Unit 1: Requirements and Data Identification

Contents

Lab Questions

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Getting Started

This notebook makes use of several third-party libraries including pandas (https://pandas.pydata.org/), lxml (http://lxml.de/), and requests (http://docs.python-requests.org/en/master/). Additionally, pandas makes use of xlrd (https://pypi.python.org/pypi/xlrd) to load data from Excel files.

We can use the the pip (https://pip.pypa.io/en/stable/) tool to install these libraries. Typically this tool is executed from the command line but we can call it from within the notebook using `!pip` .

To execute a cell, click in it and press `SHIFT` and `ENTER` on the keyboard, click the "run cell" button, or select "Run" and "Run Cells" from the menus above.

In [ ]:
```
!pip install pandas lxml requests xlrd
```

Requirements

The Cross-industry standard process for data mining (https://en.wikipedia.org/wiki/Cross-

industry_standard_process_for_data_mining), or CRISP-DM, is a model for the data mining process. With data mining and data analytics being very interrelated, much of the CRISP-DM model can be applied to analytics. CRISP-DM breaks the data mining process into phases with the first phase focusing on business understanding and proper planning based on that understanding. We can separate the business understanding phase into four parts: determining business objectives, assessing the situation, determining goals, and creating a plan.

Determine Business Objectives

As a first step, it is important to understand, as throughly as possible, what the customer wants to accomplish. While there is often a primary goal the customer would like to achieve, there are often related goals that could be addressed. Identifying related goals early in the process could save time later.

When determining objectives, it is also important to note what constrains exists such as limitations on access to data or potential data quality issues.

For example, marketing might want to examine previous efforts and results in an attempt to determine which strategies were effective or characterize the most likely customers. In this case, past marketing campaign data and data about potential customers would need to be available.

If the goal were to minimize distribution costs, it would be important to be aware of the factors that influence those costs. If the company relies on a third party for product transportation, we would likely need to collect competitor pricing data.

Assess the Situation

In order to quickly establish the customer's objectives in the previous step, only a rough idea available resources and constraining factors was necessary. Before analytics goals can be formulated and a plan created, it is important to develop a more detailed understanding of the availability of resources, the existence of constraints, and any assumptions that must be made.

When considering resources it is important to not only list those that are available but also when they will be available. Naturally, relevant data is a critical resource and it will be important to note any access restrictions that might exist. It is also important to consider personnel, hardware, and software when assessing resource availability.

If it is determined that there is a lack of resources, whether a shortage of personnel or insufficient data, a plan should be developed to resolve this. Can required data be collected? If so, how long will it take before the needed data is available?

Knowing constraints and assumptions as early in the process as possible, allows the analytics team to address and prepare for them rather than spend time reacting later. In addition to resource constraints due to time or cost, it's important to determine any legal or security-related constraints. Once limitations and constraints are known, a list of risks that could delay the project can be compiled. Methods to mitigate risk and contingency plans should also be developed.

As part of this cataloging process, a glossary of terminology should also be created. This glossary should include both business- and analytics-specific terminology. Such a glossary can facilitate communication and ensure that both the customer and analytics team understand each other.

Determine Goals

While the customer's existing goals drove the discussion and process in the previous steps, it's important to developed related, analytics-specific goals. Just as the business goals will use business terminology, the analytics goals should make use of analytics terminology. It's import that these goals be as specific and well-defined as possible. With each goal, success criteria and identification of who is responsible for evaluating success should be established.

Create a Plan

Once goals and success criteria are established, the steps necessary to achieve the goals should be developed. For each stage of the analytics process, inputs, resources, outputs, and duration should be determined; specific tools and techniques should also be identified. As part of the planning process, dependencies and scheduling issues should be analyzed to minimize risk.

The project plan is dynamic. At the end of each stage, a review of the remaining phases of the plan should be conducted and updates made accordingly.

Data Identification

Once a plan has been developed and data is made available, it is important to develop an initial understanding of the data, identifying what it contains, and determining whether what is available is sufficient to complete the analytics objectives or not.

To access and interrogate data, we'll make use of functionality included in the standard library, the set of tools included with python, as well as third-party libraries. One such library is pandas (https://pandas.pydata.org/), which can be used to load data in a variety of formats from a variety of sources. The pandas library includes data structures and tools developed to aid in data analysis.

Introduction to pandas

Before loading data with pandas, let's explore two of the common data structures pandas provides: the Series (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html) and the DataFrame (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html). A Series is used to store one-dimensional data while a DataFrame can be used for two-dimensional, or tabular, data.

To begin, we import the pandas module and follow convention of naming it *pd*.

In [1]:
```
import pandas as pd
```

With pandas imported, we can create a Series. While there are a variety of ways to do this, we'll use a list here.

In [2]:
```
data = pd.Series([1, 2, 3, 4, 5])
```

To view to the contents of the Series, we can use the *print()* or *display()* functions. When using a notebook, we can also display its contents by referencing the Series on the last line of a cell. Here we see that in addition to the values in the original list, an index associated with each element is also displayed. The data type of the elements is displayed as we well.

Lab 1 In the cell below, use the *display()* or *print()* to display the contents of the DataFrame stored in the `data` variable.

In [3]:
```
display(data)
```

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

To access the values stored in the Series themselves, we can use the *values* attribute. Here, the values are stored using a NumPy (http://www.numpy.org/) array. NumPy is a scientific computing package and it useful for working with numerical or high-volume data; pandas relies heavily on NumPy. NumPy arrays have methods that allow us to transform an array to a traditional Python data structure such as a list; a pandas Series exposes this functionality as well.

In [4]:
```
# access a Series values
data.values
```

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

In [5]:
```
# covert the array of values to a list
data.values.tolist()
```

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

In [6]:
```
# convert the Series to a list using the underlying array's tolist() method
data.tolist()
```

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

Notice that we didn't have to use the *print()* or *display()* functions to display content. If the last statement of a cell is an object by itself or an operation that doesn't assign the value to a variable, Jupyter notebook will display the string representation of that object or result as though the *display()*

function were used. For example, because `data.values` was the last statement of the cell, the content of `data.values` was displayed.

As noted earlier, values in a Series are also associated with an index. We can manually specify an index or allow pandas to generate one as was done for our series. To access the index, we can use the Series *index* attribute.

In [7]:
```
# Series
data.index
```

Out[7]:
```
RangeIndex(start=0, stop=5, step=1)
```

The automatically-generated index is represented by the *RangeIndex* class that describes the index. This is a more memory-efficient way of storing the index data than creating an array containing each value. If necessary, we can generate the corresponding array using the *values* attribute.

Lab 2 In the cell below, use the *values* attribute of `data.index` to display the underlying array.

In [8]:
```
data.index.values
```

Out[8]:
```
array([0, 1, 2, 3, 4])
```

To access elements within a series, we can use the same bracket notation used with other data structures in Python such lists. For example, to access the second element, we can execute `data[1]`. As with most other data structures in Python, Series are zero-indexed and begin numbering at zero.

In [9]:
```
# access the second element
data[1]
```

Out[9]:
```
2
```

List lists, Series also support slicing.

In [10]:
```
# slice of elements starting at index 1 and up to but not including index 4
data[1:4]
```

Out[10]:
```
1    2
2    3
3    4
dtype: int64
```

When creating a Series, we can specify the index using the *index* keyword argument. As shown in the example below, an index does not need to be numeric.

In [11]:
```
# series with index specified
data = pd.Series([60, 65, 68, 63, 61], index=["Mon", "Tue", "Wed", "Thu", "F
data
```

Out[11]:
```
Mon    60
Tue    65
Wed    68
Thu    63
Fri    61
dtype: int64
```

When working with a non-numeric index, we can still access elements in a series using bracket or slice notation. Note that slices include the last element in the slice when working with non-numeric indexes.

In [12]:
```
# access an element using an index label
data["Mon"]
```

Out[12]:
```
60
```

Lab 3 In the cell below, use the bracket notation and slicing to access values with index labels between `Mon` and `Thu`.

In [13]:
```
data["Mon":"Thu"]
```

Out[13]:
```
Mon    60
Tue    65
Wed    68
Thu    63
dtype: int64
```

The pandas library supports working with data from a variety of sources and data structures. In previous examples, we created Series objects using lists. Below is an example in which a Series is created from dictionary. Note that prior to Python 3.7, the order of dictionary keys is not guaranteed; this affects the order in which values appear in the Series.

In [14]:
```
# dictionary of temperatures
temperatures = {"Mon": 60, "Tue": 65, "Wed": 68, "Thu": 63, "Fri": 61}
temperatures
```

Out[14]:
```
{'Fri': 61, 'Mon': 60, 'Thu': 63, 'Tue': 65, 'Wed': 68}
```

In [15]:
```
# Series from a dictionary
data = pd.Series(temperatures)
data
```

Out[15]:
```
Fri    61
Mon    60
Thu    63
Tue    65
Wed    68
dtype: int64
```

In [16]:
```
# slicing
data["Mon":"Thu"]
```

Out[16]:
```
Mon    60
Thu    63
dtype: int64
```

The pandas library provides the DataFrame data structure for use with two dimensional data. One can think of the DataFrame as an extension of a Series in the sense that a DataFrame consists of multiple Series. For example, suppose we have two Series representing high and low daily temperatures.

In [17]:
```
# two Series
low_temps = pd.Series({"Mon": 52, "Tue": 49, "Wed": 55, "Thu": 53, "Fri": 51
high_temps = pd.Series({"Mon": 60, "Tue": 65, "Wed": 68, "Thu": 63, "Fri": 6
display(low_temps, high_temps)
```

```
Fri    51
Mon    52
Thu    53
Tue    49
Wed    55
dtype: int64

Fri    61
Mon    60
Thu    63
Tue    65
Wed    68
dtype: int64
```

When creating a DataFrame, we can specify a dictionary where keys represent column names and the corresponding values are the Series containing data. Below, the `forecast` DataFrame is created with the two Series created earlier.

In [18]:
```
# create a dataframe from two series
forecast = pd.DataFrame({"high": high_temps, "low": low_temps})
forecast
```

Out[18]:

	high	low
Fri	61	51
Mon	60	52
Thu	63	53
Tue	65	49
Wed	68	55

Notice that pandas automatically aligns Series' data based on index value. The combined indexes of the Series serve as the index for the DataFrame. We can specify the order of the index when we create the DataFrame.

In [19]:
```
# specify index
forecast = pd.DataFrame({"high": high_temps, "low": low_temps}, index=["Mon'
forecast
```

Out[19]:

	high	low
Mon	60	52
Tue	65	49
Wed	68	55
Thu	63	53
Fri	61	51

Just as we could with Series, we can access the index of a DataFrame using the *index* attribute.

In [20]:
```
# display index
forecast.index
```

Out[20]:
```
Index(['Mon', 'Tue', 'Wed', 'Thu', 'Fri'], dtype='object')
```

A DataFrame's index serves to identify values in one dimension. For one-dimensional Series objects, a value for the index is sufficient to identify a specific value. Because DataFrames represent two-dimension data, an index is not enough to identify a specific value. In a DataFrame, column labels are used to identify the second dimension. To view a DataFrame's columns, we can use the *column* attribute.

In [21]:
```
# display columns
forecast.columns
```

Out[21]:
```
Index(['high', 'low'], dtype='object')
```

To access values associated with a column, we can use bracket notation with the column name.

