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 [1]:
```
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 instances of this class
        print(f"{self.name} (employee number: {self.employee_num}) - Dept: {self.department}")
```

In [2]:
```
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 [3]:
```
#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 [4]:
```
pay1 = Payment("Peter", "Seamus", 100)
```

In [5]:
```
print(pay1.amount)
```

```
100
```

In [6]:
```
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

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

In [7]:
```
#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 [8]:
```
#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 [9]:
```
#Display top 5 rows
features_df.head()

#nan values are essentially empty entries
```

Out[9]:

	Store	Date	Temperature	Fuel_Price	MarkDown1	MarkDown2	MarkDown3	MarkDown4	MarkDown5	CPI	Unemployment	IsH
0	1	2010-02-05	42.31	2.572	NaN	NaN	NaN	NaN	NaN	211.096358	8.106	
1	1	2010-02-12	38.51	2.548	NaN	NaN	NaN	NaN	NaN	211.242170	8.106	
2	1	2010-02-19	39.93	2.514	NaN	NaN	NaN	NaN	NaN	211.289143	8.106	
3	1	2010-02-26	46.63	2.561	NaN	NaN	NaN	NaN	NaN	211.319643	8.106	
4	1	2010-03-05	46.50	2.625	NaN	NaN	NaN	NaN	NaN	211.350143	8.106	

In [10]:
```
#Display bottom 5 rows
features_df.tail()
```

Out[10]:

	Store	Date	Temperature	Fuel_Price	MarkDown1	MarkDown2	MarkDown3	MarkDown4	MarkDown5	CPI	Unemployment	IsHoli
8185	45	2013-06-28	76.05	3.639	4842.29	975.03	3.00	2449.97	3169.69	NaN	NaN	Fa
8186	45	2013-07-05	77.50	3.614	9090.48	2268.58	582.74	5797.47	1514.93	NaN	NaN	Fa
8187	45	2013-07-12	79.37	3.614	3789.94	1827.31	85.72	744.84	2150.36	NaN	NaN	Fa
8188	45	2013-07-19	82.84	3.737	2961.49	1047.07	204.19	363.00	1059.46	NaN	NaN	Fa
8189	45	2013-07-26	76.06	3.804	212.02	851.73	2.06	10.88	1864.57	NaN	NaN	Fa

In [11]:
```
#Print dimensions of DataFrame as tuple
features_df.shape
```

Out[11]:
```
(8190, 12)
```

In [12]:
```
#Print list of column values
features_df.columns
```

Out[12]:
```
Index(['Store', 'Date', 'Temperature', 'Fuel_Price', 'MarkDown1', 'MarkDown2',
       'MarkDown3', 'MarkDown4', 'MarkDown5', 'CPI', 'Unemployment',
       'IsHoliday'],
      dtype='object')
```

In [13]:
```
#To only rename specific columns
features_df.rename(columns={'Temperature': 'Temp', 'MarkDown1':'MD1', 'MarkDown2':'MD2',
       'MarkDown3':'MD3', 'MarkDown4':'MD4', 'MarkDown5':'MD5'}, inplace=True)
```

In [14]:
```
#Print Pandas-specific data types of all columns
features_df.dtypes
```

Out[14]:
```
Store            int64
Date            object
Temp           float64
Fuel_Price     float64
MD1            float64
MD2            float64
MD3            float64
MD4            float64
MD5            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 [15]:
```
#Extract CPI column of features_df
features_df["CPI"].head()
```

Out[15]:
```
0    211.096358
1    211.242170
2    211.289143
3    211.319643
4    211.350143
Name: CPI, dtype: float64
```

In [16]:
```
#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, 12)
98280
```

In [17]:
```
#Maximum value in Series
features_df["CPI"].max()
```

Out[17]:
```
228.9764563
```

In [18]:
```
#Median value in Series
features_df["CPI"].median()
```

Out[18]:
```
182.7640032
```

In [19]:
```
#Minimum value in Series
features_df["CPI"].min()
```

Out[19]:
```
126.064
```

In [20]:
```
#Basic Statistical Summary of a column
features_df['Temp'].describe()
```

Out[20]:
```
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 [21]:
```
#Print list of unique values
features_df["Store"].unique()
```

Out[21]:
```
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 [22]:
```
#Print unique values and frequency
features_df["Date"].value_counts()
```

Out[22]:
```
2010-06-11    45
2013-02-22    45
2010-07-09    45
2011-03-25    45
2010-02-19    45
              ..
2010-05-21    45
2010-12-03    45
2011-07-01    45
2010-02-26    45
2011-02-18    45
Name: Date, Length: 182, dtype: int64
```

In [23]:
```
#Return a sorted DataFrame acording to specified column
features_df.sort_values(by = "Date", ascending = True)
features_df.head()
```

Out[23]:

