

# Experiment 1

## Manipulating Datasets using Pandas

### 1.1 Introduction to Pandas

Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data.

Pandas allows us to analyze big data and draw conclusions based on statistical theories. Pandas can clean up messy data sets and make them readable and relevant. Relevant data is very important in data science.

**Pandas Installation:** If you have Python and PIP already installed on a system, then installing Pandas is very easy. Install it using this command:

```
C:\users\your name> pip install pandas
```

**Import Pandas:** Once Pandas is installed, import it into your applications by adding the import keyword:

```
import pandas
```

Now the Pandas are imported and ready to use.

**Pandas as pd:** Pandas is usually imported under the pd alias. alias: In Python, an alias is an alternate name for referring to the same thing. Create an alias with the as keyword while importing:

```
import pandas as pd
```

Now the Pandas package can be referred to as pd instead of Pandas.

### 2.2 Series and DataFrames

The primary two components of Pandas are the Series and DataFrame. A Series is essentially a column, and a DataFrame is a multi-dimensional table made up of a collection of Series. DataFrames and Series are quite similar in that there are many operations that you can do with one that you can do with the other, such as filling in null values and calculating the mean.

#### 2.2.1 Creating Pandas Series

Pandas Series can be created from lists, dictionaries, scalar values, etc. In the real world, a Pandas Series will be created by loading the datasets from existing storage, which can be a SQL Database, a CSV file, or an Excel file. Here are some ways in which we create a series:

**Example 1:** Create a simple Pandas Series from a list:

In order to create a series from a list, we have to first create a list. After that, we can create a series from a list.

```
import pandas as pd

# create a simple list
a = [1, 7, 2]

# create series from a list
my_Series = pd.Series(a)

print(my_Series)
```

Regarding labels: By default, the values in the series are labeled with their index number. The first value has index 0, second value has index 1, etc. This label can be used to access a specified value.

**Example 2:** Create a simple Pandas Series from a dictionary

In order to create a series from the dictionary, we have to first create a dictionary. After that, we can make a series using the dictionary. Dictionary keys are used to construct indexes for series.

```
import pandas as pd

# a simple dictionary
calories = {"day1": 420, "day2": 380, "day3": 390}

# create series from dictionary
my_Series = pd.Series(calories)

print(my_Series)
```

Pandas DataFrame can be created in multiple ways: from lists, from lists of lists, from dict of lists, from lists of dicts, from series, from files. etc. Note that the DataFrame() function is used to create a DataFrame in Pandas.

**2.2.2 Creating Pandas DataFrame**

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns. Pandas DataFrame can be created from lists, dictionaries, lists of dictionaries, etc. In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, which can be a SQL Database, CSV file, or an Excel file. To create a DataFrame, the .DataFrame() method will be used. The syntax for using the .DataFrame() method is as follows:

```
class pandas.DataFrame(data=None, index=None, columns=None, dtype=None,
copy=None)
```

Please refer to Pandas documentation for more details regarding .DataFrame() arguments. Here are some ways in which we create a dataframe:

**Example 3:** Create a DataFrame from a Series:

```
import pandas as pd

# Initialize data to a series.
d = pd.Series([10, 20, 30, 40])

# creates Dataframe.
df = pd.DataFrame(d)

# print the data.
print(df)
```

**Example 4:** Creating DataFrame from a Dictionary of lists

```
import pandas as pd

# initialize the data of lists.
data = {'Name': ['Ahmad', 'Ali', 'Omar', 'Hamzah'],
        'Age': [20, 21, 19, 18]}

# Create DataFrame
df = pd.DataFrame(data)

# Print the output.
print(df)
```

**Example 5:** Creating dataframe from two series

```
import pandas as pd

data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
}

myvar = pd.DataFrame(data)

print(myvar)
```

Dataframes can also be created from Comma Separated Files (CSV). Therefore, If your data sets are stored in a file, Pandas can load them into a DataFrame. The first step in creating a DataFrame from a CSV file is to read the file into Python. This can be done using the `pandas` library, which provides a simple way to read CSV files as DataFrames.

