COSC 130 – Group Project Instructions

Introduction

The goal of this project is to implement a data frame, or **`DFrame`**, class to be used for working with tabular data.

General Instructions

Create a Python script file named **`DFrame.py`** and a Jupyter notebook file named **`Project_04_GroupN.ipynb`**. The **`N`** character in the notebook file name should be replaced with your actual group number. You will use the script to define your **`DFrame`** class and the notebook will be used to load the script and to test the class.

Please download the files **`iris.txt`** and **`titanic.txt`**, storing these in the same directory as your script file and notebook file. It is important that these files are all in the same directory. Otherwise, your code will not run correctly when I run it.

Instructions for the Script File

Define a class named **`DFrame`** containing each of the methods shown below. Descriptions of each of these methods are provided in these instructions.

`__init__()`	**`get_row()`**	**`get_col()`**	**`filter()`**
`__str__()`	**`add_row()`**	**`add_col()`**	**`sort_by()`**
`head()`	**`del_row()`**	**`del_col()`**	**`mean()`**

Your script file should also define a function named **read_file_to_dframe()**. This function should NOT be part of the **`DFrame`** class. A description of this function is provided later in these instructions.

`__init__()`

The constructor should accept two parameters: **`self`** and **`data`**.

- **`self`** will refer to a new instance of the **`DFrame`** class.
- **`data`** is expected to be a dictionary of key/value pairs representing columns. The key for each pair represents the name of the column, while the value should be a list of values contained within that column.

The constructor should define the following attributes:

- **`self.data`** – This should be set equal to the dictionary containing the data.
- **`self.columns`** – This is a list of names for each of the columns. This can be taken from the keys in **`data`**.
- **`self.size`** – This will be a list of two integers. The first element is the number of rows in the **`DFrame`**, and the second element is equal to the number of columns.

A summary of the tasks that should be performed by the constructor is provided below:

1. To be a valid data frame, all of the columns must contain the same number of elements. Check that this is true. If it is not, raise an exception stating: "The size of the columns in data is inconsistent." If the column sizes are consistent, continue with the following steps.
2. Set **`self.data`** to **`data`**.
3. Set **`self.columns`** to be equal to a list of keys found in **`data`**.
4. Determine the number of rows and columns in the **`DFrame`**. The number of columns will be equal to the length of **`self.columns`**. The number of rows will be equal to the length of any one list found in **`self.data`**.
5. Use the number of rows and columns to define **`self.size`**.

`get_col()`

This method should accept two parameters: **`self`** and **`c`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **`c`** is used to specify the column to return. This is allowed to be either an integer specifying the column index, or a string referring the name of the column.

The method should return a list containing the values in the specified column.

The method should first check to see if the if **`c`** is an integer or a string.

- If **`c`** is a string, then return the list in **`self.data`** whose key is equal to `c`.
- If **`c`** is an integer, then use **`self.columns`** to find the associated column name, and return the list in **`self.data`** whose key is equal to this column name.

`get_row()`

This method should accept two parameters: **`self`** and **`n`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **`n`** is expected to be an integer representing the index for a particular row.

The method should return a list containing the values in the specified row.

The list returned by this method will be constructed by selecting a single element from each of the lists stored in **`self.data`**. This can be accomplished using a loop and the **`append()`** method, or with a list comprehension.

`add_col()`

This method should accept three parameters: **`self`**, **`name`**, and **`col`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **`name`** is expected to be a string containing the name for a new column to be created in the data frame.
- **`col`** is expected to be a list of values representing a new column to be added to the data frame.

The method should add a new column to the data frame.

This method should perform the following steps:

1. It should first check to see if the length of the new column is consistent with the length of the existing columns. If this is not the case, it should raise an exception reading "Inconsistent column sizes."
2. If the column size is correct, add a new pair to **`self.data`** with key equal to **`name`** and value equal to **`col`**.
3. Append **`name`** to **`self.columns`**.
4. Update **`self.size`** to reflect the new number of columns.

`add_row()`

This method should accept two parameters: **`self`**, and **`row`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **`row`** is expected to be a list of values representing a new row to be added to the data frame.

The method should add a new row to the data frame.

This method should perform the following steps:

1. Loop over the values of **`row`**, adding each value to the appropriate list in **`self.data`**. You will need to use **`self.columns`** to obtain the name (and key) associated with each column.
2. Update **`self.size`** to reflect the new number of rows.