	Store	Date	Temp	Fuel_Price	MD1	MD2	MD3	MD4	MD5	CPI	Unemployment	IsHoliday
0	1	2010-02-05	42.31	2.572	NaN	NaN	NaN	NaN	NaN	211.096358	8.106	False
1	1	2010-02-12	38.51	2.548	NaN	NaN	NaN	NaN	NaN	211.242170	8.106	True
2	1	2010-02-19	39.93	2.514	NaN	NaN	NaN	NaN	NaN	211.289143	8.106	False
3	1	2010-02-26	46.63	2.561	NaN	NaN	NaN	NaN	NaN	211.319643	8.106	False
4	1	2010-03-05	46.50	2.625	NaN	NaN	NaN	NaN	NaN	211.350143	8.106	False

In [24]:
```
features_df.head()
```

Out[24]:

	Store	Date	Temp	Fuel_Price	MD1	MD2	MD3	MD4	MD5	CPI	Unemployment	IsHoliday
0	1	2010-02-05	42.31	2.572	NaN	NaN	NaN	NaN	NaN	211.096358	8.106	False
1	1	2010-02-12	38.51	2.548	NaN	NaN	NaN	NaN	NaN	211.242170	8.106	True
2	1	2010-02-19	39.93	2.514	NaN	NaN	NaN	NaN	NaN	211.289143	8.106	False
3	1	2010-02-26	46.63	2.561	NaN	NaN	NaN	NaN	NaN	211.319643	8.106	False
4	1	2010-03-05	46.50	2.625	NaN	NaN	NaN	NaN	NaN	211.350143	8.106	False

In [25]:
```
# delete one column
features_df.drop(columns = "MD1").tail()
```

Out[25]:

	Store	Date	Temp	Fuel_Price	MD2	MD3	MD4	MD5	CPI	Unemployment	IsHoliday
8185	45	2013-06-28	76.05	3.639	975.03	3.00	2449.97	3169.69	NaN	NaN	False
8186	45	2013-07-05	77.50	3.614	2268.58	582.74	5797.47	1514.93	NaN	NaN	False
8187	45	2013-07-12	79.37	3.614	1827.31	85.72	744.84	2150.36	NaN	NaN	False
8188	45	2013-07-19	82.84	3.737	1047.07	204.19	363.00	1059.46	NaN	NaN	False
8189	45	2013-07-26	76.06	3.804	851.73	2.06	10.88	1864.57	NaN	NaN	False

In [26]:
```
# Check for missing values and how many
features_df.isnull().sum()
```

Out[26]:
```
Store             0
Date              0
Temp              0
Fuel_Price        0
MD1            4158
MD2            5269
MD3            4577
MD4            4726
MD5            4140
CPI             585
Unemployment    585
IsHoliday         0
dtype: int64
```

In [27]:
```
# delete multiple columns
features_df.drop(columns = ['MD1', 'MD2', 'MD3', 'MD4', 'MD5'], inplace = True)
```

In [28]:
```
features_df.head()
```

Out[28]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday
0	1	2010-02-05	42.31	2.572	211.096358	8.106	False
1	1	2010-02-12	38.51	2.548	211.242170	8.106	True
2	1	2010-02-19	39.93	2.514	211.289143	8.106	False
3	1	2010-02-26	46.63	2.561	211.319643	8.106	False
4	1	2010-03-05	46.50	2.625	211.350143	8.106	False

In [29]:
```
#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 [30]:
```
#With the apply() function we can apply our custom function to each value of the Series

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

In [31]:
```
features_df['Temp'].tail()
```

Out[31]:
```
8185    Warm
8186    Warm
8187    Warm
8188     Hot
8189    Warm
Name: Temp, dtype: object
```

In [32]:
```
#If we would like to define a 'one time use' anonymous function, we can use the 'lambda' keyord

features_df['Unemployment'].apply(lambda num: num + 1).head()
```

Out[32]:
```
0    9.106
1    9.106
2    9.106
3    9.106
4    9.106
Name: Unemployment, dtype: float64
```

In [33]:
```
#More efficient way method
#Uses matrix manipulation instead of row by row increments
features_df['Unemployment'] += 1
```

In [34]:
```
features_df.head()
```

Out[34]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday
0	1	2010-02-05	Mild	2.572	211.096358	9.106	False
1	1	2010-02-12	Mild	2.548	211.242170	9.106	True
2	1	2010-02-19	Mild	2.514	211.289143	9.106	False
3	1	2010-02-26	Mild	2.561	211.319643	9.106	False
4	1	2010-03-05	Mild	2.625	211.350143	9.106	False

In [35]:
```
#Say a colleague of yours asks for a new metric called "customerCost"
#Add a column that is equal to Fuel_Price * CPI

features_df['customerCost'] = features_df['Fuel_Price'] * features_df['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 [36]:
```
#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[36]:

