the most common type automatically
```

```
df['num-of-doors'].value_counts().idxmax()
```

#replace the missing 'num-of-doors' values by the most frequent

```
df["num-of-doors"].replace(np.nan, "four", inplace=True)
# Finally, let's drop all rows that do not have price
data
# simply drop whole row with NaN in "price" column
df.dropna(subset=["price"], axis=0, inplace=True)
# reset index, because we droped two rows
df reset index(drem=True, inplace=True)
```

Data Formatting

The last step in data cleaning is checking and making sure that all data is in the correct format (int, float, text or other). In Pandas, we use:

.dtype() to check the data type .astype() to change the data type

```
#### Let's list the data types for each column
df.dtypes
#### Convert data types to proper format
df[["bore", "stroke"]] = df[["bore",
"stroke"]].astype("float")
df[["normalized-losses"]] = df[["normalized-
losses"]].astype("int")
```

df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")

Data Standardization

Data is usually collected from different agencies in different formats. (Data standardization is also a term for a particular type of data normalization where we subtract the mean and divide by the standard deviation.)

Standardization is the process of transforming data into a common format, allowing the researcher to make the meaningful comparison.

Transform mpg to L/100km

In our dataset, the fuel consumption columns "city-mpg" and "highway-mpg" are represented by mpg (miles per gallon) unit. Assume we are developing an application in a country that accepts the fuel consumption with L/100km standard.

We will need to apply **data transformation** to transform mpg into L/100km.

The formula for unit conversion is:

L/100km = 235 / mpg

We can do many mathematical operations directly in Pandas.

```
# Convert mpg to L/100km by mathematical operation (235
divided by mpg)
df['city-L/100km'] = 235/df["city-mpg"]
# rename column name from "city-mpg" to "city-L/100km"
df.rename(columns={'"city-mpg"':'city-L/100km'},
inplace=True)
# check your transformed data
df.head()
```

Data Normalization

Normalization is the process of transforming values of several variables into a similar range. Typical normalizations include scaling the variable so the variable average is 0, scaling the variable so the variance is 1, or scaling the variable so the variable values range from 0 to 1.

Example

To demonstrate normalization, let's say we want to scale the columns "length", "width" and "height".

Target: would like to normalize those variables so their value ranges from 0 to 1

Approach: replace original value by (original value)/(maximum value)

```
# replace (original value) by (original value)/(maximum
value)
df['length'] = df['length']/df['length'].max()
df['width'] = df['width']/df['width'].max()
```

Data Binning

Binning is a process of transforming continuous numerical variables into discrete categorical 'bins' for grouped analysis.

In our dataset, "horsepower" is a real valued variable ranging from 48 to 288 and it has 57 unique values. What if we only care about the price difference between cars with high horsepower, medium horsepower, and little horsepower (3 types)? Can we rearrange them into three 'bins' to simplify analysis?

We will use the pandas method 'cut' to segment the 'horsepower' column into 3 bins.

```
# Convert data to correct format
df["horsepower"]=df["horsepower"].astype(int, copy=True)
# plot the histogram of horsepower to see what the
distribution of horsepower
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
```

plt.pyplot.hist(df["horsepower"])

set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")

We would like 3 bins of equal size bandwidth so we use numpy's `linspace(start_value, end_value, numbers_generated` function.

Since we want to include the minimum value of horsepower, we want to set start_value = min(df["horsepower"]).

Since we want to include the maximum value of horsepower, we want to set end_value = max(df["horsepower"]).

Since we are building 3 bins of equal length, there should be 4 dividers, so numbers_generated = 4.

We build a bin array with a minimum value to a maximum value by using the bandwidth calculated above. The values will determine when one bin ends and another begins

```
bins = np.linspace(min(df["horsepower"]),
max(df["horsepower"]), 4)
```

```
group_names = ['Low', 'Medium', 'High']
```

```
# apply the function "cut" to determine what each value
of `df['horsepower']` belongs to
```

```
df['horsepower-binned'] = pd.cut(df['horsepower'], bins,
labels=group_names, include_lowest=True )
```

df[['horsepower', 'horsepower-binned']].head(20)

```
# the number of vehicles in each bin
df["horsepower-binned"].value_counts()
```

plot the distribution of each bin

%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
pyplot.bar(group_names, df["horsepowerbinned"].value_counts())

```
# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

Complete Code

```
import pandas as pd
import matplotlib.pylab as plt
import numpy as np
```

filename = "https://cf-courses-data.s3.us.cloud-objectstorage.appdomain.cloud/IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/auto.csv"

```
headers = ["symboling","normalized-losses","make","fuel-
type","aspiration", "num-of-doors","body-style",
                "drive-wheels","engine-location","wheel-base",
"length","width","height","curb-weight","engine-type",
                "num-of-cylinders", "engine-size","fuel-
system","bore","stroke","compression-
ratio","horsepower",
                "engine-size","height","engine-type",
                "assisted as a stroke","compression-
ratio","horsepower",
```

```
df = pd.read_csv(filename, names = headers)
```

df.head()

Convert "?" to NaN (Not a Number)
replace "?" to NaN

```
df.replace("?", np.nan, inplace = True)
df.head(5)
```

The missing values are converted by default. We use the following functions to identify these missing values. There are two methods to detect missing data:

Whole columns should be dropped only if most entries in the column are empty. In our dataset, none of the columns are empty enough to drop entirely. We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns

Replace by mean

"normalized-losses": 41 missing data, replace them with mean ### "stroke": 4 missing data, replace them with mean ### "bore": 4 missing data, replace them with mean ### "horsepower": 2 missing data, replace them with mean ### "peak-rpm": 2 missing data, replace them with mean

Replace by frequency

"num-of-doors": 2 missing data, replace them with "four". ### Reason: 84% sedans is four doors. Since four doors is most frequent, it is most likely to occur

Drop the whole row

"price": 4 missing data, simply delete the whole row ### Reason: price is what we want to predict. Any data entry without price data cannot be used for prediction; therefore any row now without price data is not useful to us

Calculate the mean value for the "normalizedlosses" column

```
avg_norm_loss = df["normalized-
losses"].astype("float").mean(axis=0)
```

```
print("Average of normalized-losses:", avg_norm_loss)
```

Replace "NaN" with mean value in "normalizedlosses" column

```
df["normalized-losses"].replace(np.nan, avg_norm_loss,
inplace=True)
```

Calculate the mean value for the "bore" column

avg_bore=df['bore'].astype('float').mean(axis=0)

```
print("Average of bore:", avg_bore)
```

df["bore"].replace(np.nan, avg_bore, inplace=True)

avg_stroke=df["stroke"].astype("float").mean(axis=0)

```
print("Average of stroke:", avg_stroke)
```

df["stroke"].replace(np.nan, avg_stroke, inplace=True)

Calculate the mean value for the "horsepower" column

```
avg_horsepower =
df['horsepower'].astype('float').mean(axis=0)
```

print("Average horsepower:", avg_horsepower)

df['horsepower'].replace(np.nan, avg_horsepower,
inplace=True)

Calculate the mean value for "peak-rpm" column

avg_peakrpm=df['peak-rpm'].astype('float').mean(axis=0)

print("Average peak rpm:", avg_peakrpm)

df['peak-rpm'].replace(np.nan, avg_peakrpm, inplace=True)

To see which values are present in a particular column, we can use the ".value_counts()" method

df['num-of-doors'].value_counts()

We can also use the ".idxmax()" method to calculate the most common type automatically

df['num-of-doors'].value_counts().idxmax()

#replace the missing 'num-of-doors' values by the most
frequent

df["num-of-doors"].replace(np.nan, "four", inplace=True)

Finally, let's drop all rows that do not have price
data

simply drop whole row with NaN in "price" column
df.dropna(subset=["price"], axis=0, inplace=True)

reset index, because we droped two rows
df.reset_index(drop=True, inplace=True)

Let's list the data types for each column
df.dtypes

Convert data types to proper format df[["bore", "stroke"]] = df[["bore", "stroke"]].astype("float")

df[["normalized-losses"]] = df[["normalizedlosses"]].astype("int")

```
df[["price"]] = df[["price"]].astype("float")
```

df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")

Convert mpg to L/100km by mathematical operation (235)

```
divided by mpg)
```

rename column name from "city-mpg" to "city-L/100km"

```
df.rename(columns={'"city-mpg"':'city-L/100km'},
inplace=True)
```

check your transformed data
df.head()

replace (original value) by (original value)/(maximum value)

```
df['length'] = df['length']/df['length'].max()
```

```
df['width'] = df['width']/df['width'].max()
```

Convert data to correct format
df["horsepower"].astype(int, copy=True)

plot the histogram of horsepower to see what the distribution of horsepower

```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
plt.pyplot.hist(df["horsepower"])
```

set x/y labels and plot title

```
pyplot.bar(group_names, df["horsepower-
binned"].value_counts())
```

Bins Visualization with Histogram



What is an indicator variable?

An indicator variable (or dummy variable) is a numerical variable used to label categories. They are called 'dummies' because the numbers themselves don't have inherent meaning.

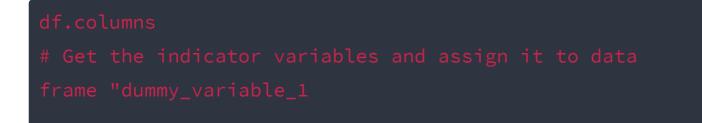
Why we use indicator variables?

We use indicator variables so we can use categorical variables for regression analysis in the later modules.

Example

We see the column "fuel-type" has two unique values: "gas" or "diesel". Regression doesn't understand words, only numbers. To use this attribute in regression analysis, we convert "fuel-type" to indicator variables.

We will use pandas' method 'get_dummies' to assign numerical values to different categories of fuel type



Code Updated

```
import pandas as pd
import matplotlib.pylab as plt
import numpy as np
filename = "https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
DA0101EN-SkillsNetwork/labs/Data%20files/auto.csv"
```

The missing values are converted by default. We use the following functions to identify these missing values. There are two methods to detect missing data:

```
### 1. **.isnull()**
### 2. **.notnull()**
```

The output is a boolean value indicating whether the value that is passed into the argument is in fact missing data.

missing_data = df.isnull()

```
missing_data.head(5)
```

Count missing values in each column
In the body of the for loop the method
".value_counts()" counts the number of "True" values

for column in missing_data.columns.values.tolist():
 print(column)
 print (missing_data[column].value_counts())
 print("")

Deal with missing data
How to deal with missing data?

Drop data
a. Drop the whole row
b. Drop the whole column
Replace data
a. Replace it by mean
b. Replace it by frequency
c. Replace it based on other functions

Whole columns should be dropped only if most entries in the column are empty. In our dataset, none of the columns are empty enough to drop entirely. We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns

Replace by mean

"normalized-losses": 41 missing data, replace them with mean

"stroke": 4 missing data, replace them with mean
"bore": 4 missing data, replace them with mean
"horsepower": 2 missing data, replace them with mean
"peak-rpm": 2 missing data, replace them with mean

Replace by frequency

"num-of-doors": 2 missing data, replace them with "four". ### Reason: 84% sedans is four doors. Since four doors is most frequent, it is most likely to occur

Drop the whole row

"price": 4 missing data, simply delete the whole row ### Reason: price is what we want to predict. Any data entry without price data cannot be used for prediction; therefore any row now without price data is not useful to us

Calculate the mean value for the "normalizedlosses" column

```
avg_norm_loss = df["normalized-
losses"].astype("float").mean(axis=0)
```

print("Average of normalized-losses:", avg_norm_loss)

Replace "NaN" with mean value in "normalizedlosses" column

df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)

Calculate the mean value for the "bore" column

avg_bore=df['bore'].astype('float').mean(axis=0)

print("Average of bore:", avg_bore)

df["bore"].replace(np.nan, avg_bore, inplace=True)

avg_stroke=df["stroke"].astype("float").mean(axis=0)

print("Average of stroke:", avg_stroke)

df["stroke"].replace(np.nan, avg_stroke, inplace=True)

Calculate the mean value for the "horsepower" column

```
avg_horsepower =
df['horsepower'].astype('float').mean(axis=0)
```

print("Average horsepower:", avg_horsepower)

df['horsepower'].replace(np.nan, avg_horsepower,
inplace=True)

Calculate the mean value for "peak-rpm" column

```
avg_peakrpm=df['peak-rpm'].astype('float').mean(axis=0)
```

print("Average peak rpm:", avg_peakrpm)

df['peak-rpm'].replace(np.nan, avg_peakrpm,
inplace=True)

To see which values are present in a particular column, we can use the ".value_counts()" method

df['num-of-doors'].value_counts()

We can also use the ".idxmax()" method to calculate the most common type automatically

df['num-of-doors'].value_counts().idxmax()

#replace the missing 'num-of-doors' values by the most
frequent

df["num-of-doors"].replace(np.nan, "four", inplace=True)

Finally, let's drop all rows that do not have price
data

check your transformed data

df.head()

replace (original value) by (original value)/(maximum
value)

```
df['length'] = df['length']/df['length'].max()
```

```
df['width'] = df['width']/df['width'].max()
```

```
# Convert data to correct format
df["horsepower"].astype(int, copy=True)
```

plot the histogram of horsepower to see what the distribution of horsepower

```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
plt.pyplot.hist(df["horsepower"])
```

```
# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

```
bins = np.linspace(min(df["horsepower"]),
max(df["horsepower"]), 4)
```

```
group_names = ['Low', 'Medium', 'High']
```

apply the function "cut" to determine what each value
of `df['horsepower']` belongs to

```
df['horsepower-binned'] = pd.cut(df['horsepower'], bins,
labels=group_names, include_lowest=True )
```

df[['horsepower', 'horsepower-binned']].head(20)

```
# the number of vehicles in each bin
df["horsepower-binned"].value_counts()
```

plot the distribution of each bin

%matplotlib inline import matplotlib as plt from matplotlib import pyplot pyplot.bar(group_names, df["horsepowerbinned"].value_counts())

Bins Visualization with Histogram

```
plt.pyplot.hist(df["horsepower"], bins = 3)
```

set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")

df.columns

```
Exploratory Data Analysis
```

mport matplotlוb.pyplot as p

import seaborn as sns

%matplotlib inline

```
path='https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
DA0101EN-
SkillsNetwork/labs/Data%20files/automobileEDA.csv'
```

```
df = pd.read_csv(path)
df.head()
```

we can calculate the correlation between variables of type "int64" or "float64" using the method "corr

df.corr()

The correlation between the following columns: bore, stroke, compression-ratio, and horsepower

```
df[['bore','stroke','compression-
ratio','horsepower']].corr()
```

Continuous numerical variables are variables that may contain any value within some range. They can be of type "int64" or "float64". A great way to visualize these variables is by using scatterplots with fitted lines

In order to start understanding the (linear) relationship between an individual variable and the price, we can use "regplot" which plots the scatterplot plus the fitted regression line for the data. # Let's find the scatterplot of "engine-size" and "price"

Engine size as potential predictor variable of price

```
sns.regplot(x="engine-size", y="price", data=df)
plt.ylim(0,)
```

As the engine-size goes up, the price goes up: this indicates a positive direct correlation between these two variables. Engine size seems like a pretty good predictor of price since the regression line is almost a perfect diagonal line

We can examine the correlation between 'engine-size' and 'price' and see that it's approximately 0.87.

df[["engine-size", "price"]].corr()

Highway mpg is a potential predictor variable of
price. Let's find the scatterplot of "highway-mpg" and
"price".