**Example 6:** Load a CSV file named data.csv into a DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df)
```

It's worth noting that the `read\_csv` function has many optional parameters that can be used to customize how the CSV file is read. For example, you can specify the delimiter used in the file (in case it's not a comma), the encoding, and whether or not the file contains a header row.

## 2.3 Exploring DataFrames

**Viewing your data:** The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference. We accomplish this with `.head()` function.

**Example 7:** using `.head()` function to print the first five rows of the data set

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df.head())
```

The `.head()` will output the first five rows of your DataFrame by default, but we could also pass a number. For example: `df.head(10)` would output the top ten rows. To see the last five rows, use `.tail()`. The `.tail()` also accepts a number, and in this case we are printing the bottom rows:

**Example 8:** using `.tail()` function to print the last five rows of the data set

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df.tail())
```

**DataFrame Shape:** The shape of a DataFrame is a tuple of array dimensions that tells the number of rows and columns of a given DataFrame. The `DataFrame.shape` attribute in Pandas enables us to obtain the shape of a DataFrame. For example, if a

DataFrame has a shape of (80, 10) , this implies that the DataFrame is made up of 80 rows and 10 columns of data.

**Example 9:** printing the shape of the dataframe

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df.shape)
```

**Information About the Data:** The DataFrames object has a method called `info()`, that gives you more information about the data set. The `.info()` method provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

**Example 10:** using `.info()` function

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df.info())
```

## 2.4 DataFrames Indexing

Pandas dataframes have an intrinsic tabular structure represented by rows and columns where each row and column has a unique label (name) and position number (like a coordinate) inside the DataFrame, and each data point is characterised by its location at the intersection of a specific row and column. Indexing a Pandas DataFrame means selecting particular subsets of data (such as rows, columns, and individual cells) from that DataFrame.

When you create a DataFrame in Pandas, the DataFrame will automatically have certain properties. Specifically, each row and each column will have an integer “location” in the DataFrame. These integer locations for the rows and columns start at zero. So the first column will have an integer location of 0, the second column will have an integer location of 1, and so on. The same numbering pattern applies to the rows. We can use these numeric indexes to retrieve specific rows and columns.

To select a subset from a DataFrame, we use the indexing operator `[]`, attribute operator `.`, and an appropriate method of Pandas dataframe indexing such as `loc`, `iloc`, `at`, `iat`, and some others. Essentially, there are two main ways of indexing Panda DataFrames: label-based and position-based (aka location-based or integer-based). Also, it is possible to apply boolean DataFrame indexing based on predefined conditions, or even mix different types of DataFrame indexing.

For the next examples, we will create the following DataFrame to explore the effects of various dataframe indexing methods:

```
import pandas as pd
```

```
df = pd.DataFrame({'col_1': list(range(1, 11)), 'col_2':  
list(range(11, 21)), 'col_3': list(range(21, 31)), 'col_4':  
list(range(31, 41)), 'col_5': list(range(41, 51)), 'col_6':  
list(range(51, 61)), 'col_7': list(range(61, 71)), 'col_8':  
list(range(71, 81)), 'col_9': list(range(81, 91))})
```

```
print(df)
```

### 2.4.1 Label-based Dataframe Indexing

As its name suggests, this approach implies selecting DataFrame subsets based on the row and column labels. Let's explore four methods of label-based dataframe indexing: using the indexing operator [], the loc indexer, and the at indexer.

- **Using the indexing operator:** If we need to select all data from one or multiple columns of a Pandas DataFrame, we can simply use the indexing operator []. To select all data from a single column, we pass the name of this column. Before proceeding, let's take a look at what the current labels of our DataFrame are. For this purpose, we will use the attributes columns and index:

```
print(df.columns)  
print(df.index)
```

**Example 11:** printing the second column using the label indexing operator.