	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 [37]:
```
features_df.loc[[100,105]]
```

Out[37]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
100	1	2012-01-06	Mild	3.157	219.714258	8.348	False	693.637913
105	1	2012-02-10	Mild	3.409	220.265178	8.348	True	750.883993

In [38]:
```
#Retrieve the CPI and customerCost of rows 500 to 505
features_df.loc[500:505, ["CPI", "customerCost"]]
```

Out[38]:

	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 [39]:
```
#Retrieve a couple rows from their ROW index values
features_df.iloc[[0, 1]]
```

Out[39]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
0	1	2010-02-05	Mild	2.572	211.096358	9.106	False	542.939833
1	1	2010-02-12	Mild	2.548	211.242170	9.106	True	538.245049

In [40]:
```
#We may also provide specific row/column values to access specific values
features_df.iloc[0, 1]
```

Out[40]:
```
'2010-02-05'
```

In [41]:
```
#Multiple rows and specific columns
features_df.iloc[[0, 2], [1, 3]]
```

Out[41]:

	Date	Fuel_Price
0	2010-02-05	2.572
2	2010-02-19	2.514

In [42]:
```
#Access rows 1 to 3 for Store column to Fuel_Price
features_df.iloc[1:3, 0:3]
```

Out[42]:

	Store	Date	Temp
1	1	2010-02-12	Mild
2	1	2010-02-19	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 [43]:
```
# Split a value of the Date column yy/mm/dd
# Use 2010-02-05 as an example

print('2010-02-05'.split("-"))
```

```
['2010', '02', '05']
```

In [44]:
```
features_df.head()
```

Out[44]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost
0	1	2010-02-05	Mild	2.572	211.096358	9.106	False	542.939833
1	1	2010-02-12	Mild	2.548	211.242170	9.106	True	538.245049
2	1	2010-02-19	Mild	2.514	211.289143	9.106	False	531.180905
3	1	2010-02-26	Mild	2.561	211.319643	9.106	False	541.189605
4	1	2010-03-05	Mild	2.625	211.350143	9.106	False	554.794125

In [45]:
```
#By accessing .str, we gain access to all the string methods we covered in Python 1!
#new data frame with split value columns

new = features_df["Date"].str.split("-", expand = True)

new.head()
```

Out[45]:

	0	1	2
0	2010	02	05
1	2010	02	12
2	2010	02	19
3	2010	02	26
4	2010	03	05

In [46]:
```
#Declare new column named Year to be first column of new DataFrame
features_df["Year"]= new[0]

#Do the same for Month
features_df["Month"]= new[1]
```

In [47]:
```
features_df.head()
```

Out[47]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month
0	1	2010-02-05	Mild	2.572	211.096358	9.106	False	542.939833	2010	02
1	1	2010-02-12	Mild	2.548	211.242170	9.106	True	538.245049	2010	02
2	1	2010-02-19	Mild	2.514	211.289143	9.106	False	531.180905	2010	02
3	1	2010-02-26	Mild	2.561	211.319643	9.106	False	541.189605	2010	02
4	1	2010-03-05	Mild	2.625	211.350143	9.106	False	554.794125	2010	03

In [48]:
```
#Format float
features_df.round(2).head()
```

Out[48]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month
0	1	2010-02-05	Mild	2.57	211.10	9.11	False	542.94	2010	02
1	1	2010-02-12	Mild	2.55	211.24	9.11	True	538.25	2010	02
2	1	2010-02-19	Mild	2.51	211.29	9.11	False	531.18	2010	02
3	1	2010-02-26	Mild	2.56	211.32	9.11	False	541.19	2010	02
4	1	2010-03-05	Mild	2.62	211.35	9.11	False	554.79	2010	03

Conditional Indexing

- Conditional Operators (>, ==, >=) can be used to return rows based on their values
- Bitwise Operators (|, &) can be used to combine conditonal statements

In [49]:
```
features_df.head()
```

Out[49]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month
0	1	2010-02-05	Mild	2.572	211.096358	9.106	False	542.939833	2010	02
1	1	2010-02-12	Mild	2.548	211.242170	9.106	True	538.245049	2010	02
2	1	2010-02-19	Mild	2.514	211.289143	9.106	False	531.180905	2010	02
3	1	2010-02-26	Mild	2.561	211.319643	9.106	False	541.189605	2010	02
4	1	2010-03-05	Mild	2.625	211.350143	9.106	False	554.794125	2010	03

In [50]:
```
#Check data types of new columns
features_df.dtypes
```

Out[50]:
```
Store             int64
Date             object
Temp             object
Fuel_Price      float64
CPI             float64
Unemployment    float64
IsHoliday          bool
customerCost    float64
Year             object
Month            object
dtype: object
```

In [51]:
```
#Convert Year and Month to integers from string
features_df['Year'] = features_df['Year'].astype('int64')
features_df['Month'] = features_df['Year'].astype('int64')
```

In [52]:
```
#Return rows with year value of 2011
year_filt = features_df["Year"] == "2011"

feb_df = features_df[year_filt]
feb_df.head()
```

```
C:\Users\devra\anaconda3\lib\site-packages\pandas\core\ops\array_ops.py:253: FutureWarning: elementwise compar
ison failed; returning scalar instead, but in the future will perform elementwise comparison
  res_values = method(rvalues)
```

Out[52]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month

In [53]:
```
#Return rows with CPI lower than 130
CPI_filt = features_df["CPI"] < 130
low_CPI = features_df[CPI_filt]
low_CPI.head()
```

Out[53]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month
546	4	2010-02-05	Mild	2.598	126.442065	9.623	False	328.496484	2010	2010
547	4	2010-02-12	Mild	2.573	126.496258	9.623	True	325.474872	2010	2010
548	4	2010-02-19	Mild	2.540	126.526286	9.623	False	321.376766	2010	2010
549	4	2010-02-26	Mild	2.590	126.552286	9.623	False	327.770420	2010	2010
550	4	2010-03-05	Mild	2.654	126.578286	9.623	False	335.938770	2010	2010

In [54]:
```
#Return rows with year equal to 2010 AND unemployment larger than 8
filt1 = features_df["Year"]  == 2010
filt2 = features_df["Unemployment"]  > 8.00

unemployment_2010 = features_df[ filt1 & filt2 ]
unemployment_2010.head()
```

Out[54]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month
0	1	2010-02-05	Mild	2.572	211.096358	9.106	False	542.939833	2010	2010
1	1	2010-02-12	Mild	2.548	211.242170	9.106	True	538.245049	2010	2010
2	1	2010-02-19	Mild	2.514	211.289143	9.106	False	531.180905	2010	2010
3	1	2010-02-26	Mild	2.561	211.319643	9.106	False	541.189605	2010	2010
4	1	2010-03-05	Mild	2.625	211.350143	9.106	False	554.794125	2010	2010

In [55]:
```
#Return rows with temp larger than 40 OR Store number equal to 4
filt1 = features_df["Temp"]  == 'Cold'
filt2 = features_df["Store"]  == 4

features_df[filt1 | filt2].head()
```

Out[55]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month
546	4	2010-02-05	Mild	2.598	126.442065	9.623	False	328.496484	2010	2010
547	4	2010-02-12	Mild	2.573	126.496258	9.623	True	325.474872	2010	2010
548	4	2010-02-19	Mild	2.540	126.526286	9.623	False	321.376766	2010	2010
549	4	2010-02-26	Mild	2.590	126.552286	9.623	False	327.770420	2010	2010
550	4	2010-03-05	Mild	2.654	126.578286	9.623	False	335.938770	2010	2010

In [56]:
```
##CLASS EXERCISE
# find the rows with Fuel_Price larger than 3.00 AND IsHoliday is True
filt1 = features_df["IsHoliday"]  == True
filt2 = features_df["Fuel_Price"]  > 3.00

holiday_Fuel = features_df[filt1 & filt2]
```

In [57]:
```
holiday_Fuel.head()
```

Out[57]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month
53	1	2011-02-11	Mild	3.022	212.936705	8.742	True	643.494721	2011	2011
83	1	2011-09-09	Warm	3.546	215.861056	8.962	True	765.443305	2011	2011
94	1	2011-11-25	Warm	3.236	218.467621	8.866	True	706.961222	2011	2011
99	1	2011-12-30	Mild	3.129	219.535990	8.866	True	686.928112	2011	2011
105	1	2012-02-10	Mild	3.409	220.265178	8.348	True	750.883993	2012	2012

In [58]:
```
# find the rows with CPI < 200  OR Unemployment < 5
filt1 = features_df["CPI"]  < 200
filt2 = features_df["Unemployment"]  < 5.00

CPI_unemployment = features_df[filt1 | filt2]
```

In [59]:
```
CPI_unemployment.head()
```

Out[59]:

	Store	Date	Temp	Fuel_Price	CPI	Unemployment	IsHoliday	customerCost	Year	Month
546	4	2010-02-05	Mild	2.598	126.442065	9.623	False	328.496484	2010	2010
547	4	2010-02-12	Mild	2.573	126.496258	9.623	True	325.474872	2010	2010
548	4	2010-02-19	Mild	2.540	126.526286	9.623	False	321.376766	2010	2010
549	4	2010-02-26	Mild	2.590	126.552286	9.623	False	327.770420	2010	2010
550	4	2010-03-05	Mild	2.654	126.578286	9.623	False	335.938770	2010	2010

In [60]:
```
#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)
```