```
sns.regplot(x="highway-mpg", y="price", data=df)
```

As highway-mpg goes up, the price goes down: this indicates an inverse/negative relationship between these two variables. Highway mpg could potentially be a

```
predictor of price.
```

```
df[['highway-mpg', 'price']].corr()
```

Weak Linear Relationship

Let's see if "peak-rpm" is a predictor variable of "price".

```
sns.regplot(x="peak-rpm", y="price", data=df)
```

Peak rpm does not seem like a good predictor of the price at all since the regression line is close to horizontal. Also, the data points are very scattered and far from the fitted line, showing lots of variability. Therefore, it's not a reliable variable.

We can examine the correlation between 'peak-rpm'
and 'price' and see it's approximately -0.101616.

```
df[['peak-rpm','price']].corr()
```

Categorical Variables

These are variables that describe a 'characteristic' of a data unit, and are selected from a small group of categories. The categorical variables can have the type "object" or "int64". A good way to visualize categorical variables is by using boxplots

sns.boxplot(x="body-style", y="price", data=df)

We see that the distributions of price between the different body-style categories have a significant overlap, so body-style would not be a good predictor of price. Let's examine engine "engine-location" and "price":

```
sns.boxplot(x="engine-location", y="price", data=df)
```

Here we see that the distribution of price between these two engine-location categories, front and rear, are distinct enough to take engine-location as a potential good predictor of price.

```
# drive-wheels
sns.boxplot(x="drive-wheels", y="price", data=df)
```

Descriptive Statistical Analysis

Let's first take a look at the variables by utilizing
a description method.

The **describe** function automatically computes
basic statistics for all continuous variables. Any NaN
values are automatically skipped in these statistics.

This will show:

| ## the IQR (Interquartile Range: 25%, 50% and 75%) |
|--|
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| units of each characteristic/variable we have We can |

units of each characteristic/variable we have. We can apply the "value_counts" method on the column "drivewheels". Don't forget the method "value_counts" only works on pandas series, not pandas dataframes. As a result, we only include one bracket `df['drivewheels']`, not two brackets `df[['drive-wheels']]`

df['drive-wheels'].value_counts()

We can convert the series to a dataframe as follows:

df['drive-wheels'].value_counts().to_frame()

```
## Let's repeat the above steps but save the results to
the dataframe "drive_wheels_counts" and rename the
column 'drive-wheels' to 'value_counts'
drive_wheels_counts = df['drive-
wheels'].value_counts().to_frame()
drive_wheels_counts.rename(columns={'drive-wheels':
    'value_counts'}, inplace=True)
drive_wheels_counts.index.name = 'drive-wheels'
```

drive_wheets_counts

Basics of Grouping

The "groupby" method groups data by different categories. The data is grouped based on one or several variables, and analysis is performed on the individual groups.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
path='https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
DA0101EN-
```

SkillsNetwork/labs/Data%20files/automobileEDA.csv'

```
df = pd.read_csv(path)
df.head()
df['drive-wheels'].unique()
```

If we want to know, on average, which type of drive wheel is most valuable, we can group "drive-wheels" and then average them.

We can select the columns 'drive-wheels', 'bodystyle' and 'price', then assign it to the variable "df_group_one".

df_group_one = df[['drive-wheels','body-style','price']]
grouping results

df_group_one = df_group_one.groupby(['drivewheels'],as_index=False).mean()

df_group_one

You can also group by multiple variables. For example, let's group by both 'drive-wheels' and 'bodystyle'. This groups the dataframe by the unique combination of 'drive-wheels' and 'body-style'. We can store the results in the variable 'grouped_test1'.

grouping results df_gptest = df[['drive-wheels','body-style','price']]

```
grouped_test1 = df_gptest.groupby(['drive-wheels','body-
style'],as_index=False).mean()
```

grouped_test1

A pivot table is like an Excel spreadsheet, with one variable along the column and another along the row. We can convert the dataframe to a pivot table using the method "pivot" to create a pivot table from the groups

```
grouped_pivot = grouped_test1.pivot(index='drive-
wheels',columns='body-style')
```

grouped_pivot

```
## fill missing values with 0
```

```
grouped_pivot = grouped_pivot.fillna(0)
grouped_pivot
```

```
## Use the "groupby" function to find the average
"price" of each car based on "body-style".
```

```
df_gpprice = df[['body-style','price']]
```

```
grouped_price = df_gpprice.groupby(['body-
style'],as_index=False).mean()
```

grouped_price

Let's use a heat map to visualize the relationship between Body Style vs Price

```
## use the grouped results
plt.pcolor(grouped_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
```

The heatmap plots the target variable (price) proportional to colour with respect to the variables 'drive-wheel' and 'body-style' on the vertical and horizontal axis, respectively. This allows us to visualize how the price is related to 'drive-wheel' and 'body-style'.

The default labels convey no useful information to us. Let's change that

```
fig, ax = plt.subplots()
im = ax.pcolor(grouped_pivot, cmap='RdBu')
```

```
#label names
row_labels = grouped_pivot.columns.levels[1]
col_labels = grouped_pivot.index
```

```
#move ticks and labels to the center
ax.set_xticks(np.arange(grouped_pivot.shape[1]) + 0.5,
minor=False)
```

```
ax.set_yticks(np.arange(grouped_pivot.shape[0]) + 0.5,
minor=False)
#insert labels
ax.set_xticklabels(row_labels, minor=False)
ax.set_yticklabels(col_labels, minor=False)
## rotate label if too long
plt.xticks(rotation=90)
fig.colorbar(im)
plt.show()
```

5. Correlation and Causation

Correlation: a measure of the extent of interdependence between variables.

Causation: the relationship between cause and effect between two variables.

It is important to know the difference between these two. Correlation does not imply causation. Determining correlation is much simpler the determining causation as causation may require independent experimentation.

Pearson Correlation

The Pearson Correlation measures the linear dependence between two variables X and Y.

The resulting coefficient is a value between -1 and 1 inclusive, where:

- 1: Perfect positive linear correlation.
- **0**: No linear correlation, the two variables most likely do not affect each other.
- -1: Perfect negative linear correlation.

Pearson Correlation is the default method of the function "corr". Like before, we can calculate the Pearson Correlation of the of the 'int64' or 'float64' variables. [df.corr()]

P-value

What is this P-value? The P-value is the probability value that the correlation between these two variables is statistically significant. Normally, we choose a significance level of 0.05, which means that we are 95% confident that the correlation between the variables is significant.

By convention, when the

- p-value is << 0.001: we say there is strong evidence that the correlation is significant.
- the p-value is << 0.05: there is moderate evidence that the correlation is significant.
- the p-value is << 0.1: there is weak evidence that the correlation is significant.
- the p-value is >> 0.1: there is no evidence that the correlation is significant.

We can obtain this information using "stats" module in the "scipy" library

```
from scipy import stats
```

Let's calculate the Pearson Correlation Coefficient
and P-value of 'wheel-base' and 'price

```
pearson_coef, p_value = stats.pearsonr(df['wheel-base'],
df['price'])
```

```
print("The Pearson Correlation Coefficient is",
pearson_coef, " with a P-value of P =", p_value)
```

Let's calculate the Pearson Correlation Coefficient
and P-value of 'horsepower' and 'price

```
pearson_coef, p_value = stats.pearsonr(df['horsepower'],
df['price'])
```

print("The Pearson Correlation Coefficient is",
pearson_coef, " with a P-value of P = ", p_value)

ANOVA : Analysis of Variance

The Analysis of Variance (ANOVA) is a statistical method used to test whether there are significant differences between the means of two or more groups. ANOVA returns two parameters:

F-test score: ANOVA assumes the means of all groups are the same, calculates how much the actual means deviate from

the assumption, and reports it as the F-test score. A larger score means there is a larger difference between the means.

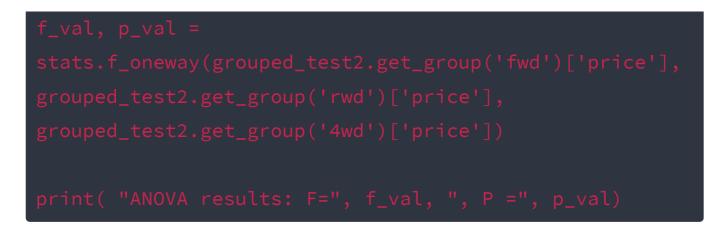
P-value: P-value tells how statistically significant our calculated score value is.

If our price variable is strongly correlated with the variable we are analyzing, we expect ANOVA to return a sizeable F-test score and a small p-value

Since ANOVA analyzes the difference between different groups of the same variable, the groupby function will come in handy. Because the ANOVA algorithm averages the data automatically, we do not need to take the average before hand

To see if different types of 'drive-wheels' impact 'price', we group the data.

```
grouped_test2=df_gptest[['drive-wheels',
 'price']].groupby(['drive-wheels'])
grouped_test2.head(2)
## We can obtain the values of the method group using
the method "get_group
grouped_test2.get_group('4wd')['price']
#### We can use the function 'f_oneway' in the module
```



We now have a better idea of what our data looks like and which variables are important to take into account when predicting the car price. We have narrowed it down to the following variables:

Continuous numerical variables:

- Length
- Width
- Curb-weight
- Engine-size
- Horsepower
- City-mpg
- Highway-mpg
- Wheel-base
- Bore

Categorical variables:

• Drive-wheels

As we now move into building machine learning models to automate our analysis, feeding the model with variables that meaningfully affect our target variable will improve our model's prediction performance.

Model Development

In data analytics, we often use **Model Development** to help us predict future observations from the data we have.

A model will help us understand the exact relationship between different variables and how these variables are used to predict the result.

Simple Linear Regression is a method to help us understand the relationship between two variables:

- The predictor/independent variable (X)
- The response/dependent variable (that we want to predict)(Y)

The result of Linear Regression is a **linear function** that predicts the response (dependent) variable as a function of the predictor (independent) variable.

Y:Response VariableX:Predictor Variables Linear Function

Yhat = a + bX

- a refers to the intercept of the regression line, in other words: the value of Y when X is 0
- b refers to the slope of the regression line, in other words: the value with which Y changes when X increases by 1 unit

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
# path of data
```

```
path = 'https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
DA0101EN-
```

SkillsNetwork/labs/Data%20files/automobileEDA.csv'

```
df = pd.read_csv(path)
df.head()
```

```
#### Create the linear regression object
lm = LinearRegression()
```

lm

```
#### How could "highway-mpg" help us predict car price?
```

```
#### For this example, we want to look at how highway-
mpg can help us predict car price. Using simple linear
regression, we will create a linear function with
"highway-mpg" as the predictor variable and the "price"
as the response variable
```

```
X = df[['highway-mpg']]
Y = df['price']
```

```
lm.fit(X,Y)
```

Multiple Linear Regression

What if we want to predict car price using more than one variable?

If we want to use more variables in our model to predict car price, we can use **Multiple Linear Regression**. Multiple Linear Regression is very similar to Simple Linear Regression, but this method is used to explain the relationship between one continuous response (dependent) variable and **two or more** predictor (independent) variables. Most of the real-world regression models involve multiple predictors. We will illustrate the structure by using four predictor variables, but these results can generalize to any integer:

Y:Response Variable X_1:Predictor Variable 1 X_2:Predictor Variable 2 X_3:Predictor Variable 3 X_4:Predictor Variable 4

a:intercept b_1:coefficients of Variable 1 b_2:coefficients of Variable 2 b_3:coefficients of Variable 3 b_4:coefficients of Variable 4

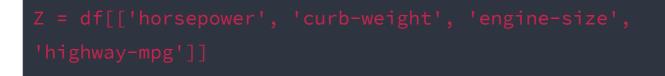
The equation is given by:

 $Yhat = a + b_1X_1 + b_2X_2 + b_3X_3 + b_4X_4$

From the previous section we know that other good predictors of price could be:

- Horsepower
- Curb-weight
- Engine-size
- Highway-mpg

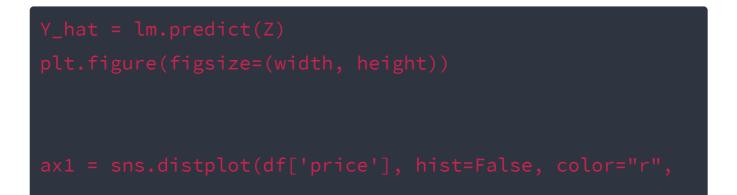
Let's develop a model using these variables as the predictor variables.

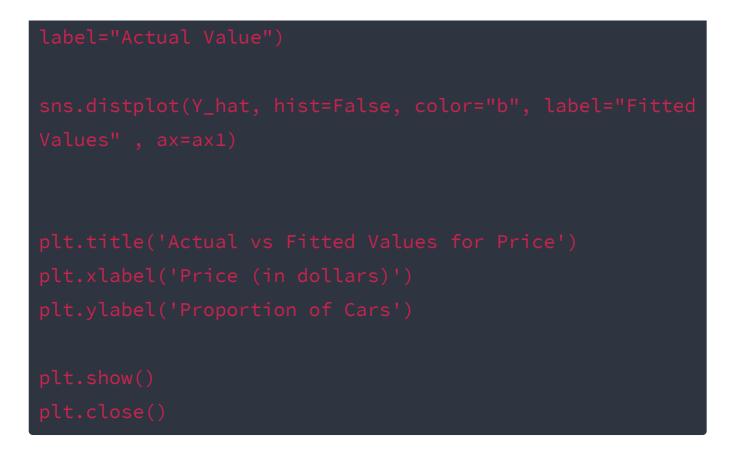