```
print(df['col_2'])
```

Now try to run the following and see what is returned:

```
print(df[['col_5', 'col_1', 'col_8']])  
print(df[['col_5', 'col_1', 'col_8', 'col_100']])
```

- **Using the loc indexer:** If we need to select not only columns but also rows (or only rows) from a dataframe, we can use the loc method, aka loc indexer. This method implies using the indexing operator [] as well. This is the most common way of accessing dataframe rows and columns by label. In general, the syntax `df.loc[row_label]` is used to pull a specific row from a DataFrame as a Panda Series object.

**Example 12:** selecting the values of the first row of the DataFrame using the loc indexer

```
print(df.loc[0])
```

However, for our further experiments with the **loc** indexer, let's rename the row labels to something more meaningful and of a string data type:

```
df.index = ['row_1', 'row_2', 'row_3', 'row_4', 'row_5', 'row_6',  
'row_7', 'row_8', 'row_9', 'row_10']
```

Now try to run the following samples and figure out what returns:

```
print(df.loc[['row_6', 'row_2', 'row_9']])  
print(df.loc[['row_6', 'row_2', 'row_9', 'row_100']])  
print(df.loc['row_7':'row_9'])
```

**Task 1.1:** Print the first 4 rows using the slicing method.

**Task 1.2:** Print the last row with columns from col\_5 to col\_7.

- **Using the at indexer:** For the last case from the previous section, i.e., for selecting only one value from a DataFrame, there is a faster method – using the at indexer. The syntax is identical to that of the loc indexer, except that here we always use exactly two labels (for the row and column) separated by a comma:

```
df.at['row_6', 'col_3']
```

## 2.4.2 Position-based Dataframe Indexing

Using this approach, each DataFrame element (row, column, or data point) is referred to by its position number rather than its label. The position numbers are integers starting from 0 for the first row or column (typical Python 0-based indexing) and increasing by 1 for each subsequent row/column. Position-based indexing is purely Python-style, i.e., the start bound of the range is inclusive while the stop bound is *exclusive*. Position-based indexing uses indexing operator [], iloc, and iat methods.

Try to run the following samples and figure out what returns:

```
print(df[3:6])  
print(df.iloc[3])  
print(df.iloc[[9, 8, 7]])  
print(df.iloc[0, [2, 8, 3]])  
Print(df.iat[1, 2])
```

**Task 1.3:** Use iloc with position-based indexing to select rows 8, and 1 and columns 5 to 9.

**Task 1.4:** Use iloc with position-based indexing to select all rows and columns 1, 3, and 7.

## 2.4.3 Boolean Dataframe Indexing

Apart from label-based or position-based Panda DataFrame indexing, it is possible to select a subset of a DataFrame based on a certain condition. For example: `df[df['col_2'] > 15]` Will select all the rows of our DataFrame where the values of the `col_2` column are greater than 15. We can also use any other comparison operators: such as `==` equals, `!=` not equals, `>` greater than, `<` less than, `>=` greater than or equal to, `<=` less than or equal to. Also, it is possible to define a boolean condition for a string

column too (the comparison operators `==` and `!=` make sense in such cases). In addition, we can define several criteria for the same column or multiple columns. The operators to be used for this purpose are `&` (*and*), `|` (*or*), `~` (*not*). Each condition must be put in a separate pair of brackets.

**Task 1.5:** Selecting all the rows of the dataframe where the value of `col_2` is NOT greater than 15

**Task 1.6:** Selecting all the rows of the dataframe where the value of `col_2` is greater than 15 but not equal to 19

## 2.5 Saving Dataframe to CSV file

To save Pandas DataFrames, use the `.to_csv()` method. The `.to_csv()` method is a built-in function in Pandas that allows you to save a Pandas DataFrame as a CSV file. This method exports the DataFrame into a comma-separated values (CSV) file, which is a simple and widely used format for storing tabular data. The syntax for using the `.to_csv()` method is as follows:

```
DataFrame.to_csv(filename, sep=',', index=False, encoding='utf-8')
```

Here, DataFrame refers to the Pandas DataFrame that we want to export, and filename refers to the name of the file that you want to save your data to.

The sep parameter specifies the separator that should be used to separate values in the CSV file. By default, it is set to comma-separated values. We can also set it to a different separator, like `\t` for tab-separated values.

The index parameter is a boolean value that determines whether to include the index of the DataFrame in the CSV file. By default, it is set to False, which means the index is not included.

The encoding parameter specifies the character encoding to be used for the CSV file. By default, it is set to utf-8, which is a standard encoding for text files.

Example: Saving DataFrame to CSV

```
import pandas as pd
```

### Example 13: Saving a DataFrame to a CSV file

```
import pandas as pd

# Create a sample dataframe
Biodata = {'Name': ['Ahmad', 'Ali', 'Omar', 'Hamzah'],
           'Age': [28, 23, 35, 31],
           'Gender': ['M', 'F', 'M', 'F']}

df = pd.DataFrame(Biodata)

