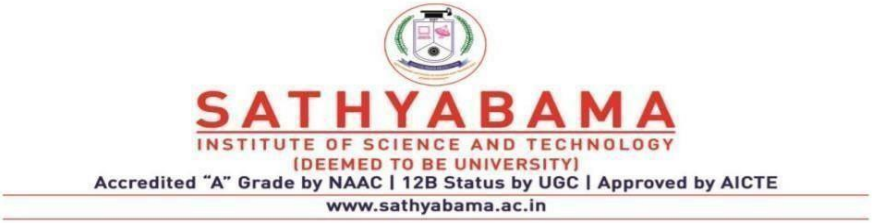


**SCHOOL OF COMPUTING**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

# UNIT-III SCSB3018 – Python for Data Science

# DATA PREPROCESSING USING PYTHON

Python Data Operations - Python Data cleansing - Python Processing CSV Data, JSON Data, XLS Data - Scaling – Normalization - Relational databases - NoSQL Databases - Date and Time - Data Wrangling - Data Aggregation - Reading HTML Pages - Processing Unstructured Data - word tokenization - Stemming and Lemmatization.

## 3.1 Python Data Operations

Python handles data of various formats mainly through the two libraries, Pandas and Numpy. We have already seen the important features of these two libraries in the previous chapters. In this chapter we will see some basic examples from each of the libraries on how to operate on data.

### 3.1.1 Data Operations in Numpy

The core object in NumPy is the ndarray, an N-dimensional array of elements of the same type. Elements are accessed using zero-based indexing. An ndarray can be created using various array creation functions, with the most basic one being

```
numpy.array
```

**Example**

```
# more than one dimensions
import numpy as np
a = np.array([[1, 2], [3, 4]])
print a
```

**Output**

```
[[1, 2]
 [3, 4]]
```

### 3.1.2 Data Operations in Pandas

Pandas handles data through Series, Data Frame, and Panel. We will see some examples from each of these.

**Pandas Series**

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index. A pandas Series can be created using the following constructor −

```
pandas.Series( data, index, dtype, copy)
```

### 3.1.3 Pandas DataFrame

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. A pandas DataFrame can be created using the following constructor –

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print df
```

**Output**

```
          Age     Name
rank1     28       Tom
rank2     34      Jack
rank3     29     Steve
rank4     42     Ricky
```

### 3.1.4 Pandas Panel

A **panel** is a 3D container of data. The term **Panel data** is derived from econometrics and is partially responsible for the name pandas − **pan(el)-da(ta)**-s.

A Panel can be created using the following constructor −

```
pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)
```

**Example to create a panel from dict of DataFrame Objects**

```
#creating an empty panel
import pandas as pd
import numpy as np

data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),
        'Item2' : pd.DataFrame(np.random.randn(4, 2))}
p = pd.Panel(data)
print p
```

**O/P**

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 4
```

## 3.2 Python Data Cleansing

Missing data is always a problem in real life scenarios. Areas like machine learning and data mining face severe issues in the accuracy of their model predictions because of poor quality of data caused by missing values. In these areas, missing value treatment is a major point of focus to make their models more accurate and valid.

Let us consider an online survey for a product. Many a times, people do not share all the information related to them. Few people share their experience, but not how long they are using the product; few people share how long they are using the product, their experience but not their contact information. Thus, in some or the other way a part of data is always missing, and this is very common in real time.

**Example for how we can handle missing values (say NA or NaN) using Pandas:**

```
# import the pandas library
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df
```

**Output**

```
        one       two     three
a  0.077988  0.476149  0.965836
b       NaN       NaN       NaN
c -0.390208 -0.551605 -2.301950
d       NaN       NaN       NaN
e -2.000303 -0.788201  1.510072
f -0.930230 -0.670473  1.146615
g       NaN       NaN       NaN
h  0.085100  0.532791  0.887415
```

### 3.2.1 Check for Missing Values

To make detecting missing values easier (and across different array dtypes),

Pandas provides the **isnull()** and **notnull()** functions, which are also methods on Series and DataFrame objects

**Example**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].isnull()
```

**O/P**

```
a  False
b  True
c  False
d  True
e  False
f  False
g  True
h  False
Name: one, dtype: bool
```

### 3.2.2 Cleaning / Filling Missing Data

Pandas provides various methods for cleaning the missing values. The fillna function can fill in NA values with non-null data in a couple of ways, which we have illustrated in the following sections.

#### 3.2.2.1 Replace NaN with a Scalar Value

The following program shows how you can replace "NaN" with "0".

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'c', 'e'],columns=['one',
'two', 'three'])
df = df.reindex(['a', 'b', 'c'])
print df
print ("NaN replaced with '0':")
print df.fillna(0)
```

**O/P**

```
          one          two        three
a   -0.576991   -0.741695   0.553172
b          NaN          NaN         NaN
c    0.744328   -1.735166   1.749580

NaN replaced with '0':
          one          two        three
a   -0.576991   -0.741695   0.553172
b    0.000000    0.000000   0.000000
c    0.744328   -1.735166   1.749580
```

### 3.2.3 Fill NA Forward and Backward

| Method | Action |
|---|---|
| pad/fill | Fill methods Forward |
| bfill/backfill | Fill methods Backward |

**Example**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.fillna(method='pad')
```

**O/P**

```
        one       two     three
a  0.077988  0.476149  0.965836
b  0.077988  0.476149  0.965836
c -0.390208 -0.551605 -2.301950
d -0.390208 -0.551605 -2.301950
e -2.000303 -0.788201  1.510072
f -0.930230 -0.670473  1.146615
g -0.930230 -0.670473  1.146615
h  0.085100  0.532791  0.887415
```

### 3.2.4 Drop Missing Values

If you want to simply exclude the missing values, then use the **dropna** function along with the **axis** argument. By default, axis=0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

**Example**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.dropna()
```

**O/P**

```
          one         two       three
a    0.077988    0.476149    0.965836
c   -0.390208   -0.551605   -2.301950
e   -2.000303   -0.788201    1.510072
f   -0.930230   -0.670473    1.146615
h    0.085100    0.532791    0.887415
```

### 3.2.5 Replace Missing (or) Generic Values

Many times, we have to replace a generic value with some specific value. We can achieve this by applying the replace method.