In [22]:
```
# access a column using bracket notation
forecast['high']
```

Out[22]:
```
Mon    60
Tue    65
Wed    68
Thu    63
Fri    61
Name: high, dtype: int64
```

A DataFrame also has attributes corresponding to its columns allowing us to access column data using the dot operator.

In [23]:
```
# accessing a column using an attribute
forecast.high
```

Out[23]:
```
Mon    60
Tue    65
Wed    68
Thu    63
Fri    61
Name: high, dtype: int64
```

Each DataFrame column is a Series, we can verify this using the *type()* or *isinstance()* function.

In [24]:
```
# columns are Series
type(forecast.high)
```

Out[24]:
```
pandas.core.series.Series
```

In [25]:
```
# columns are Series
isinstance(forecast.high, pd.Series)
```

Out[25]:
```
True
```

As with Series, we can access data stored in a DataFrame using the *values* attribute.

In [26]:
```
# underlying values
forecast.values
```

Out[26]:
```
array([[60, 52],
       [65, 49],
       [68, 55],
       [63, 53],
       [61, 51]])
```

Because the DataFrame represents two-dimensional data, its values are stored as a NumPy array of nested NumPy arrays where each inner array corresponds to a row. Like most objects that represent a collection, arrays support bracket notation.

In [27]:
```
# first row
forecast.values[0]
```

Out[27]:
```
array([60, 52])
```

In [28]:
```
# first row, second value
forecast.values[0][1]
```

Out[28]:
```
52
```

While this method of accessing data based on the DataFrames underlying array works, it is cumbersome. We can instead use the DataFrame's index and columns. As we saw earlier, we can use a column's name to access its values.

In [29]:
```
# access column by name
forecast['low']
```

Out[29]:
```
Mon    52
Tue    49
Wed    55
Thu    53
Fri    51
Name: low, dtype: int64
```

To access a row, we rely on the index corresponding to the row and the DataFrame's *loc* attribute.

In [30]:
```
# access a row by index label
forecast.loc['Wed']
```

Out[30]:
```
high    68
low     55
Name: Wed, dtype: int64
```

The `forecast` DataFrame has string index values but we can still use integers to specify a specific row through the use of the *iloc* attribute.

In [31]:
```
# access a row by position
forecast.iloc[2]
```

Out[31]:
```
high    68
low     55
Name: Wed, dtype: int64
```

We can combine these methods of accessing specific rows and columns to access a specific value within the DataFrame.

In [32]:
```
# access a value by row name and index
forecast.loc['Wed']['low']
```

Out[32]:
```
55
```

In [33]:
```
# access a value by row name and index
forecast['low'].loc['Wed']
```

Out[33]: 55

Alternatively, we can specify both the index and column name simultaneously when using *iloc* or *loc*.

In [34]:
```
# access a value by row name and index
forecast.loc['Wed', 'low']
```

Out[34]: 55

Using column names and index values/labels to access data gives more context to what the corresponding data represents than using the DataFrame's underlying array.

We can also use slicing with a DataFrame's index or columns. To specify a column slice, we must use the *loc* or *iloc* properties.

Lab 4 In the cell below, slicing with the DataFrame's *loc* attribute to display rows where the index is between `Mon` and `Wed`.

In [35]:
```
forecast.loc['Mon':'Wed']
```

Out[35]:

	high	low
Mon	60	52
Tue	65	49
Wed	68	55

We can use slicing with columns and rows simultaneously.

In [36]:
```
# row and column slicing
forecast.loc['Mon':'Wed', 'high':'low']
```

Out[36]:

	high	low
Mon	60	52
Tue	65	49
Wed	68	55

We can also use bracket notation to apply a mask to a DataFrame. For example, if we want to view rows in which the high temperature is 65 or greater we can create the following mask.

In [37]:
```
# create a mask
mask = forecast.high >= 65
mask
```

Out[37]:
```
Mon    False
Tue     True
Wed     True
Thu    False
Fri    False
Name: high, dtype: bool
```

Applying the mask to the DataFrame effectively filters the data.

In [38]:
```
# apply a mask
forecast[mask]
```

Out[38]:

	high	low
Tue	65	49
Wed	68	55

Another advantage to using DataFrames is the ease with which we can manipulate the data. For example, the following line calculates the mean temperature value for each row and stores the value in a new column. Had we used another data structure, we might have had to write a for-loop to do this calculation.

In [39]:
```
# calculate new column's values
forecast['mean'] = (forecast['high'] + forecast['low']) / 2
forecast
```

Out[39]:

	high	low	mean
Mon	60	52	56.0
Tue	65	49	57.0
Wed	68	55	61.5
Thu	63	53	58.0
Fri	61	51	56.0

In fact, for performance reasons, we should try to avoid the use of for-loops when manipulating data stored in a DataFrame. A for loop will certainly work if needed though. We can iterate through a DataFrame's rows using the *iterrows()* method which returns the index and row content separately as we move from row to row.

In [40]:
```
# display index and row data for each row
for index, row in forecast.iterrows():
    print("Index:", index)
    print("Row:", row)
```

```
Index: Mon
Row: high    60.0
low     52.0
mean    56.0
Name: Mon, dtype: float64
Index: Tue
Row: high    65.0
low     49.0
mean    57.0
Name: Tue, dtype: float64
Index: Wed
Row: high    68.0
low     55.0
mean    61.5
Name: Wed, dtype: float64
Index: Thu
Row: high    63.0
low     53.0
mean    58.0
Name: Thu, dtype: float64
Index: Fri
Row: high    61.0
low     51.0
mean    56.0
Name: Fri, dtype: float64
```

Similarly, we can use a loop to manipulate row values.

In [41]:
```
# calculate difference using a for loop
forecast['difference'] = 0  # create new column with all zeros
for index, row in forecast.iterrows():
    row['difference'] = row['high'] - row['low']
    forecast.loc[index] = row  # iterrows creates a copy, we need to explici

forecast
```

Out[41]:

	high	low	mean	difference
Mon	60.0	52.0	56.0	8.0
Tue	65.0	49.0	57.0	16.0
Wed	68.0	55.0	61.5	13.0
Thu	63.0	53.0	58.0	10.0
Fri	61.0	51.0	56.0	10.0

As we continue working with pandas, additional features will explored. For a more in-depth introduction to pandas, see the Python Data Science Handbook (https://jakevdp.github.io/PythonDataScienceHandbook/).

Access Data

Accessing data will depend on how it is stored. If the relevant data is stored as a spreadsheet, one will need access to the spreadsheet file and the necessary tools to read the file. Similarly, the appropriate tools are required to access data stored in a database.

Loading Data from Files

Files tend to be a commonly used container for data, creating the files is relatively easy with spreadsheet software and sharing the data can be as easy as attaching a file to an email. Accessing data from a file in Python is also relatively easy. The file we'll use for these examples is from Kaggle (https://www.kaggle.com/), which hosts both datasets and notebooks. Specifically, we'll use an HR dataset (https://www.kaggle.com/pavansubhasht/ibm-hr-analytics-attrition-dataset) created by IBM that includes HR data about employees.

We can use Python's standard library, the collection of data structures and tools included with Python, to load the data contained in `data/01-attrition.csv`.

As a first step, let's examine the source CSV. When working with text data like a CSV, it's helpful to know how fields are separated, using a delimiter, relying on a fixed width for each column, or by some other means.

In the code below, we start by opening the source CSV for reading and assigning it to the variable `infile`. To read only the first five lines of the CSV, we'll use a while-loop that continues as long as the `line_number` variable has a value less than 5. Before entering the loop, we'll initialize the variable with a value of 0. Each iteration through the loop, we'll read a line from the file, print the line, and increment the `line_number` variable.

In [42]:
```
with open("./data/01-attrition.csv") as infile:
    line_number = 0
    while line_number < 5:
        print(infile.readline())
        line_number += 1
```

```
Age,Attrition,BusinessTravel,DailyRate,Department,DistanceFromHome,Educat
ion,EducationField,EmployeeCount,EmployeeNumber,EnvironmentSatisfaction,G
ender,HourlyRate,JobInvolvement,JobLevel,JobRole,JobSatisfaction,MaritalS
tatus,MonthlyIncome,MonthlyRate,NumCompaniesWorked,Over18,OverTime,Percen
tSalaryHike,PerformanceRating,RelationshipSatisfaction,StandardHours,Stoc
kOptionLevel,TotalWorkingYears,TrainingTimesLastYear,WorkLifeBalance,Year
sAtCompany,YearsInCurrentRole,YearsSinceLastPromotion,YearsWithCurrManage
r

41,Yes,Travel_Rarely,1102,Sales,1,2,Life Sciences,1,1,2,Female,94,3,2,Sal
es Executive,4,Single,5993,19479,8,Y,Yes,11,3,1,80,0,8,0,1,6,4,0,5

49,No,Travel_Frequently,279,Research & Development,8,1,Life Sciences,1,2,
3,Male,61,2,2,Research Scientist,2,Married,5130,24907,1,Y,No,23,4,4,80,1,
10,3,3,10,7,1,7

37,Yes,Travel_Rarely,1373,Research & Development,2,2,Other,1,4,4,Male,92,
2,1,Laboratory Technician,3,Single,2090,2396,6,Y,Yes,15,3,2,80,0,7,3,3,0,
0,0,0

33,No,Travel_Frequently,1392,Research & Development,3,4,Life Sciences,1,
5,4,Female,56,3,1,Research Scientist,3,Married,2909,23159,1,Y,Yes,11,3,3,
80,0,8,3,3,8,7,3,0
```

Based on the output, it appears that the data is comma-separated as we would expect in a CSV (comma-separated values) file. We can also see that the first line contains header data.

The Python standard library includes a CSV module (https://docs.python.org/3/library/csv.html) for working with CSV data. Among the tools in this module is the *Sniffer* (https://docs.python.org/3/library/csv.html#csv.Sniffer) class that can be used to determine formating information for a CSV. The code below reads the first ten kilobytes of the file to determine formatting information; the value can be increased as needed.

In [43]:
```
import csv
with open("./data/01-attrition.csv") as infile:
    dialect = csv.Sniffer().sniff(infile.read(10000))

print(f"Delimter: {dialect.delimiter}")
```

```
Delimter: ,
```

This is consistent with what we saw from the first five lines of the data.

Typically, we load the contents of a file one line at a time. To store all the file's lines, we'll create an empty list and store it in the variable named `csv_data`. Next, we can open the file and specify its encoding (https://en.wikipedia.org/wiki/Character_encoding) as `UTF-8-sig`, which indicates that

the file contains UTF-8 encoded data with an optional byte order mark (https://en.wikipedia.org/wiki/Byte_order_mark). The variable used to refer to the file object is named `csv_file`. Using `with` will keep the file open only for as long as we need it and allows us to avoid having to write code to explicitly close the file.

We can use the csv module's *reader()* function to iterate through the file line by line. The *reader()* function takes an optional *dialect* parameter that will allow us to specify dialect information extracted by a *Sniffer* if its available. Each line is represented by a list with elements corresponding to the different field values in the line. As we read through the file, we will append each line to the first list we created, `csv_data`.

In [44]:
```
# load data from CSV file
import csv

csv_data = []
# open the file for reading
with open('./data/01-attrition.csv', encoding='utf-8-sig') as csv_file:
    # create a csv reader using the exsiting dialect information
    reader = csv.reader(csv_file, dialect=dialect)
    # iterate through the csv's rows and append to the csv_data list
    for row in reader:
        csv_data.append(row)
```

We can iterate through `csv_data` to display the first few rows of data.