```
#### Fit the linear model using the four above-mentioned
variables.
lm.fit(Z, df['price'])
lm.intercept_
lm.coef_
#### **Price** = -15678.742628061467 + 52.65851272 x
**horsepower** + 4.69878948 x **curb-weight** +
81.95906216 x **engine-size** + 33.58258185 x **highway-
mpg**
```

How do we visualize a model for Multiple Linear Regression? This gets a bit more complicated because you can't visualize it with regression or residual plot.

One way to look at the fit of the model is by looking at the **distribution plot**. We can look at the distribution of the fitted values that result from the model and compare it to the distribution of the actual values.





Model Evaluation Using Visualization

Now that we've developed some models, how do we evaluate our models and choose the best one? One way to do this is by using a visualization.

When it comes to simple linear regression, an excellent way to visualize the fit of our model is by using **regression plots**.

This plot will show a combination of a scattered data points (a **scatterplot**), as well as the fitted **linear regression** line going through the data. This will give us a reasonable estimate of the relationship between the two variables, the strength of the correlation, as well as the direction (positive or negative correlation).

Let's visualize **highway-mpg** as potential predictor variable of price

import the visualization package: seaborn
import seaborn as sns
%matplotlib inline

```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.regplot(x="highway-mpg", y="price", data=df)
plt.ylim(0,)
```

We can see from this plot that price is negatively correlated to highway-mpg since the regression slope is negative.

One thing to keep in mind when looking at a regression plot is to pay attention to how scattered the data points are around the regression line. This will give you a good indication of the variance of the data and whether a linear model would be the best fit or not. If the data is too far off from the line, this linear model might not be the best model for this data.

```
### Let's compare this plot to the regression plot of
"peak-rpm".
```

```
plt.figure(figsize=(width, height))
sns.regplot(x="peak-rpm", y="price", data=df)
plt.ylim(0,)
```

Residual Plot

A good way to visualize the variance of the data is to use a residual plot.

What is a **residual**?

The difference between the observed value (y) and the predicted value (Yhat) is called the residual (e). When we look at a regression plot, the residual is the distance from the data point to the fitted regression line.

So what is a **residual plot**?

A residual plot is a graph that shows the residuals on the vertical y-axis and the independent variable on the horizontal x-axis.

What do we pay attention to when looking at a residual plot?

We look at the spread of the residuals:

 If the points in a residual plot are randomly spread out around the x-axis, then a linear model is appropriate for the data.

Why is that? Randomly spread out residuals means that the variance is constant, and thus the linear model is a good fit for this data.

```
width = 12
height = 10
plt.figure(figsize=(width, height))
```

Polynomial Regression and Pipelines

Polynomial regression is a particular case of the general linear regression model or multiple linear regression models.

We get non-linear relationships by squaring or setting higherorder terms of the predictor variables.

There are different orders of polynomial regression:

Quadratic - 2nd Order

Yhat = a + b1X + b2X2Yhat = a + b1X + b2X2

Cubic - 3rd Order

Yhat=*a*+*b*1*X*+*b*2*X*2+*b*3*X*3Yhat=*a*+*b*1X+*b*2X2+*b*3X3

Higher-Order:

Y = a + b1X + b2X2 + b3X3....Y = a + b1X + b2X2 + b3X3....

We saw earlier that a linear model did not provide the best fit while using "highway-mpg" as the predictor variable. Let's see if we can try fitting a polynomial model to the data instead.

We will use the following function to plot the data:

```
def PlotPolly(model, independent_variable,
  dependent_variabble, Name):
    x_new = np.linspace(15, 55, 100)
```

```
y_new = model(x_new)
```

```
plt.plot(independent_variable, dependent_variabble,
'.', x_new, y_new, '-')
plt.title('Polynomial Fit with Matplotlib for Price
~ Length')
    ax = plt.gca()
    ax.set_facecolor((0.898, 0.898, 0.898))
    fig = plt.gcf()
    plt.xlabel(Name)
    plt.ylabel('Price of Cars')
    plt.show()
    plt.close()
x = df['highway-mpg']
y = df['price']
```

Let's fit the polynomial using the function
polyfit, then use the function **poly1d** to display
the polynomial function.

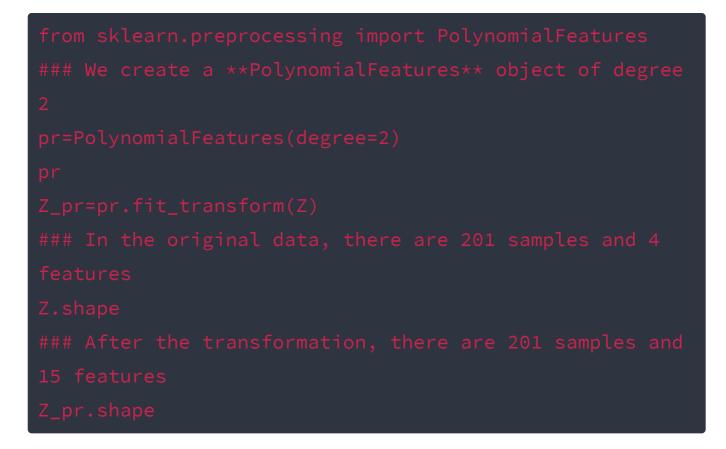
```
# Here we use a polynomial of the 3rd order (cubic)
f = np.polyfit(x, y, 3)
p = np.poly1d(f)
print(p)
### plot the function
PlotPolly(p, x, y, 'highway-mpg')
np.polyfit(x, y, 3)
```

Polynomial Regression with more than one predictor variable

The analytical expression for Multivariate Polynomial function gets complicated. For example, the expression for a second-order (degree=2) polynomial with two variables is given by:

 $Yhat = a + b_1X_1 + b_2X_2 + b_3X_1X_2 + b_4X_12 + b_5X_22Yhat = a + b_1X_1 + b_2X_2 + b_3X_1X_2 + b_4X_12 + b_5X_22$

We can perform a polynomial transform on multiple features.



Pipeline

Data Pipelines simplify the steps of processing the data. We use the module **Pipeline** to create a pipeline. We also use **StandardScaler** as a step in our pipeline

```
pipe.fit(Z,y)
```

Similarly, we can normalize the data, perform a transform and produce a prediction simultaneously.

Measures for In-Sample Evaluation

When evaluating our models, not only do we want to visualize the results, but we also want a quantitative measure to determine how accurate the model is.

Two very important measures that are often used in Statistics to determine the accuracy of a model are:

- R² / R-squared
- Mean Squared Error (MSE)

R-squared

R squared, also known as the coefficient of determination, is a measure to indicate how close the data is to the fitted regression line.

The value of the R-squared is the percentage of variation of the response variable (y) that is explained by a linear model.

Mean Squared Error (MSE)

The Mean Squared Error measures the average of the squares of errors. That is, the difference between actual value (y) and the estimated value (\hat{y}).

mse = mean_squared_error(df['price'], Yhat)

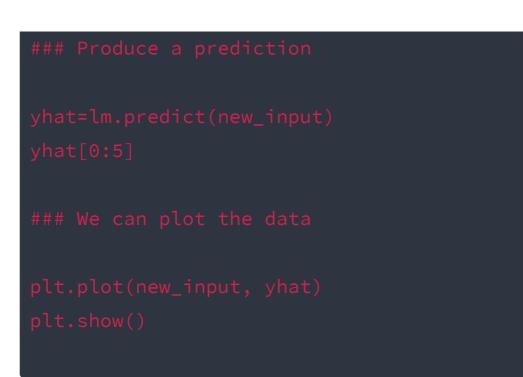
Model 3: Polynomial Fit

```
### We apply the function to get the value of R^2
r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
### We can say that ~67.419 % of the variation of price
is explained by this polynomial fit.
```

```
Prediction and Decision Making
```

In the previous section, we trained the model using the method **fit**. Now we will use the method **predict** to produce a prediction. Lets import **pyplot** for plotting; we will also be using some functions from numpy.

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
### Create a new input
new_input=np.arange(1, 100, 1).reshape(-1, 1)
lm.fit(X, Y)
lm
```



Decision Making: Determining a Good Model Fit

Now that we have visualized the different models, and generated the R-squared and MSE values for the fits, how do we determine a good model fit?

• What is a good R-squared value?

When comparing models, **the model with the higher Rsquared value is a better fit** for the data.

• What is a good MSE?

When comparing models, **the model with the smallest MSE value is a better fit** for the data.

Let's take a look at the values for the different models.

Simple Linear Regression: Using Highway-mpg as a Predictor Variable of Price.

- R-squared: 0.49659118843391759
- MSE: 3.16 ×10^7

Multiple Linear Regression: Using Horsepower, Curb-weight, Engine-size, and Highway-mpg as Predictor Variables of Price.

- R-squared: 0.80896354913783497
- MSE: 1.2 ×10^7

Polynomial Fit: Using Highway-mpg as a Predictor Variable of Price.

- R-squared: 0.6741946663906514
- MSE: 2.05 × 10^7

Simple Linear Regression Model (SLR) vs Multiple Linear Regression Model (MLR)

Usually, the more variables you have, the better your model is at predicting, but this is not always true. Sometimes you may not have enough data, you may run into numerical problems, or many of the variables may not be useful and even act as noise. As a result, you should always check the MSE and R^2.

In order to compare the results of the MLR vs SLR models, we look at a combination of both the R-squared and MSE to make the best conclusion about the fit of the model.

- **MSE**: The MSE of SLR is 3.16×10^7 while MLR has an MSE of 1.2 ×10^7. The MSE of MLR is much smaller.
- R-squared: In this case, we can also see that there is a big difference between the R-squared of the SLR and the Rsquared of the MLR. The R-squared for the SLR (~0.497) is very small compared to the R-squared for the MLR (~0.809).

This R-squared in combination with the MSE show that MLR seems like the better model fit in this case compared to SLR.

Simple Linear Model (SLR) vs. Polynomial Fit

- **MSE**: We can see that Polynomial Fit brought down the MSE, since this MSE is smaller than the one from the SLR.
- **R-squared**: The R-squared for the Polynomial Fit is larger than the R-squared for the SLR, so the Polynomial Fit also brought up the R-squared quite a bit.

Since the Polynomial Fit resulted in a lower MSE and a higher R-squared, we can conclude that this was a better fit model than the simple linear regression for predicting "price" with "highway-mpg" as a predictor variable.

Multiple Linear Regression (MLR) vs. Polynomial Fit

• **MSE**: The MSE for the MLR is smaller than the MSE for the Polynomial Fit.

• **R-squared**: The R-squared for the MLR is also much larger than for the Polynomial Fit.

Conclusion

Comparing these three models, we conclude that **the MLR model is the best model** to be able to predict price from our dataset. This result makes sense since we have 27 variables in total and we know that more than one of those variables are potential predictors of the final car price.

Visualization

The Dataset: Immigration to Canada from 1980 to 2013

```
# useful for many scientific computing in Python
import numpy as np
import pandas as pd
# primary data structure library
df_can = pd.read_excel(
    'https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
DV0101EN-SkillsNetwork/Data%20Files/Canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2)
```

print('Data read into a pandas dataframe!')

df_can.head()

When analyzing a dataset, it's always a good idea to start by getting basic information about your dataframe. We can do this by using the `info()` method.

This method can be used to get a short summary of the dataframe

df_can.info(verbose=False)

To get the list of column headers we can call upon the data frame's `columns` instance variable

df_can.columns

Similarly, to get the list of indices we use the
`.index` instance variables

```
df_can.index
```

To get the index and columns as lists, we can use the `tolist()` method.

```
df_can.columns.tolist()
df_can.index.tolist()
```

To view the dimensions of the dataframe, we use the `shape` instance variable of it

df_can.shape

```
## in pandas axis=0 represents rows (default) and axis=1
represents columns.
```

```
df_can.drop(['AREA','REG','DEV','Type','Coverage'],
axis=1, inplace=True)
df_can.head(2)
```

Let's rename the columns so that they make sense. We can use `rename()` method by passing in a dictionary of old and new names as follows

```
df_can.rename(columns={'OdName':'Country',
    'AreaName':'Continent', 'RegName':'Region'},
    inplace=True)
```

df_can.columns

```
## We will also add a 'Total' column that sums up the
total immigrants by country over the entire period 1980
- 2013, as follows
```

df_can['Total'] = df_can.sum(axis=1)

We can check to see how many null objects we have in the dataset as follows

```
df_can.isnull().sum()
```

Finally, let's view a quick summary of each column in our dataframe using the `describe()` method.

df_can.describe()

There are two ways to filter on a column name:

Method 1: Quick and easy, but only works if the column name does NOT have spaces or special characters.