# Save the dataframe to a CSV file
df.to_csv('Biodata.csv', index=False)
```

**Task 1.7:** Save the above dataframe using tab-separated values



## 2.6 Dealing with Rows and Columns in Panda's DataFrame

We can perform basic operations on rows/columns like deleting, adding, and renaming.

### 2.6.1 Column Addition:

In Order to add a column to a Pandas DataFrame, we can declare a new list as a column and add it to an existing DataFrame.

**Example 14:** Column addition using the basic method

```
import pandas as pd

# Define a dictionary containing Students data
data = {'Name': ['Ahmad', 'Ali', 'Omar', 'Hamzah'],
        'Height': [5.1, 6.2, 5.1, 5.2],
        'Qualification': ['Msc', 'MA', 'Msc', 'Msc']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data)
print(df)

# Declare a list that is to be converted into a column
address = ['Nablus', 'Alquds', 'Gaza', 'Haifa']

# Using 'Address' as the column name and equating it to the list
df['Address'] = address

# Observe the result
print(df)
```

Note that the basic method will add the column at the end. By using DataFrame.insert() method, it gives us the freedom to add a column at any position we like, not just at the end. It also provides different options for inserting the column values.

**Task 1.8:** Add a new column (on the dataframe above) named Age with the following values [21, 23, 24, 21] and make it the third column.

### 2.6.2 Column Deletion:

In Order to delete a column in Pandas DataFrame, we can use the drop() method. Columns are deleted by dropping columns with column names.

**Example 15:** Deleting a column

```
import pandas as pd

# Define a dictionary containing Students data
data = {'Name': ['Ahmad', 'Ali', 'Omar', 'Hamzah'],
```

```

'Height': [5.1, 6.2, 5.1, 5.2],
'Qualification': ['Msc', 'MA', 'Msc', 'Msc'],
'Age': [21, 23, 24, 21]}

```

```

# Convert the dictionary into DataFrame
df = pd.DataFrame(data)
print(df)

```

```

# dropping passed columns
df1 = df.drop(["Height"], axis = 1)

```

```

# Observe the result
print(df)
print(df1)

```

As you can see from the output, the new output doesn't have the passed columns. Those values were dropped since the axis was set equal to 1 and the changes were made in the original data frame since inplace was True. Note that, Use axis=1 or columns param to remove columns. By default, Pandas return a copy of the DataFrame after deleting columns, use the inplace=True parameter to remove it from the existing referring DataFrame.

**Task 1.9:** Remove column Height and column Age from the existing referring dataframe.

### 2.6.3 Adding new row to DataFrame

There are multiple ways to add or insert a row to a Pandas DataFrame, in this section, we will explain how to add a row to a Pandas DataFrame using the `loc[]` method. By using `df.loc[index]=list` you can append a list as a row to the DataFrame at a specified Index. In order to add at the end, get the index of the last record using the `len(df)` function. The below example adds the list `["Hyperion",27000,"60days",2000]` to the end of the Pandas DataFrame.

**Example 16:** Adding a new row to the end of the Pandas DataFrame.

```

import pandas as pd

technologies= {
    'Courses':["Spark", "PySpark", "Hadoop", "Python", "Pandas"],
    'Fee' :[22000,25000,23000,24000,26000],
    'Duration':['30days', '50days', '35days', '40days', '55days'],
    'Discount':[1000,2300,1000,1200,2500]
}

df = pd.DataFrame(technologies)
print(df)

# New list to append Row to DataFrame

```

```
list = ["Hyperion", 27000, "60days", 2000]
df.loc[len(df)] = list
print(df)
```

**Task 1.10:** use `.iloc` to insert the new row above into the second position of the DataFrame.

### 2.6.3 Deleting row From DataFrame

By using the `pandas.DataFrame.drop()` method, you can drop/remove/delete rows from the DataFrame. Let's create a DataFrame, run some examples, and explore the output. Note that our DataFrame contains index labels for rows, which we are going to use to demonstrate removing rows by labels. **Please run the samples below (from example 17 to example 20) and figure out what returns.**