Lab 5 In the cell below, use a for-loop to display the first five rows of data in `csv_data`.

In [45]:
```
for row in csv_data[:5]:
    print(row)
```

```
['Age', 'Attrition', 'BusinessTravel', 'DailyRate', 'Department', 'Distan
ceFromHome', 'Education', 'EducationField', 'EmployeeCount', 'EmployeeNum
ber', 'EnvironmentSatisfaction', 'Gender', 'HourlyRate', 'JobInvolvemen
t', 'JobLevel', 'JobRole', 'JobSatisfaction', 'MaritalStatus', 'MonthlyIn
come', 'MonthlyRate', 'NumCompaniesWorked', 'Over18', 'OverTime', 'Percen
tSalaryHike', 'PerformanceRating', 'RelationshipSatisfaction', 'StandardH
ours', 'StockOptionLevel', 'TotalWorkingYears', 'TrainingTimesLastYear',
'WorkLifeBalance', 'YearsAtCompany', 'YearsInCurrentRole', 'YearsSinceLas
tPromotion', 'YearsWithCurrManager']
['41', 'Yes', 'Travel_Rarely', '1102', 'Sales', '1', '2', 'Life Science
s', '1', '1', '2', 'Female', '94', '3', '2', 'Sales Executive', '4', 'Sin
gle', '5993', '19479', '8', 'Y', 'Yes', '11', '3', '1', '80', '0', '8',
'0', '1', '6', '4', '0', '5']
['49', 'No', 'Travel_Frequently', '279', 'Research & Development', '8',
'1', 'Life Sciences', '1', '2', '3', 'Male', '61', '2', '2', 'Research Sc
ientist', '2', 'Married', '5130', '24907', '1', 'Y', 'No', '23', '4',
'4', '80', '1', '10', '3', '3', '10', '7', '1', '7']
['37', 'Yes', 'Travel_Rarely', '1373', 'Research & Development', '2',
'2', 'Other', '1', '4', '4', 'Male', '92', '2', '1', 'Laboratory Technici
an', '3', 'Single', '2090', '2396', '6', 'Y', 'Yes', '15', '3', '2', '8
0', '0', '7', '3', '3', '0', '0', '0', '0']
['33', 'No', 'Travel_Frequently', '1392', 'Research & Development', '3',
'4', 'Life Sciences', '1', '5', '4', 'Female', '56', '3', '1', 'Research
Scientist', '3', 'Married', '2909', '23159', '1', 'Y', 'Yes', '11', '3',
'3', '80', '0', '8', '3', '3', '8', '7', '3', '0']
```

Examining the output, we see that the first five elements of the `csv_data` list are themselves lists; each of these lists corresponds to a line from the original file. Each of these lists, in turn, have elements that correspond to individual values.

There are a variety of ways we can structure this data for convenient access. One way is to create a list of dictionaries where each dictionary corresponds to a row of data with field names as dictionary keys and row data as dictionary values. To construct this list of dictionaries, we'll iterate through `csv_data`. As a first step, lets assign the first row to a new variable named `fields`.

In [46]:
```
fields = csv_data[0]
fields
```

Out[46]:
```
['Age',
 'Attrition',
 'BusinessTravel',
 'DailyRate',
 'Department',
 'DistanceFromHome',
 'Education',
 'EducationField',
 'EmployeeCount',
 'EmployeeNumber',
 'EnvironmentSatisfaction',
 'Gender',
 'HourlyRate',
 'JobInvolvement',
 'JobLevel',
 'JobRole',
 'JobSatisfaction',
 'MaritalStatus',
 'MonthlyIncome',
 'MonthlyRate',
 'NumCompaniesWorked',
 'Over18',
 'OverTime',
 'PercentSalaryHike',
 'PerformanceRating',
 'RelationshipSatisfaction',
 'StandardHours',
 'StockOptionLevel',
 'TotalWorkingYears',
 'TrainingTimesLastYear',
 'WorkLifeBalance',
 'YearsAtCompany',
 'YearsInCurrentRole',
 'YearsSinceLastPromotion',
 'YearsWithCurrManager']
```

Next, we'll create a list to store the dictionaries representing each row; we'll assign this list to the `hr_data` variable. We next iterate through the remaining rows of `csv_data` and create a dictionary for each row. Recall that each row is itself stored as a list with elements using the `enumerate()` function. The `enumerate()` function will return both the index of an element in a list as well as the element. Having the index will allow us to us to access the corresponding field name from the `fields` list. In the newly created dictionary, we'll pair the field names and field values. Once we've finished iterating through the row, we'll store the dictionary in the `hr_data` list.

In [47]:
```
hr_data = []

# iterate through the remaining data rows
for row in csv_data[1:]:
    row_dictionary = {}

    # iterate through the row elements using
    for index, element in enumerate(row):
        # get the corresponding field name
        field_name = fields[index]
        # store the element with its field
        row_dictionary[field_name] = element

    #add the dictionary to the list
    hr_data.append(row_dictionary)

# diplay the data from the second row of data
hr_data[1]
```

Out[47]:
```
{'Age': '49',
 'Attrition': 'No',
 'BusinessTravel': 'Travel_Frequently',
 'DailyRate': '279',
 'Department': 'Research & Development',
 'DistanceFromHome': '8',
 'Education': '1',
 'EducationField': 'Life Sciences',
 'EmployeeCount': '1',
 'EmployeeNumber': '2',
 'EnvironmentSatisfaction': '3',
 'Gender': 'Male',
 'HourlyRate': '61',
 'JobInvolvement': '2',
 'JobLevel': '2',
 'JobRole': 'Research Scientist',
 'JobSatisfaction': '2',
 'MaritalStatus': 'Married',
 'MonthlyIncome': '5130',
 'MonthlyRate': '24907',
 'NumCompaniesWorked': '1',
 'Over18': 'Y',
 'OverTime': 'No',
 'PercentSalaryHike': '23',
 'PerformanceRating': '4',
 'RelationshipSatisfaction': '4',
 'StandardHours': '80',
 'StockOptionLevel': '1',
 'TotalWorkingYears': '10',
 'TrainingTimesLastYear': '3',
 'WorkLifeBalance': '3',
 'YearsAtCompany': '10',
 'YearsInCurrentRole': '7',
 'YearsSinceLastPromotion': '1',
 'YearsWithCurrManager': '7'}
```

This structure will allow us to easily access data for each record. For example, we can print the ages

of the first ten records using the following code.

In [48]:
```
for record in hr_data[:10]:
    print(record['Age'])
```

```
41
49
37
33
27
32
59
30
38
36
```

While this is an improvement over the list-of-lists structure we initially created when loading the CSV data, there are still issues with the way we've stored the data. One issue is that all the data is stored as strings. While this is fine for some fields such as `Business Travel` or `Department`, it would be preferable to store other fields like `Age` as numeric values. To demonstrate this problem, let's try to calculate the mean age of the first ten records. This code will result in an error.

In [49]:
```
total_age = 0

# iterate through the first 10 records and add each age to the total
for record in hr_data[:10]:
    total_age += record['Age']


total_age/10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-8c2527541582> in <module>()
      3 # iterate through the first 10 records and add each age to the total
      4 for record in hr_data[:10]:
----> 5     total_age += record['Age']
      6
      7

TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Our code caused an error because we tried to add an integer, 0, and a string, the value associated with `Age`. To fix this we would have had to convert the value when we loaded the data or when we tried to do a calculation with it as demonstrated below.

In [50]:
```
total_age = 0

# iterate through the first 10 records and add each age to the total
for record in hr_data[:10]:
    # convert age value to an integer then add to total_age
    total_age += int(record['Age'])


total_age/10
```

Out[50]:
```
38.2
```

We could continue to refine the code we wrote to load the data to include necessary data type conversions. However, third party libraries exist that try to do this automatically. One such library is pandas. With pandas loaded, we can use its `read_csv()` function to load data from a file; we can specify the location of the file as an argument to the function. The `read_csv` method returns a DataFrame object which we can store in a variable. To display the first few rows of a DataFrame, we can use the `head()` method.

In [51]:
```
# load data
hr_data = pd.read_csv('data/01-attrition.csv')