Method 2: More robust, and can filter on multiple columns

```
df['column']  # returns series
df[['column 1', 'column 2']] # returns dataframe
```

df_can.Country # returns a series

Let's try filtering on the list of countries
('Country') and the data for years: 1980 - 1985

df_can[['Country', 1980, 1981, 1982, 1983, 1984, 1985]] # returns a dataframe

notice that 'Country' is string, and the years are integers. ## There are main 2 ways to select rows:

df.loc[label] # filters by the labels of the
index/column

df.iloc[index] # filters by the positions of the index/column

Before we proceed, notice that the default index of the dataset is a numeric range from 0 to 194. This makes it very difficult to do a query by a specific country. For example to search for data on Japan, we need to know the corresponding index value.

This can be fixed very easily by setting the 'Country' column as the index using `set_index()` method

df_can.set_index('Country', inplace=True)

tip: The opposite of set is reset. So to reset the index, we can use df_can.reset_index()

df_can.head(3)

Example: Let's view the number of immigrants from Japan (row 87) for the following scenarios: 1. The full row data (all columns) 2. For year 2013 3. For years 1980 to 1985.

r_can.columns = list(map(str, dt_can.column

```
## [print (type(x)) for x in df_can.columns.values] #<--
uncomment to check type of column headers</pre>
```

Since we converted the years to string, let's declare
a variable that will allow us to easily call upon the
full range of years

useful for plotting later on
years = list(map(str, range(1980, 2014)))
vears

Filtering based on a criteria

To filter the dataframe based on a condition, we simply pass the condition as a boolean vector.

For example, Let's filter the dataframe to show the data on Asian countries (AreaName = Asia)

1. create the condition boolean series condition = df_can['Continent'] == 'Asia' print(condition)

2. pass this condition into the dataFrame
df_can[condition]

we can pass multiple criteria in the same line.
let's filter for AreaNAme = Asia and RegName =
Southern Asia

```
df_can[(df_can['Continent']=='Asia') &
 (df_can['Region']=='Southern Asia')]
# note: When using 'and' and 'or' operators, pandas
requires we use '&' and '|' instead of 'and' and 'or'
# don't forget to enclose the two conditions in
parentheses
print('data dimensions:', df_can.shape)
print(df_can.columns)
df_can.head(2)
```

Matplotlib.Pyplot

One of the core aspects of Matplotlib is **matplotlib.pyplot**. It is Matplotlib's scripting layer which we studied in details in the videos about Matplotlib. Recall that it is a collection of command style functions that make Matplotlib work like MATLAB. Each **pyplot** function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. In this lab, we will work with the scripting layer to learn how to generate line plots. In future labs, we will get to work with the Artist layer as well to experiment first hand how it differs from the scripting layer.

Let's start by importing matplotlib and matplotlib.pyplot as follows

```
# we are using the inline backend
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
print('Matplotlib version: ', mpl.__version__) # >=
2.0.0
print(plt.style.available)
mpl.style.use(['ggplot']) # optional: for ggplot-like
style
```

Plotting in _pandas

Fortunately, pandas has a built-in implementation of Matplotlib that we can use. Plotting in *pandas* is as simple as appending a .plot() method to a series or dataframe

```
## passing in years 1980 - 2013 to exclude the 'total'
column
haiti = df_can.loc['Haiti', years]
haiti.head()
haiti.plot()
## let's label the x and y axis using `plt.title()`,
`plt.ylabel()`, and `plt.xlabel()` as follows
haiti.index = haiti.index.map(int) # let's change the
index values of Haiti to type integer for plotting
haiti.plot(kind='line')
```

Area Plots, Histograms, and Bar Plots

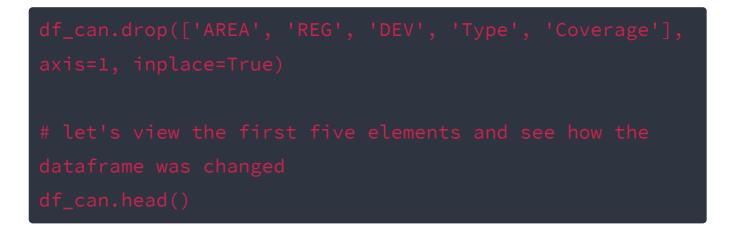
```
import numpy as np
# useful for many scientific computing in Python
import pandas as pd
```

```
# primary data structure tibrary

df_can = pd.read_excel(
    'https://cf-courses-data.s3.us.cloud-object-
    storage.appdomain.cloud/IBMDeveloperSkillsNetwork-
DV0101EN-SkillsNetwork/Data%20Files/Canada.xlsx',
        sheet_name='Canada by Citizenship',
        skiprows=range(20),
        skipfooter=2)
print('Data downloaded and read into a dataframe!')

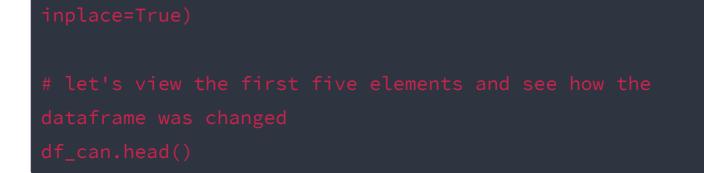
# print the dimensions of the dataframe
print(df_can.shape)
```

Clean up the dataset to remove columns that are not informative to us for visualization (eg. Type, AREA, REG).



Rename some of the columns so that they make sense.

df_can.rename(columns={'OdName':'Country',
 'AreaName':'Continent', 'RegName':'Region'},

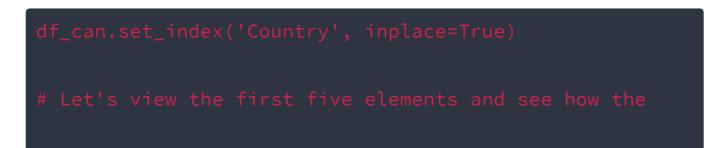


For consistency, ensure that all column labels of type string

```
# let's examine the types of the column labels
all(isinstance(column, str) for column in
df_can.columns)
## Notice how the above line of code returned _False_
when we tested if all the column labels are of type
**string**. So let's change them all to **string** type
df_can.columns = list(map(str, df_can.columns))
# let's check the column labels types now
all(isinstance(column, str) for column in
```

df_can.columns)

Set the country name as index - useful for quickly looking up countries using .loc method



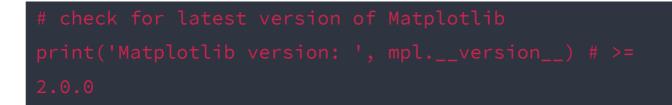
```
dataframe was change
df_can.head()
```

Add total column

```
df_can['Total'] = df_can.sum(axis=1)
# let's view the first five elements and see how the
dataframe was changed
df_can.head()
# finally, let's create a list of years from 1980 - 2013
# this will come in handy when we start plotting the
data
years = list(map(str, range(1980, 2014)))
```

Visualizing Data using Matplotlib

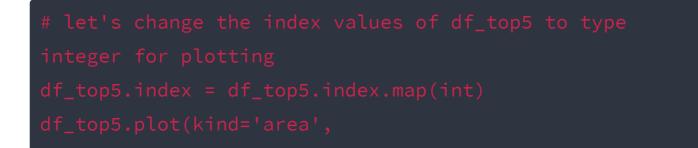
| # use the inline backend to generate the plots within |
|--|
| the browser |
| % matplotlib inline |
| import matplotlib as mpl |
| import matplotlib.pyplot as plt |
| <pre>mpl.style.use('ggplot') # optional: for ggplot-like style</pre> |



In the last module, we created a line plot that visualized the top 5 countries that contribued the most immigrants to Canada from 1980 to 2013. With a little modification to the code, we can visualize this plot as a cumulative plot, also knows as a **Stacked Line Plot** or **Area plot**.

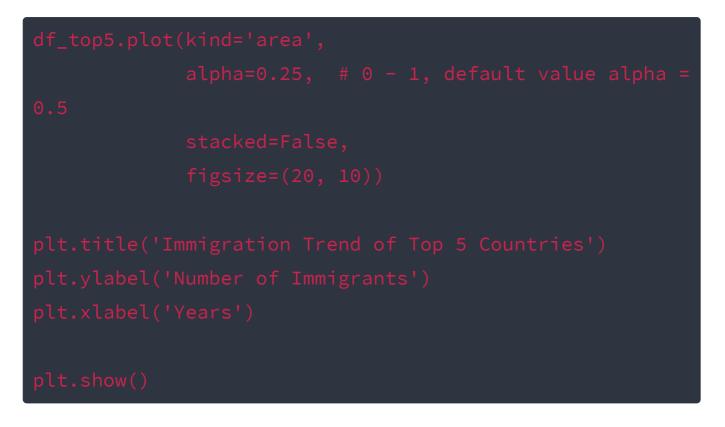
```
df_can.sort_values(['Total'], ascending=False, axis=0,
inplace=True)
# get the top 5 entries
df_top5 = df_can.head()
# transpose the dataframe
df_top5 = df_top5[years].transpose()
df_top5.head()
```

Area plots are stacked by default. And to produce a stacked area plot, each column must be either all positive or all negative values (any NaN, i.e. not a number, values will default to 0). To produce an unstacked plot, set parameter **stacked** to value **False**.





The unstacked plot has a default transparency (alpha value) at 0.5. We can modify this value by passing in the alpha parameter.

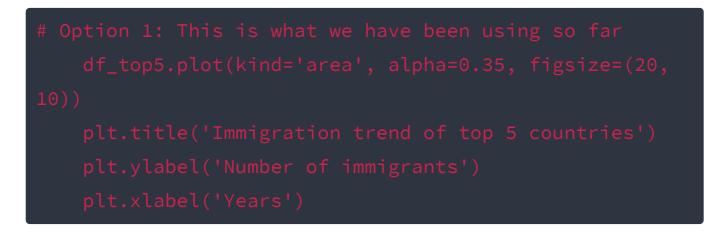


Two types of plotting

As we discussed in the video lectures, there are two styles/options of plotting with matplotlib, plotting using the Artist layer and plotting using the scripting layer.

Option 1: Scripting layer (procedural method) - using matplotlib.pyplot as 'plt'

You can use plt i.e. matplotlib.pyplot and add more elements by calling different methods procedurally; for example, plt.title(...) to add title or plt.xlabel(...) to add label to the x-axis

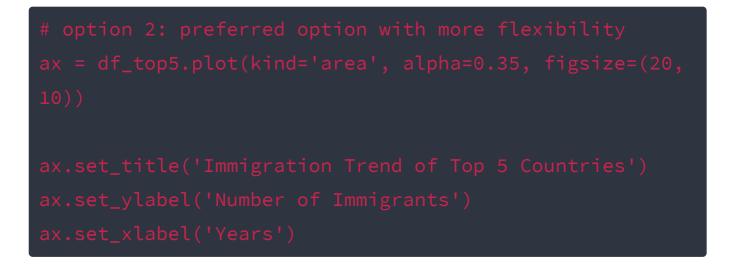


Option 2: Artist layer (Object oriented method) - using an Axes instance from Matplotlib (preferred)

You can use an Axes instance of your current plot and store it in a variable (eg. ax). You can add more elements by calling methods with a little change in syntax (by adding "set_" to the previous methods). For example, use <code>ax.set_title()</code> instead of <code>plt.title()</code> to add title, or <code>ax.set_xlabel()</code> instead of <code>plt.xlabel()</code> to add label to the x-axis.

This option sometimes is more transparent and flexible to use for advanced plots (in particular when having multiple plots, as you will see later).

In this course, we will stick to the **scripting layer**, except for some advanced visualizations where we will need to use the **artist layer** to manipulate advanced aspects of the plots



Use the scripting layer to create a stacked area plot of the 5 countries that contributed the least to immigration to Canada **from** 1980 to 2013. Use a transparency value of 0.45.

| #The correct answer is: |
|---|
| # get the 5 countries with the least contribution |
| df_least5 = df_can.tail(5) |
| <pre># transpose the dataframe</pre> |
| <pre>df_least5 = df_least5[years].transpose()</pre> |
| df_least5.head() |
| <pre>df_least5.index = df_least5.index.map(int) # let's</pre> |
| change the index values of df_least5 to type integer for |
| plotting |
| df_least5.plot(kind='area', alpha=0.45, figsize=(20, |
| 10)) |
| <pre>plt.title('Immigration Trend of 5 Countries with</pre> |
| Least Contribution to Immigration') |
| <pre>plt.ylabel('Number of Immigrants')</pre> |
| <pre>plt.xlabel('Years')</pre> |

plt.show()

Use the artist layer to create an unstacked area plot of the 5 countries that contributed the least to immigration to Canada **from** 1980 to 2013. Use a transparency value of 0.55

| <pre># get the 5 countries with the least contribution df_least5 = df_can.tail(5)</pre> |
|---|
| |
| |
| |
| |
| <pre>ax.set_title('Immigration Trend of 5 Countries with Least Contribution to Immigration') ax.set_ylabel('Number of Immigrants') ax.set_xlabel('Years')</pre> |

Histograms

A histogram is a way of representing the *frequency* distribution of numeric dataset. The way it works is it

partitions the x-axis into *bins*, assigns each data point in our dataset to a bin, and then counts the number of data points that have been assigned to each bin. So the y-axis is the frequency or the number of data points in each bin. Note that we can change the bin size and usually one needs to tweak it so that the distribution is displayed nicely.

Question: What is the frequency distribution of the number (population) of new immigrants from the various countries to Canada in 2013?

