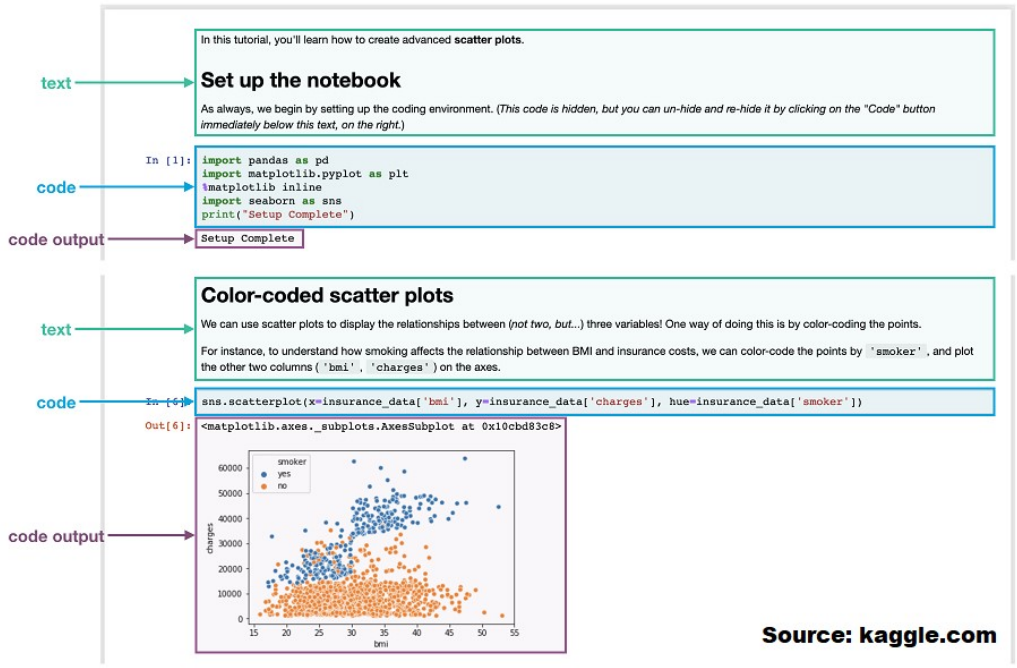
Jupyter Notebook

This is a web-based application (runs in the browser) that is used to interpret Python code.

- To add more code cells (or blocks) click on the **'+'** button in the top left corner
- There are 3 cell types in Jupyter:
  - Code: Used to write Python code
  - Markdown: Used to write texts (can be used to write explanations and other key information)
  - NBConvert: Used convert Jupyter (.ipynb) files to other formats (HTML, LaTex, etc.)
- To run Python code in a specific cell, you can click on the **'Run'** button at the top or press **Shift + Enter**
- The number sign (#) is used to insert comments when coding to leave messages for yourself or others. These comments will not be interpreted as code and are overlooked by the program



Classes

- Object-orientated programming approach popular and efficient
- Define classes of real-world things or situations (can be thought of as creating your own data type)
  - Attributes of various data types
  - Functions inside of a class are the same except called methods
  - Methods may be accessed using the dot operator
- Instanciate objects of your classes
- __init()__ method used to prefill attributes
- Capitalize class names

In [50]:
```
class Employee():
    """A simple attempt to represent am employee."""
    def __init__(self, name, employee_num, department ):
        self.name = name
        self.employee_num = employee_num
        self.department = department
```

```
    def description(self): # Creating a function (a.k.a method) that can be used by i
        print(f"{self.name} (employee number: {self.employee_num}) - Dept: {self.depa
```

In [51]:
```
employee1 = Employee("Mike", 12210, "Marketing")
employee2 = Employee("Peter", 31445, "IT")
employee1.description()
employee2.description()
```

```
Mike (employee number: 12210) - Dept: Marketing
Peter (employee number: 31445) - Dept: IT
```

In [52]:
```
#Create a Payment class and assign it 3 attributes: payer, payee, amount
class Payment:
    def __init__(self, payer, payee, amount):
        self.payer = payer
        self.payee = payee
        self.amount = amount
```

In [53]:
```
pay1 = Payment("Peter", "Seamus", 100)
```

In [54]:
```
print(pay1.amount)
```

```
100
```

In [55]:
```
print(pay1.payee)
```

```
Seamus
```

Pandas

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

It will seamlessly bridge the gap between Python and Excel.

Built Around 2 Main Classes:

- DataFrames
- Series

In [56]:
```
#Import pandas and assign it to a shorthand name pd
import pandas as pd
```

Reading CSV Files

- Function to use in Pandas: read_csv()
- Value passed to read_csv() must be string and the **exact** name of the file
- CSV Files must be in the same directory as the python file/notebook

In [57]:
```
#Read our data into a DataFrame names features_df
#read_excel does the same but for spreadsheet files
features_df = pd.read_csv('features.csv')

#print(df)
```

Basic DataFrame Functions

- head() will display the first 5 values of the DataFrame
- tail() will display the last 5 values of the DataFrame
- shape will display the dimensions of the DataFrame
- columns() will return the columns of the DataFrame as a list
- dtypes will display the types of each column of the DataFrame
- drop() will remove a column from the DataFrame

In [58]:
```
#Display top 5 rows
features_df.head()

#nan values are essentially empty entries
```

Out[58]:

	Store	Date	Temperature	Fuel_Price	MarkDown1	CPI	Unemployment	IsHoliday
0	1	2/5/2010	42.31	2.572	NaN	211.096358	8.106	False
1	1	2/12/2010	38.51	2.548	NaN	211.242170	8.106	True
2	1	2/19/2010	39.93	2.514	NaN	211.289143	8.106	False
3	1	2/26/2010	46.63	2.561	NaN	211.319643	8.106	False
4	1	3/5/2010	46.50	2.625	NaN	211.350143	8.106	False

In [59]:
```
#Display bottom 5 rows
features_df.tail()
```

Out[59]:

	Store	Date	Temperature	Fuel_Price	MarkDown1	CPI	Unemployment	IsHoliday
8185	45	6/28/2013	76.05	3.639	4842.29	NaN	NaN	False
8186	45	7/5/2013	77.50	3.614	9090.48	NaN	NaN	False
8187	45	7/12/2013	79.37	3.614	3789.94	NaN	NaN	False
8188	45	7/19/2013	82.84	3.737	2961.49	NaN	NaN	False
8189	45	7/26/2013	76.06	3.804	212.02	NaN	NaN	False

In [60]:
```
#Print dimensions of DataFrame as tuple
features_df.shape
```

Out[60]: (8190, 8)

In [61]:
```
#Print list of column values
features_df.columns
```

Out[61]:

```
Index(['Store', 'Date', 'Temperature', 'Fuel_Price', 'MarkDown1', 'CPI',
```

In [62]:
```
#To only rename specific columns
features_df.rename(columns={'Temperature': 'Temp', 'MarkDown1':'MD1'}, inplace=True)
```

In [63]:
```
#Print Pandas-specific data types of all columns
features_df.dtypes
```

Out[63]:
```
Store             int64
Date             object
Temp            float64
Fuel_Price      float64
MD1             float64
CPI             float64
Unemployment    float64
IsHoliday          bool
dtype: object
```

Indexing and Series Functions

- Columns of a DataFrame can be accessed through the following format: df_name["name_of_column"]
- Columns will be returned as a Series, which have different methods than DataFrames
- A couple useful Series functions: max(), median(), min(), value_counts(), sort_values()

In [64]:
```
#Extract CPI column of features_df
features_df["CPI"].head()
```

Out[64]:
```
0    211.096358
1    211.242170
2    211.289143
3    211.319643
4    211.350143
Name: CPI, dtype: float64
```

In [65]:
```
#Display the dimensions with 'shape'
#Display the total number of entries with 'size'
# Example with our DataFrame
print(features_df.shape)
print(features_df.size)
```

```
(8190, 8)
65520
```

In [66]:
```
#Maximum value in Series
features_df["CPI"].max()
```

Out[66]:
```
228.9764563
```

In [67]:
```
#Median value in Series
features_df["CPI"].median()
```

Out[67]:
```
182.7640032
```

In [68]:
```
#Minimum value in Series
features_df["CPI"].min()
```

Out[68]:
```
126.064
```

In [69]:
```
#Basic Statistical Summary of a column
features_df['Temp'].describe()
```

Out[69]:
```
count    8190.000000
mean       59.356198
std        18.678607
min        -7.290000
25%        45.902500
50%        60.710000
75%        73.880000
max       101.950000
Name: Temp, dtype: float64
```

In [70]:
```
#Print list of unique values
features_df["Store"].unique()
```

Out[70]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45], dtype=int64)
```

In [71]:
```
#Print unique values and frequency
features_df["Date"].value_counts()
```

Out[71]:
```
2/25/2011    45
2/1/2013     45
3/25/2011    45
9/2/2011     45
1/18/2013    45
             ..
4/5/2013     45
8/3/2012     45
4/15/2011    45
10/7/2011    45
3/9/2012     45
Name: Date, Length: 182, dtype: int64
```

In [72]:
```
#Return a sorted DataFrame acording to specified column
features_df.sort_values(by = "Date", ascending = True)
features_df.head()
```

Out[72]:

	Store	Date	Temp	Fuel_Price	MD1	CPI	Unemployment	IsHoliday
0	1	2/5/2010	42.31	2.572	NaN	211.096358	8.106	False
1	1	2/12/2010	38.51	2.548	NaN	211.242170	8.106	True
2	1	2/19/2010	39.93	2.514	NaN	211.289143	8.106	False
3	1	2/26/2010	46.63	2.561	NaN	211.319643	8.106	False
4	1	3/5/2010	46.50	2.625	NaN	211.350143	8.106	False

In [73]:
```
features_df.head()
```

Out[73]:

	Store	Date	Temp	Fuel_Price	MD1	CPI	Unemployment	IsHoliday
0	1	2/5/2010	42.31	2.572	NaN	211.096358	8.106	False
1	1	2/12/2010	38.51	2.548	NaN	211.242170	8.106	True
2	1	2/19/2010	39.93	2.514	NaN	211.289143	8.106	False
3	1	2/26/2010	46.63	2.561	NaN	211.319643	8.106	False
4	1	3/5/2010	46.50	2.625	NaN	211.350143	8.106	False

In [74]:
```
# delete one column
features_df.drop(columns = "MD1").tail()
```

Out[74]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday
8185	45	6/28/2013	76.05	3.639	NaN	NaN	False
8186	45	7/5/2013	77.50	3.614	NaN	NaN	False
8187	45	7/12/2013	79.37	3.614	NaN	NaN	False
8188	45	7/19/2013	82.84	3.737	NaN	NaN	False
8189	45	7/26/2013	76.06	3.804	NaN	NaN	False

In [75]:
```
# Check for missing values and how many
features_df.isnull().sum()
```

Out[75]:
```
Store              0
Date               0
Temp               0
Fuel_Price         0
MD1             4158
CPI              585
Unemployment     585
IsHoliday          0
dtype: int64
```

In [76]:
```
# delete multiple columns
features_df.drop(columns = 'MD1', inplace = True)
```

In [77]:
```
features_df.head()
```

Out[77]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday
0	1	2/5/2010	42.31	2.572	211.096358	8.106	False
1	1	2/12/2010	38.51	2.548	211.242170	8.106	True
2	1	2/19/2010	39.93	2.514	211.289143	8.106	False
3	1	2/26/2010	46.63	2.561	211.319643	8.106	False
4	1	3/5/2010	46.50	2.625	211.350143	8.106	False

In [78]:
```
#Define a function to convert float values to our custom categorical ranges

def temp_categorical(temp):
    if temp < 50:
```

```
        return 'Mild'
    elif temp >= 50 and temp < 80:
        return 'Warm'
    else:
        return 'Hot'
```

In [79]:
```
#With the apply() function we can apply our custom function to each value of the Seri

features_df['Temp'] = features_df['Temp'].apply(temp_categorical)
```

In [80]:
```
features_df['Temp'].tail()
```

Out[80]:
```
8185    Warm
8186    Warm
8187    Warm
8188     Hot
8189    Warm
Name: Temp, dtype: object
```

In [81]:
```
#More efficient way method
#Uses matrix manipulation instead of row by row increments
features_df['Unemployment'] += 1
```

In [82]:
```
features_df.head()
```

Out[82]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday
0	1	2/5/2010	Mild	2.572	211.096358	9.106	False
1	1	2/12/2010	Mild	2.548	211.242170	9.106	True
2	1	2/19/2010	Mild	2.514	211.289143	9.106	False
3	1	2/26/2010	Mild	2.561	211.319643	9.106	False
4	1	3/5/2010	Mild	2.625	211.350143	9.106	False

In [83]:
```
#Say a colleague of yours asks for a new metric called "customerCost"
#Add a column that is equal to Fuel_Price * CPI
```

Indexing

- Because Pandas will select entries based on column values by default, selecting data based on row values requires the use of the iloc method.
- Allowed inputs are:
  - An integer, e.g. 5.
  - A list or array of integers, e.g. [4, 3, 0].
  - A slice object with ints, e.g. 1:7.

In [84]:
```
#Return Fuel_Price to IsHoliday columns of 0-10th rows
#Note how LOC can reference columns by their names
features_df.loc[0:10,"Fuel_Price":"IsHoliday"]
```

Out[84]:

	Fuel_Price	CPI	Unemployment	IsHoliday
0	2.572	211.096358	9.106	False
1	2.548	211.242170	9.106	True
2	2.514	211.289143	9.106	False
3	2.561	211.319643	9.106	False
4	2.625	211.350143	9.106	False
5	2.667	211.380643	9.106	False
6	2.720	211.215635	9.106	False
7	2.732	211.018042	9.106	False
8	2.719	210.820450	8.808	False
9	2.770	210.622857	8.808	False
10	2.808	210.488700	8.808	False

In [85]:
```
features_df.loc[[100,105]]
```

Out[85]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
100	1	1/6/2012	Mild	3.157	219.714258	8.348	False	693.637913
105	1	2/10/2012	Mild	3.409	220.265178	8.348	True	750.883993

In [86]:
```
#Retrieve the CPI and customerCost of rows 500 to 505
features_df.loc[500:505, ["CPI", "customerCost"]]
```

Out[86]:

	CPI	customerCost
500	226.112207	840.459072
501	226.315150	842.118672
502	226.518093	830.415327
503	226.721036	820.049986
504	226.923979	817.153247
505	226.968844	815.726026

In [87]:
```
#We can also retrieve rows with a condition
features_df.loc[features_df['Store'] == 2]
```

Out[87]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
182	2	2/5/2010	Mild	2.572	210.752605	9.324	False	542.055701
183	2	2/12/2010	Mild	2.548	210.897994	9.324	True	537.368087
184	2	2/19/2010	Mild	2.514	210.945160	9.324	False	530.316133
185	2	2/26/2010	Mild	2.561	210.975957	9.324	False	540.309427
186	2	3/5/2010	Mild	2.625	211.006754	9.324	False	553.892730
...	...	...	...	...	...	...	...	...
359	2	6/28/2013	Hot	3.495	NaN	NaN	False	NaN
360	2	7/5/2013	Warm	3.422	NaN	NaN	False	NaN
361	2	7/12/2013	Hot	3.400	NaN	NaN	False	NaN
362	2	7/19/2013	Warm	3.556	NaN	NaN	False	NaN

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
363	2	7/26/2013	Hot	3.620	NaN	NaN	False	NaN

In [88]:
```
#We can layer conditions with &
filt1 = features_df['Store'] == 2
filt2 = features_df['CPI'] > 211
features_df.loc[filt1 & filt2]
```

Out[88]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
186	2	3/5/2010	Mild	2.625	211.006754	9.324	False	553.892730
187	2	3/12/2010	Warm	2.667	211.037551	9.324	False	562.837149
200	2	6/11/2010	Hot	2.668	211.112002	9.200	False	563.246821
201	2	6/18/2010	Hot	2.637	211.109654	9.200	False	556.696158
207	2	7/30/2010	Hot	2.640	211.026468	9.099	False	557.109877
...	...	...	...	...	...	...	...	...
346	2	3/29/2013	Warm	3.606	224.635985	7.237	False	810.037363
347	2	4/5/2013	Warm	3.583	224.719258	7.112	False	805.169102
348	2	4/12/2013	Warm	3.529	224.802531	7.112	False	793.328133
349	2	4/19/2013	Warm	3.451	224.802531	7.112	False	775.793536
350	2	4/26/2013	Warm	3.417	224.802531	7.112	False	768.150250

148 rows × 8 columns

In [89]:
```
#Retrieve all rows with a isHoliday of True and customerCost larger than 550
filt1 = features_df['IsHoliday'] == True
filt2 = features_df['customerCost'] > 550
features_df.loc[filt1 & filt2]
```

Out[89]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
42	1	11/26/2010	Warm	2.735	211.748433	8.838	True	579.131965
47	1	12/31/2010	Mild	2.943	211.404932	8.838	True	622.164715
53	1	2/11/2011	Mild	3.022	212.936705	8.742	True	643.494721
83	1	9/9/2011	Warm	3.546	215.861056	8.962	True	765.443305
94	1	11/25/2011	Warm	3.236	218.467621	8.866	True	706.961222
...	...	...	...	...	...	...	...	...
8113	45	2/10/2012	Mild	3.640	189.707605	9.424	True	690.535681
8143	45	9/7/2012	Warm	3.911	191.577676	9.684	True	749.260289
8154	45	11/23/2012	Mild	3.748	192.283032	9.667	True	720.676804
8159	45	12/28/2012	Mild	3.563	192.559264	9.667	True	686.088659
8165	45	2/8/2013	Mild	3.753	192.897089	9.625	True	723.942776

265 rows × 8 columns

In [90]:
```
#Retrieve a couple rows from their ROW index values
features_df.iloc[[0, 1]]
```

Out[90]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
0	1	2/5/2010	Mild	2.572	211.096358	9.106	False	542.939833
1	1	2/12/2010	Mild	2.548	211.242170	9.106	True	538.245049

In [91]:
```
#We may also provide specific row/column values to access specific values
features_df.iloc[0, 1]
```

Out[91]: '2/5/2010'

In [92]:
```
#Multiple rows and specific columns
features_df.iloc[[0, 2], [1, 3]]
```

Out[92]:

	Date	Fuel_Price
0	2/5/2010	2.572
2	2/19/2010	2.514

In [93]:
```
#Access rows 1 to 3 for Store column to Fuel_Price
features_df.iloc[1:3, 0:3]
```

Out[93]:

	Store	Date	Temp
1	1	2/12/2010	Mild
2	1	2/19/2010	Mild

Formatting Data

- To access and format the string values of a DataFrame, we can access methods within the "str" module of the DataFrame
- We may also format float values using options.display.float_format() in Pandas

In [94]:
```
# We can access all the same string methods from Python 1 using .str
features_df['Temp'] = features_df['Temp'].str.upper()
```

In [95]:
```
features_df.head()
```

Out[95]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
0	1	2/5/2010	MILD	2.572	211.096358	9.106	False	542.939833
1	1	2/12/2010	MILD	2.548	211.242170	9.106	True	538.245049
2	1	2/19/2010	MILD	2.514	211.289143	9.106	False	531.180905
3	1	2/26/2010	MILD	2.561	211.319643	9.106	False	541.189605
4	1	3/5/2010	MILD	2.625	211.350143	9.106	False	554.794125

In [96]:
```
#Format float
features_df.round(2).head()
```

Out[96]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
0	1	2/5/2010	MILD	2.57	211.10	9.11	False	542.94
1	1	2/12/2010	MILD	2.55	211.24	9.11	True	538.25
2	1	2/19/2010	MILD	2.51	211.29	9.11	False	531.18
3	1	2/26/2010	MILD	2.56	211.32	9.11	False	541.19
4	1	3/5/2010	MILD	2.62	211.35	9.11	False	554.79

In [97]:
```
#Export the current version of our DataFrame to a .csv file
features_df.to_csv("features_final.csv", index=False, header=True)

#to_excel also an option to export to Excel Spreadsheet
features_df.to_excel("features_final.xlsx", index=False, header=True)
```