# display first 5 rows of data
hr_data.head(5)
```

Out[51]:

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Educatio
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life Sc
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life Sc
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life Sc
4	27	No	Travel_Rarely	591	Research & Development	2	1	M

5 rows × 35 columns

Notice that the output is much easier to view and understand than what was displayed when we loaded the contents of the file as a list of lists; this is one of the advantages to using pandas to work with data. This is also do to how pandas integrates with the notebook software, Jupyter.

By default, pandas will only display twenty columns. If there are more than twenty columns, as in this example, pandas will indicate this with ellipses. To increase the number of columns, we can use the `set_option()` function, specifying the option we want to change as the first argument and its new value as the second. The code below increases the number of displayed columns to fifty then displays the first five rows again.

In [52]:
```
# increase number of displayed columns
pd.set_option('display.max_column', 50)
```

Lab 6 In the cell below, use the DataFrame's *head()* method to display the first five rows of data.

In [53]:
```
hr_data.head(5)
```

Out[53]:

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Educatio
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life Sc
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life Sc
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life Sc
4	27	No	Travel_Rarely	591	Research & Development	2	1	M

When loading data, pandas will automatically assign an index to each row. The index appears before the first named column above. To access a row we can use the `iloc` attribute to access a row by specifying its position in the DataFrame or the `loc` attribute by specifying the label associated with the row in the DataFrame's index; when the index is automatically assigned these two properties can be used interchangeably.

In [54]:
```
# get the third row
hr_data.iloc[2]
```

Out[54]:
```
Age                                37
Attrition                         Yes
BusinessTravel          Travel_Rarely
DailyRate                       1373
Department      Research & Development
DistanceFromHome                   2
Education                          2
EducationField                 Other
EmployeeCount                      1
EmployeeNumber                     4
EnvironmentSatisfaction            4
Gender                          Male
HourlyRate                        92
JobInvolvement                     2
JobLevel                           1
JobRole          Laboratory Technician
JobSatisfaction                    3
MaritalStatus                 Single
MonthlyIncome                  2090
MonthlyRate                    2396
NumCompaniesWorked                 6
Over18                             Y
OverTime                         Yes
PercentSalaryHike                 15
PerformanceRating                  3
RelationshipSatisfaction           2
StandardHours                     80
StockOptionLevel                   0
TotalWorkingYears                  7
TrainingTimesLastYear              3
WorkLifeBalance                    3
YearsAtCompany                     0
YearsInCurrentRole                 0
YearsSinceLastPromotion            0
YearsWithCurrManager               0
Name: 2, dtype: object
```

In [55]:
```
# get the row with index label "2"
hr_data.loc[2]
```

Out[55]:
```
Age                                37
Attrition                         Yes
BusinessTravel          Travel_Rarely
DailyRate                       1373
Department      Research & Development
DistanceFromHome                   2
Education                          2
EducationField                 Other
EmployeeCount                      1
EmployeeNumber                     4
EnvironmentSatisfaction            4
Gender                          Male
HourlyRate                        92
JobInvolvement                     2
JobLevel                           1
JobRole          Laboratory Technician
JobSatisfaction                    3
MaritalStatus                 Single
MonthlyIncome                  2090
MonthlyRate                    2396
NumCompaniesWorked                 6
Over18                             Y
OverTime                         Yes
PercentSalaryHike                 15
PerformanceRating                  3
RelationshipSatisfaction           2
StandardHours                     80
StockOptionLevel                   0
TotalWorkingYears                  7
TrainingTimesLastYear              3
WorkLifeBalance                    3
YearsAtCompany                     0
YearsInCurrentRole                 0
YearsSinceLastPromotion            0
YearsWithCurrManager               0
Name: 2, dtype: object
```

Now that we have the data loaded into a DataFrame, we can do a variety of things including calculating summary statistics or creating visualizations. For example, we can calculate the mean age of the first ten records using the following code.

In [56]:
```
# mean age of
hr_data.iloc[:10]['Age'].mean()
```

Out[56]:
```
38.2
```

To calculate the average age, we didn't have to restructure the data or explicitly change the data type. This is one of the advantages of using a library like pandas.

To explain the line of code above, let's look at each part. The `hr_data` variable refers to the DataFrame we created using the CSV data. We can specify multiple rows by using Python's slice notation with `iloc`. To specify a particualr column, we can use bracket notation along with the columns name. When we access a subset of the rows, the data structure is still a DataFrame. When we access a specific column, the data structure used is a related one-dimensional structure known as a Series (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html). Both DataFrames and Series have methods that can be used to manipulate the data stored within them.

In [57]:
```
# first ten rows, a DataFrame
hr_data.iloc[:10]
```

Out[57]:

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Educatio
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life Sc
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life Sc
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life Sc
4	27	No	Travel_Rarely	591	Research & Development	2	1	M
5	32	No	Travel_Frequently	1005	Research & Development	2	2	Life Sc
6	59	No	Travel_Rarely	1324	Research & Development	3	3	M
7	30	No	Travel_Rarely	1358	Research & Development	24	1	Life Sc
8	38	No	Travel_Frequently	216	Research & Development	23	3	Life Sc
9	36	No	Travel_Rarely	1299	Research & Development	27	3	M

In [58]:
```
# Age column of first ten rows, a Series
hr_data.iloc[:10]['Age']
```

Out[58]:
```
0    41
1    49
2    37
3    33
4    27
5    32
6    59
7    30
8    38
9    36
Name: Age, dtype: int64
```

In [59]:
```
# mean of the age of the first ten rows
hr_data.iloc[:10]['Age'].mean()
```

Out[59]:
```
38.2
```

Because slicing is such a common operation, pandas supports specifying slices of rows using bracket notation on a DataFrame directly without having to use the `iloc` attribute.

In [60]:
```
hr_data[:10]['Age'].mean()
```

Out[60]:
```
38.2
```

We can also access columns within a DataFrame with the dot operator.

Lab 7 In the cell below, calculate the mean of the first ten values in the `Age` column. Use the dot operator rather than bracket notation to access the `Age` column. The result should be the same as the previous result.

In [61]:
```
hr_data[:10].Age.mean()
```

Out[61]:
```
38.2
```

We'll explore more of the features of a DataFrame later but let's continue looking at how we can load data from different sources.

Loading Data from a Database

It is often convenient to store data in a database rather than in individual files. There are a variety of Python libraries available (https://wiki.python.org/moin/DatabaseInterfaces) to interact with databases. One of the most common is SQLAlchemy (http://docs.sqlalchemy.org/en/latest/dialects/index.html) and pandas includes functionality to work with it. Before we can use SQLAlchemy, we should make sure it is installed using `pip`. Typically, we will install the libraries we expect to use before starting work in a notebook but for completeness, this step is included in this notebook.

In [ ]:
```
!pip install sqlalchemy
```

For convenience, we'll use a SQLite database for this example. SQLite databases are convenient because they can be stored as a single file. The process we'll use to connect to and load data from the SQLite database can be used with other database engines such as Microsoft SQL Server or MySQL.

In this example, we'll load data from a commonly used example database, the Chinook database (https://chinookdatabase.codeplex.com/), which contains data for a fictitious media store. An diagram of the database structure appears below.

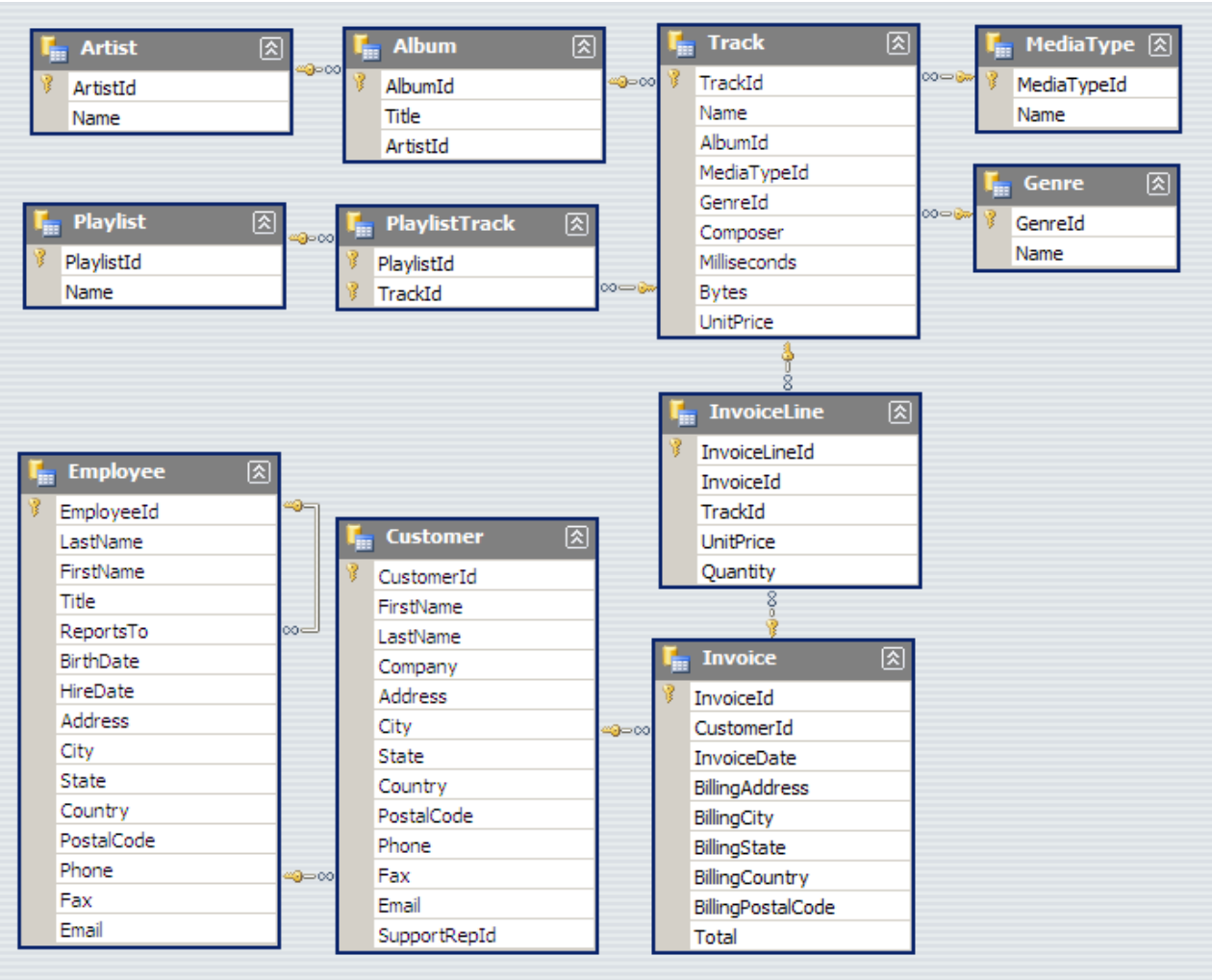

Diagram of the Chinook Database

In order to load data from a database using pandas, we must first create a connection to the database. This is done using SQLAlchemy's `create_engine()` function. Using a resource identifier, we can specify the location to the SQLite database file as an argument when we call the function; see the pandas documentation (https://pandas.pydata.org/pandas-docs/stable/io.html#engine-connection-examples) and SQLAlchemy documentation (http://docs.sqlalchemy.org/en/latest/core/engines.html) for examples of how to connect to other database engines.

In [63]:
```
# import the create_engine function
from sqlalchemy import create_engine

# create a database connection
engine = create_engine('sqlite:///data/01-chinook.sqlite')
```

We can use SQLAlchemy alone to load data from the database. One we've defined an engine, we can use its `connect` method to establish a connection to the database and query the database

with the connection. To demonstrate how we can query the database, we'll query the database for the first five records from the inner join on the `Invoice` and `Customer` tables ordered by `InvoiceID`. First we'll store the query as a string for clarity, then query the database, and finally iterate through the result to display each record.

In [64]:
```
# database query
query = """
SELECT *
FROM Invoice INNER JOIN Customer
ON Invoice.CustomerId = Customer.CustomerId
ORDER BY Invoice.InvoiceID
LIMIT 5
"""

# open database connection, close when finished
with engine.connect() as connection:
    # execute query
    result = connection.execute(query)
    # iterate through results
    for row in result:
        print(row)
```

```
(1, 2, '2009-01-01 00:00:00', 'Theodor-Heuss-Straße 34', 'Stuttgart', Non
e, 'Germany', '70174', 1.98, 2, 'Leonie', 'Köhler', None, 'Theodor-Heuss-
Straße 34', 'Stuttgart', None, 'Germany', '70174', '+49 0711 2842222', No
ne, 'leonekohler@surfeu.de', 5)
(2, 4, '2009-01-02 00:00:00', 'Ullevålsveien 14', 'Oslo', None, 'Norway',
'0171', 3.96, 4, 'Bjørn', 'Hansen', None, 'Ullevålsveien 14', 'Oslo', Non
e, 'Norway', '0171', '+47 22 44 22 22', None, 'bjorn.hansen@yahoo.no', 4)
(3, 8, '2009-01-03 00:00:00', 'Grétrystraat 63', 'Brussels', None, 'Belgi
um', '1000', 5.94, 8, 'Daan', 'Peeters', None, 'Grétrystraat 63', 'Brusse
ls', None, 'Belgium', '1000', '+32 02 219 03 03', None, 'daan_peeters@app
le.be', 4)
(4, 14, '2009-01-06 00:00:00', '8210 111 ST NW', 'Edmonton', 'AB', 'Canad
a', 'T6G 2C7', 8.91, 14, 'Mark', 'Philips', 'Telus', '8210 111 ST NW', 'E
dmonton', 'AB', 'Canada', 'T6G 2C7', '+1 (780) 434-4554', '+1 (780) 434-5
565', 'mphilips12@shaw.ca', 5)
(5, 23, '2009-01-11 00:00:00', '69 Salem Street', 'Boston', 'MA', 'USA',
'2113', 13.86, 23, 'John', 'Gordon', None, '69 Salem Street', 'Boston',
'MA', 'USA', '2113', '+1 (617) 522-1333', None, 'johngordon22@yahoo.com',
4)
```

We can see that each record is represented by a tuple with elements corresponding to field values; this is similar to the structure of the data we initially had when working with the CSV file. We could write code to make it easier to work with this data or we could rely instead on a library like pandas.

With a connection to the database, we can query the database for data and store the result in a DataFrame using pandas' `read_sql_query()` function. For this example, we'll load the entire result of the join.

In [65]:
```
# query
query = """
SELECT *
FROM Invoice INNER JOIN Customer
ON Invoice.CustomerId = Customer.CustomerId
ORDER BY Invoice.InvoiceID
"""
# query database
invoice_customer = pd.read_sql_query(query, engine)