Before we proceed with creating the histogram plot, let's first examine the data split into intervals. To do this, we will us **Numpy**'s **histrogram** method to get the bin ranges and frequency counts as follows:

```
# let's quickly view the 2013 data
df_can['2013'].head()
# np.histogram returns 2 values
count, bin_edges = np.histogram(df_can['2013'])
print(count) # frequency count
print(bin_edges) # bin ranges, default = 10 bins
## output:
[178 11 1 2 0 0 0 0 1 2]
[ 0. 3412.9 6825.8 10238.7 13651.6 17064.5 20477.4
23890.3 27303.2
30716.1 34129. ]
```

```
## We can easily graph this distribution by passing
`kind=hist` to `plot()`.
df_can['2013'].plot(kind='hist', figsize=(8, 5))
# add a title to the histogram
plt.title('Histogram of Immigration from 195 Countries
in 2013')
# add y-label
plt.ylabel('Number of Countries')
# add x-label
plt.xlabel('Number of Immigrants')
```

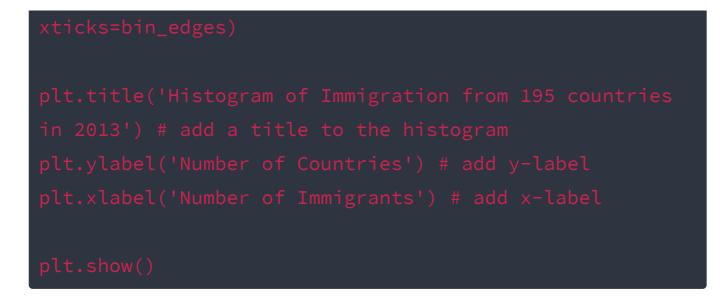
plt.show()

In the above plot, the x-axis represents the population range of immigrants in intervals of 3412.9. The y-axis represents the number of countries that contributed to the aforementioned population.

Notice that the x-axis labels do not match with the bin size. This can be fixed by passing in a `xticks` keyword that contains the list of the bin sizes, as follows

```
# 'bin_edges' is a list of bin intervals
count, bin_edges = np.histogram(df_can['2013'])
```

df_can['2013'].plot(kind='hist', figsize=(8, 5),



We can also plot multiple histograms on the same plot. For example, let's try to answer the following questions using a histogram.

Question: What is the immigration distribution for Denmark, Norway, and Sweden for years 1980 - 2013?

```
# let's quickly view the dataset
df_can.loc[['Denmark', 'Norway', 'Sweden'], years]
# transpose dataframe
df_t = df_can.loc[['Denmark', 'Norway', 'Sweden'],
years].transpose()
df_t.head()
# generate histogram
df_t.plot(kind='hist', figsize=(10, 6))
plt.title('Histogram of Immigration from Denmark,
Norway, and Sweden from 1980 - 2013')
plt.ylabel('Number of Years')
plt.xlabel('Number of Immigrants')
```

plt.show()

Let's make a few modifications to improve the impact and aesthetics of the previous plot:

```
-increase the bin size to 15 by passing in bins parameter;
-set transparency to 60% by passing in alpha parameter;
-label the x-axis by passing in x-label parameter;
-change the colors of the plots by passing in color paramete
```

If we do not want the plots to overlap each other, we can stack them using the **stacked** parameter. Let's also adjust the min and max x-axis labels to remove the extra gap on the edges of the plot. We can pass a tuple (min,max) using the **xlim** paramater, as show below.

plt.show()

Use the scripting layer to display the immigration distribution for Greece, Albania, and Bulgaria for years 1980 - 2013? Use an overlapping plot with 15 bins and a transparency value of 0.35.

```
plt.ylabel('Number of Years')
```

plt.xlabel('Number of Immigrants')

Bar Charts (Dataframe)

A bar plot is a way of representing data where the *length* of the bars represents the magnitude/size of the feature/variable. Bar graphs usually represent numerical and categorical variables grouped in intervals.

To create a bar plot, we can pass one of two arguments via kind parameter in plot():

- kind=bar creates a vertical bar plot
- kind=barh creates a horizontal bar plot

Vertical bar plot

In vertical bar graphs, the x-axis is used for labelling, and the length of bars on the y-axis corresponds to the magnitude of the variable being measured. Vertical bar graphs are particularly useful in analyzing time series data. One disadvantage is that they lack space for text labelling at the foot of each bar.

Let's start off by analyzing the effect of Iceland's Financial Crisis:

The 2008 - 2011 Icelandic Financial Crisis was a major economic and political event in Iceland. Relative to the size of its economy, Iceland's systemic banking collapse was the largest experienced by any country in economic history. The crisis led to a severe economic depression in 2008 - 2011 and significant political unrest.

Let's compare the number of Icelandic immigrants (country = 'Iceland') to Canada from year 1980 to 2013

```
# step 1: get the data
df_iceland = df_can.loc['Iceland', years]
df_iceland.head()
# step 2: plot data
df_iceland.plot(kind='bar', figsize=(10, 6))
plt.xlabel('Year') # add to x-label to the plot
plt.ylabel('Number of immigrants') # add y-label to the
plot
plt.title('Icelandic immigrants to Canada from 1980 to
2013') # add title to the plot
plt.show()
```

Let's annotate this on the plot using the **annotate** method of the **scripting layer** or the **pyplot interface**. We will pass in the following parameters:

- **s**: str, the text of annotation.
- xy: Tuple specifying the (x,y) point to annotate (in this case, end point of arrow).
- xytext: Tuple specifying the (x,y) point to place the text (in this case, start point of arrow).

- xycoords: The coordinate system that xy is given in -'data' uses the coordinate system of the object being annotated (default).
- **arrowprops**: Takes a dictionary of properties to draw the arrow:
 - arrowstyle: Specifies the arrow style, '->' is standard arrow.
 - **connectionstyle**: Specifies the connection type. **arc3** is a straight line.
 - **color**: Specifies color of arrow.
 - Iw: Specifies the line width



Let's also annotate a text to go over the arrow. We will pass in the following additional parameters:

- **rotation**: rotation angle of text in degrees (counter clockwise)
- va: vertical alignment of text ['center' | 'top' | 'bottom' | 'baseline']
- ha: horizontal alignment of text ['center' | 'right' | 'left']

```
df_iceland.plot(kind='bar', figsize=(10, 6), rot=90)
```

system of the object being annotated

Horizontal Bar Plot

Sometimes it is more practical to represent the data horizontally, especially if you need more room for labelling the bars. In horizontal bar graphs, the y-axis is used for labelling, and the length of bars on the x-axis corresponds to the magnitude of the variable being measured. As you will see, there is more room on the y-axis to label categorical variables.

Using the scripting later and the df_can dataset, create a *horizontal* bar plot showing the *total* number of immigrants to

Canada from the top 15 countries, for the period 1980 - 2013. Label each country with the total immigrant count

plt.annotate(label, xy=(value - 47000, index 0.10), color='white')

plt.show()

Pie Charts

A **pie chart** is a circular graphic that displays numeric proportions by dividing a circle (or pie) into proportional slices. You are most likely already familiar with pie charts as it is widely used in business and media. We can create pie charts in Matplotlib by passing in the **kind=pie** keyword.

Let's use a pie chart to explore the proportion (percentage) of new immigrants grouped by continents for the entire time period from 1980 to 2013

Step 1: Gather data.

We will use *pandas* groupby method to summarize the immigration data by **Continent**. The general process of groupby involves the following steps:

- 1. **Split:** Splitting the data into groups based on some criteria.
- 2. **Apply:** Applying a function to each group independently: .sum() .count() .mean() .std() .aggregate() .apply() .etc..
- 3. Combine: Combining the results into a data structure.

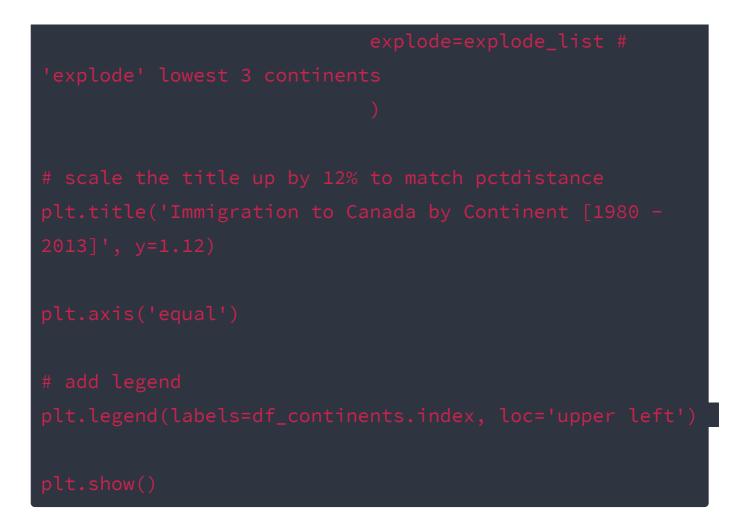
```
# group countries by continents and apply sum() function
df_continents = df_can.groupby('Continent',
axis=0).sum()
# note: the output of the groupby method is a `groupby'
object.
# we can not use it further until we apply a function
(eg .sum())
```



Step 2: Plot the data. We will pass in kind = 'pie' keyword, along with the following additional parameters:

- autopct is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be fmt%pct.
- **startangle** rotates the start of the pie chart by angle degrees counterclockwise from the x-axis.
- shadow Draws a shadow beneath the pie (to give a 3D feel).

- Remove the text labels on the pie chart by passing in legend and add it as a seperate legend using plt.legend().
- Push out the percentages to sit just outside the pie chart by passing in pctdistance parameter.
- Pass in a custom set of colors for continents by passing in colors parameter.
- Explode the pie chart to emphasize the lowest three continents (Africa, North America, and Latin America and Caribbean) by passing in explode parameter.



Using a pie chart, explore the proportion (percentage) of new immigrants grouped by continents in the year 2013

| <pre>explode_list = [0.0, 0, 0, 0.1, 0.1, 0.2] # ratio for each continent with which to offset each wedge.</pre> | | |
|--|--|--|
| <pre>df_continents['2013'].plot(kind='pie',</pre> | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| <pre># turn off labels on pie chart</pre> | | |
| | | |
| # the ratio between the pie cen | | |
| label | | |

Box Plots

A **box plot** is a way of statistically representing the *distribution* of the data through five main dimensions:

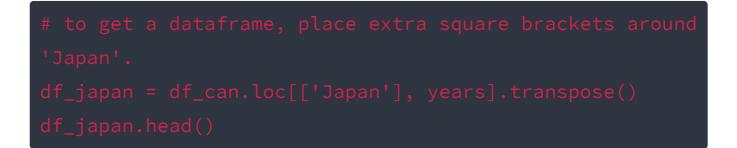
- **Minimum:** The smallest number in the dataset excluding the outliers.
- First quartile: Middle number between the minimum and the median.
- Second quartile (Median): Middle number of the (sorted) dataset.
- Third quartile: Middle number between median and maximum.
- **Maximum:** The largest number in the dataset excluding the outliers.

To make a **boxplot**, we can use **kind=box** in **plot** method invoked on a *pandas* series or dataframe.

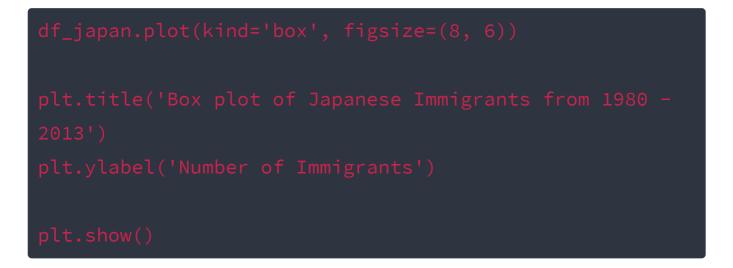
Let's plot the box plot for the Japanese immigrants between 1980 - 2013.

Step 1: Get the subset of the dataset. Even though we are extracting the data for just one country, we will obtain it as a dataframe. This will help us with calling the

dataframe.describe() method to view the percentiles.



Step 2: Plot by passing in kind='box'



One of the key benefits of box plots is comparing the distribution of multiple datasets. In one of the previous labs, we observed that China and India had very similar immigration trends. Let's analyze these two countries further using box plots

Compare the distribution of the number of new immigrants from India and China for the period 1980 - 2013

```
df_CI= df_can.loc[['China', 'India'], years].transpose()
df_CI.head()
df_CI.describe()
df_CI.plot(kind='box', figsize=(10, 7))
    plt.title('Box plots of Immigrants from China and
India (1980 - 2013)')
    plt.ylabel('Number of Immigrants')
    plt.show()
```

If you prefer to create horizontal box plots, you can pass the **vert** parameter in the **plot** function and assign it to *False*. You can also specify a different color in case you are not a big fan of the default red color.

```
# horizontal box plots
df_CI.plot(kind='box', figsize=(10, 7), color='blue',
vert=False)
plt.title('Box plots of Immigrants from China and India
(1980 - 2013)')
plt.xlabel('Number of Immigrants')
plt.show()
```

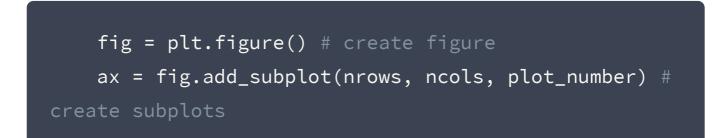
Subplots

Often times we might want to plot multiple plots within the same figure. For example, we might want to perform a side by

side comparison of the box plot with the line plot of China and India's immigration.

To visualize multiple plots together, we can create a **figure** (overall canvas) and divide it into **subplots**, each containing a plot. With **subplots**, we usually work with the **artist layer** instead of the **scripting layer**.

Typical syntax is :



Where

 nrows and ncols are used to notionally split the figure into (nrows * ncols) sub-axes,

 plot_number is used to identify the particular subplot that this function is to create within the notional grid.
 plot_number starts at 1, increments across rows first and

has a maximum of nrows * ncols as shown below.