Replacing NA with a scalar value is equivalent behavior of the **fillna()** function.

**Example**

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'one':[10,20,30,40,50,2000],
'two':[1000,0,30,40,50,60]})
print df.replace({1000:10,2000:60})
```

**O/P**

```
   one  two
0   10   10
1   20    0
2   30   30
3   40   40
4   50   50
5   60   60
```

### 3.3 Python - Processing CSV Data

**Reading CSV Data** is a key step in data science, as data from various sources is often exported to CSV (Comma Separated Values) format. Python's pandas library makes it easy to read and manipulate this data.

### 3.3.1 Input as CSV File

The csv file is a text file in which the values in the columns are separated by a comma. Let's consider the following data present in the file named **input.csv**.

You can create this file using windows notepad by copying and pasting this data. Save the file as **input.csv** using the save As All files(*.*) option in notepad.

```
id,name,salary,start_date,dept
1,Rick,623.3,2012-01-01,IT
2,Dan,515.2,2013-09-23,Operations
3,Tusar,611,2014-11-15,IT
4,Ryan,729,2014-05-11,HR
5,Gary,843.25,2015-03-27,Finance
6,Rasmi,578,2013-05-21,IT
7,Pranab,632.8,2013-07-30,Operations
8,Guru,722.5,2014-06-17,Finance
```

### 3.3.2 Reading a CSV File

The **read_csv** function of the pandas library is used read the content of a CSV file into the python environment as a pandas DataFrame. The function can read the files from the OS by using proper path to the file.

**Example**

```
import pandas as pd
data = pd.read_csv('path/input.csv')
print (data)
```

**O/P**

```
   id     name   salary  start_date         dept
0   1     Rick   623.30  2012-01-01           IT
1   2      Dan   515.20  2013-09-23   Operations
2   3    Tusar   611.00  2014-11-15           IT
3   4     Ryan   729.00  2014-05-11           HR
4   5     Gary   843.25  2015-03-27      Finance
5   6    Rasmi   578.00  2013-05-21           IT
6   7  Pranab   632.80  2013-07-30   Operations
7   8     Guru   722.50  2014-06-17      Finance
```

### 3.3.3 Reading Specific Rows

The read_csv function of the pandas library can also be used to read some specific rows for a given column. We slice the result from the read_csv function using the code shown below for first 5 rows for the column named salary.

**Example**

```
import pandas as pd
data = pd.read_csv('path/input.csv')

# Slice the result for first 5 rows
print (data[0:5]['salary'])
```

**O/P**

```
0    623.30
1    515.20
2    611.00
3    729.00
4    843.25
Name: salary, dtype: float64
```

### 3.3.4 Reading Specific Columns

The **read_csv** function of the pandas library can also be used to read some specific columns. We use the multi-axes indexing method called **.loc()** for this purpose. We choose to display the salary and name column for all the rows.

**Example**

```
import pandas as pd
data = pd.read_csv('path/input.csv')

# Use the multi-axes indexing funtion
print (data.loc[:,['salary','name']])
```

**O/P**

```
    salary     name
0   623.30     Rick
1   515.20      Dan
2   611.00    Tusar
3   729.00     Ryan
4   843.25     Gary
5   578.00    Rasmi
6   632.80   Pranab
7   722.50     Guru
```

### 3.3.5 Reading Specific Columns and Rows

The **read_csv** function of the pandas library can also be used to read some specific columns and specific rows. We use the multi-axes indexing method called **.loc()** for this purpose. We choose to display the salary and name column for some of the rows.

**Example**

```
import pandas as pd
data = pd.read_csv('path/input.csv')

# Use the multi-axes indexing funtion
print (data.loc[[1,3,5],['salary','name']])
```

**O/P**

```
    salary    name
1    515.2     Dan
3    729.0    Ryan
5    578.0   Rasmi
```

### 3.3.6 Reading Specific Columns for a Range of Rows

The **read_csv** function of the pandas library can also be used to read some specific columns and a range of rows. We use the multi-axes indexing method called **.loc()** for this purpose. We choose to display the salary and name column for some of the

rows.

**Example:**

```
import pandas as pd
data = pd.read_csv('path/input.csv')

# Use the multi-axes indexing funtion
print (data.loc[2:6,['salary','name']])
```

**O/P**

```
     salary     name
2   611.00    Tusar
3   729.00     Ryan
4   843.25     Gary
5   578.00    Rasmi
6   632.80   Pranab
```

### 3.4 Python Processing JSON Data

JSON file stores data as text in human-readable format. JSON stands for JavaScript Object Notation. Pandas can read JSON files using the **read_json** function.

#### 3.4.1 Input Data

Create a JSON file by copying the below data into a text editor like notepad. Save the file with **.json** extension and choosing the file type as **all files(*.*)**.

```
{
   "ID":["1","2","3","4","5","6","7","8" ],
   "Name":["Rick","Dan","Michelle","Ryan","Gary","Nina","Simon","Guru" ]
   "Salary":["623.3","515.2","611","729","843.25","578","632.8","722.5" ],

   "StartDate":[ "1/1/2012","9/23/2013","11/15/2014","5/11/2014","3/27/2015","5/21/2013",
      "7/30/2013","6/17/2014"],
   "Dept":[ "IT","Operations","IT","HR","Finance","IT","Operations","Finance"]
}
```

### 3.4.2 Read the JSON File

The **read_json** function of the pandas library can be used to read the JSON file into a pandas DataFrame.

**Example**

```
import pandas as pd

data = pd.read_json('path/input.json')
print (data)
```

**O/P**

```
         Dept  ID    Name  Salary   StartDate
0          IT   1    Rick  623.30    1/1/2012
1  Operations   2     Dan  515.20   9/23/2013
2          IT   3   Tusar  611.00  11/15/2014
3          HR   4    Ryan  729.00   5/11/2014
4     Finance   5    Gary  843.25   3/27/2015
5          IT   6   Rasmi  578.00   5/21/2013
6  Operations   7  Pranab  632.80   7/30/2013
7     Finance   8    Guru  722.50   6/17/2014
```

### 3.4.3 Reading Specific Columns and Rows

Similar to what we have already seen in the previous chapter to read the CSV file, the **read_json** function of the pandas library can also be used to read some specific columns and specific rows after the JSON file is read to a DataFrame. We use the multi-axes indexing method called **.loc()** for this purpose. We choose to display the Salary and Name column for some of the rows.

**Example:**

```
import pandas as pd
data = pd.read_json('path/input.xlsx')

# Use the multi-axes indexing funtion
print (data.loc[[1,3,5],['salary','name']])
```

**O/P**

```
   salary   name
1   515.2    Dan
3   729.0   Ryan
5   578.0  Rasmi
```

### 3.4.4 Reading JSON file as Records

We can also apply the **to_json** function along with parameters to read the JSON file content into individual records.

**Example:**

```
import pandas as pd
data = pd.read_json('path/input.xlsx')

print(data.to_json(orient='records', lines=True))
```

**O/P**

```
{"Dept":"IT","ID":1,"Name":"Rick","Salary":623.3,"StartDate":"1\/1\/2012"}
{"Dept":"Operations","ID":2,"Name":"Dan","Salary":515.2,"StartDate":"9\/23\/2013"}
{"Dept":"IT","ID":3,"Name":"Tusar","Salary":611.0,"StartDate":"11\/15\/2014"}
{"Dept":"HR","ID":4,"Name":"Ryan","Salary":729.0,"StartDate":"5\/11\/2014"}
{"Dept":"Finance","ID":5,"Name":"Gary","Salary":843.25,"StartDate":"3\/27\/2015"}
{"Dept":"IT","ID":6,"Name":"Rasmi","Salary":578.0,"StartDate":"5\/21\/2013"}
{"Dept":"Operations","ID":7,"Name":"Pranab","Salary":632.8,"StartDate":"7\/30\/2013"}
{"Dept":"Finance","ID":8,"Name":"Guru","Salary":722.5,"StartDate":"6\/17\/2014"}
```

### 3.5 Python Processing XLS Data

Microsoft Excel is a very widely used spread sheet program. Its user friendliness

and appealing features makes it a very frequently used tool in Data Science. The Panadas library provides features using which we can read the Excel file in full as well as in parts for only a selected group of Data. We can also read an Excel file with multiple sheets in it. We use the **read_excel** function to read the data from it.

### 3.5.1 Input as Excel File

We Create an excel file with multiple sheets in the windows OS. The Data in the different sheets is as shown below.

You can create this file using the Excel Program in windows OS. Save the file as **input.xlsx**.

```
# Data in Sheet1

id,name,salary,start_date,dept
1,Rick,623.3,2012-01-01,IT
2,Dan,515.2,2013-09-23,Operations
3,Tusar,611,2014-11-15,IT
4,Ryan,729,2014-05-11,HR
5,Gary,843.25,2015-03-27,Finance
6,Rasmi,578,2013-05-21,IT
7,Pranab,632.8,2013-07-30,Operations
8,Guru,722.5,2014-06-17,Finance

# Data in Sheet2

id  name    zipcode
1   Rick    301224
2   Dan 341255
3   Tusar   297704
4   Ryan    216650
5   Gary    438700
6   Rasmi   665100
7   Pranab  341211
8   Guru    347480
```

### 3.5.2 Reading an Excel File

The **read_excel** function of the pandas library is used read the content of an Excel file

into the python environment as a pandas DataFrame. The function can read the files from the OS by using proper path to the file. By default, the function will read Sheet1.

**Example:**

```
import pandas as pd
data = pd.read_excel('path/input.xlsx')
print (data)
```

**O/P**

```
   id    name  salary  start_date        dept
0   1    Rick  623.30  2012-01-01          IT
1   2     Dan  515.20  2013-09-23  Operations
2   3   Tusar  611.00  2014-11-15          IT
3   4    Ryan  729.00  2014-05-11          HR
4   5    Gary  843.25  2015-03-27     Finance
5   6   Rasmi  578.00  2013-05-21          IT
6   7  Pranab  632.80  2013-07-30  Operations
7   8    Guru  722.50  2014-06-17     Finance
```

### 3.5.3 Reading Specific Columns and Rows

Similar to what we have already seen in the previous chapter to read the CSV file, the **read_excel** function of the pandas library can also be used to read some specific columns and specific rows. We use the multi-axes indexing method called **.loc()** for this purpose. We choose to display the salary and name column for some of the rows.

**Example:**

```
import pandas as pd
data = pd.read_excel('path/input.xlsx')

# Use the multi-axes indexing funtion
print (data.loc[[1,3,5],['salary','name']])
```

**O/P**

```
    salary    name
1    515.2     Dan
3    729.0    Ryan
5    578.0   Rasmi
```

### 3.5.4 Reading Multiple Excel Sheets

Multiple sheets with different Data formats can also be read by using read_excel function with help of a wrapper class named **ExcelFile**. It will read the multiple sheets into memory only once. I

**Example:** Read sheet1 and sheet2 into two data frames and print them out individually.

```
import pandas as pd
with pd.ExcelFile('C:/Users/Rasmi/Documents/pydatasci/input.xlsx') as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')

print("****Result Sheet 1****")
print (df1[0:5]['salary'])
print("")
print("***Result Sheet 2****")
print (df2[0:5]['zipcode'])
```

**O/P**

```
****Result Sheet 1****
0     623.30
1     515.20
2     611.00
3     729.00
4     843.25
Name: salary, dtype: float64

***Result Sheet 2****
0     301224
1     341255
2     297704
3     216650
4     438700
Name: zipcode, dtype: int64
```

### 3.6 Scaling

We use this preprocessing technique for scaling the feature vectors. Scaling of feature vectors is important, because the features should not be synthetically large or small.

**Example**

```
import numpy as np
from sklearn import preprocessing
Input_data = np.array(
    [
        [2.1, -1.9, 5.5],
        [-1.5, 2.4, 3.5],
        [0.5, -7.9, 5.6],
        [5.9, 2.3, -5.8]
    ]
 )
data_scaler_minmax = preprocessing.MinMaxScaler(feature_range=(0,1))
data_scaled_minmax = data_scaler_minmax.fit_transform(input_data)
print ("\nMin max scaled data:\n", data_scaled_minmax)
```

**Output**

```
Min max scaled data:
[
  [ 0.48648649 0.58252427 0.99122807]
  [ 0. 1. 0.81578947]
  [ 0.27027027 0. 1. ]
  [ 1. 0.99029126 0. ]
]
```

### 3.7 Normalisation

We use this preprocessing technique for modifying the feature vectors.

Normalisation of feature vectors is necessary so that the feature vectors can be measured at common scale. There are two types of normalisation as follows −

**L1 Normalisation**

It is also called Least Absolute Deviations. It modifies the value in such a manner that the sum of the absolute values remains always up to 1 in each row. Following example shows the implementation of L1 normalisation on input data.

**Example**

```
import numpy as np
from sklearn import preprocessing Input_data
=np.array(
[
  [2.1, -1.9, 5.5],
  [-1.5, 2.4, 3.5],
  [0.5, -7.9, 5.6],
  [5.9, 2.3, -5.8]
  ])
  data_normalized_l1 = preprocessing.normalize(input_data, norm='l1')
  print("\nL1 normalized data:\n", data_normalized_l1)
```

**Output**

```
L1 normalized data:
[
  [ 0.22105263 -0.2 0.57894737]
  [-0.2027027 0.32432432 0.47297297]
  [ 0.03571429 -0.56428571 0.4 ]
  [ 0.42142857 0.16428571 -0.41428571]
]
```

**L2 Normalisation**

Also called Least Squares. It modifies the value in such a manner that the sum of the squares remains always up to 1 in each row. Following example shows the implementation of L2 normalisation on input data.

**Example**

```
import numpy as np
from sklearn import preprocessing
Input_data = np.array(
    [
        [2.1, -1.9, 5.5],
        [-1.5, 2.4, 3.5],
        [0.5, -7.9, 5.6],
        [5.9, 2.3, -5.8]
    ]
)
data_normalized_l2 = preprocessing.normalize(input_data, norm='l2')
print("\nL1 normalized data:\n", data_normalized_l2)
```

**Output**

```
L2 normalized data:
[
  [ 0.33946114 -0.30713151 0.88906489]
  [-0.33325106 0.53320169 0.7775858 ]
  [ 0.05156558 -0.81473612 0.57753446]
  [ 0.68706914 0.26784051 -0.6754239 ]
]
```

### 3.8 Python – Relational Databases

We can connect to relational databases for analysing data using the **pandas** library as well as another additional library for implementing database connectivity. This package is named as **sqlalchemy** which provides full SQL language functionality to be used in python.

#### 3.8.1 Installing SQLAlchemy

The installation is very straight forward using Anaconda which we have discussed in the chapter Data Science Environment. Assuming you have installed Anaconda as described in this chapter, run the following command in the Anaconda Prompt Window to install the SQLAlchemy package.

```
conda install sqlalchemy
```

#### 3.8.2 Reading Relational Tables

We will use Sqlite3 as our relational database as it is very light weight and easy to use. Though the SQLAlchemy library can connect to a variety of relationa sources including MySql, Oracle and Postgresql and Mssql. We first create a database engine and then connect to the database engine using the **to_sql** function

of the SQLAlchemy library.

In the below example we create the relational table by using the **to_sql** function from a dataframe already created by reading a csv file. Then we use the **read_sql_query** function from pandas to execute and capture the results from various SQL queries.

**Example:**

```
from sqlalchemy import create_engine
import pandas as pd

data = pd.read_csv('/path/input.csv')

# Create the db engine
engine = create_engine('sqlite:///:memory:')

# Store the dataframe as a table
data.to_sql('data_table', engine)

# Query 1 on the relational table
res1 = pd.read_sql_query('SELECT * FROM data_table', engine)
print('Result 1')
print(res1)
print('')

# Query 2 on the relational table
res2 = pd.read_sql_query('SELECT dept,sum(salary) FROM data_table group by dept', engine)
print('Result 2')
print(res2)
```

**O/P**

```
Result 1
   index  id     name  salary  start_date         dept
0      0   1     Rick  623.30  2012-01-01           IT
1      1   2      Dan  515.20  2013-09-23   Operations
2      2   3    Tusar  611.00  2014-11-15           IT
3      3   4     Ryan  729.00  2014-05-11           HR
4      4   5     Gary  843.25  2015-03-27      Finance
5      5   6    Rasmi  578.00  2013-05-21           IT
6      6   7   Pranab  632.80  2013-07-30   Operations
7      7   8     Guru  722.50  2014-06-17      Finance

Result 2
          dept  sum(salary)
0      Finance      1565.75
1           HR       729.00
2           IT      1812.30
3  Operations      1148.00
```

### 3.8.3 Inserting Data to Relational Tables

We can also insert data into relational tables using sql.execute function available in pandas. In the below code we previous csv file as input data set, store it in a relational table and then insert another record using sql.execute.

```
from sqlalchemy import create_engine
from pandas.io import sql

import pandas as pd

data = pd.read_csv('C:/Users/Rasmi/Documents/pydatasci/input.csv')
engine = create_engine('sqlite:///:memory:')

# Store the Data in a relational table
data.to_sql('data_table', engine)

# Insert another row
sql.execute('INSERT INTO data_table VALUES(?,?,?,?,?,?)', engine,
params=[('id',9,'Ruby',711.20,'2015-03-27','IT')])

# Read from the relational table
res = pd.read_sql_query('SELECT ID,Dept,Name,Salary,start_date FROM
data_table', engine)
print(res)
```

**O/P**

```
   id         dept     name  salary  start_date
0   1           IT     Rick  623.30  2012-01-01
1   2   Operations      Dan  515.20  2013-09-23
2   3           IT    Tusar  611.00  2014-11-15
3   4           HR     Ryan  729.00  2014-05-11
4   5      Finance     Gary  843.25  2015-03-27
5   6           IT    Rasmi  578.00  2013-05-21
6   7   Operations   Pranab  632.80  2013-07-30
7   8      Finance     Guru  722.50  2014-06-17
8   9           IT     Ruby  711.20  2015-03-27
```

### 3.8.4 Deleting Data from Relational Tables

We can also delete data into relational tables using sql.execute function available in pandas. The below code deletes a row based on the input condition given.

**Example:**

```
from sqlalchemy import create_engine
from pandas.io import sql

import pandas as pd

data = pd.read_csv('C:/Users/Rasmi/Documents/pydatasci/input.csv')
engine = create_engine('sqlite:///:memory:')
data.to_sql('data_table', engine)

sql.execute('Delete from data_table where name = (?) ', engine,  params=[('Gary')])

res = pd.read_sql_query('SELECT ID,Dept,Name,Salary,start_date FROM data_table', engine)
print(res)
```

**O/P**

```
   id         dept     name  salary  start_date
0   1           IT     Rick   623.3  2012-01-01
1   2   Operations      Dan   515.2  2013-09-23
2   3           IT    Tusar   611.0  2014-11-15
3   4           HR     Ryan   729.0  2014-05-11
4   6           IT    Rasmi   578.0  2013-05-21
5   7   Operations   Pranab   632.8  2013-07-30
6   8      Finance     Guru   722.5  2014-06-17
```

### 3.9 Python - NoSQL Databases

As unstructured and semi-structured data increases, the use of **NoSQL databases** like **MongoDB** becomes essential. Python interacts with NoSQL databases similarly to relational databases. To connect Python with MongoDB, the **pymongo** library is used. You can install it using the following command in Anaconda:

```
conda install pymongo
```

This library enables python to connect to MOngoDB using a db client. Once connected we select the db name to be used for various operations.

### 3.9.1 Inserting Data

To insert data into MongoDB we use the insert() method which is available in the database environment. First we connect to the db using python code shown below and then we provide the document details in form of a series of key-value pairs.

**Example:**

```
# Import the python libraries
from pymongo import MongoClient
from pprint import pprint

# Choose the appropriate client
client = MongoClient()

# Connect to the test db
db=client.test

# Use the employee collection
employee = db.employee
employee_details = {
    'Name': 'Raj Kumar',
    'Address': 'Sears Streer, NZ',
    'Age': '42'
}
```

**O/P :**

```
{u'Address': u'Sears Streer, NZ',
 u'Age': u'42',
 u'Name': u'Raj Kumar',
 u'_id': ObjectId('5adc5a9f84e7cd3940399f93')}
```

### 3.9.2. Updating Data

Updating an existing MongoDB data is similar to inserting. We use the update() method which is native to mongoDB. In the below code we are replacing the

existing record with new key-value pairs. Please note how we are using the condition criteria to decide which record to update.

**Example:**

```
# Import the python libraries
from pymongo import MongoClient
from pprint import pprint

# Choose the appropriate client
client = MongoClient()

# Connect to db
db=client.test
employee = db.employee

# Use the condition to choose the record
# and use the update method
```

```
db.employee.update_one(
        {"Age":'42'},
        {
        "$set": {
            "Name":"Srinidhi",
            "Age":'35',
            "Address":"New Omsk, WC"
        }
        }
    )

Queryresult = employee.find_one({'Age':'35'})

pprint(Queryresult)
```

**O/P:**

```
{u'Address': u'New Omsk, WC',
 u'Age': u'35',
 u'Name': u'Srinidhi',
 u'_id': ObjectId('5adc5a9f84e7cd3940399f93')}
```

### 3.9.3. Deleting Data

Deleting a record is also straight forward where we use the delete method. Here also we mention the condition which is used to choose the record to be deleted.

**Example:**

```
# Import the python libraries
from pymongo import MongoClient
from pprint import pprint

# Choose the appropriate client
client = MongoClient()

# Connect to db
db=client.test
employee = db.employee

# Use the condition to choose the record
# and use the delete method
db.employee.delete_one({"Age":'35'})

Queryresult = employee.find_one({'Age':'35'})

pprint(Queryresult)
```

**O/P:**

None

### 3.10 Python - Date and Time

Often in data science we need analysis which is based on temporal values. Python can handle the various formats of date and time gracefully. The **datetime** library provides necessary methods and functions to handle the following scenarios.

- Date Time Representation
- Date Time Arithmetic
- Date Time Comparison

#### 3.10.1 Date Time Representation

A date and its various parts are represented by using different datetime functions. Also, there are format specifiers which play a role in displaying the alphabetical parts of a date like name of the month or week day. The following code shows

today's date and various parts of the date.

**Example:**

```
import datetime

print 'The Date Today is  :', datetime.datetime.today()

date_today = datetime.date.today()
print date_today
print 'This Year   :', date_today.year
print 'This Month    :', date_today.month
print 'Month Name:',date_today.strftime('%B')
print 'This Week Day    :', date_today.day
print 'Week Day Name:',date_today.strftime('%A')
```

**O/P:**

```
The Date Today is  : 2018-04-22 15:38:35.835000
2018-04-22
This Year   : 2018
This Month    : 4
Month Name: April
This Week Day    : 22
Week Day Name: Sunday
```

### 3.10.2 Date Time Arithmetic

For calculations involving dates we store the various dates into variables and apply the relevant mathematical operator to these variables.

**Example:**

```
import datetime

#Capture the First Date
day1 = datetime.date(2018, 2, 12)
print 'day1:', day1.ctime()

# Capture the Second Date
day2 = datetime.date(2017, 8, 18)
print 'day2:', day2.ctime()

# Find the difference between the dates
print 'Number of Days:', day1-day2


date_today  = datetime.date.today()

# Create a delta of Four Days
no_of_days = datetime.timedelta(days=4)

# Use Delta for Past Date
before_four_days = date_today - no_of_days
print 'Before Four Days:', before_four_days

# Use Delta for future Date
after_four_days = date_today + no_of_days
print 'After Four Days:', after_four_days
```

O/P:

```
day1: Mon Feb 12 00:00:00 2018
day2: Fri Aug 18 00:00:00 2017
Number of Days: 178 days, 0:00:00
Before Four Days: 2018-04-18
After Four Days: 2018-04-26
```

### 3.10.3 Date Time Comparison

Date and time are compared using logical operators. But we must be careful in comparing the right parts of the dates with each other. In the below examples we

take the future and past dates and compare them using the python if clause along with logical operators.

**Example:**

```
import datetime

date_today  = datetime.date.today()

print 'Today is: ', date_today
# Create a delta of Four Days
no_of_days = datetime.timedelta(days=4)

# Use Delta for Past Date
before_four_days = date_today - no_of_days
print 'Before Four Days:', before_four_days

after_four_days =  date_today + no_of_days

date1 = datetime.date(2018,4,4)

print 'date1:',date1

if date1 == before_four_days :
    print 'Same Dates'
if date_today > date1:
    print 'Past Date'
if date1 < after_four_days:
    print 'Future Date'
```

**O/P:**

```
Today is:  2018-04-22
Before Four Days: 2018-04-18
date1: 2018-04-04
Past Date
Future Date
```

### 3.11.Python - Data Wrangling

Data wrangling involves processing the data in various formats like - merging, grouping, concatenating etc. for the purpose of analysing or getting them ready to be used with another set of data. Python has built-in features to apply these

wrangling methods to various data sets to achieve the analytical goal. In this chapter we will look at few examples describing these methods.

### 3.11.1 Merging Data

The Pandas library in python provides a single function, **merge**, as the entry point for all standard database join operations between DataFrame objects –

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
left_index=False, right_index=False, sort=True)
```

**Example**

```
# import the pandas library
import pandas as pd
left = pd.DataFrame({
         'id':[1,2,3,4,5],
         'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
         'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right = pd.DataFrame(
         {'id':[1,2,3,4,5],
         'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
         'subject_id':['sub2','sub4','sub3','sub6','sub5']})
print left
print right
```

**O/P:**

```
    Name  id   subject_id
0   Alex   1         sub1
1    Amy   2         sub2
2  Allen   3         sub4
3  Alice   4         sub6
4  Ayoung  5         sub5

    Name  id   subject_id
0  Billy   1         sub2
1  Brian   2         sub4
2  Bran    3         sub3
3  Bryce   4         sub6
4  Betty   5         sub5
```

### 3.11.2 Grouping Data

Grouping data sets is a frequent need in data analysis where we need the result in terms of various groups present in the data set. Panadas has in-built methods which can roll the data into various groups.

**Example:**

```
# import the pandas library
import pandas as pd

ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',
         'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],
         'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],
         'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
         'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)

grouped = df.groupby('Year')
print grouped.get_group(2014)
```

**O/P**

```
   Points  Rank     Team    Year
0     876     1   Riders    2014
2     863     2   Devils    2014
4     741     3   Kings     2014
9     701     4   Royals    2014
```

### 3.11.3 Concatenating Data

Pandas provides various facilities for easily combining together **Series, DataFrame**, and **Panel** objects. In the below example the **concat** function performs concatenation operations along an axis. Let us create different objects and do concatenation.

**Example:**

```
import pandas as pd
one = pd.DataFrame({
         'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
         'subject_id':['sub1','sub2','sub4','sub6','sub5'],
         'Marks_scored':[98,90,87,69,78]},
         index=[1,2,3,4,5])
two = pd.DataFrame({
         'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
         'subject_id':['sub2','sub4','sub3','sub6','sub5'],
         'Marks_scored':[89,80,79,97,88]},
         index=[1,2,3,4,5])
print pd.concat([one,two])
```

**O/P**

```
   Marks_scored     Name   subject_id
1            98     Alex         sub1
2            90      Amy         sub2
3            87    Allen         sub4
4            69    Alice         sub6
5            78   Ayoung         sub5
1            89    Billy         sub2
2            80    Brian         sub4
3            79     Bran         sub3
4            97    Bryce         sub6
5            88    Betty         sub5
```

### 3.12.Python - Data Aggregation

Python has several methods are available to perform aggregations on data. It is done using the pandas and numpy libraries. The data must be available or converted to a dataframe to apply the aggregation functions.

### 3.12.1 Applying Aggregations on DataFrame

**Example:**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
      index = pd.date_range('1/1/2000', periods=10),
      columns = ['A', 'B', 'C', 'D'])

print df

r = df.rolling(window=3,min_periods=1)
print r
```

**O/P**

```
                   A           B           C           D
2000-01-01   1.088512   -0.650942   -2.547450   -0.566858
2000-01-02   0.790670   -0.387854   -0.668132    0.267283
2000-01-03  -0.575523   -0.965025    0.060427   -2.179780
2000-01-04   1.669653    1.211759   -0.254695    1.429166
2000-01-05   0.100568   -0.236184    0.491646   -0.466081
2000-01-06   0.155172    0.992975   -1.205134    0.320958
2000-01-07   0.309468   -0.724053   -1.412446    0.627919
2000-01-08   0.099489   -1.028040    0.163206   -1.274331
2000-01-09   1.639500   -0.068443    0.714008   -0.565969
2000-01-10   0.326761    1.479841    0.664282   -1.361169

Rolling [window=3,min_periods=1,center=False,axis=0]
```

### 3.12.2 Apply Aggregation on a Whole Dataframe

**Example:**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
      index = pd.date_range('1/1/2000', periods=10),
      columns = ['A', 'B', 'C', 'D'])
print df

r = df.rolling(window=3,min_periods=1)
print r.aggregate(np.sum)
```

**O/P**

```
                   A          B          C          D
2000-01-01   1.088512  -0.650942  -2.547450  -0.566858
2000-01-02   1.879182  -1.038796  -3.215581  -0.299575
2000-01-03   1.303660  -2.003821  -3.155154  -2.479355
2000-01-04   1.884801  -0.141119  -0.862400  -0.483331
2000-01-05   1.194699   0.010551   0.297378  -1.216695
2000-01-06   1.925393   1.968551  -0.968183   1.284044
2000-01-07   0.565208   0.032738  -2.125934   0.482797
2000-01-08   0.564129  -0.759118  -2.454374  -0.325454
2000-01-09   2.048458  -1.820537  -0.535232  -1.212381
2000-01-10   2.065750   0.383357   1.541496  -3.201469

                   A          B          C          D
2000-01-01   1.088512  -0.650942  -2.547450  -0.566858
2000-01-02   1.879182  -1.038796  -3.215581  -0.299575
2000-01-03   1.303660  -2.003821  -3.155154  -2.479355
2000-01-04   1.884801  -0.141119  -0.862400  -0.483331
2000-01-05   1.194699   0.010551   0.297378  -1.216695
2000-01-06   1.925393   1.968551  -0.968183   1.284044
2000-01-07   0.565208   0.032738  -2.125934   0.482797
2000-01-08   0.564129  -0.759118  -2.454374  -0.325454
2000-01-09   2.048458  -1.820537  -0.535232  -1.212381
2000-01-10   2.065750   0.383357   1.541496  -3.201469
```

### 3.12.3 Apply Aggregation on a Single Column of a Dataframe

**Example:**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
      index = pd.date_range('1/1/2000', periods=10),
      columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
print r['A'].aggregate(np.sum)
```

**O/P:**

```
                   A         B         C         D
2000-01-01   1.088512  -0.650942  -2.547450  -0.566858
2000-01-02   1.879182  -1.038796  -3.215581  -0.299575
2000-01-03   1.303660  -2.003821  -3.155154  -2.479355
2000-01-04   1.884801  -0.141119  -0.862400  -0.483331
2000-01-05   1.194699   0.010551   0.297378  -1.216695
2000-01-06   1.925393   1.968551  -0.968183   1.284044
2000-01-07   0.565208   0.032738  -2.125934   0.482797
2000-01-08   0.564129  -0.759118  -2.454374  -0.325454
2000-01-09   2.048458  -1.820537  -0.535232  -1.212381
2000-01-10   2.065750   0.383357   1.541496  -3.201469
2000-01-01   1.088512
2000-01-02   1.879182
2000-01-03   1.303660
2000-01-04   1.884801
2000-01-05   1.194699
2000-01-06   1.925393
2000-01-07   0.565208
2000-01-08   0.564129
2000-01-09   2.048458
2000-01-10   2.065750
Freq: D, Name: A, dtype: float64
```

### 3.12.4 Apply Aggregation on Multiple Columns of a DataFrame

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10, 4),
      index = pd.date_range('1/1/2000', periods=10),
      columns = ['A', 'B', 'C', 'D'])
print df
r = df.rolling(window=3,min_periods=1)
print r[['A','B']].aggregate(np.sum)
```

**O/P:**

```
                   A         B         C         D
2000-01-01  1.088512 -0.650942 -2.547450 -0.566858
2000-01-02  1.879182 -1.038796 -3.215581 -0.299575
2000-01-03  1.303660 -2.003821 -3.155154 -2.479355
2000-01-04  1.884801 -0.141119 -0.862400 -0.483331
2000-01-05  1.194699  0.010551  0.297378 -1.216695
2000-01-06  1.925393  1.968551 -0.968183  1.284044
2000-01-07  0.565208  0.032738 -2.125934  0.482797
2000-01-08  0.564129 -0.759118 -2.454374 -0.325454
2000-01-09  2.048458 -1.820537 -0.535232 -1.212381
2000-01-10  2.065750  0.383357  1.541496 -3.201469
                   A         B
2000-01-01  1.088512 -0.650942
2000-01-02  1.879182 -1.038796
2000-01-03  1.303660 -2.003821
2000-01-04  1.884801 -0.141119
2000-01-05  1.194699  0.010551
2000-01-06  1.925393  1.968551
2000-01-07  0.565208  0.032738
2000-01-08  0.564129 -0.759118
2000-01-09  2.048458 -1.820537
2000-01-10  2.065750  0.383357
```

### 3.13. Python – Reading HTML Pages

#### 3.13.1 Install Beautifulsoup

Use the Anaconda package manager to install the required package and its dependent packages.

```
conda install Beaustifulsoap
```

#### 3.13.2 Reading the HTML file

In the below example we make a request to an url to be loaded into the python environment. Then use the html parser parameter to read the entire html file. Next, we print first few lines of the html page.

**Example:**

```
import urllib2
from bs4 import BeautifulSoup

# Fetch the html file
response = urllib2.urlopen('http://tutorialspoint.com/python/python_overview.htm')
html_doc = response.read()

# Parse the html file
soup = BeautifulSoup(html_doc, 'html.parser')

# Format the parsed html file
strhtm = soup.prettify()

# Print the first few characters
print (strhtm[:225])
```

**O/P**

```
<!DOCTYPE html>
<!--[if IE 8]><html class="ie ie8"> <![endif]-->
<!--[if IE 9]><html class="ie ie9"> <![endif]-->
<!--[if gt IE 9]><!-->
<html>
 <!--<![endif]-->
 <head>
  <!-- Basic -->
  <meta charset="utf-8"/>
  <title>
```

### 3.13.3 Extracting Tag Value

We can extract tag value from the first instance of the tag using the following code.

**Example:**

```
import urllib2
from bs4 import BeautifulSoup

response = urllib2.urlopen('http://tutorialspoint.com/python/python_overview.htm')
html_doc = response.read()

soup = BeautifulSoup(html_doc, 'html.parser')

print (soup.title)
print(soup.title.string)
print(soup.a.string)
print(soup.b.string)
```

**O/P**

```

Python Overview
None
Python is Interpreted
```

### 3.13.4 Extracting All Tags

Example to extract tag value from all the instances of a tag using the following code.

```
import urllib2
from bs4 import BeautifulSoup

response = urllib2.urlopen('http://tutorialspoint.com/python/python_overview.htm')
html_doc = response.read()
soup = BeautifulSoup(html_doc, 'html.parser')

for x in soup.find_all('b'): print(x.string)
```

**O/P**

```
Python is Interpreted
Python is Interactive
Python is Object-Oriented
Python is a Beginner's Language
Easy-to-learn
Easy-to-read
Easy-to-maintain
A broad standard library
Interactive Mode
Portable
Extendable
Databases
GUI Programming
Scalable
```

### 3.13. Python - Processing Unstructured Data

**Structured data** is organized in rows and columns and can be easily stored in databases. Examples include **CSV, TXT, and XLS** files, which use delimiters and may have fixed or variable widths. Missing values are typically shown as blanks between delimiters. Unstructured data, such as HTML, images, PDFs, social media feeds, or **plain text**, doesn't follow a tabular format. HTML can be processed using tags, but other formats lack such structure. In such cases,

**Python libraries** with built-in functions are used to handle and process the data.

### 3.14.1 Reading Data

**Example:**

In this we take a text file and read the file segregating each of the lines in it. Next we can divide the output into further lines and words. The original file is a text file containing some paragraphs describing the python language.

```
filename = 'path\input.txt'

with open(filename) as fn:

# Read each line
   ln = fn.readline()

# Keep count of lines
   lncnt = 1
   while ln:
       print("Line {}: {}".format(lncnt, ln.strip()))
       ln = fn.readline()
       lncnt += 1
```

**O/P:**

Line 1: Python is an interpreted high-level programming language for general-purpose programming. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales.

Line 2: Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

Line 3: Python interpreters are available for many operating systems. CPython, the

reference implementation of Python, is open source software and has a community-based development model, as do nearly all of its variant implementations. CPython is managed by the non-profit Python Software Foundation.

### 3.14.2 Counting Word Frequency

We can count the frequency of the words in the file using the counter function as follows.

```
from collections import Counter

with open(r'pathinput2.txt') as f:
                p = Counter(f.read().split())
                print(p)
```

**O/P**

```
Counter({'and': 3, 'Python': 3, 'that': 2, 'a': 2, 'programming': 2, 'code': 1, '1991,': 1, 'is': 1, 'programming.': 1, 'dynamic': 1, 'an': 1, 'design': 1, 'in': 1, 'high-level': 1, 'management.': 1, 'features': 1, 'readability,': 1, 'van': 1, 'both': 1, 'for': 1, 'Rossum': 1, 'system': 1, 'provides': 1, 'memory': 1, 'has': 1, 'type': 1, 'enable': 1, 'Created': 1, 'philosophy': 1, 'constructs': 1, 'emphasizes': 1, 'general-purpose': 1, 'notably': 1, 'released': 1, 'significant': 1, 'Guido': 1, 'using': 1, 'interpreted': 1, 'by': 1, 'on': 1, 'language': 1, 'whitespace.': 1, 'clear': 1, 'It': 1, 'large': 1, 'small': 1, 'automatic': 1, 'scales.': 1, 'first': 1})
```

### 3.14.Python - Word Tokenization

Word tokenization is the process of splitting a large sample of text into words. This is a requirement in natural language processing tasks where each word needs to be

captured and subjected to further analysis like classifying and counting them for a particular sentiment etc. The Natural Language Tool kit(NLTK) is a library used to achieve this. Install NLTK before proceeding with the python program for word tokenization.

```
conda install -c anaconda nltk
```

Next we use the **word_tokenize** method to split the paragraph into individual words.

**Example:**

```
import nltk

word_data = "It originated from the idea that there are readers who prefer
learning new skills from the comforts of their drawing rooms"
nltk_tokens = nltk.word_tokenize(word_data)
print (nltk_tokens)
```

**O/P**

```
['It', 'originated', 'from', 'the', 'idea', 'that', 'there', 'are', 'readers',
'who', 'prefer', 'learning', 'new', 'skills', 'from', 'the',
'comforts', 'of', 'their', 'drawing', 'rooms']
```

### 15.1 Tokenizing Sentences

We can also tokenize the sentences in a paragraph like we tokenized the words. We

use the method **sent_tokenize** to achieve this.

**Example:**

```
import nltk
sentence_data = "Sun rises in the east. Sun sets in the west."
nltk_tokens = nltk.sent_tokenize(sentence_data)
print (nltk_tokens)
```

**O/P**

```
['Sun rises in the east.', 'Sun sets in the west.']
```

### 3.15.Python - Stemming and Lemmatization

In the areas of Natural Language Processing we come across situation where two or more words have a common root. For example, the three words - agreed, agreeing and agreeable have the same root word agree. A search involving any of these words should treat them as the same word which is the root word. So it becomes essential to link all the words into their root word. The NLTK library has methods to do this linking and give the output showing the root word.

**Example:**

The below program uses the Porter Stemming Algorithm for stemming.

```
import nltk
from nltk.stem.porter import PorterStemmer
porter_stemmer = PorterStemmer()

word_data = "It originated from the idea that there are readers who prefer
learning new skills from the comforts of their drawing rooms"
# First Word tokenization
nltk_tokens = nltk.word_tokenize(word_data)
#Next find the roots of the word
for w in nltk_tokens:
      print "Actual: %s  Stem: %s"  % (w,porter_stemmer.stem(w))
```

**O/P**

```
Actual: It  Stem: It
Actual: originated  Stem: origin
Actual: from  Stem: from
Actual: the  Stem: the
Actual: idea  Stem: idea
Actual: that  Stem: that
Actual: there  Stem: there
Actual: are  Stem: are
Actual: readers  Stem: reader
Actual: who  Stem: who
Actual: prefer  Stem: prefer
Actual: learning  Stem: learn
Actual: new  Stem: new
Actual: skills  Stem: skill
Actual: from  Stem: from
Actual: the  Stem: the
Actual: comforts  Stem: comfort
Actual: of  Stem: of
Actual: their  Stem: their
Actual: drawing  Stem: draw
Actual: rooms  Stem: room
```

Lemmatization is similar ti stemming but it brings context to the words. So it goes a steps further by linking words with similar meaning to one word. For example if a paragraph has words like cars, trains and automobile, then it will link all of them to automobile. In the below program we use the WordNet lexical database for lemmatization.

**Example:**

```
import nltk
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()

word_data = "It originated from the idea that there are readers who prefer
learning new skills from the comforts of their drawing rooms"
nltk_tokens = nltk.word_tokenize(word_data)
for w in nltk_tokens:
      print "Actual: %s  Lemma: %s"  %
(w,wordnet_lemmatizer.lemmatize(w))
```

**O/P**

```
Actual: It  Lemma: It
Actual: originated  Lemma: originated
Actual: from  Lemma: from
Actual: the  Lemma: the
Actual: idea  Lemma: idea
Actual: that  Lemma: that
Actual: there  Lemma: there
Actual: are  Lemma: are
Actual: readers  Lemma: reader
Actual: who  Lemma: who
Actual: prefer  Lemma: prefer
Actual: learning  Lemma: learning
Actual: new  Lemma: new
Actual: skills  Lemma: skill
Actual: from  Lemma: from
Actual: the  Lemma: the
Actual: comforts  Lemma: comfort
Actual: of  Lemma: of
Actual: their  Lemma: their
Actual: drawing  Lemma: drawing
Actual: rooms  Lemma: room
```