# display first five rows
invoice_customer.head(5)
```

Out[65]:

	InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingI
0	1	2	2009-01-01 00:00:00	Theodor-Heuss-Straße 34	Stuttgart	None	Germany	
1	2	4	2009-01-02 00:00:00	Ullevålsveien 14	Oslo	None	Norway	
2	3	8	2009-01-03 00:00:00	Grétrystraat 63	Brussels	None	Belgium	
3	4	14	2009-01-06 00:00:00	8210 111 ST NW	Edmonton	AB	Canada	
4	5	23	2009-01-11 00:00:00	69 Salem Street	Boston	MA	USA	

Loading Data from an API

Many websites that provide access to data often provide a means of accessing that data programmatically using an application programming interface (https://www.programmableweb.com/api-university/what-are-apis-and-how-do-they-work) or API. A common way to access an API is by making HTTP (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) requests of a server in much the same way a web browser would. If the request is valid, the server hosting the API will respond with the requested data. The response is often JSON (https://en.wikipedia.org/wiki/JSON) or XML (https://en.wikipedia.org/wiki/XML) text.

To demonstrate this, let's look at the IEX API (https://iextrading.com/developer/docs/#getting-started), a free API providing stock market data. To make an HTTP request, we'll use the popular Requests (http://docs.python-requests.org/en/master/) library. In addition to providing the ability to make HTTP requests, the Requests library also includes functionality to convert JSON data into Python objects like lists and dictionaries. First, we'll request historic data for a specific stock then display the response as plain text.

In [66]:
```
# import requests module to make HTTP requests
import requests

# Make an HTTP GET request and store the response
response = requests.get("https://api.iextrading.com/1.0/stock/aapl/chart")

# display the response content
response.content
```

Out[66]:

```
b'[{"date":"2018-03-01","open":178.54,"high":179.775,"low":172.66,"clos
e":175,"volume":48801970,"unadjustedVolume":48801970,"change":-3.12,"chan
gePercent":-1.752,"vwap":176.0889,"label":"Mar 1","changeOverTime":0},{"d
ate":"2018-03-02","open":172.8,"high":176.3,"low":172.45,"close":176.2
1,"volume":38453950,"unadjustedVolume":38453950,"change":1.21,"changePerc
ent":0.691,"vwap":174.7679,"label":"Mar 2","changeOverTime":0.00691428571
428576},{"date":"2018-03-05","open":175.21,"high":177.74,"low":174.52,"cl
ose":176.82,"volume":28401366,"unadjustedVolume":28401366,"change":0.6
1,"changePercent":0.346,"vwap":176.5212,"label":"Mar 5","changeOverTime":
0.010399999999999961},{"date":"2018-03-06","open":177.91,"high":178.25,"l
ow":176.13,"close":176.67,"volume":23788506,"unadjustedVolume":2378850
6,"change":-0.15,"changePercent":-0.085,"vwap":177.0443,"label":"Mar
6","changeOverTime":0.009542857142857072},{"date":"2018-03-07","open":17
4.94,"high":175.85,"low":174.27,"close":175.03,"volume":31703462,"unadjus
tedVolume":31703462,"change":-1.64,"changePercent":-0.928,"vwap":175.068
4,"label":"Mar 7","changeOverTime":0.00017142857142857793},{"date":"2018-
03-08","open":175.48,"high":177.12,"low":175.07,"close":176.94,"volume":2
3774107,"unadjustedVolume":23774107,"change":1.91,"changePercent":1.09
1,"vwap":176.1886,"label":"Mar 8","changeOverTime":0.011085714285714273},
{"date":"2018-03-09","open":177.96,"high":180,"low":177.39,"close":179.9
8,"volume":32185162,"unadjustedVolume":32185162,"change":3.04,"changePerc
ent":1.718,"vwap":179.1455,"label":"Mar 9","changeOverTime":0.02845714285
71428},{"date":"2018-03-12","open":180.29,"high":182.39,"low":180.21,"clo
se":181.72,"volume":32207081,"unadjustedVolume":32207081,"change":1.74,"c
hangePercent":0.967,"vwap":181.657,"label":"Mar 12","changeOverTime":0.03
84},{"date":"2018-03-13","open":182.59,"high":183.5,"low":179.24,"close":
179.97,"volume":31693529,"unadjustedVolume":31693529,"change":-1.75,"chan
gePercent":-0.963,"vwap":181.586,"label":"Mar 13","changeOverTime":0.0283
99999999999995},{"date":"2018-03-14","open":180.32,"high":180.52,"low":17
7.81,"close":178.44,"volume":29368356,"unadjustedVolume":29368356,"chang
e":-1.53,"changePercent":-0.85,"vwap":178.7619,"label":"Mar 14","changeOv
erTime":0.019657142857142845},{"date":"2018-03-15","open":178.5,"high":18
0.24,"low":178.0701,"close":178.65,"volume":22743798,"unadjustedVolume":2
2743798,"change":0.21,"changePercent":0.118,"vwap":179.0623,"label":"Mar
15","changeOverTime":0.02085714285714289},{"date":"2018-03-16","open":17
8.65,"high":179.12,"low":177.62,"close":178.02,"volume":39404688,"unadjus
tedVolume":39404688,"change":-0.63,"changePercent":-0.353,"vwap":178.338
3,"label":"Mar 16","changeOverTime":0.017257142857142915},{"date":"2018-0
3-19","open":177.32,"high":177.47,"low":173.66,"close":175.3,"volume":334
46771,"unadjustedVolume":33446771,"change":-2.72,"changePercent":-1.52
8,"vwap":175.2302,"label":"Mar 19","changeOverTime":0.001714285714285779
2},{"date":"2018-03-20","open":175.24,"high":176.8,"low":174.94,"close":1
75.24,"volume":19649350,"unadjustedVolume":19649350,"change":-0.06,"chang
ePercent":-0.034,"vwap":175.7335,"label":"Mar 20","changeOverTime":0.0013
714285714286234},{"date":"2018-03-21","open":175.04,"high":175.09,"low":1
71.26,"close":171.27,"volume":37054935,"unadjustedVolume":37054935,"chang
e":-3.97,"changePercent":-2.265,"vwap":173.1398,"label":"Mar 21","changeO
verTime":-0.021314285714285657},{"date":"2018-03-22","open":170,"high":17
```

2.68,"low":168.6,"close":168.85,"volume":41490767,"unadjustedVolume":4149
0767,"change":-2.42,"changePercent":-1.413,"vwap":170.167,"label":"Mar 2
2","changeOverTime":-0.03514285714285718},{"date":"2018-03-23","open":16
8.39,"high":169.92,"low":164.94,"close":164.94,"volume":41028784,"unadjus
tedVolume":41028784,"change":-3.91,"changePercent":-2.316,"vwap":167.580
6,"label":"Mar 23","changeOverTime":-0.057485714285714 3},{"date":"2018-03
-26","open":168.07,"high":173.1,"low":166.44,"close":172.77,"volume":3754
1236,"unadjustedVolume":37541236,"change":7.83,"changePercent":4.747,"vwa
p":170.0402,"label":"Mar 26","changeOverTime":-0.012742857142857084},{"da
te":"2018-03-27","open":173.68,"high":175.15,"low":166.92,"close":168.3
4,"volume":40922579,"unadjustedVolume":40922579,"change":-4.43,"changePer
cent":-2.564,"vwap":171.5377,"label":"Mar 27","changeOverTime":-0.0380571
4285714284},{"date":"2018-03-28","open":167.25,"high":170.02,"low":165.1
9,"close":166.48,"volume":41668545,"unadjustedVolume":41668545,"change":-
1.86,"changePercent":-1.105,"vwap":167.1902,"label":"Mar 28","changeOverT
ime":-0.048685714285714346},{"date":"2018-03-29","open":167.805,"high":17
1.75,"low":166.9,"close":167.78,"volume":38398505,"unadjustedVolume":3839
8505,"change":1.3,"changePercent":0.781,"vwap":169.2088,"label":"Mar 2
9","changeOverTime":-0.04125714285714285}]'

Examining the response content or reading the API documentation reveals that the data is JSON formatted. We can use the `json()` method associated with the response to process the data.

In [67]:
```
# process JSON data and convert to python lists and dictionaries
stock_data = response.json()

# display data
stock_data
```

Out[67]:
```
[{'change': -3.12,
  'changeOverTime': 0,
  'changePercent': -1.752,
  'close': 175,
  'date': '2018-03-01',
  'high': 179.775,
  'label': 'Mar 1',
  'low': 172.66,
  'open': 178.54,
  'unadjustedVolume': 48801970,
  'volume': 48801970,
  'vwap': 176.0889},
 {'change': 1.21,
  'changeOverTime': 0.00691428571428576,
  'changePercent': 0.691,
  'close': 176.21,
  'date': '2018-03-02',
  'high': 176.3,
  'label': 'Mar 2',
```

The data is stored as a list of dictionaries with each dictionary corresponding to a specific data. Each dictionary also contains information related to the stock price. While we could work with the data in this structure, we could also use pandas to make the HTTP request and process the response into a DataFrame.

In [68]:
```
# use pandas to load API data
stock_history = pd.read_json("https://api.iextrading.com/1.0/stock/aapl/char