`del_col()`

This method should accept two parameters: **`self`** and **`c`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **`c`** is used to specify a column. This is allowed to be either an integer specifying the column index, or a string referring the name of the column.

The method should delete the specified column from the data frame.

This method should perform the following steps:

1. First check to see if the if **`c`** is an integer or a string.
    - If **`c`** is a string, then delete the list in **`self.data`** whose key is equal to `c`.
    - If **`c`** is an integer, then use **`self.columns`** to find the associated column name, and delete the list in **`self.data`** whose key is equal to this column name.
2. Update **`self.size`** to reflect the new number of columns.

Note: You can use the **del** keyword to delete an item from a dictionary based on its key.

`del_row()`

This method should accept two parameters: **`self`** and **`n`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **n** is expected to be an integer representing the index for a particular row.

The method should delete the specified row from the data frame.

This method should perform the following steps:

1. Loop over the values in **`self.data`**. Delete from each value the element at index **`n`**.
2. Update **`self.size`** to reflect the new number of rows.

Note: You can use the **del** keyword to delete an item from a list based on its index.

`head()`

This method should accept two parameters: **`self`** and **`n`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **n** is expected to be an integer. This parameter should have a default value of 6.

The method should return a new **DFrame** consisting of only the first **n** rows of the **`DFrame`** from which it was called.

This can be accomplished as follows:

1. Create an empty dictionary named **`new_data`**.
2. Loop over the pairs of **`self.data`**. For each pair, and a key/value pair to **`new_data`** with the same key, and whose value is a list consisting of only the first **n** elements of the original list. You can create the new list using slicing.
3. Use **`new_data`** to create a new **`DFrame`** object, and then return that object.

Alternately: You could create **`new_data`** using a dictionary comprehension.

`filter()`

This method should accept four parameters: **`self`**, **`c`**, **`mode`**, and **`val`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **`c`** is expected to be a string or an integer representing the name or index of a column.
- **`mode`** is expected to be one of the following strings: **`'<'`**, **`'<='`**, **`'>'`**, **`'>='`**, **`'=='`**
- **`val`** is expected to be any value of the time stored in column **`c`**.

The method should return a new **`DFrame`** that contains a subset of the records from the original **`DFrame`**. A row should be included in the new **`DFrame`** only if its value in column c satisfies the relationship specified by **`mode`** when compared against **`val`**.

This method should perform the following steps:

1. Use **`self.get_col()`** to obtain the column associated with **`c`**. Store the resulting list in a variable named **`col`**.
2. Create a dictionary with the same keys as those found in **`self.data`**, but with each value being equal to an empty list. Use this dictionary to create a **`DFrame`** named **`result`**.
3. Loop over elements of **`col`**. Perform the following steps each time the loop executes:
    a. Set a variable named **`keep`** equal to **`False`**.
    b. Use the value in **`mode`** with an **`if-elif-else`** statement to determine which comparison to perform. Compare the current value of **`col`** with **`val`**, storing the result of the comparison in **`keep`**.
    c. If **`keep`** is **`True`**, use **`self.get_row()`** to obtain the row for the current element of column, and then use the **`add_row()`** method to add this row to **`result`**.
4. Return **`result`**.

`sort_by()`

This method should accept three parameters: **`self`**, **`c`**, and **`reverse`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **`c`** is expected to be a string or an integer representing the name or index of a column.
- **`reverse`** is expected to be a Boolean value, and should have a default value of **`False`**.

The method should return a new **`DFrame`** that contains the same contents as the original **`DFrame`**, but with the rows sorted according to the specified column. By default the method should sort in increasing order by the values in column **`c`**, but if **`reverse`** is set to False, then the sorting should be performed in decreasing order.

This method should perform the following steps:

1. Within the method, define a helper function named **`argsort()`**. This function should be identical to the function of the same name from Homework 05.
2. Call the **`self.get_col()`** method, passing it the parameter **`c`**. Store the resulting list in a variable named **`col`**.
3. Call **`argsort()`** on **`col`**, storing the result in a list named **`idx_list`**. Use the parameter **`reverse`** when calling **`argsort()`** to specify the desired order of the sorting.
4. Create a dictionary with the same keys as those found in **`self.data`**, but with each value being equal to an empty list. Use this dictionary to create a **`DFrame`** named **`result`**.
5. Loop over the elements of **`idx_list`**. For each such element, extract the corresponding row using **`self.get_row()`** and then add this row to **`result`**.
6. Return **`result`**.

`mean()`

This method should accept two parameters: **`self`** and **`cols`**.

- **`self`** will refer to an instance of the **`DFrame`** class.
- **`cols`** is expected to be a list of integers corresponding to indices of columns.

The method should return a new **`DFrame`** the contains the means (or averages) of the columns specified by the parameter **`cols`**. The new **`DFrame`** should have one column for every value in **`cols`**, and should have only one row containing the average for each column.

This can be accomplished as follows:

1. Create an empty dictionary named **`mean_data`**.
2. Loop over the values stored in **`cols`**. Each time the loop executes, perform the following steps:
    a. Use **`self.columns`** to get the name of the current column.
    b. Calculate the mean (average) of the column associated with the name found in Part (a). Store the resulting mean in a list (with only a single value).
    c. Add the values from Parts (a) and (b) to **mean_data** as a new key/value pair. The column name should be the key and the list containing the column mean should be the value for the pair.
3. Use the dictionary **`mean_data`** to create a new **`DFrame`**, and then return that **`DFrame`**.

`__str__()`

This method should accept a single parameter named **`self`** that will refer to an instance of the **`DFrame`** class. The method should construct a string that can be printed to display the contents of the **`DFrame`** object in a tabular format. This is a special method that will be called whenever a **`DFrame`** object is passed to **`print()`**. It such a scenario the string returned by this method will be displayed.

This method should perform the following steps:

1. Start by creating a helper function named **`row_helper`**. This function should accept two parameters, named **`row`** and **`widths`**. The **`row`** parameter should be a list representing a row from the **`DFrame`**, while **`widths`** should be a list of integers indicating the desired width of the displayed column. The function should return a string the following format: **`| xxxx | xxxx | xxxx | xxxx |`**
    a. Create a string variable named **`row_str`** to contain the initial vertical bar followed by a space.
    b. Look over the lists **`row`** and **`widths`** (which should have the same number of elements). In each iteration of the loop, concentrate an element of **`row`** and another vertical bar to **`row_string`**. This can be down using the following code: **`row_str += f'{row[i]:<{widths[i]}} | '`**
    c. Concatenate a newline character to the end of **`row_str`**.
    d. Return **row_str**.
2. Next we will create a list named **`widths`** to store the desired widths for each column. Begin by creating an empty list named **`widths`**. Then loop over the values in **`self.columns`**. For each iteration of the loop, perform the following steps:
    a. Set **`temp`** equal to the length of the name of the current column being considered.
    b. Use **`self.get_col()`** to obtain the values stored in the current column, storing them in **`col`**.
    c. Loop over the values in **`col`**. For each iteration of the loop, covert the current value of **`col`** to a string, and find the length of the resulting string. If that value is larger than **`temp`**, then replace **`temp`** with the larger value.
    d. After the inner loop finishes, append **`temp`** to the list **`widths`**.
3. Finally, we will create our output string. Begin by calling **`row_helper`** on **`self.columns`** and **`widths`**, storing the result in a variable named **`out`**.
4. Add a string consisting of several dashes followed by a newline character to **`out`**. The number of dashes should be equal to the sum of the values in **`widths`**, plus 3 times the number of columns, plus 1.
5. Create a for loop that will execute once for each row in the **`DFrame`**. With each iteration of the loop, use **`self.get_row()`** to obtain a new row of the **`DFrame`**. Pass that row and **`widths`** to **`row_helper()`**, concatenating the result to **`out`**.
6. Return **`out`**.

`read_file_to_dframe()`

This function should accept three parameters: **`path`**, **`schema`**, and **`sep`**.

1. **`path`** is expected to be a string containing the path to a text file.
2. **`schema`** is expected to be a list of data types.
3. **`sep`** is expected to be a string.

This method is intended to read in a data file and use the contents of that file to create a **`DFrame`** object. The lines of the file will be read in as strings. These strings will be tokenized by splitting them according to the parameter **`sep`**. The individual tokens will each be coerced into data types as indicated by the **`schema`** parameter.

Note that the first line of the text file is expected to contain the names of the columns.

This method should perform the following steps:

4. Recreate the **`process_line()`** and **`read_list_to_list()`** functions from Homework 05. This will be used as helper functions to read in the data file and return the contents as a list of lists.
5. Call **`read_list_to_list()`** passing it the parameters **`path`**, **`schema`**, and **`sep`**. Store the results in a variable named **`rows`**. This will be a list of lists. The first element of **`rows`** will be a list containing the column names.
6. Create a dictionary named **data**. The keys in this dictionary should be equal to the column names that are stored in **`rows[0]`**. The values for each key/value pair should initially be set as empty lists. One way of creating this dictionary would be to start with an empty dictionary, then loop over the elements of **`rows[0]`** to add the desired key/value pairs.
7. Use **`data`** to create a **`DFrame`** object named **`df`**.
8. Loop over all of the elements of **`rows`** except for the first (you may delete the first element of **`rows`** if you wish). Each time the loop executes, used the **`get_row()`** method of **`df`** to add the current element of **`rows`** to the **`DFrame`**.
9. Return **`df`**.

