```
'Output':
   age state_of_origin
0   15           Lagos
1   17     Cross River
2   21            Kano
3   29            Abia
4   25           Benue

# drop all rows less than 20
my_DF.drop(my_DF[my_DF['age'] < 20].index, inplace=True)
my_DF

'Output':
   age state_of_origin
2   21            Kano
3   29            Abia
4   25           Benue
```

Adding a Row/Column

We can add a new column to a Pandas DataFrame by using the **assign** method.

```
# show dataframe
my_DF = pd.DataFrame({'age': [15,17,21,29,25], \
            'state_of_origin':['Lagos', 'Cross River', 'Kano', 'Abia',
            'Benue']})
my_DF
'Output':
   age state_of_origin
0   15           Lagos
1   17     Cross River
2   21            Kano
3   29            Abia
4   25           Benue
# add column to data frame
```

```
my_DF = my_DF.assign(capital_city = pd.Series(['Ikeja', 'Calabar', \
                                               'Kano', 'Umuahia',
                                               'Makurdi']))
my_DF
'Output':
   age state_of_origin capital_city
0   15           Lagos        Ikeja
1   17     Cross River      Calabar
2   21            Kano         Kano
3   29            Abia      Umuahia
4   25           Benue      Makurdi
```

We can also add a new DataFrame column by computing some function on another column. Let's take an example by adding a column computing the absolute difference of the ages from their mean.

```
mean_of_age = my_DF['age'].mean()
my_DF['diff_age'] = my_DF['age'].map(lambda x: abs(x-mean_of_age))
my_DF
'Output':
   age state_of_origin  diff_age
0   15           Lagos       6.4
1   17     Cross River       4.4
2   21            Kano       0.4
3   29            Abia       7.6
4   25           Benue       3.6
```

Typically in practice, a fully formed dataset is converted into Pandas for cleaning and data analysis, which does not ideally involve adding a new observation to the dataset. But in the event that this is desired, we can use the **append()** method to achieve this. However, it may not be a computationally efficient action. Let's see an example.

```
# show dataframe
my_DF = pd.DataFrame({'age': [15,17,21,29,25], \
            'state_of_origin':['Lagos', 'Cross River', 'Kano', 'Abia',
            'Benue']})
my_DF
```

```
'Output':
   age state_of_origin
0   15           Lagos
1   17     Cross River
2   21            Kano
3   29            Abia
4   25           Benue
# add a row to data frame
my_DF = my_DF.append(pd.Series([30 , 'Osun'], index=my_DF.columns), \
                                                    ignore_index=True)
my_DF
'Output':
   age state_of_origin
0   15           Lagos
1   17     Cross River
2   21            Kano
3   29            Abia
4   25           Benue
5   30            Osun
```

We observe that adding a new row involves passing to the **append** method, a **Series** object with the **index** attribute set to the columns of the main DataFrame. Since typically, in given datasets, the index is nothing more than the assigned defaults, we set the attribute **ignore_index** to create a new set of default index values with the new row(s).

Data Alignment

Pandas utilizes data alignment to align indices when performing some binary arithmetic operation on DataFrames. If two or more DataFrames in an arithmetic operation do not share a common index, a **NaN** is introduced denoting missing data. Let's see examples of this.

```
# create a 3x3 dataframe - remember randint(low, high, size)
df_A = pd.DataFrame(np.random.randint(1,10,[3,3]),\
            columns=['First','Second','Third'])
df_A
```