# display historic stock data
stock_history
```

Out[68]:

	change	changeOverTime	changePercent	close	date	high	label	low	open	una
0	-3.12	0.000000	-1.752	175.00	2018-03-01	179.775	Mar 1	172.6600	178.540	
1	1.21	0.006914	0.691	176.21	2018-03-02	176.300	Mar 2	172.4500	172.800	
2	0.61	0.010400	0.346	176.82	2018-03-05	177.740	Mar 5	174.5200	175.210	
3	-0.15	0.009543	-0.085	176.67	2018-03-06	178.250	Mar 6	176.1300	177.910	
4	-1.64	0.000171	-0.928	175.03	2018-03-07	175.850	Mar 7	174.2700	174.940	
5	1.91	0.011086	1.091	176.94	2018-03-08	177.120	Mar 8	175.0700	175.480	
6	3.04	0.028457	1.718	179.98	2018-03-09	180.000	Mar 9	177.3900	177.960	
7	1.74	0.038400	0.967	181.72	2018-03-12	182.390	Mar 12	180.2100	180.290	
8	-1.75	0.028400	-0.963	179.97	2018-03-13	183.500	Mar 13	179.2400	182.590	
9	-1.53	0.019657	-0.850	178.44	2018-03-14	180.520	Mar 14	177.8100	180.320	
10	0.21	0.020857	0.118	178.65	2018-03-15	180.240	Mar 15	178.0701	178.500	
11	-0.63	0.017257	-0.353	178.02	2018-03-16	179.120	Mar 16	177.6200	178.650	
12	-2.72	0.001714	-1.528	175.30	2018-03-19	177.470	Mar 19	173.6600	177.320	
13	-0.06	0.001371	-0.034	175.24	2018-03-20	176.800	Mar 20	174.9400	175.240	
14	-3.97	-0.021314	-2.265	171.27	2018-03-21	175.090	Mar 21	171.2600	175.040	
15	-2.42	-0.035143	-1.413	168.85	2018-03-22	172.680	Mar 22	168.6000	170.000	
16	-3.91	-0.057486	-2.316	164.94	2018-03-23	169.920	Mar 23	164.9400	168.390	
17	7.83	-0.012743	4.747	172.77	2018-03-26	173.100	Mar 26	166.4400	168.070	
18	-4.43	-0.038057	-2.564	168.34	2018-03-27	175.150	Mar 27	166.9200	173.680	
19	-1.86	-0.048686	-1.105	166.48	2018-03-28	170.020	Mar 28	165.1900	167.250	

	change	changeOverTime	changePercent	close	date	high	label	low	open	una
20	1.30	-0.041257	0.781	167.78	2018-03-29	171.750	Mar 29	166.9000	167.805	

Scraping Data

The final method of acquiring data that we'll examine is known as scraping (https://en.wikipedia.org/wiki/Web_scraping). Scraping involves extracting data from a source that is typically meant for human use rather than programmatic manipulation. Often, data is extracted from a website. Because many websites explicitly prohibit the use of scraping in their terms of service, we'll use a HTML file created for this example.

Before we extract the data, let's see what the page defined by the HTML looks like. To render HTML within a notebook, we will use the IPython (https://ipython.org/) `display()` function and `HTML` class. Jupyter notebooks are closely related to IPython; the Python code written in a Jypter notebook is executed using IPython so we don't need to make sure IPython library is installed before importing parts of it.

In [69]:
```
# display HTML content
from IPython.display import HTML
HTML(filename="./data/01-table.html")
```

Out[69]:

Salary information

Name	Position	Office	Age	Start date	Salary
Tiger Nixon	System Architect	Edinburgh	61	2011/04/25	$320,800
Garrett Winters	Accountant	Tokyo	63	2011/07/25	$170,750
Ashton Cox	Junior Technical Author	San Francisco	66	2009/01/12	$86,000
Cedric Kelly	Senior Javascript Developer	Edinburgh	22	2012/03/29	$433,060
Airi Satou	Accountant	Tokyo	33	2008/11/28	$162,700
Brielle Williamson	Integration Specialist	New York	61	2012/12/02	$372,000
Herrod Chandler	Sales Assistant	San Francisco	59	2012/08/06	$137,500
Rhona Davidson	Integration Specialist	Tokyo	55	2010/10/14	$327,900
Colleen Hurst	Javascript Developer	San Francisco	39	2009/09/15	$205,500
Sonya Frost	Software Engineer	Edinburgh	23	2008/12/13	$103,600
Jena Gaines	Office Manager	London	30	2008/12/19	$90,560
Quinn Flynn	Support Lead	Edinburgh	22	2013/03/03	$342,000
Charde Marshall	Regional Director	San Francisco	36	2008/10/16	$470,600
Haley Kennedy	Senior Marketing Designer	London	43	2012/12/18	$313,500
Tatyana Fitzpatrick	Regional Director	London	19	2010/03/17	$385,750
Michael Silva	Marketing Designer	London	66	2012/11/27	$198,500
Paul Byrd	Chief Financial Officer (CFO)	New York	64	2010/06/09	$725,000
Gloria Little	Systems Administrator	New York	59	2009/04/10	$237,500
Bradley Greer	Software Engineer	London	41	2012/10/13	$132,000
Dai Rios	Personnel Lead	Edinburgh	35	2012/09/26	$217,500
Jenette Caldwell	Development Lead	New York	30	2011/09/03	$345,000
Yuri Berry	Chief Marketing Officer (CMO)	New York	40	2009/06/25	$675,000
Caesar Vance	Pre-Sales Support	New York	21	2011/12/12	$106,450
Doris Wilder	Sales Assistant	Sidney	23	2010/09/20	$85,600
Angelica Ramos	Chief Executive Officer (CEO)	London	47	2009/10/09	$1,200,000
Gavin Joyce	Developer	Edinburgh	42	2010/12/22	$92,575
Jennifer Chang	Regional Director	Singapore	28	2010/11/14	$357,650
Brenden Wagner	Software Engineer	San Francisco	28	2011/06/07	$206,850
Fiona Green	Chief Operating Officer (COO)	San Francisco	48	2010/03/11	$850,000
Shou Itou	Regional Marketing	Tokyo	20	2011/08/14	$163,000
Name	**Position**	**Office**	**Age**	**Start date**	**Salary**

Name	Position	Office	Age	Start date	Salary
Michelle House	Integration Specialist	Sidney	37	2011/06/02	$95,400
Suki Burks	Developer	London	53	2009/10/22	$114,500
Prescott Bartlett	Technical Author	London	27	2011/05/07	$145,000
Gavin Cortez	Team Leader	San Francisco	22	2008/10/26	$235,500
Martena Mccray	Post-Sales support	Edinburgh	46	2011/03/09	$324,050
Unity Butler	Marketing Designer	San Francisco	47	2009/12/09	$85,675
Howard Hatfield	Office Manager	San Francisco	51	2008/12/16	$164,500
Hope Fuentes	Secretary	San Francisco	41	2010/02/12	$109,850
Vivian Harrell	Financial Controller	San Francisco	62	2009/02/14	$452,500
Timothy Mooney	Office Manager	London	37	2008/12/11	$136,200
Jackson Bradshaw	Director	New York	65	2008/09/26	$645,750
Olivia Liang	Support Engineer	Singapore	64	2011/02/03	$234,500
Bruno Nash	Software Engineer	London	38	2011/05/03	$163,500
Sakura Yamamoto	Support Engineer	Tokyo	37	2009/08/19	$139,575
Thor Walton	Developer	New York	61	2013/08/11	$98,540
Finn Camacho	Support Engineer	San Francisco	47	2009/07/07	$87,500
Serge Baldwin	Data Coordinator	Singapore	64	2012/04/09	$138,575
Zenaida Frank	Software Engineer	New York	63	2010/01/04	$125,250
Zorita Serrano	Software Engineer	San Francisco	56	2012/06/01	$115,000
Jennifer Acosta	Junior Javascript Developer	Edinburgh	43	2013/02/01	$75,650
Cara Stevens	Sales Assistant	New York	46	2011/12/06	$145,600
Hermione Butler	Regional Director	London	47	2011/03/21	$356,250
Lael Greer	Systems Administrator	London	21	2009/02/27	$103,500
Jonas Alexander	Developer	San Francisco	30	2010/07/14	$86,500
Shad Decker	Regional Director	Edinburgh	51	2008/11/13	$183,000
Michael Bruce	Javascript Developer	Singapore	29	2011/06/27	$183,000
Donna Snider	Customer Support	New York	27	2011/01/25	$112,000
Name	**Position**	**Office**	**Age**	**Start date**	**Salary**

We can see that the HTML file contains salary data. Though we've rendered the HTML content, we haven't extracted the data itself. There are a variety of web scraping libraries available for Python such as Beautiful Soup (https://www.crummy.com/software/BeautifulSoup/) but we'll rely on pandas. The pandas `read_html()` function can be used to extract data stored in HTML tables. First, let's make sure that the data is, in fact, stored in an HTML table. To do this, we'll display part of the content of the HTML file as plain text.

In [70]:
```
# open the html file
with open("./data/01-table.html", encoding="UTF-8") as html_file:
    # read the entire file
    html_data = html_file.readlines()

# display the first 40 lines
html_data[:40]
```

Out[70]:
```
['<!DOCTYPE html>\n',
 '<html>\n',
 '\n',
 '<head>\n',
 '    <meta charset="utf-8">\n',
 '    <title>Table Examplpe</title>\n',
 '    <meta name="description" content="An HTML table exmaple">\n',
 '</head>\n',
 '\n',
 '<body>\n',
 '    <h4>Salary information</h4>\n',
 '    <table>\n',
 '        <thead>\n',
 '            <tr>\n',
 '                <th>Name</th>\n',
 '                <th>Position</th>\n',
 '                <th>Office</th>\n',
 '                <th>Age</th>\n',
 '                <th>Start date</th>\n',
 '                <th>Salary</th>\n',
 '            </tr>\n',
 '        </thead>\n',
 '        <tfoot>\n',
 '            <tr>\n',
 '                <th>Name</th>\n',
 '                <th>Position</th>\n',
 '                <th>Office</th>\n',
 '                <th>Age</th>\n',
 '                <th>Start date</th>\n',
 '                <th>Salary</th>\n',
 '            </tr>\n',
 '        </tfoot>\n',
 '        <tbody>\n',
 '            <tr>\n',
 '                <td>Tiger Nixon</td>\n',
 '                <td>System Architect</td>\n',
 '                <td>Edinburgh</td>\n',
 '                <td>61</td>\n',
 '                <td>2011/04/25</td>\n',
 '                <td>$320,800</td>\n']
```

The HTML includes at least one `table` element so we'll be able to use pandas to extract data.

The pandas `read_html()` function will attempt to extract data from each table on a page. The value returned by `read_html()` is a list of DataFrames with each DataFrame corresponding to a table.

In [71]:
```
# extract table data from HTML
html_data = pd.read_html('./data/01-table.html')
```

The pandas *read_html()* returns is a list of DataFrames. Because the HTML document contained only one table, we are interested only in the first element in the returned list.

Lab 8 In the cell below, write the code to access the DataFrame corresponding to the first element from this list and display the first five rows of data.

In [72]:
```
salary_data = html_data[0]
salary_data.head(5)
```

Out[72]:

	Name	Position	Office	Age	Start date	Salary
0	Tiger Nixon	System Architect	Edinburgh	61	2011/04/25	$320,800
1	Garrett Winters	Accountant	Tokyo	63	2011/07/25	$170,750
2	Ashton Cox	Junior Technical Author	San Francisco	66	2009/01/12	$86,000
3	Cedric Kelly	Senior Javascript Developer	Edinburgh	22	2012/03/29	$433,060
4	Airi Satou	Accountant	Tokyo	33	2008/11/28	$162,700

Describe Data

Once we've gathered or accessed data from various sources, our next step might be to document the data itself, describing both the the source as well as information about fields including type of data, possible values, and description.

When working with a database, we can rely on the structure imposed by the database itself; when tables are created, data types for each field must be specified. Most database software provides a means of viewing the *Create* statement used to define a table. For example, all SQLite databases include a `sqlite_master` table that contains information about other tables in the database. Included in this table is a `sql` column that stores the SQL statements used to create the other tables. To retrieve the a table's *Create* statement we can use a query in the following form:

```
SELECT sql FROM sqlite_master WHERE type='table' AND name='{TABLE_NAME}'
```

where `{TABLE_NAME}` represents the name of the specific table we're interested in.

The following code connects to the database and queries for the *Create* statement for the `Customer` and `Invoice` tables. The query returns an iterable collection of rows where each element is a dictionary. The dictionary for each row has keys that correspond to field names and values that correspond to field values.

In [73]:
```
# query to be used
query_template = "SELECT sql FROM sqlite_master WHERE type='table' AND name=

# tables names
tables = ['Customer', 'Invoice']

# connect to the databalse
with engine.connect() as connection:
    for table in tables:
        # substitue table name into query template
        query = query_template.format(table_name=table)
        # execute the query
        result = connection.execute(query)
        # iterate through each record
        for record in result:
            # print the value associated with the 'sql' column
            print(record['sql'])
```

