Pandas DataFrames

The `pandas` library is a Python package that provides fast, flexible, and expressive data structures that are designed to make working with relational or labeled data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis/manipulation tool that's available in any language.

The two primary data structures of pandas are Series (one-dimensional) and DataFrames (two-dimensional) and they handle the vast majority of typical use cases. `pandas` is built on top of `NumPy` and is intended to integrate well within a scientific computing environment with many other third-party libraries.

Let's look at a few exercises in order to understand data handling techniques using the `pandas` library.

Activity 1 : Creating a Pandas Series

In this exercise, we will learn how to create a **pandas** series object from the data structures that we created previously. If you have imported pandas as pd, then the function to create a series is simply **pd.Series**. Let's go through the following steps:

1. Import the **NumPy** library and initialize the labels, lists, and a dictionary:
```
import numpy as np
labels = ['a','b','c']
my_data = [10,20,30]
array_1 = np.array(my_data)
d = {'a':10,'b':20,'c':30}
```
2. Import **pandas** as **pd** by using the following command:
```
import pandas as pd
```
3. Create a series from the **my_data** list by using the following command:
```
print("\nHolding numerical data\n",'-
'*25, sep='')
print(pd.Series(array_1))
```
The output is as follows:
```
Holding numerical data
-------------------------
0  10
1  20
2  30
dtype: int64
```

4. Create a series from the **`my_data`** list along with the labels as follows:

```
print("\nHolding text labels\n",'-'*20,
sep='')
print(pd.Series(labels))
```

The output is as follows:

```
Holding text labels
--------------------
0     a
1     b
2     c
dtype: object
```

5. Then, create a series from the **`NumPy`** array, as follows:

```
print("\nHolding functions\n",'-'*20,
sep='')
print(pd.Series(data=[sum,print,len]))
```

The output is as follows:

```
Holding functions
--------------------
0     <built-in function sum>
1     <built-in function print>
2     <built-in function len>
dtype: object
```

6. Create a series from the dictionary, as follows:

```
print("\nHolding objects from a
dictionary\n",'-'*40, sep='')
print(pd.Series(data=[d.keys, d.items,
d.values]))
```

The output is as follows:

```
Holding objects from a dictionary
```

```
----------------------------------------
0     <built-in method keys of dict
object at 0x7fb8...
1     <built-in method items of dict
object at 0x7fb...
2     <built-in method values of dict
object at 0x7f...
dtype: object
```

Note

You may get a different final output because the system may store the object in the memory differently.

In this exercise, we created **pandas** series, which are the building blocks of **pandas** DataFrames. The **pandas series** object can hold many types of data, such as integers, objects, floats, doubles, and others. This is the key to constructing a bigger table where multiple series objects are stacked together to create a database-like entity.

Activity 2: Pandas Series and Data Handling

In this exercise, we will create a **pandas** series using the **pd.series** function. Then, we will manipulate the data in the DataFrame using various handling techniques. Perform the following steps:

1. Create a **`pandas`** series with numerical data by using the following command:

```
import numpy as np
import pandas as pd
labels = ['a','b','c']
my_data = [10,20,30]
array_1 = np.array(my_data)
d = {'a':10,'b':20,'c':30}
print("\nHolding numerical data\n",'-
'*25, sep='')
print(pd.Series(array_1))
```

The output is as follows:

```
Holding numerical data
-------------------------
0     10
1     20
2     30
dtype: int32
```

2. Create a **`pandas`** series with labels by using the following command:

```
print("\nHolding text labels\n",'-'*20,
sep='')
print(pd.Series(labels))
```

The output is as follows:

```
Holding text labels
--------------------
0     a
1     b
2     c
dtype: object
```

3. Create a **pandas** series with functions by using the following command:
```
print("\nHolding functions\n",'-'*20,
sep='')
print(pd.Series(data=[sum,print,len]))
```
The output is as follows:
```
Holding functions
--------------------
0     <built-in function sum>
1     <built-in function print>
2     <built-in function len>
dtype: object
```
4. Create a **pandas** series with a dictionary by using the following command:
```
print("\nHolding objects from a
dictionary\n",'-'*40, sep='')
print(pd.Series(data=[d.keys, d.items,
d.values]))
```
The output is as follows:
```
Holding objects from a dictionary
----------------------------------------
0     <built-in method keys of dict
object at 0x0000...
1     <built-in method items of dict
object at 0x000...
2     <built-in method values of dict
object at 0x00...
dtype: object
```
In this exercise, we created pandas **series** objects using various types of lists.

Activity 3: Creating Pandas DataFrames

The **`pandas`** DataFrame is similar to an Excel table or relational database (SQL) table, which consists of three main components: the data, the index (or rows), and the columns. Under the hood, it is a stack of **`pandas`** series objects, which are themselves built on top of **`NumPy`** arrays. So, all of our previous knowledge of NumPy arrays applies here. Let's perform the following steps:

1. Create a simple DataFrame from a two-dimensional matrix of numbers. First, the code draws **`20`** random integers from the uniform distribution. Then, we need to reshape it into a (**`5,4`**) NumPy array – **`5`** rows and **`4`** columns:
   ```
   import numpy as np
   import pandas as pd
   matrix_data =
   np.random.randint(1,10,size=20).reshape(
   5,4)
   ```
2. Define the rows labels as (**`'A','B','C','D','E'`**) and column labels as (**`'W','X','Y','Z'`**):
   ```
   row_labels = ['A','B','C','D','E']
   column_headings = ['W','X','Y','Z']
   ```
3. Create a DataFrame using **`pd.DataFrame`**:
   ```
   df = pd.DataFrame(data=matrix_data,
   index=row_labels, \
   ```

```
                          columns=column_heading
s)
```

4. Print the DataFrame:

```
print("\nThe data frame looks like\n",'-
'*45, sep='')
print(df)
```

The sample output is as follows:

```
The data frame looks like
---------------------------------------------
   W  X  Y  Z
A  4  3  8  9
B  7  8  1  2
C  7  8  1  1
D  7  9  5  7
E  7  6  1  8
```

Figure 1: Output of the DataFrame

5. Create a DataFrame from a Python dictionary of the lists of integers by using the following command:

```
d={'a':[10,20],'b':[30,40],'c':[50,60]}
```

6. Pass this dictionary as a data argument to the **`pd.DataFrame`** function. Pass on a list of rows or indices. Notice how the dictionary keys became the column names and that the values were distributed among multiple rows:

```
df2=pd.DataFrame(data=d,index=['X','Y'])
print(df2)
```

The output is as follows:

```
    a   b   c
X  10  30  50
Y  20  40  60
```

Figure 2: Output of DataFrame df2

In this exercise, we created DataFrames manually from scratch, which will allow us to understand DataFrames better.

Note

The most common way that you will create a pandas DataFrame will be to read tabular data from a file on your local disk or over the internet – CSV, text, JSON, HTML, Excel, and so on. We will cover some of these in the next chapter.

Activity 4: Viewing a DataFrame Partially

In the previous exercise, we used **`print(df)`** to print the whole DataFrame. For a large dataset, we would like to print only sections of data. In this exercise, we will read a part of the DataFrame. Let's learn how to do so:

1. Import the **`NumPy`** library and execute the following code to create a DataFrame with **`25`** rows. Then, fill it with random numbers:
```
# 25 rows and 4 columns
import numpy as np
import pandas as pd
matrix_data =
np.random.randint(1,100,100).reshape(25,
4)
```

```
column_headings = ['W','X','Y','Z']
df =
pd.DataFrame(data=matrix_data,columns=co
lumn_headings)
```

2. Run the following code to view only the first five rows of the DataFrame:

```
df.head()
```

The sample output is as follows (note that your output could be different due to randomness):

	W	X	Y	Z
0	70	96	7	77
1	96	73	15	74
2	50	52	61	33
3	62	4	10	37
4	3	54	59	8

Figure 3: The first five rows of the DataFrame

By default, **head** shows only five rows. If you want to see any specific number of rows, just pass that as an argument.

3. Print the first eight rows by using the following command:

```
df.head(8)
```

The sample output is as follows:

	W	X	Y	Z
0	70	96	7	77
1	96	73	15	74
2	50	52	61	33
3	62	4	10	37
4	3	54	59	8
5	49	57	41	94
6	21	24	48	23
7	7	2	53	2

Figure 4: The first eight rows of the DataFrame

Just like **`head`** shows the first few rows, **`tail`** shows the last few rows.

4. Print the DataFrame using the **`tail`** command, as follows:

```
df.tail(10)
```

The sample output (partially shown) is as follows:

	W	X	Y	Z
17	27	21	88	63
18	58	50	35	66
19	50	77	14	10
20	29	54	68	26
21	13	61	89	84
22	11	37	42	16
23	83	22	12	43
24	13	58	13	27

Figure 5: The last few rows of the DataFrame

In this section, we learned how to view portions of the DataFrame without looking at the whole

DataFrame. In the next section, we're going to look at two functionalities: indexing and slicing columns in a DataFrame.

Activity 5: Creating and Deleting a New Column or Row

In this exercise, we're going to create and delete a new column or a row from the **`stock.csv`** dataset. We'll also use the **`inplace`** function to modify the original DataFrame.

Note

The **`stock.csv`** *file can be found in the folder.*

Let's go through the following steps:

1. Import the necessary Python modules, load the **`stocks.csv`** file, and create a new column using the following snippet:
```
import pandas as pd
df = pd.read_csv("stock.csv")
df.head()
print("\nA column is created by
assigning it in relation\n",\
      '-'*75, sep='')
df['New'] = df['Price']+df['Price']
df['New (Sum of X and Z)'] =
df['New']+df['Price']
print(df)
```

Note

Don't forget to change the path (highlighted) based on the location of the file on your system.

The sample output (partially shown) is as follows:

```
A column is created by assigning it in relation
-----------------------------------------------------------------
   Symbol  Price  New  New (Sum of X and Z)
0     MMM    100  200                   300
1     AOS    101  202                   303
2     ABT    102  204                   306
3    ABBV    103  206                   309
4     ACN    104  208                   312
5    ATVI    105  210                   315
6     AYI    106  212                   318
7    ADBE    107  214                   321
8     AAP    108  216                   324
9     AMD    109  218                   327
10    AES    110  220                   330
11    AET    111  222                   333
12    AMG    112  224                   336
13    AFL    113  226                   339
14      A    114  228                   342
15    APD    115  230                   345
16   AKAM    116  232                   348
17    ALK    117  234                   351
18    ALB    118  236                   354
```

Figure 6: Partial output of the DataFrame

2. Drop a column using the **`df.drop`** method:

```
print("\nA column is dropped by using df.drop() method\n",\
      '-'*55, sep='')
df = df.drop('New', axis=1) # Notice the axis=1 option
# axis = 0 is default, so one has to change it to 1
print(df)
```

The sample output (partially shown) is as follows:

```
A column is dropped by using df.drop() method
-----------------------------------------------------
   Symbol  Price  New (Sum of X and Z)
0     MMM    100                   300
1     AOS    101                   303
2     ABT    102                   306
3    ABBV    103                   309
4     ACN    104                   312
5    ATVI    105                   315
6     AYI    106                   318
7    ADBE    107                   321
8     AAP    108                   324
9     AMD    109                   327
10    AES    110                   330
11    AET    111                   333
12    AMG    112                   336
13    AFL    113                   339
14      A    114                   342
15    APD    115                   345
16   AKAM    116                   348
17    ALK    117                   351
```

Figure 7: Partial output of the DataFrame

3. Drop a specific row using the **`df.drop`** method:

```
df1=df.drop(1)
print("\nA row is dropped by using
df.drop method and axis=0\n",\
      '-'*65, sep='')
print(df1)
```

The partial output is as follows:

```
A row is dropped by using df.drop method and axis=0
---------------------------------------------------------
   Symbol  Price  New (Sum of X and Z)
0     MMM    100                   300
2     ABT    102                   306
3    ABBV    103                   309
4     ACN    104                   312
5    ATVI    105                   315
6     AYI    106                   318
7    ADBE    107                   321
8     AAP    108                   324
9     AMD    109                   327
10    AES    110                   330
11    AET    111                   333
12    AMG    112                   336
13    AFL    113                   339
14      A    114                   342
```

Figure 8: Partial output of the DataFrame

Dropping methods creates a copy of the DataFrame and does not change the original DataFrame.

4. Change the original DataFrame by setting the **`inplace`** argument to **`True`**:

```
print("\nAn in-place change can be done by making ",\
      "inplace=True in the drop method\n",\
      '-'*75, sep='')
df.drop('New (Sum of X and Z)', axis=1, inplace=True)
print(df)
```

The sample output is as follows:

```
An in-place change can be done by making inplace=True in the drop method
--------------------------------------------------------------------------
   Symbol  Price
0     MMM    100
1     AOS    101
2     ABT    102
3    ABBV    103
4     ACN    104
5    ATVI    105
6     AYI    106
7    ADBE    107
8     AAP    108
9     AMD    109
10    AES    110
11    AET    111
12    AMG    112
13    AFL    113
14      A    114
```

Figure 9: Partial Output of the DataFrame

We have now learned how to modify DataFrames by dropping or adding rows and columns.

Note

All the normal operations are not in-place, that is, they do not impact the original DataFrame object and return a copy of the original with addition (or deletion) instead. The last bit of the preceding code shows how to make a change in the existing DataFrame with the **`inplace=True`** *argument. Please note that this change is irreversible and should be used with caution.*