Instructions for the Notebook

Any set of instructions you see in this document with an orange bar to the left will indicate a place where you should create a markdown cell. If no instructions are provided regarding formatting, then the text should be unformatted.

Any set of instructions you see with a blue bar to the left will provide instructions for creating a single code cell.

Assignment Header

Create a markdown cell with a level 1 header that reads: "COSC – Group Project". Add your group number as a level 3 header. Beneath that, list the names of the group members in bold.

Introduction

Create a markdown cell with a level 2 header that reads "**Introduction**".

Add unformatted text explaining the purpose of this project. Explain that you will be using this notebook to test the **`DFrame`** class that you have developed.

We will start by running the script.

Use the Magic command **`%run -i DFrame.py`** to run the contents of your script.

Nonsense Dataset

Create a markdown cell with a level 2 header that reads "**Nonsense Dataset**".

Add unformatted text explaining that you will test the **`DFrame`** class on an example consisting of meaningless values. Explain that you will start by creating the **`DFrame`** instance.

Use the dictionary shown below to create a **`DFrame`** instance named **`df`**. Then print **`df`**.

```
my_dict = {
    'x1': [12, 15, 18, 13, 15, 20],
    'x2': ['dog', 'aardvark', 'buffalo', 'cat', 'elephant', 'fish'],
    'x3': [1.8025, 4.61, 1.347, 4.7, 2.809, 3.6185]
}
```

We will now check the size of the **`DFrame`**.

Create a new code cell, and use it to print the **`size`** attribute of **`df`**.

Create a markdown cell to explain that you will now test the **`sort_by()`** method.

Create the three code cells described below. Each one tests a different aspect of the **`sort_by()`** method.

Create a new code cell containing the following code: **`print(df.sort_by('x1'))`**

Create a new code cell containing the following code: **`print(df.sort_by(0))`**

Create a new code cell containing the following code: **`print(df.sort_by('x3', reverse=True))`**

Create a markdown cell to explain that we will now display the first 3 rows of the **DFrame** to confirm that the order of elements in the original **DFrame** have not been changed.

Use the **head()** method to display the first three rows of **df**.

Create a markdown cell to explain that you will now test the **filter()** method.

Create the three code cells described below. Each one tests a different aspect of the **filter()** method.

Create a new code cell containing the following code: **print(df.filter('x2', '<', 'c'))**

Create a new code cell containing the following code: **print(df.filter('x2', '>', 'd'))**

Create a new code cell containing the following code: **print(df.filter(0, '<=', 15))**

Create a new code cell containing the following code: **print(df.filter(0, '>=', 15))**

Create a new code cell containing the following code: **print(df.filter(0, '==', 15))**

Create a markdown cell to explain that you will now test the **mean()** method.

Create a new code cell containing the following code: **print(df.mean([0,2]))**

Create a markdown cell to explain that you will now test the **add_row()**, **del_row()**, **add_col()**, and **del_col()** methods.

Create a new code cell containing the following code:

```
df.add_row([14, 'gorilla', 3.142])
df.del_row(1)
df.del_row(1)
df.del_col('x3')
df.add_row([19, 'horse'])
df.add_col('x4', [12345, 123, 1234, 1, 123456, 12])
print(df)
```

Iris Dataset

Create a markdown cell with a level 2 header that reads "**Iris Dataset**".

Add unformatted text explaining that you will test the **DFrame** class on the Iris Dataset. You can learn more about this Dataset here: Iris Dataset

Use the **read_file_to_dframe()** method to load the contents of the file **'iris.txt'** into a **DFrame** instance named **iris**. Use schema **[float, float, float, float, str]** and use the tab character as the separator. Use the **head()** method to bring the first 6 rows of this **DFrame**.

Create a markdown cell explaining that you will now check the number of rows and columns in the **iris DFrame**.

Print the value of the **size** attribute of **iris**.

Create a markdown cell to explain that we you now use the **mean()** method to calculate the average value of the first four columns in the **DFrame**.

Use the **mean()** method to calculate the average of the first four columns in the **DFrame**. Print the results.

Create a markdown cell to explain that we you now use the **filter()** and **mean()** method to calculate the average value of the first four columns in the **DFrame** for each of the three species of flower represented in the dataset. The three species are "setosa", "versicolor", and "virginica").

Create three code cells to calculate the averages for each of the three species.

Use **filter()** to select the rows of **iris** for which the **species** column is equal to **'setosa'**. Then use **mean()** to calculate the average of the first four columns in the **DFrame**. Print the results. This can all be done in one line of code by chaining calls to the **filter()** and **mean()** methods.

Use **filter()** to select the rows of **iris** for which the **species** column is equal to **'versicolor'**. Then use **mean()** to calculate the average of the first four columns in the **DFrame**. Print the results. This can all be done in one line of code by chaining calls to the **filter()** and **mean()** methods.

Use **filter()** to select the rows of **iris** for which the **species** column is equal to **'virginica'**. Then use **mean()** to calculate the average of the first four columns in the **DFrame**. Print the results. This can all be done in one line of code by chaining calls to the **filter()** and **mean()** methods.

Titanic Dataset

Create a markdown cell with a level 2 header that reads "**Titanic Dataset**".

Add unformatted text explaining that you will test the **DFrame** class on the Titanic Dataset. You can learn more about this Dataset here: Titanic Dataset

Use the **read_file_to_dframe()** method to load the contents of the file **'titanic.txt'** into a **DFrame** instance named **titanic**. Use schema **[int, int, str, str, float, float]** and use the tab character as the separator. Use the **head()** method to bring the first 10 rows of this **DFrame**.

Create a markdown cell explaining that you will now check the number of rows and columns in the **titanic DFrame**.

Print the value of the **size** attribute of **titanic**.

Create a markdown cell explaining you will now use the **sort_by()** method to determine the ages of the 10 youngest passengers on the Titanic, as well as the ages of the 10 oldest passengers of the titanic.

Perform these tasks using two separate code cells.

Use **sort_by()** to sort the rows of **titanic** in increasing order by **Age**. Use **head()** to display the first 10 rows of the results. This can all be done in one line of code by chaining calls to the **sort_by()** and **head()** methods.

Use **sort_by()** to sort the rows of **titanic** in decreasing order by **Age**. Use **head()** to display the first 10 rows of the results. This can all be done in one line of code by chaining calls to the **sort_by()** and **head()** methods.

Create a markdown cell explaining you will now determine the survival rate for female passengers on the Titanic, as well as the survival rate for male passengers on the Titanic.

Perform these tasks using two separate code cells.

Use **`filter()`** to select the rows of **`titanic`** for which **`Sex`** is equal to **`'female'`**. Then use **`mean()`** to calculate the average of the **`Survived`** column. Print the results. This can all be done in one line of code by chaining calls to the **`filter()`** and **`mean()`** methods.

Use **`filter()`** to select the rows of **`titanic`** for which **`Sex`** is equal to **`'male'`**. Then use **`mean()`** to calculate the average of the **`Survived`** column. Print the results. This can all be done in one line of code by chaining calls to the **`filter()`** and **`mean()`** methods.

Create a markdown cell explaining you will now determine the survival rates for passengers in each of the three passenger classes on the Titanic.

Perform these tasks using three separate code cells.

Use **`filter()`** to select the rows of **`titanic`** for which **`Pclass`** is equal to **1**. Then use **`mean()`** to calculate the average of the **`Survived`** column. Print the results. This can all be done in one line of code by chaining calls to the **`filter()`** and **`mean()`** methods.

Use **`filter()`** to select the rows of **`titanic`** for which **`Pclass`** is equal to **2**. Then use **`mean()`** to calculate the average of the **`Survived`** column. Print the results. This can all be done in one line of code by chaining calls to the **`filter()`** and **`mean()`** methods.

Use **`filter()`** to select the rows of **`titanic`** for which **`Pclass`** is equal to **3**. Then use **`mean()`** to calculate the average of the **`Survived`** column. Print the results. This can all be done in one line of code by chaining calls to the **`filter()`** and **`mean()`** methods.

Submission Instructions

When you are done, click **Kernel > Restart and Run All**. If any cell produces an error, then manually run every cell after that one, in order. Save your notebook, and then export the notebook as an HTML file. Upload the HTML file to Canvas and upload the IPYNB file to CoCalc, placing the file into the folder **Projects/Group Project**.