We can then specify which subplot to place each plot by passing in the ax paramemter in plot() method as follows:

fig = plt.figure() # create figure ax0 = fig.add_subplot(1, 2, 1) # add subplot 1 (1 row, 2 columns, first plot) ax1 = fig.add_subplot(1, 2, 2) # add subplot 2 (1 row, 2

```
columns, second plot). See tip below**
```

```
# Subplot 1: Box plot
df_CI.plot(kind='box', color='blue', vert=False,
figsize=(20, 6), ax=ax0) # add to subplot 1
ax0.set_title('Box Plots of Immigrants from China and
India (1980 - 2013)')
ax0.set_xlabel('Number of Immigrants')
ax0.set_ylabel('Countries')
# Subplot 2: Line plot
df_CI.plot(kind='line', figsize=(20, 6), ax=ax1) # add
to subplot 2
ax1.set_title ('Line Plots of Immigrants from China and
India (1980 - 2013)')
ax1.set_ylabel('Number of Immigrants')
ax1.set_xlabel('Years')
```

In the case when **nrows**, **ncols**, and **plot_number** are all less than 10, a convenience exists such that a 3-digit number can be given instead, where the hundreds represent **nrows**, the tens represent **ncols** and the units represent **plot_number**. For instance,

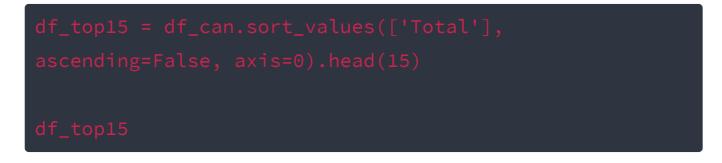
```
subplot(211) = subplot(2, 1, 1)
```

produces a subaxes in a figure which represents the top plot (i.e. the first) in a 2 rows by 1 column notional grid (no grid

actually exists, but conceptually this is how the returned subplot has been positioned).

Create a box plot to visualize the distribution of the top 15 countries (based on total immigration) grouped by the *decades* 1980s, 1990s, and 2000s.

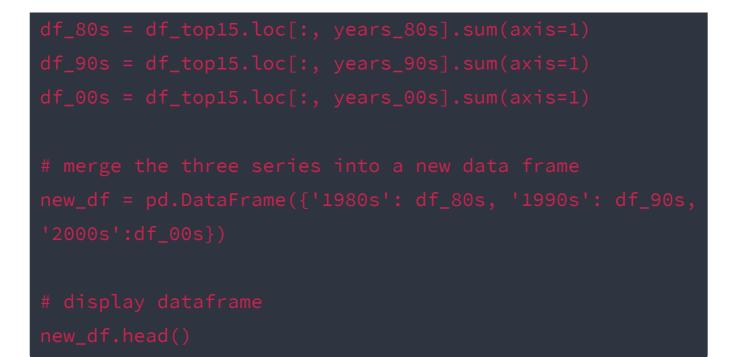
Step 1: Get the dataset. Get the top 15 countries based on Total immigrant population. Name the dataframe **df_top15**



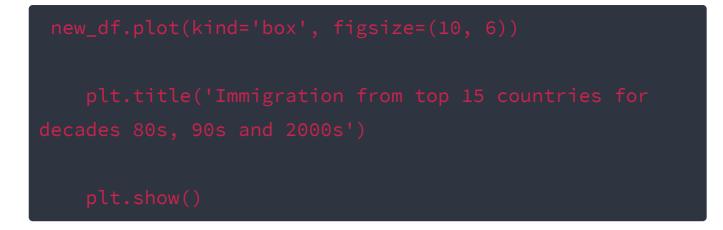
Step 2: Create a new dataframe which contains the aggregate for each decade. One way to do that:

- 1. Create a list of all years in decades 80's, 90's, and 00's.
- 2. Slice the original dataframe df_can to create a series for each decade and sum across all years for each country.
- Merge the three series into a new data frame. Call your dataframe new_df

```
# create a list of all years in decades 80's, 90's, and
00's
years_80s = list(map(str, range(1980, 1990)))
years_90s = list(map(str, range(1990, 2000)))
years_00s = list(map(str, range(2000, 2010)))
# slice the original dataframe df_can to create a series
for each decade
```



Step 3: Plot the box plots.



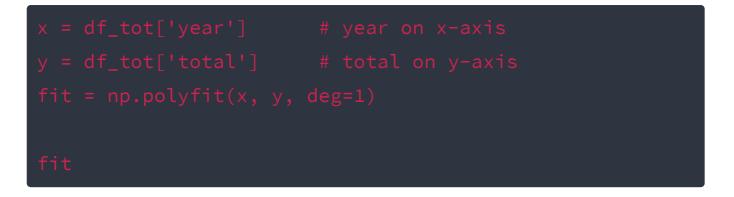
Scatter Plots

A scatter plot (2D) is a useful method of comparing variables against each other. Scatter plots look similar to line plots in that they both map independent and dependent variables on a 2D graph. While the data points are connected together by a line in a line plot, they are not connected in a scatter plot. The data in a scatter plot is considered to express a trend. With further analysis using tools like regression, we can mathematically calculate this relationship and use it to predict trends outside the dataset. Using a scatter plot, let's visualize the trend of total immigrantion to Canada (all countries combined) for the years 1980 - 2013.

let's try to plot a linear line of best fit, and use it to predict the number of immigrants in 2015.

Step 1: Get the equation of line of best fit. We will use **Numpy**'s **polyfit()** method by passing in the following:

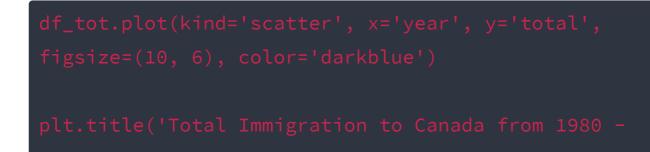
- **x**: x-coordinates of the data.
- y: y-coordinates of the data.
- deg: Degree of fitting polynomial. 1 = linear, 2 = quadratic, and so on.



The output is an array with the polynomial coefficients, highest powers first. Since we are plotting a linear regression

y= $a \times x + b$, our output has 2 elements [5.56709228e+03, -1.09261952e+07] with the the slope in position 0 and intercept in position 1

Step 2: Plot the regression line on the scatter plot.



```
2013')
plt.xlabel('Year')
plt.ylabel('Number of Immigrants')
# plot line of best fit
plt.plot(x, fit[0] * x + fit[1], color='red') # recall
that x is the Years
plt.annotate('y={0:.0f} x + {1:.0f}'.format(fit[0],
fit[1]), xy=(2000, 150000))
plt.show()
# print out the line of best fit
'No. Immigrants = {0:.0f} * Year +
{1:.0f}'.format(fit[0], fit[1])
```

'No. Immigrants = 5567 * Year + -10926195'

Using the equation of line of best fit, we can estimate the number of immigrants in 2015:

No. Immigrants = 5567 *Year - 10926195 No. Immigrants = 5567* 2015 - 10926195 No. Immigrants = 291,310

Create a scatter plot of the total immigration from Denmark, Norway, and Sweden to Canada from 1980 to 2013?

```
# create df_countries dataframe
    df_countries = df_can.loc[['Denmark', 'Norway',
    'Sweden'], years].transpose()
```

```
# create df_total by summing across three countries
for each year
   df_total = pd.DataFrame(df_countries.sum(axis=1))
   # reset index in place
   df_total.reset_index(inplace=True)
```

```
# rename columns
df_total.columns = ['year', 'total']
```

change column year from string to int to create
scatter plot

df_total['year'] = df_total['year'].astype(int)

show resulting dataframe
df_total.head()

```
# generate scatter plot
    df_total.plot(kind='scatter', x='year', y='total',
figsize=(10, 6), color='darkblue')
```

```
# add title and label to axes
plt.title('Immigration from Denmark, Norway, and
Sweden to Canada from 1980 - 2013')
plt.xlabel('Year')
plt.ylabel('Number of Immigrants')
# show plot
```

Bubble Plots

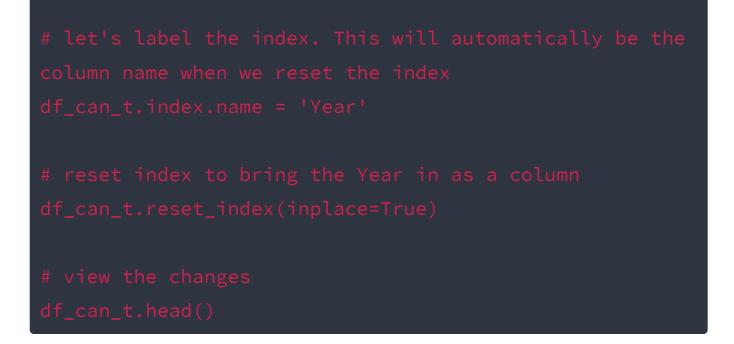
A bubble plot is a variation of the scatter plot that displays three dimensions of data (x, y, z). The data points are replaced with bubbles, and the size of the bubble is determined by the third variable z, also known as the weight. In maplotlib, we can pass in an array or scalar to the parameter s to plot(), that contains the weight of each point.

Let's start by analyzing the effect of Argentina's great depression

Argentina suffered a great depression from 1998 to 2002, which caused widespread unemployment, riots, the fall of the government, and a default on the country's foreign debt. In terms of income, over 50% of Argentines were poor, and seven out of ten Argentine children were poor at the depth of the crisis in 2002.

Let's analyze the effect of this crisis, and compare Argentina's immigration to that of it's neighbour Brazil. Let's do that using a **bubble plot** of immigration from Brazil and Argentina for the years 1980 - 2013. We will set the weights for the bubble as the *normalized* value of the population for each year

```
# transposed dataframe
df_can_t = df_can[years].transpose()
# cast the Years (the index) to type int
df_can_t.index = map(int, df_can_t.index)
```



Step 2: Create the normalized weights.

There are several methods of normalizations in statistics, each with its own use. In this case, we will use [feature scaling] to bring all values into the range [0, 1]. The general formula is:

$$X' = rac{X - X_{\min}}{X_{\max} - X_{\min}}$$

where XX is the original value, X'X' is the corresponding normalized value. The formula sets the max value in the dataset to 1, and sets the min value to 0. The rest of the data points are scaled to a value between 0-1 accordingly.

```
# normalize Brazil data
norm_brazil = (df_can_t['Brazil'] -
df_can_t['Brazil'].min()) / (df_can_t['Brazil'].max() -
df_can_t['Brazil'].min())
```

Step 3: Plot the data.

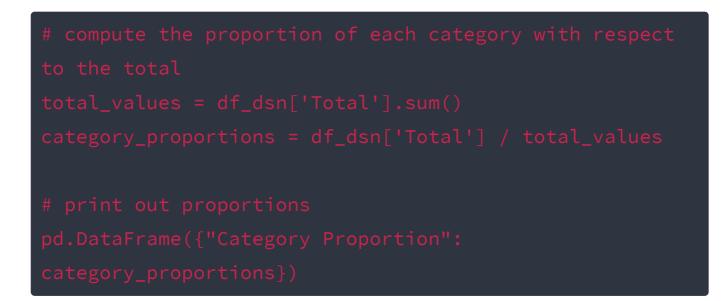
- To plot two different scatter plots in one plot, we can include the axes one plot into the other by passing it via the ax parameter.
- We will also pass in the weights using the sparameter. Given that the normalized weights are between 0-1, they won't be visible on the plot. Therefore, we will:
 - multiply weights by 2000 to scale it up on the graph, and,
 - add 10 to compensate for the min value (which has a 0 weight and therefore scale with $\times 2000 \times 2000$).

Waffle Charts

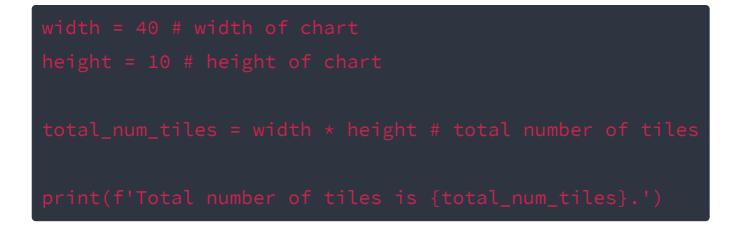
A waffle chart is an interesting visualization that is normally created to display progress toward goals. It is commonly an effective option when you are trying to add interesting visualization features to a visual that consists mainly of cells, such as an Excel dashboard.

Unfortunately, unlike R, waffle charts are not built into any of the Python visualization libraries. Therefore, we will learn how to create them from scratch.

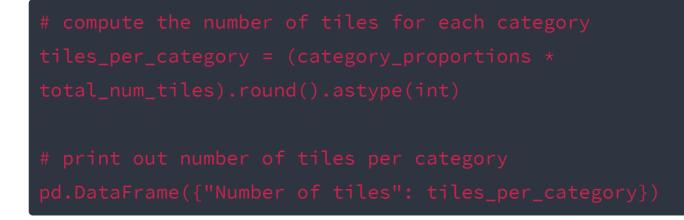
Step 1. The first step into creating a waffle chart is determing the proportion of each category with respect to the total.



Step 2. The second step is defining the overall size of the waffle chart

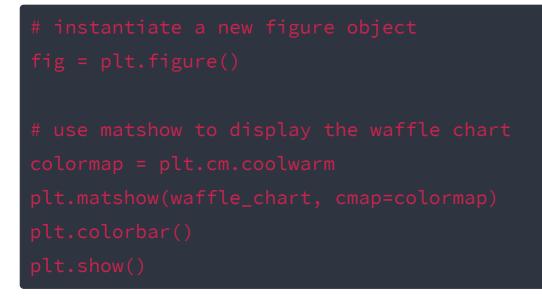


Step 3. The third step is using the proportion of each category to determe it respective number of tiles



Step 4. The fourth step is creating a matrix that resembles the waffle chart and populating it.

Step 5. Map the waffle chart matrix into a visual.



Step 6. Prettify the chart.

plt.xticks([])
plt.yticks([])
plt.show()

Step 7. Create a legend and add it to chart.

values_cumsum = np.cumsum(df_dsn['Total'])

Now it would very inefficient to repeat these seven steps every time we wish to create a waffle chart. So let's combine all seven steps into one function called *create_waffle_chart*. This function would take the following parameters as input:

- 1. categories: Unique categories or classes in dataframe.
- 2. values: Values corresponding to categories or classes.
- 3. height: Defined height of waffle chart.
- 4. width: Defined width of waffle chart.
- 5. colormap: Colormap class

 value_sign: In order to make our function more generalizable, we will add this parameter to address signs that could be associated with a value such as %, \$, and so on. value_sign has a default value of empty string.

```
print (df_dsn.index.values[i] + ': ' +
str(tiles))
```

initialize the waffle chart as an empty matrix
waffle_chart = np.zeros((height, width))

```
str(values[i]) + ')'
```

```
color_val =
colormap(float(values_cumsum[i])/total_values)
legend_handles.append(mpatches.Patch(color=color_val,
label=label_str))

# add legend to chart
plt.legend(
    handles=legend_handles,
    loc='lower center',
    ncol=len(categories),
    bbox_to_anchor=(0., -0.2, 0.95, .1)
)
plt.show()
```

Now to create a waffle chart, all we have to do is call the function create_waffle_chart. Let's define the input parameters

```
width = 40 # width of chart
height = 10 # height of chart
categories = df_dsn.index.values # categories
values = df_dsn['Total'] # correponding values of
categories
colormap = plt.cm.coolwarm # color map class
create_waffle_chart(categories, values, height, width,
colormap)
```

Word Clouds

word clouds (also known as text clouds or tag clouds) work in a simple way: the more a specific word appears in a source of textual data (such as a speech, blog post, or database), the bigger and bolder it appears in the word cloud. Luckily, a Python package already exists in Python for generating word clouds. The package, called word_cloud

install wordcloud ! pip3 install wordcloud

from wordcloud import WordCloud, STOPWORDS

print ('Wordcloud is installed and imported!')

word clouds are commonly used to perform high-level analysis and visualization of text data. Accordinly, let's digress from the immigration dataset and work with an example that involves analyzing text data. Let's try to analyze a short novel written by **Lewis Carroll** titled *Alice's Adventures in Wonderland*. Let's go ahead and download a .*txt* file of the novel.

import urllib

```
# open the file and read it into a variable alice_novel
alice_novel = urllib.request.urlopen('https://cf-
courses-data.s3.us.cloud-object-
```

storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/alice_novel.txt').read().deco de("utf-8")

Next, let's use the stopwords that we imported from

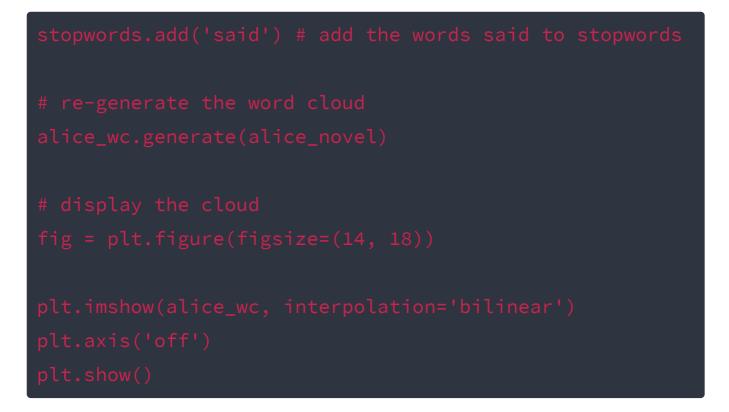
word_cloud. We use the function *set* to remove any redundant stopwords

stopwords = set(STOPWORDS)

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

```
# instantiate a word cloud object
alice_wc = WordCloud(
    background_color='white',
    max_words=2000,
    stopwords=stopwords
)
# generate the word cloud
alice_wc.generate(alice_novel)
# display the word cloud
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

said isn't really an informative word. So let's add it to our stopwords and re-generate the cloud.

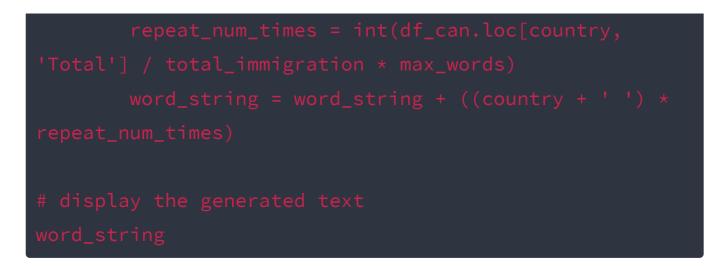


Another example

Unfortunately, our immigration data does not have any text data, but where there is a will there is a way. Let's generate sample text data from our immigration dataset, say text data of 90 words.

Using countries with single-word names, let's duplicate each country's name based on how much they contribute to the total immigration.

```
total_immigration = df_can['Total'].sum()
total_immigration
max_words = 90
word_string = ''
for country in df_can.index.values:
    # check if country's name is a single-word name
    if country.count(" ") == 0:
```

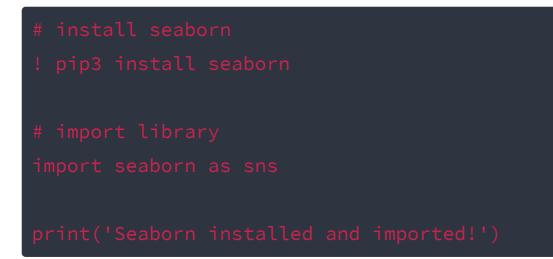


We are not dealing with any stopwords here, so there is no need to pass them when creating the word cloud.

```
# create the word cloud
wordcloud =
WordCloud(background_color='white').generate(word_string
)
print('Word cloud created!')
# display the cloud
plt.figure(figsize=(14, 18))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

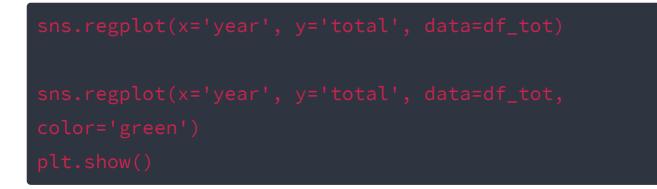
Regression Plots

In lab *Pie Charts, Box Plots, Scatter Plots, and Bubble Plots,* we learned how to create a scatter plot and then fit a regression line. It took ~20 lines of code to create the scatter plot along with the regression fit. In this final section, we will explore *seaborn* and see how efficient it is to create regression lines and fits using this library!

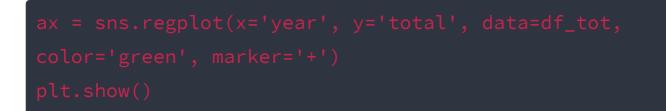


Create a new dataframe that stores that total number of landed immigrants to Canada per year from 1980 to 2013

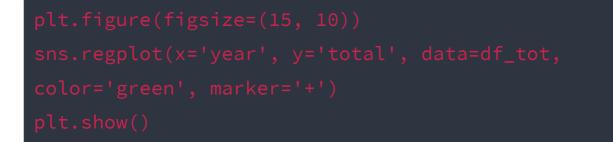
With *seaborn*, generating a regression plot is as simple as calling the **regplot** function



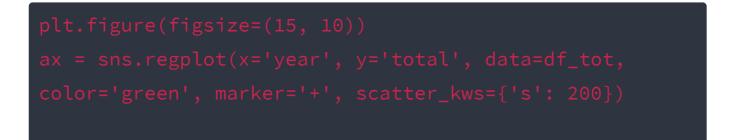
You can always customize the marker shape, so instead of circular markers, let's use



Let's blow up the plot a little so that it is more appealing to the sight.



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



```
ax.set(xlabel='Year', ylabel='Total Immigration') # add
x- and y-labels
ax.set_title('Total Immigration to Canada from 1980 -
2013') # add title
plt.show()
```

And finally increase the font size of the tickmark labels, the title, and the x- and y-labels so they don't feel left out

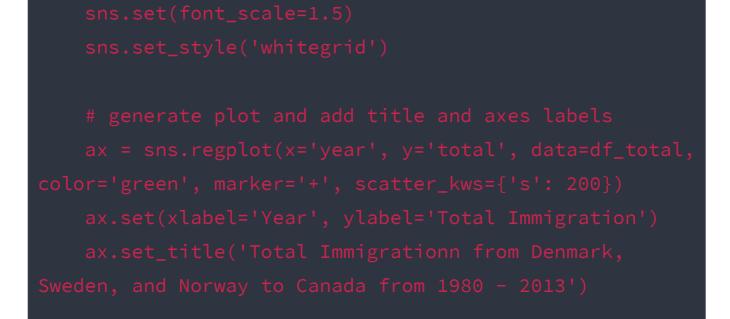
```
plt.figure(figsize=(15, 10))
sns.set(font_scale=1.5)
ax = sns.regplot(x='year', y='total', data=df_tot,
color='green', marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 -
2013')
plt.show()
```

If you are not a big fan of the purple background, you can easily change the style to a white plain background

```
plt.figure(figsize=(15, 10))
sns.set(font_scale=1.5)
sns.set_style('ticks') # change background to white
background
ax = sns.regplot(x='year', y='total', data=df_tot,
color='green', marker='+', scatter_kws={'s': 200})
```

```
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 -
2013')
plt.show()
```

Use seaborn to create a scatter plot with a regression line to visualize the total immigration from Denmark, Sweden, and Norway to Canada from 1980 to 2013.



Geospatial visualization with Folium

Folium is that it was developed for the sole purpose of visualizing geospatial data. While other libraries are available to visualize geospatial data, such as **plotly**, they might have a cap on how many API calls you can make within a defined time frame. **Folium**, on the other hand, is completely free. Folium is a powerful Python library that helps you create several types of Leaflet maps. The fact that the Folium results are interactive makes this library very useful for dashboard building

Scenario

Datasets:

 San Francisco Police Department Incidents for the year 2016 - [Police Department Incidents] from San Francisco public data portal. Incidents derived from San Francisco Police Department (SFPD) Crime Incident Reporting system. Updated daily, showing data for the entire year of 2016. Address and location has been anonymized by moving to mid-block or to an intersection.

2. Immigration to Canada from 1980 to 2013 - [International migration flows to and from selected countries - The 2015 revision] from United Nation's website. The dataset contains annual data on the flows of international migrants as recorded by the countries of destination. The data presents both inflows and outflows according to the place of birth, citizenship or place of previous / next residence both for foreigners and nationals. For this lesson, we will focus on the Canadian Immigration data

```
import numpy as np # useful for many scientific
computing in Python
import pandas as pd # primary data structure library
!conda install -c conda-forge folium=0.5.0 --yes
import folium
```

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



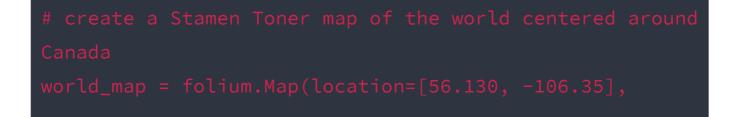
display world map world_map

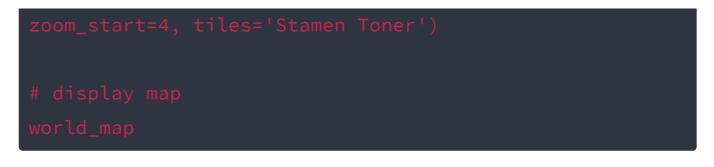
You can customize this default definition of the world map by specifying the centre of your map, and the initial zoom level. All locations on a map are defined by their respective *Latitude* and *Longitude* values. So you can create a map and pass in a center of *Latitude* and *Longitude* values of **[0, 0]**. For a defined center, you can also define the initial zoom level into that location when the map is rendered. **The higher the zoom level the more the map is zoomed into the center**. Let's create a map centered around Canada and play with the zoom level to see how it affects the rendered map

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

A. Stamen Toner Maps

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





B. Stamen Terrain Maps

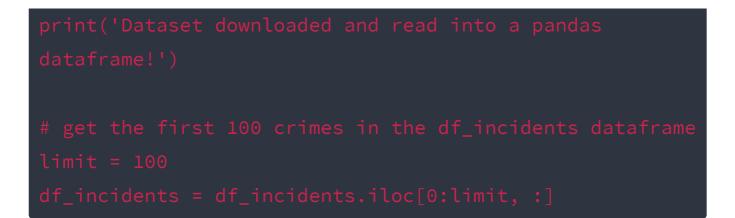
These are maps that feature hill shading and natural vegetation colors. They showcase advanced labeling and linework generalization of dual-carriageway roads.



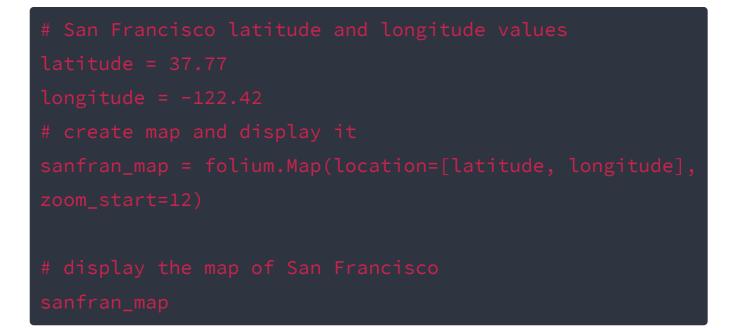
Maps with Markers

Download and import the data on police department incidents using *pandas* read_csv() method



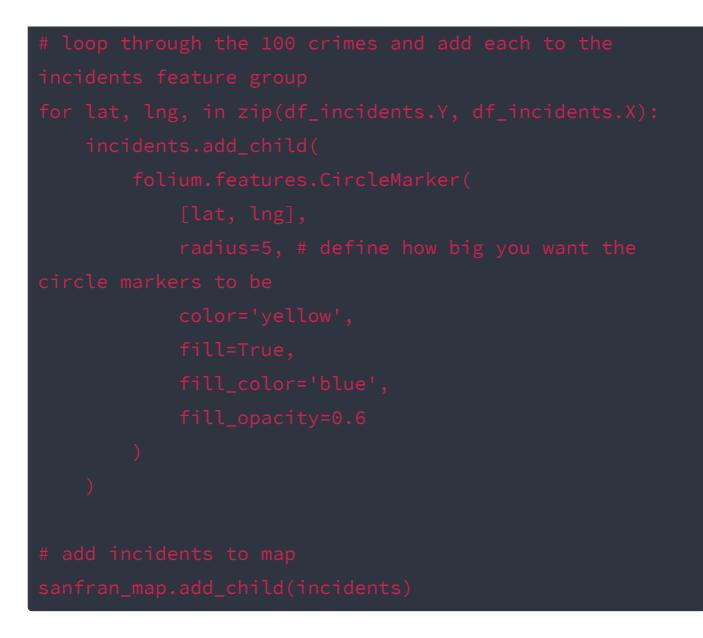


Now that we reduced the data a little, let's visualize where these crimes took place in the city of San Francisco. We will use the default style, and we will initialize the zoom level to 12.



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

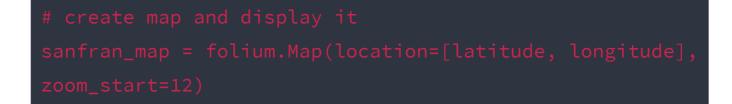
sanfran_map
instantiate a feature group for the incidents in the
dataframe
incidents = folium.map.FeatureGroup()



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

```
# instantiate a feature group for the incidents in the
dataframe
incidents = folium.map.FeatureGroup()
# loop through the 100 crimes and add each to the
incidents feature group
for lat, lng, in zip(df_incidents.Y, df_incidents.X):
    incidents.add_child(
        folium.features.CircleMarker(
```

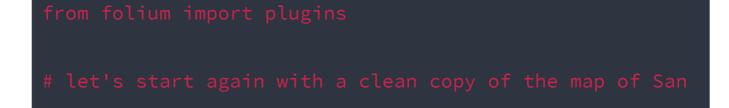
If you find the map to be so congested will all these markers, there are two remedies to this problem. The simpler solution is to remove these location markers and just add the text to the circle markers themselves as follows:



```
# loop through the 100 crimes and add each to the map
for lat, lng, label in zip(df_incidents.Y,
df_incidents.X, df_incidents.Category):
    folium.features.CircleMarker(
        [lat, lng],
        radius=5, # define how big you want the circle
markers to be
        color='yellow',
        fill=True,
        popup=label,
        fill_color='blue',
        fill_opacity=0.6
    ).add_to(sanfran_map)
# show map
sanfran_map
```

The other proper remedy is to group the markers into different clusters. Each cluster is then represented by the number of crimes in each neighborhood. These clusters can be thought of as pockets of San Francisco which you can then analyze separately.

To implement this, we start off by instantiating a *MarkerCluster* object and adding all the data points in the dataframe to this object.



Choropleth Maps

A **Choropleth** map is a thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map, such as population density or per-capita income. The choropleth map provides an easy way to visualize how a measurement varies across a geographic area, or it shows the level of variability within a region

```
print(df_can.shape)
```

Clean up data

```
# clean up the dataset to remove unnecessary columns
(eg. REG)
df_can.drop(['AREA','REG','DEV','Type','Coverage'],
axis=1, inplace=True)
```

```
# let's rename the columns so that they make sense
df_can.rename(columns={'OdName':'Country',
'AreaName':'Continent','RegName':'Region'},
inplace=True)
```

```
# for sake of consistency, let's also make all column
labels of type string
df_can.columns = list(map(str, df_can.columns))
```

```
df_can['Total'] = df_can.sum(axis=1)
# years that we will be using in this lesson - useful
for plotting later on
years = list(map(str, range(1980, 2014)))
print ('data dimensions:', df_can.shape)
```

In order to create a **Choropleth** map, we need a GeoJSON file that defines the areas/boundaries of the state, county, or country that we are interested in. In our case, since we are endeavoring to create a world map, we want a GeoJSON that defines the boundaries of all world countries



Now that we have the GeoJSON file, let's create a world map, centered around **[0, 0]** *latitude* and *longitude* values, with an initisal zoom level of 2

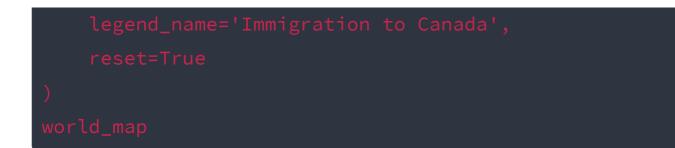
```
world_geo = r'world_countries.json' # geojson file
# create a plain world map
world_map = folium.Map(location=[0, 0], zoom_start=2)
```

And now to create a **Choropleth** map, we will use the *choropleth* method with the following main parameters:

- 1. geo_data, which is the GeoJSON file.
- 2. data, which is the dataframe containing the data.
- 3. **columns**, which represents the columns in the dataframe that will be used to create the **Choropleth** map.
- 4. key_on, which is the key or variable in the GeoJSON file that contains the name of the variable of interest. To determine that, you will need to open the GeoJSON file using any text editor and note the name of the key or variable that contains the name of the countries, since the countries are our variable of interest. In this case, name is the key in the GeoJSON file that contains the name of the countries. Note that this key is case_sensitive, so you need to pass exactly as it exists in the GeoJSON file.

```
# generate choropleth map using the total immigration of
each country to Canada from 1980 to 2013
world_map.choropleth(
    geo_data=world_geo,
    data=df_can,
    columns=['Country', 'Total'],
    key_on='feature.properties.name',
    fill_color='YlOrRd',
    fill_opacity=0.7,
    line_opacity=0.2,
    legend_name='Immigration to Canada'
```

Defining our own thresholds and starting with 0 instead of -6,918!



Creating Dashboards

Basic Plotly Charts

Airline Reporting Carrier On-Time Performance Dataset

The Reporting Carrier On-Time Performance Dataset contains information on approximately 200 million domestic US flights reported to the United States Bureau of Transportation Statistics. The dataset contains basic information about each flight (such as date, time, departure airport, arrival airport) and, if applicable, the amount of time the flight was delayed and information about the reason for the delay. This dataset can be used to predict the likelihood of a flight arriving on time.

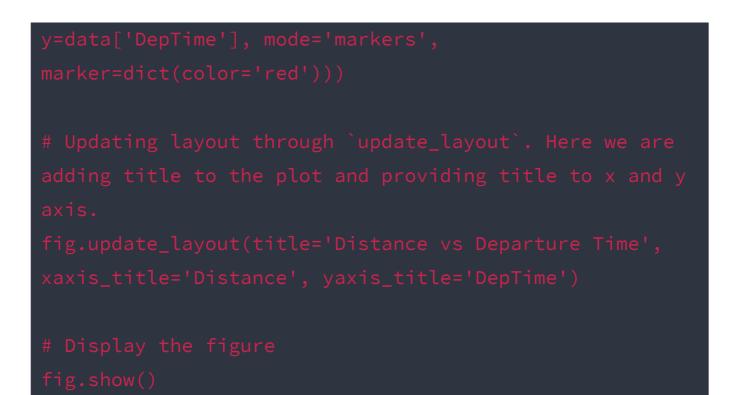
```
# Import required libraries
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
# Read the airline data into pandas dataframe
airline_data = pd.read_csv('https://cf-courses-
data.s3.us.cloud-object-
```

plotly.graph_objects

Scatter Plot

Idea: How departure time changes with respect to airport distance

First we create a figure using go.Figure and adding trace to it through go.scatter fig = go.Figure(data=go.Scatter(x=data['Distance'],



Line Plot

Extract average monthly arrival delay time and see how it changes over the year



- Create a line plot with x-axis being the month and y-axis being computed average delay time. Update plot title, xaxis, and yaxis title.
- Hint: Scatter and line plot vary by updating mode parameter

```
fig = go.Figure(data=go.Scatter(x=line_data['Month'],
y=line_data['ArrDelay'], mode='lines',
marker=dict(color='green')))
```

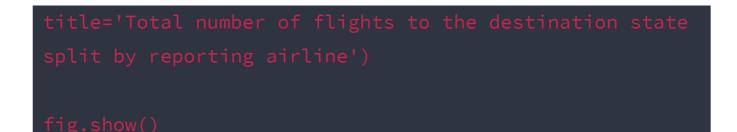
fig.update_layout(title='Month vs Average Flight Delay
Time', xaxis_title='Month', yaxis_title='ArrDelay')

fig.show()

plotly.express

Extract number of flights from a specific airline that goes to a destination

```
# Group the data by destination state and reporting
airline. Compute total number of flights in each
combination
bar_data = data.groupby(['DestState'])
['Flights'].sum().reset_index()
# Display the data
bar_data
# Use plotly express bar chart function px.bar. Provide
input data, x and y axis variable, and title of the
chart.
# This will give total number of flights to the
destination state.
fig = px.bar(bar_data, x="DestState", y="Flights",
```



Bubble Chart

Get number of flights as per reporting airline



- Create a bubble chart using the bub_data with x-axis being reporting airline and y-axis being flights.
- Provide title to the chart
- Update size of the bubble based on the number of flights.
 Use size parameter.
- Update name of the hover tooltip to reporting_airline using hover_name parameter.

Histogram

Get distribution of arrival delay

Set missing values to 0
data['ArrDelay'] = data['ArrDelay'].fillna(0)

- Use px.histogram and pass the dataset.
- Pass ArrDelay to x parameter.

fig = px.histogram(data, x="ArrDelay")
fig.show()

Pie Chart

Proportion of distance group by month (month indicated by numbers)

```
# Use px.pie function to create the chart. Input
dataset.
# Values parameter will set values associated to the
sector. 'Month' feature is passed to it.
# labels for the sector are passed to the `names`
parameter.
fig = px.pie(data, values='Month',
names='DistanceGroup', title='Distance group proportion
by month')
```

Sunburst Charts

Hierarchical view in other order of month and destination state holding value of number of flights

- Create sunburst chart using px.sunburst.
- Define hierarchy of sectors from root to leaves in path parameter. Here, we go from Month to DestStateName feature.
- Set sector values in values paramter. Here, we can pass in Flights feature.
- Show the figure

```
fig = px.sunburst(data, path=['Month', 'DestStateName'],
values='Flights')
fig.show()
```

Dash Components

- Create a dash application layout
- Add HTML H1, P, and Div components
- Add core graph component
- Add multiple charts



fig = px.pie(data, values='Month',
names='DistanceGroup', title='Distance group proportion

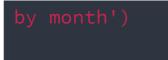


fig.show()

Theme

Proportion of distance group (250 mile distance interval group) by month (month indicated by numbers).

- 1. Import required libraries and create an application layout
- 2. Add title to the dashboard using HTML H1 component
- 3. Add a paragraph about the chart using HTML P component
- 4. Add the pie chart created above using core graph component
- 5. Run the app
- For step 1 (only review), this is very specific to running app from Jupyerlab.
 - For Jupyterlab, we will be using jupyter-dash library. Adding from jupyter_dash import JupyterDash import statement.
 - Instead of creating dash application using app = dash.Dash(), we will be using app = JupyterDash(__name__).
- For step 2,
 - Title as Airline Performance Dashboard
 - Use style parameter and make the title center aligned, with color code #503D36, and font-size as 40. Check More about HTML section

- For step 3,
 - Paragraph as Proportion of distance group (250 mile distance interval group) by month (month indicated by numbers).
 - Use style parameter to make the description center aligned and with color **#F57241**.
- For step 4, refer dcc.Graph component usage.
- For step 5, you can refer examples provided here.

NOTE: Run the solution cell multiple times if you are not seeing the result.



Capstone Project

Reivewing Web APIs

| ## Requests is a python Library that allows you to send |
|--|
| `HTTP/1.1` requests easily. We can import the library as |
| |
| |
| |
| |
| from PIL import Image |

from IPython.display import IFrame

You can make a `GET` request via the method `get` to

r.encoding

Using URL parameters in GET Requests

```
url_get='http://httpbin.org/get'
payload={"name":"Joseph","ID":"123"}
r=requests.get(url_get,params=payload)
r.url
print("request body:", r.request.body)
print(r.status_code)
print(r.text)
r.headers['Content-Type']
r.json()
```

Collecting Job Data Using APIs

- Collect job data from GitHub Jobs API
- Store the collected data into an excel spreadsheet

Scenario 1

Using an API, let us find out who currently are on the International Space Station (ISS)

The API at [http://api.open-notify.org/astros.json] gives us the information of astronauts currently on ISS in json format.



#Print the names of the astronauts currently on ISS

Scenario 2

Collect Jobs Data using GitHub Jobs API Objective: Determine the number of jobs currently open for various technologies

Collect the number of job postings for the following languages using the API:

- C
- C#
- C++
- Java
- JavaScript
- Python
- Scala
- Oracle
- SQL Server
- MySQL Server
- PostgreSQL
- MongoDB

Write a function to get the number of jobs for the given technology.

Note: The API gives a maximum of 50 jobs per page.

If you get 50 jobs per page, it means there could be some more job listings available.

So if you get 50 jobs per page you should make another API call for next page to check for more jobs.

If you get less than 50 jobs per page, you can take it as the final count

```
import requests
```

```
baseurl = "https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBM-DA0321EN-
SkillsNetwork/labs/module%201/datasets/githubposting.jso
n"
```

def get_number_of_jobs(technology):
 number_of_jobs = 0
 #your code goes here
 return technology,number_of_jobs

print(get_number_of_jobs('python'))