```
import pandas as pd
import numpy as np

technologies = {
    'Courses': ["Spark", "PySpark", "Hadoop", "Python"],
    'Fee' : [20000, 25000, 26000, 22000],
    'Duration': ['30day', '40days', np.nan, None],
    'Discount': [1000, 2300, 1500, 1200]
}

indexes=['r1', 'r2', 'r3', 'r4']
df = pd.DataFrame(technologies, index=indexes)
print(df)
```

#### Example 17: Drop rows by Index Labels or Names

If you have a DataFrame with row labels (index labels), you can specify what rows you want to remove by label names. Here is an example:

```
# Drop rows by Index Label
df1 = df.drop(['r1', 'r2'])
print(df1)
```

Alternatively, you can also write the same statement by using the field name 'index'.

```
df1 = df.drop(index=['r1', 'r2'])
print(df1)
```

#### Example 18: Drop Rows by Index Number (Row Number)

Similarly, by using the `drop()` method, you can also remove rows by index position from a Pandas DataFrame. The `drop()` method doesn't have a position index as a parameter; hence, we need to get the row labels from the index and pass these to the `drop` method. We will use `df.index` to get row labels for the indexes we want to delete. Below are some examples:

```
# Delete Rows 1 and 3 by Index numbers
df1=df.drop(df.index[[1,3]])
print(df1)
```

```
# Removes First Row
df=df.drop(df.index[0])
```

```
# Removes Last Row
df=df.drop(df.index[-1])
```

```
# Delete Rows by Index Range
df1=df.drop(df.index[2:])
print(df1)
```

#### **Example 19:** Delete Rows with Default Indexes

By default, Pandas assign a sequence number to all rows, also called an index. The row index starts at zero and increments by 1 for every row. To remove rows with the default index, you can try the steps below.

```
# Remove rows when you have a default index.
df = pd.DataFrame(technologies)
df1 = df.drop(0)
df3 = df.drop([0, 3])
df4 = df.drop(range(0,2))
```

#### **Example 20:** Drop Rows by Checking Conditions

Most of the time, we would also need to remove DataFrame rows based on some conditions (column value), Below are some quick examples applied to technology dataframe.

```
# Using drop() to delete rows based on column value
df.drop(df[df['Fee'] >= 24000].index, inplace = True)
```

```
# Remove rows
df2 = df[df.Fee >= 24000]
```

```
# Using loc
df2 = df.loc[df["Fee"] >= 24000]
```

**Task 1.11:** delete all rows with Fee >= 2200 and Discount == 2300. Save the result in a file named task.csv

**Task 1.12:** Write a code to delete all rows with a Non/NaN value.

**Task 1.13:** Write a code to delete all rows having courses equal to Spark or Hadoop.

## 2.7 Combining DataFrames in Pandas

### 2.7.1 DataFrame Concatenation

With concatenation, your datasets are just stitched together along an axis — either the row axis or column axis. To concatenate two or more DataFrames vertically, you can use the `.concat()` method with no parameters:

**Example 14:** Concatenate two DataFrames vertically

```
import pandas as pd

# Create two sample DataFrames
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']})

df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']})

# Concatenate the DataFrames vertically
result = pd.concat([df1, df2])

print(result)
```

**Task 1.14:** Perform the concatenation in the above example along columns.

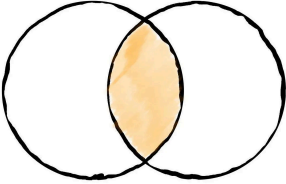
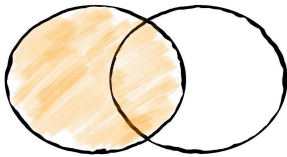
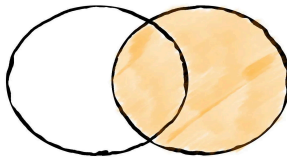
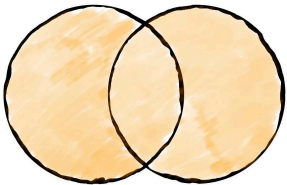
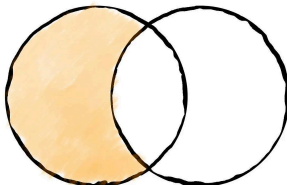
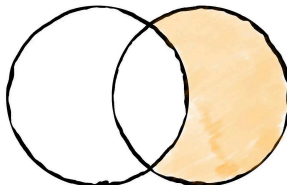
### 2.7.1 DataFrame Merging

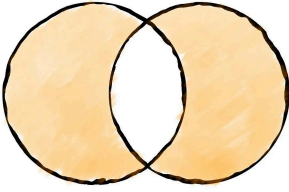
Merging functionality is similar to a database's `join` operations. When you want to combine data objects based on one or more keys, similar to what you'd do in a relational database, `merge()` is the tool you need. More specifically, `merge()` is most useful when you want to combine rows that share data. You can achieve both many-to-one and many-to-many joins with `merge()`. When you use `merge()`, you'll provide two required arguments: The `left` DataFrame and The `right` DataFrame

After that, you can provide a number of optional arguments to define how your datasets are merged:

- `how` defines what kind of merge to make. It defaults to `'inner'`, but other possible options include `'outer'`, `'left'`, and `'right'`.
- `on` tells `merge()` which columns or indices, also called key columns or key indices, you want to join on. This is optional. If it isn't specified, and `left_index` and `right_index` (covered below) are `False`, then columns from the two DataFrames that share names will be used as join keys. If you use `on`, then the column or index that you specify must be present in both objects.

The table below illustrates the merging types along with details on how to customize the .merge() method in each case.

 <p><b>Inner Join:</b> To perform an inner join between two DataFrames using a single column, all we need is to provide the on argument when calling merge().</p> <pre>df = pd.merge(df1, df2, on='id')</pre>	 <p><b>Left Outer Join:</b> Use keys from the left frame only. To perform a left join between two pandas DataFrames, you now have to specify how='left' when calling merge().</p> <pre>df1.merge(df2, on='id', how='left')</pre>	 <p><b>Right Outer Join:</b> Use keys from the right frame only. To perform a right join between two pandas DataFrames, you now have to specify how='right' when calling merge().</p> <pre>df = pd.merge(df1, df2, how='right')</pre>
 <p><b>Full Outer Join:</b> Use union of keys from both frames. To perform a full outer join, you need to specify how='outer' when calling merge().</p> <pre>df = pd.merge(df1, df2, how='outer')</pre>	 <p><b>Left Anti-Join:</b> Use only keys from the left frame that don't appear in the right frame. A left anti-join in pandas can be performed in two steps. In the first step, we need to perform a left outer join with indicator=True. Then, we simply need to query() the result from the previous expression.</p> <pre>df3 = df1.merge(df2, on='A', how='left', indicator=True) df = df3.loc[df3['_merge']</pre>	 <p><b>Right Anti-Join:</b> Use only keys from the right frame that don't appear in the left frame. Similar to Left anti-join.</p> <pre>df3 = df1.merge(df2, on='A', how='right', indicator=True) df = df3.loc[df3['_merge'] == 'right_only', 'A'] d = df2[df2['A'].isin(df)]</pre>

	<pre>== 'left_only', 'A'] d = df1[df1['A'].isin(df)]</pre>	
 <p><b>Full Anti-Join:</b> Take the symmetric difference of the keys of both frames.</p> <pre>df3 = df1.merge(df2, on='A', how='outer', indicator=True)  df = df3.loc[df3['_merge'] == 'both', 'A']  d = df3[~df3['A'].isin(df)]</pre>	<p><b>Merging on Different Column Names.</b></p> <p>In this case, instead of providing the on argument, we have to provide the left_on and right_on arguments to specify the columns of the left and right DataFrames to be considered when merging them together.</p> <pre>d = df1.merge(df2, left_on='A', right_on='B')</pre> <p>Will merge df1 and df2 based on the A and B columns, respectively.</p>	<p><b>Merging on Multiple Columns.</b></p> <p>If you want to merge on multiple columns, you can simply pass all the desired columns into the on argument as a list:</p> <pre>d = df1.merge(df2, on=['A', 'B'])</pre> <p>If the columns in the left and right frames have different names then once again, you can make use of right_on and left_on arguments</p>

**Task 1.15:** Run the samples above and figure out what returns.

**Task 1.16:** Let's say that we want to merge frames df1 and df2 using a left outer join, Select all the columns from df1 but only column colE from df2.