```
CREATE TABLE [Customer]
(
    [CustomerId] INTEGER  NOT NULL,
    [FirstName] NVARCHAR(40)  NOT NULL,
    [LastName] NVARCHAR(20)  NOT NULL,
    [Company] NVARCHAR(80),
    [Address] NVARCHAR(70),
    [City] NVARCHAR(40),
    [State] NVARCHAR(40),
    [Country] NVARCHAR(40),
    [PostalCode] NVARCHAR(10),
    [Phone] NVARCHAR(24),
    [Fax] NVARCHAR(24),
    [Email] NVARCHAR(60)  NOT NULL,
    [SupportRepId] INTEGER,
    CONSTRAINT [PK_Customer] PRIMARY KEY  ([CustomerId]),
    FOREIGN KEY ([SupportRepId]) REFERENCES [Employee] ([EmployeeId])
                ON DELETE NO ACTION ON UPDATE NO ACTION
)
CREATE TABLE [Invoice]
(
    [InvoiceId] INTEGER  NOT NULL,
    [CustomerId] INTEGER  NOT NULL,
    [InvoiceDate] DATETIME  NOT NULL,
    [BillingAddress] NVARCHAR(70),
    [BillingCity] NVARCHAR(40),
    [BillingState] NVARCHAR(40),
    [BillingCountry] NVARCHAR(40),
    [BillingPostalCode] NVARCHAR(10),
    [Total] NUMERIC(10,2)  NOT NULL,
    CONSTRAINT [PK_Invoice] PRIMARY KEY  ([InvoiceId]),
    FOREIGN KEY ([CustomerId]) REFERENCES [Customer] ([CustomerId])
                ON DELETE NO ACTION ON UPDATE NO ACTION
)
```

While this gives us information about the type of data stored in each field - for example, the `Total` field is a number with with ten digits where up to two can be used for decimal values - it doesn't tell us about the range of values.

For more insight into the data, we can use pandas. To see how pandas can be used to document data, let's load a new dataset. This dataset contains sanitized loan data from Lending Club (https://www.lendingclub.com/info/download-data.action).

Lab 9 Use the pandas *read_csv()* function to load loan data from the `./data/01-loan.csv` file and store the DataFrame in a variable named `loan_data`. Display the first few rows of the DataFrame.

In [74]:
```
loan_data = pd.read_csv("./data/01-loan.csv")
loan_data.head()
```

Out[74]:

	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade	e
0	5000.0	5000.0	4975.0	36 months	10.65%	162.87	B	B2	
1	2500.0	2500.0	2500.0	60 months	15.27%	59.83	C	C4	
2	2400.0	2400.0	2400.0	36 months	15.96%	84.33	C	C5	
3	10000.0	10000.0	10000.0	36 months	13.49%	339.31	C	C1	RES(
4	3000.0	3000.0	3000.0	60 months	12.69%	67.79	B	B5	U

First, we can get a full list of list of column names.

In [75]:
```
# use the tolist() method for an easier-to-read listing of column names
loan_data.columns.tolist()
```

Out[75]:
```
['loan_amnt',
 'funded_amnt',
 'funded_amnt_inv',
 'term',
 'int_rate',
 'installment',
 'grade',
 'sub_grade',
 'emp_title',
 'emp_length',
 'home_ownership',
 'annual_inc',
 'verification_status',
 'issue_d',
 'loan_status',
 'purpose',
 'title',
 'zip_code',
 'addr_state',
 'dti',
 'delinq_2yrs',
 'earliest_cr_line',
 'inq_last_6mths',
 'mths_since_last_delinq',
 'mths_since_last_record',
 'open_acc',
 'pub_rec',
 'revol_bal',
 'revol_util',
 'total_acc',
 'total_pymnt',
 'total_pymnt_inv',
 'total_rec_prncp',
 'total_rec_int',
 'total_rec_late_fee',
 'recoveries',
 'collection_recovery_fee',
 'last_pymnt_d',
 'last_pymnt_amnt',
 'pub_rec_bankruptcies']
```

Every DataFrame has a `describe()` method that provides summary statistics for each column. By default, `describe()` will only provide information for columns with numeric data. To see this, we start by loading another dataset containing sales data.

In [76]:
```
# descriptive stats
loan_data.describe()
```

Out[76]:

	loan_amnt	funded_amnt	funded_amnt_inv	installment	annual_inc	dti	c
count	42535.000000	42535.000000	42535.000000	42535.000000	4.253100e+04	42535.000000	425
mean	11089.722581	10821.585753	10139.830603	322.623063	6.913656e+04	13.373043	
std	7410.938391	7146.914675	7131.686447	208.927216	6.409635e+04	6.726315	
min	500.000000	500.000000	0.000000	15.670000	1.896000e+03	0.000000	
25%	5200.000000	5000.000000	4950.000000	165.520000	4.000000e+04	8.200000	
50%	9700.000000	9600.000000	8500.000000	277.690000	5.900000e+04	13.470000	
75%	15000.000000	15000.000000	14000.000000	428.180000	8.250000e+04	18.680000	
max	35000.000000	35000.000000	35000.000000	1305.190000	6.000000e+06	29.990000	

We can have pandas give some descriptive information for the non-numeric fields using the `include` parameter with a value of "all".

In [77]:
```
# descriptive stats for all columns
loan_data.describe(include="all")
```

Out[77]:

	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_g
count	42535.000000	42535.000000	42535.000000	42535	42535	42535.000000	42535	4
unique	NaN	NaN	NaN	2	394	NaN	7	
top	NaN	NaN	NaN	36 months	10.99%	NaN	B	
freq	NaN	NaN	NaN	31534	970	NaN	12389	
mean	11089.722581	10821.585753	10139.830603	NaN	NaN	322.623063	NaN	
std	7410.938391	7146.914675	7131.686447	NaN	NaN	208.927216	NaN	
min	500.000000	500.000000	0.000000	NaN	NaN	15.670000	NaN	
25%	5200.000000	5000.000000	4950.000000	NaN	NaN	165.520000	NaN	
50%	9700.000000	9600.000000	8500.000000	NaN	NaN	277.690000	NaN	
75%	15000.000000	15000.000000	14000.000000	NaN	NaN	428.180000	NaN	
max	35000.000000	35000.000000	35000.000000	NaN	NaN	1305.190000	NaN	

This tells us the number of unique values, the most frequently appearing value, and that value's frequency. It's usually helpful to know what the actual unique values ares. To do this, we'll have to iterate through the columns and work with each column individually as a Series. Series objects have a `unique` method that will display the Series' unique values.

When iterating through the columns, we'd like to see the unique values for columns with non-numeric data. There are a variety of ways (https://stackoverflow.com/questions/19900202/how-to-determine-whether-a-column-variable-is-numeric-or-not-in-pandas-numpy) to do this; one way is to

use the `is_string_dtype` function in the `pandas.api.types` module. Here, *dtype* means "data type".

In [78]:
```
# iterate through column names, display unique values for string data
for column in loan_data.columns:
    if pd.api.types.is_string_dtype(loan_data[column]):
        display(column, loan_data[column].value_counts())
```

```
'term'
```

```
 36 months    31534
 60 months    11001
Name: term, dtype: int64
```

```
'int_rate'
```

We can also view the data types of each column using the `dtypes` attribute of the DataFrame itself.

Lab 10 Use the *dtypes* attribute to display the data types for each column of the `loan_data` DataFrame

In [79]:
```
loan_data.dtypes
```

Out[79]:
```
loan_amnt                  float64
funded_amnt                float64
funded_amnt_inv            float64
term                        object
int_rate                    object
installment                float64
grade                       object
sub_grade                   object
emp_title                   object
emp_length                  object
home_ownership              object
annual_inc                 float64
verification_status         object
issue_d                     object
loan_status                 object
purpose                     object
title                       object
zip_code                    object
addr_state                  object
dti                        float64
delinq_2yrs                float64
earliest_cr_line            object
inq_last_6mths             float64
mths_since_last_delinq     float64
mths_since_last_record     float64
open_acc                   float64
pub_rec                    float64
revol_bal                  float64
revol_util                  object
total_acc                  float64
total_pymnt                float64
total_pymnt_inv            float64
total_rec_prncp            float64
total_rec_int              float64
total_rec_late_fee         float64
recoveries                 float64
collection_recovery_fee    float64
last_pymnt_d                object
last_pymnt_amnt            float64
pub_rec_bankruptcies       float64
dtype: object
```

The `object` data type is used for any non-numeric values. We can convert these values to a more appropriate data type later, if necessary.

Having a description of each column would be helpful. For columns where the meaning is unclear, we would consult the data owner for clarification. Lending Club (https://www.lendingclub.com/info/download-data.action) provides a data dictionary that contains a description for most of the columns that appear in their datasets. The contents of the dictionary are

loaded using the `read_excel()` function. We can specify a column as the index rather than have pandas generate an index automatically by specifying a column number to the `index_col` argument.

In [80]:
```
# load data dictionary
loan_dicitonary = pd.read_excel("./data/01-loan-dict.xlsx", index_col=0)
loan_dicitonary
```

Out[80]:

LoanStatNew	Description
acc_now_delinq	The number of accounts on which the borrower i...
acc_open_past_24mths	Number of trades opened in past 24 months.
addr_state	The state provided by the borrower in the loan...
all_util	Balance to credit limit on all trades
annual_inc	The self-reported annual income provided by th...
annual_inc_joint	The combined self-reported annual income provi...
application_type	Indicates whether the loan is an individual ap...
avg_cur_bal	Average current balance of all accounts
bc_open_to_buy	Total open to buy on revolving bankcards.
bc_util	Ratio of total current balance to high credit/...
chargeoff_within_12_mths	Number of charge-offs within 12 months
collection_recovery_fee	post charge off collection fee
collections_12_mths_ex_med	Number of collections in 12 months excluding m...
delinq_2yrs	The number of 30+ days past-due incidences of ...
delinq_amnt	The past-due amount owed for the accounts on w...
desc	Loan description provided by the borrower
dti	A ratio calculated using the borrower’s total ...
dti_joint	A ratio calculated using the co-borrowers' tot...
earliest_cr_line	The month the borrower's earliest reported cre...
emp_length	Employment length in years. Possible values ar...
emp_title	The job title supplied by the Borrower when ap...
fico_range_high	The upper boundary range the borrower’s FICO a...
fico_range_low	The lower boundary range the borrower’s FICO a...
funded_amnt	The total amount committed to that loan at tha...
funded_amnt_inv	The total amount committed by investors for th...
grade	LC assigned loan grade
home_ownership	The home ownership status provided by the borr...
id	A unique LC assigned ID for the loan listing.
il_util	Ratio of total current balance to high credit/...
initial_list_status	The initial listing status of the loan. Possib...
...	...

LoanStatNew	Description
sec_app_open_act_il	Number of currently active installment trades...
sec_app_num_rev_accts	Number of revolving accounts at time of appli...
sec_app_chargeoff_within_12_mths	Number of charge-offs within last 12 months a...
sec_app_collections_12_mths_ex_med	Number of collections within last 12 months e...
sec_app_mths_since_last_major_derog	Months since most recent 90-day or worse rati...
hardship_flag	Flags whether or not the borrower is on a hard...
hardship_type	Describes the hardship plan offering
hardship_reason	Describes the reason the hardship plan was off...
hardship_status	Describes if the hardship plan is active, pend...
deferral_term	Amount of months that the borrower is expected...
hardship_amount	The interest payment that the borrower has com...
hardship_start_date	The start date of the hardship plan period
hardship_end_date	The end date of the hardship plan period
payment_plan_start_date	The day the first hardship plan payment is due...
hardship_length	The number of months the borrower will make sm...
hardship_dpd	Account days past due as of the hardship plan ...
hardship_loan_status	Loan Status as of the hardship plan start date
orig_projected_additional_accrued_interest	The original projected additional interest amo...
hardship_payoff_balance_amount	The payoff balance amount as of the hardship p...
hardship_last_payment_amount	The last payment amount as of the hardship pla...
disbursement_method	The method by which the borrower receives thei...
debt_settlement_flag	Flags whether or not the borrower, who has cha...
debt_settlement_flag_date	The most recent date that the Debt_Settlement_...
settlement_status	The status of the borrower’s settlement plan. ...
settlement_date	The date that the borrower agrees to the settl...
settlement_amount	The loan amount that the borrower has agreed t...
settlement_percentage	The settlement amount as a percentage of the p...
settlement_term	The number of months that the borrower will be...
NaN	NaN
NaN	* Employer Title replaces Employer Name for al...

153 rows × 1 columns

We can use the dictionary to get descriptions for most of the columns in our DataFrame.

In [81]:
```
# print descriptions only for columns being used
for column in loan_data.columns:
    print(column, " - ", loan_dicitonary.loc[column]["Description"])
```

```
loan_amnt  -  The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.
funded_amnt  -  The total amount committed to that loan at that point in time.
funded_amnt_inv  -  The total amount committed by investors for that loan at that point in time.
term  -  The number of payments on the loan. Values are in months and can be either 36 or 60.
int_rate  -  Interest Rate on the loan
installment  -  The monthly payment owed by the borrower if the loan originates.
grade  -  LC assigned loan grade
sub_grade  -  LC assigned loan subgrade
emp_title  -  The job title supplied by the Borrower when applying for the loan.*
emp_length  -  Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years.
home_ownership  -  The home ownership status provided by the borrower during registration or obtained from the credit report. Our values are: RENT, OWN, MORTGAGE, OTHER
annual_inc  -  The self-reported annual income provided by the borrower during registration.
verification_status  -  Indicates if income was verified by LC, not verified, or if the income source was verified
issue_d  -  The month which the loan was funded
loan_status  -  Current status of the loan
purpose  -  A category provided by the borrower for the loan request.
title  -  The loan title provided by the borrower
zip_code  -  The first 3 numbers of the zip code provided by the borrower in the loan application.
addr_state  -  The state provided by the borrower in the loan application
dti  -  A ratio calculated using the borrower’s total monthly debt payments on the total debt obligations, excluding mortgage and the requested LC loan, divided by the borrower’s self-reported monthly income.
delinq_2yrs  -  The number of 30+ days past-due incidences of delinquency in the borrower's credit file for the past 2 years
earliest_cr_line  -  The month the borrower's earliest reported credit line was opened
inq_last_6mths  -  The number of inquiries in past 6 months (excluding auto and mortgage inquiries)
mths_since_last_delinq  -  The number of months since the borrower's last delinquency.
mths_since_last_record  -  The number of months since the last public record.
open_acc  -  The number of open credit lines in the borrower's credit file.
pub_rec  -  Number of derogatory public records
revol_bal  -  Total credit revolving balance
revol_util  -  Revolving line utilization rate, or the amount of credit the borrower is using relative to all available revolving credit.
total_acc  -  The total number of credit lines currently in the borrowe
```

```
r's credit file
total_pymnt  -  Payments received to date for total amount funded
total_pymnt_inv  -  Payments received to date for portion of total amount
funded by investors
total_rec_prncp  -  Principal received to date
total_rec_int  -  Interest received to date
total_rec_late_fee  -  Late fees received to date
recoveries  -  post charge off gross recovery
collection_recovery_fee  -  post charge off collection fee
last_pymnt_d  -  Last month payment was received
last_pymnt_amnt  -  Last total payment amount received
pub_rec_bankruptcies  -  Number of public record bankruptcies
```

Combining these pieces of information, we can summarize the loan data fields with a description, the type of data, and description of possible values.

Field	Description	Data Type	Possible Values
loan_amnt	The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.	float	500 - 35000
funded_amnt	The total amount committed to that loan at that point in time.	float	500 - 35000
funded_amnt_inv	The total amount committed by investors for that loan at that point in time.	float	0 - 35000
term	The number of payments on the loan. Values are in months and can be either 36 or 60.	object	'36 months', '60 months', *blank*
int_rate	Interest Rate on the loan	object	*percentage*, *blank*
installment	The monthly payment owed by the borrower if the loan originates.	float	15.67 - 1305.16
grade	LC assigned loan grade	object	A - G, *blank*
sub_grade	LC assigned loan subgrade	object	A1 - A5, ... ,G1 - G5, *blank*
emp_title	The job title supplied by the Borrower when applying for the loan.*	object	*various strings*, *blank*
emp_length	Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years.	object	'< 1 year', '1 year', ...'9 years', '10+ years', *blank*
home_ownership	The home ownership status provided by the borrower during registration or obtained from the credit report. Our values are: RENT, OWN, MORTGAGE, OTHER	object	'RENT', 'OWN', 'MORTGAGE', 'OTHER', 'NONE', *blank*
annual_inc	The self-reported annual income provided by the borrower during registration.	float	1896 - 6000000
verification_status	Indicates if income was verified by LC, not verified, or if the income source was verified	object	'Verified', 'Source Verified', 'Not Verified', *blank*

Field	Description	Data Type	Possible Values
issue_d	The month which the loan was funded	object	*month-year string*, *blank*
loan_status	Current status of the loan	object	'Fully Paid', 'Charged Off', 'Does not meet the credit policy. Status:Fully Paid', 'Does not meet the credit policy. Status:Charged Off', *blank*
purpose	A category provided by the borrower for the loan request.	object	'credit_card', 'car', 'small_business', 'other', 'wedding', 'debt_consolidation', 'home_improvement', 'major_purchase', 'medical', 'moving', 'vacation', 'house', 'renewable_energy', 'educational', *blank*
title	The loan title provided by the borrower	object	*various strings*, *blank*
zip_code	The first 3 numbers of the zip code provided by the borrower in the loan application.	object	*sanitized zip codes*
addr_state	The state provided by the borrower in the loan application	object	*state abbreviation*
dti	A ratio calculated using the borrower's total monthly debt payments on the total debt obligations, excluding mortgage and the requested LC loan, divided by the borrower's self-reported monthly income.	float	0 - 29.99
delinq_2yrs	The number of 30+ days past-due incidences of delinquency in the borrower's credit file for the past 2 years	float	0 - 13
earliest_cr_line	The month the borrower's earliest reported credit line was opened	object	*month-year string*, *blank*
inq_last_6mths	The number of inquiries in past 6 months (excluding auto and mortgage inquiries)	float	0 - 33
mths_since_last_delinq	The number of months since the borrower's last delinquency.	float	0 - 120
mths_since_last_record	The number of months since the last public record.	float	0 - 129
open_acc	The number of open credit lines in the borrower's credit file.	float	1 - 47
pub_rec	Number of derogatory public records	float	0 - 5
revol_bal	Total credit revolving balance	float	0 - 1207359
revol_util	Revolving line utilization rate, or the amount of credit the borrower is using relative to all available revolving credit.	object	*percentage*
total_acc	The total number of credit lines currently in the borrower's credit file	float	1 - 90
total_pymnt	Payments received to date for total amount funded	float	0 - 58886.47343
total_pymnt_inv	Payments received to date for portion of total amount funded by investors	float	0 - 58563.68

Field	Description	Data Type	Possible Values
total_rec_prncp	Principal received to date	float	0 - 35000.02
total_rec_int	Interest received to date	float	0 - 23886.47
total_rec_late_fee	Late fees received to date	float	0 - 209
recoveries	post charge off gross recovery	float	0 - 29623.35
collection_recovery_fee	post charge off collection fee	float	0 - 7002.19
last_pymnt_d	Last month payment was received	object	*month-year string*, *blank*
last_pymnt_amnt	Last total payment amount received	float	0 - 3170.22
pub_rec_bankruptcies	Number of public record bankruptcies	float	0 - 2

Lab Answers

1.
```
display(data)
```

   or

```
print(data)
```

2.
```
data.index.values
```

3.
```
data["Mon":"Thu"]
```

4.
```
forecast.loc['Mon':'Wed']
```

5.
```
for row in csv_data[:5]:
    print(row)
```

6.
```
hr_data.head(5)
```

7.
```
hr_data[:10].Age.mean()
```

8.
```
salary_data = html_data[0]
salary_data.head(5)
```

   or

```
html_data[0].head(5)
```

9.
```
loan_data = pd.read_csv("./data/01-loan.csv")
loan_data.head()
```

10.
```
loan_data.dtypes
```

Next Steps

Having a catalog of the data will help us determine if what we have is sufficient to continue with the project or if we'll need to gather more data. For example, if we had been asked to determine if factors such as debt-to-income ratio or an internal-assigned grade was reliable in determining whether a loan would be paid off or not, it would appear that we have enough data to proceed based on our initial description of the data. If, however, we were asked to give different weights to different types of existing debt (credit card balances versus home mortgages, for example), we would have to request additional data.

Assuming we have sufficient data, our next step might be to further explore our data characterize it and determine if any relationships exist. Before exploring data, however, the data must be transformed or cleaned to facilitate analysis. Often, data cleansing and exploration are interwoven tasks.

Resources

- CRISP-DM (https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining)
- Pandas Documentation (https://pandas.pydata.org/)
- *Python Data Analysis* by Fandango, Chapter 5: Retrieving, Processing, and Storing Data (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/programming/python/9781787127
data-analysis-second-edition/ch05_html)
- *Python Data Science Handbook* by VanderPlas (https://jakevdp.github.io/PythonDataScienceHandbook/)
- *Python for Data Analysis* by Wes McKinney, Chapter 6: Data Loading, Storage, and File Formats (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/programming/python/9781491957
loading-storage-and-file-formats/io_html)
- Requests Documentation (http://docs.python-requests.org/en/master/)
- SQLAlchemy Documentation (https://www.sqlalchemy.org/)

Exercises

Use this notebook to compete each exercise below. Add cells as necessary.

1. Use pandas to extract tabular data from the webpage at `http://testing-ground.scraping.pro/table?products=10&years=10&quarters=4` . Display the content of each of the generated DataFrames. Generally, sites restrict the use of scrapers; scraping.pro (http://testing-ground.scraping.pro/) provides a set of pages to test web scrapers. After loading and displaying the data notice that there are issues with the DataFrame including `NaN` values and formatting issues. We will address some of these issues in later units.
2. Use the MetaWeather API (https://www.metaweather.com/api/) to retrieve weather data for Columbus on January 1, 2018 using the URL `https://www.metaweather.com/api/location/2383660/2018/1/1` with panda's *read_json()* function; here `2383660` is the WOEID (https://en.wikipedia.org/wiki/WOEID) for

Columbus. Use the API documentation to write up a description of the columns in the resulting DataFrame. To write text in a cell, select *Cell*, *Cell Type*, and *Markdown* from the menus above. Do not worry about formatting the description text.

In [ ]: