

Unit-03

Capturing, Preparing and Working with data



Outline

- ✓ Basic File IO in Python
- ✓ NumPy V/S Pandas (what to use?)
- ✓ NumPy
- ✓ Pandas
- ✓ Accessing text, CSV, Excel files using pandas
- ✓ Accessing SQL Database
- ✓ Web Scrapping using BeautifulSoup

Basic IO operations in Python

- Before we can read or write a file, we have to open it using Python's built-in **open()** function.

syntax

```
fileobject = open(filename [, accessmode][, buffering])
```

- ➔ **filename** is a name of a file we want to open.
- ➔ **accessmode** is determines the mode in which file has to be opened (list of possible values given below)
- ➔ If **buffering** is set to 0, no buffering will happen, if set to 1 line buffering will happen, if grater than 1 then the number of buffer and if negative is given it will follow system default buffering behaviour.

M	Description
r	Read only (default)
rb	Read only in binary format
r+	Read and Write both
rb+	Read and Write both in binary format

M	Description (create file if not exist)
w	Write only
wb	Write only in binary format
w+	Read and Write both
wb+	Read and Write both in binary format

M	Description
a	Opens file to append, if file not exist will create it for write
ab	Append in binary format, if file not exist will create it for write
a+	Append, if file not exist it will create for read & write both
ab+	Read and Write both in binary format

Example : Read file in Python

- ▶ **read(size)** will read specified bytes from the file, if we don't specify size it will return whole file.

readfile.py

```
1 f = open('college.txt')
2 data = f.read()
3 print(data)
```

college.txt

```
Madhuben & Bhanubhai Patel Institute of Technology- Anand
Beyond Vitthal Udyognagar Anand,
Gujarat-388121, INDIA
```

- ▶ **readlines()** method will return list of lines from the file.

readlines.py

```
1 f = open('college.txt')
2 lines = f.readlines()
3 print(lines)
```

OUTPUT

```
['Madhuben & Bhanubhai Patel Institute of Technology-
Anand
Beyond Vitthal Udyognagar Anand,
Gujarat-388121, INDIA
```

- ▶ We can use for loop to get each line separately,

readlinesfor.py

```
1 f = open('college.txt')
2 lines = f.readlines()
3 for l in lines :
4     print(l)
```

OUTPUT

```
Madhuben & Bhanubhai Patel Institute of Technology- Anand
Beyond Vitthal Udyognagar Anand,
Gujarat-388121, INDIA
```

How to write path?

- ▶ We can specify relative path in argument to **open** method, alternatively we can also specify absolute path.
- ▶ To specify absolute path,
 - ➔ In windows, `f=open('D:\\folder\\subfolder\\filename.txt')`
 - ➔ In mac & linux, `f=open('/user/folder/subfolder/filename.txt')`
- ▶ We suppose to close the file once we are done using the file in the Python using **close()** method.

closefile.py

```
1 f = open('college.txt')
2 data = f.read()
3 print(data)
4 f.close()
```

Handling errors using “with” keyword

- ▶ It is possible that we may have typo in the filename or file we specified is moved/deleted, in such cases there will be an error while running the file.
- ▶ To handle such situations we can use new syntax of opening the file using **with** keyword.

fileusingwith.py

```
1 with open('college.txt') as f :  
2     data = f.read()  
3     print(data)
```

- ▶ When we open file using with we **need not to close** the file.

Example : Write file in Python

- ▶ **write()** method will write the specified data to the file.

readdemo.py

```
1 with open('college.txt','a') as f :  
2     f.write('Hello world')
```

- ▶ If we open file with '**w**' mode it will overwrite the data to the existing file or will create new file if file does not exists.
- ▶ If we open file with '**a**' mode it will append the data at the end of the existing file or will create new file if file does not exists.

Reading CSV files without any library functions

- ▶ A comma-separated values file is a delimited text file that uses a comma to separate values.
- ▶ Each line of is a data record, Each record consists of many fields, separated by commas.

▶ Example :

Book1.csv

```
studentname,enrollment,cpi  
abcd,123456,8.5  
bcde,456789,2.5  
cdef,321654,7.6
```

- ▶ We can use Microsoft Excel to access CSV files.
- ▶ In the later sessions we will access CSV files using different libraries, but we can also access CSV files without any libraries.
(Not recommend)

readlines.py

```
1 with open('Book1.csv') as f :  
2     rows = f.readlines()  
3     isFirstLine = True  
4     for r in rows :  
5         if isFirstLine :  
6             isFirstLine = False  
7             continue  
8         cols = r.split(',')  
9         print('Student Name = ', cols[0], end=" ")  
10        print('\tEn. No. = ', cols[1], end=" ")  
11        print('\tCPI = \t', cols[2])
```


NumPy v/s Pandas

- ▶ Developers built pandas on top of NumPy, as a result every task we perform using pandas also goes through NumPy.
- ▶ To obtain the **benefits of pandas**, we need to **pay a performance penalty** that some testers say is **100 times slower** than NumPy for similar task.
- ▶ Nowadays computer hardware are powerful enough to take care for the performance issue, but when **speed** of execution is **essential NumPy** is always the **best** choice.
- ▶ We can use **pandas** to make writing code **easier** and **faster**, pandas will reduce potential coding errors.
- ▶ Pandas provide rich **time-series** functionality, **data alignment**, **NA-friendly** statistics, **groupby**, **merge**, etc.. methods, if we use NumPy we have to implement all these methods manually.
- ▶ So,
 - ➔ if we want **performance** we should use **NumPy**,
 - ➔ if we want **ease of coding** we should use **pandas**.

Unit-03.01

Lets Learn

NumPy

NumPy

- ▶ NumPy (Numeric Python) is a Python library to manipulate arrays.
- ▶ Almost all the libraries in python rely on NumPy as one of their main building block.
- ▶ NumPy provides functions for domains like Algebra, Fourier transform etc..
- ▶ NumPy is incredibly fast as it has bindings to C libraries.
- ▶ Install :
 - ➔ `conda install numpy`
 - OR ➔ `pip install numpy`

NumPy Array

- ▶ The most important object defined in NumPy is an N-dimensional array type called **ndarray**.
- ▶ It describes the **collection** of **items** of the **same type**, Items in the collection can be accessed using a **zero-based index**.
- ▶ An instance of **ndarray** class can be constructed in many different ways, the basic ndarray can be created as below.

syntax

```
import numpy as np
a= np.array(list | tuple | set | dict)
```

numpyarray.py

```
1 import numpy as np
2 a= np.array(['MBIT', 'College', 'Anand'])
3 print(type(a))
4 print(a)
```

Output

```
<class 'numpy.ndarray'>
['MBIT', 'College', 'Anand']
```

NumPy Array (Cont.)

- ▶ **arange**(*start,end,step*) function will create NumPy array starting from *start* till *end* (not included) with specified *steps*.

numpyarange.py

```
1 import numpy as np
2 b = np.arange(0,10,1)
3 print(b)
```

Output

```
[0 1 2 3 4 5 6 7 8 9]
```

- ▶ **zeros**(*n*) function will return NumPy array of given shape, filled with zeros.

numpyzeros.py

```
1 import numpy as np
2 c = np.zeros(3)
3 print(c)
4 c1 = np.zeros((3,3)) #have to give as tuple
5 print(c1)
```

Output

```
[0. 0. 0.]
```

```
[[0. 0. 0.] [0. 0. 0.] [0. 0. 0.]]
```

- ▶ **ones**(*n*) function will return NumPy array of given shape, filled with ones.

NumPy Array (Cont.)

- ▶ **eye**(*n*) function will create 2-D NumPy array with ones on the diagonal and zeros elsewhere.

numpyeye.py

```
1 import numpy as np
2 b = np.eye(3)
3 print(b)
```

Output

```
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]
```

- ▶ **linspace**(*start, stop, num*) function will return evenly spaced numbers over a specified interval.

numpylinspace.py

```
1 import numpy as np
2 c = np.linspace(0,1,11)
3 print(c)
```

Output

```
[0.  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8
 0.9  1. ]
```

- ▶ **Note:** in **arange** function we have given start, stop & **step**, whereas in **linspace** function we are giving start, stop & **number of elements** we want.

Array Shape in NumPy

- ▶ We can grab the shape of ndarray using its **shape property**.

numpyarray.py

```
1 import numpy as np
2 b = np.zeros((3,3))
3 print(b.shape)
```

Output

```
(3,3)
```

- ▶ We can also **reshape** the array using reshape method of **ndarray**.

numpyarray.py

```
1 import numpy as np
2 re1 = np.random.randint(1,100,10)
3 re2 = re1.reshape(5,2)
4 print(re2)
```

Output

```
[[29 55]
 [44 50]
 [25 53]
 [59 6]
 [93 7]]
```

- ▶ **Note:** the number of elements and multiplication of rows and cols in new array must be equal.
 - ➔ **Example :** here we have old one-dimensional array of 10 elements and reshaped shape is (5,2)
so, $5 * 2 = 10$, which means it is a valid reshape

NumPy Random

- ▶ **rand**(p1,p2...,pn) function will create **n-dimensional array** with **random data** using uniform distribution, if we **do not** specify any parameter it will return **random float** number.

numpyrand.py

```
1 import numpy as np
2 r1 = np.random.rand()
3 print(r1)
4 r2 = np.random.rand(3,2) # no tuple
5 print(r2)
```

Output

```
0.23937253208490505

[[0.58924723 0.09677878]
 [0.97945337 0.76537675]
 [0.73097381 0.51277276]]
```

- ▶ **randint**(low,high,num) function will create one-dimensional array with **num** random integer data between **low** and **high**.

numpyrandint.py

```
1 import numpy as np
2 r3 = np.random.randint(1,100,10)
3 print(r3)
```

Output

```
[78 78 17 98 19 26 81 67 23 24]
```

- ▶ We can reshape the array in any shape using **reshape** method, which we learned in previous slide.

NumPy Random (Cont.)

- ▶ **randn**(p1,p2...,pn) function will create **n-dimensional array** with **random data** using standard normal distribution, if we **do not** specify any parameter it will return **random float** number.

numpyrandn.py

```
1 import numpy as np
2 r1 = np.random.randn()
3 print(r1)
4 r2 = np.random.randn(3,2) # no tuple
5 print(r2)
```

Output

```
-0.15359861758111037

[[ 0.40967905 -0.21974532]
 [-0.90341482 -0.69779498]
 [ 0.99444948 -1.45308348]]
```

- ▶ **Note:** **rand** function will generate random number using uniform distribution, whereas **randn** function will generate random number using standard normal distribution.
- ▶ We are going to learn the difference using visualization technique (as a data scientist, We have to use visualization techniques to convince the audience)

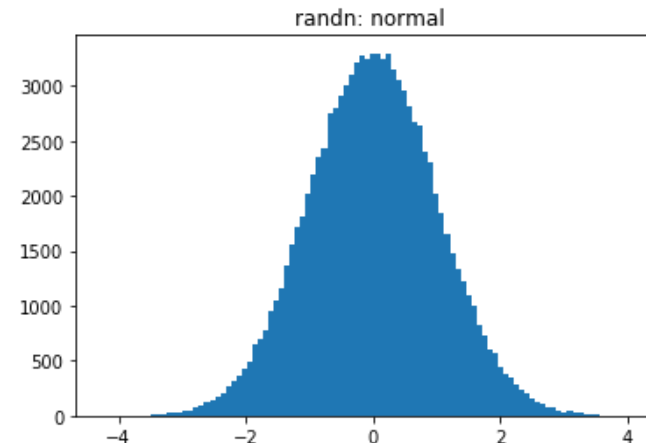
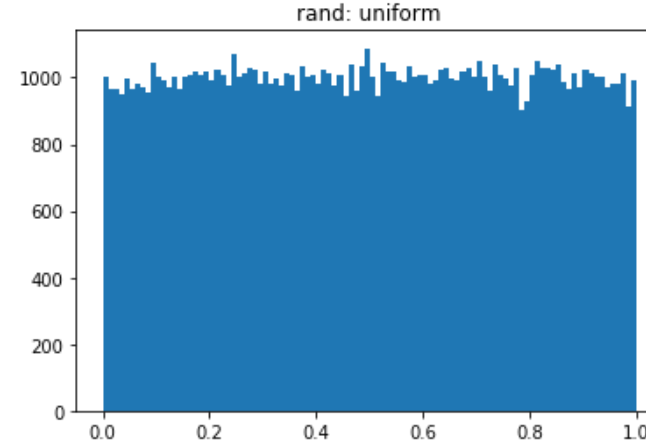
Visualizing the difference between rand & randn

► We are going to use matplotlib library to visualize the difference.

→ You need not to worry if you are not getting the syntax of matplotlib, we are going to learn it in detail in Unit-4

matplotdemo.py

```
1 import numpy as np
2 from matplotlib import pyplot as plt
3 %matplotlib inline
4 samplesize = 100000
5 uniform = np.random.rand(samplesize)
6 normal = np.random.randn(samplesize)
7 plt.hist(uniform,bins=100)
8 plt.title('rand: uniform')
9 plt.show()
10 plt.hist(normal,bins=100)
11 plt.title('randn: normal')
12 plt.show()
```



Aggregations

- ▶ **min()** function will return the minimum value from the ndarray, there are two ways in which we can use min function, example of both ways are given below.

numpymin.py

```
1 import numpy as np
2 l = [1,5,3,8,2,3,6,7,5,2,9,11,2,5,3,4,8,9,3,1,9,3]
3 a = np.array(l)
4 print('Min way1 = ',a.min())
5 print('Min way2 = ',np.min(a))
```

Output

```
Min way1 = 1
Min way2 = 1
```

- ▶ **max()** function will return the maximum value from the ndarray, there are two ways in which we can use min function, example of both ways are given below.

numpymax.py

```
1 import numpy as np
2 l = [1,5,3,8,2,3,6,7,5,2,9,11,2,5,3,4,8,9,3,1,9,3]
3 a = np.array(l)
4 print('Max way1 = ',a.max())
5 print('Max way2 = ',np.max(a))
```

Output

```
Max way1 = 11
Max way2 = 11
```

Aggregations (Cont.)

- ▶ NumPy support many aggregation functions such as min, max, argmin, argmax, sum, mean, std, etc...

numpymin.py

```
1 l = [7,5,3,1,8,2,3,6,11,5,2,9,10,2,5,3,7,8,9,3,1,9,3]
2 a = np.array(l)
3 print('Min = ',a.min())
4 print('ArgMin = ',a.argmin())
5 print('Max = ',a.max())
6 print('ArgMax = ',a.argmax())
7 print('Sum = ',a.sum())
8 print('Mean = ',a.mean())
9 print('Std = ',a.std())
```

Output

```
Min = 1
ArgMin = 3
Max = 11
ArgMax = 8
Sum = 122
Mean = 5.304347826086956
Std = 3.042235771223635
```

Using axis argument with aggregate functions

- ▶ When we apply aggregate functions with multidimensional ndarray, it will apply aggregate function to all its dimensions (axis).

numpyaxis.py

```
1 import numpy as np
2 array2d = np.array([[1,2,3],[4,5,6],[7,8,9]])
3 print('sum = ',array2d.sum())
```

Output

```
sum = 45
```

- ▶ If we want to get sum of rows or cols we can use axis argument with the aggregate functions.

numpyaxis.py

```
1 import numpy as np
2 array2d = np.array([[1,2,3],[4,5,6],[7,8,9]])
3 print('sum (cols)= ',array2d.sum(axis=0)) #Vertical
4 print('sum (rows)= ',array2d.sum(axis=1)) #Horizontal
```

Output

```
sum (cols) = [12 15 18]
sum (rows) = [6 15 24]
```

Single V/S Double bracket notations

- ▶ There are two ways in which you can access element of multi-dimensional array, example of both the method is given below

numpybrackets.py

```
1 arr =  
2 np.array(['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i'])  
3 print('double = ', arr[2][1]) # double bracket notation  
4 print('single = ', arr[2,1]) # single bracket notation
```

Output

```
double = h  
single = h
```

- ▶ Both method is valid and provides exactly the same answer, but single bracket notation is recommended as in double bracket notation it will create a temporary sub array of third row and then fetch the second column from it.
- ▶ Single bracket notation will be easy to read and write while programming.

Slicing ndarray

- ▶ Slicing in python means taking elements from one given index to another given index.
- ▶ Similar to Python List, we can use same syntax `array[start:end:step]` to slice ndarray.
 - ➔ Default start is 0
 - ➔ Default end is length of the array
 - ➔ Default step is 1

numpslice1d.py

```
1 import numpy as np
2 arr =
  np.array(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
3 print(arr[2:5])
4 print(arr[:5])
5 print(arr[5:])
6 print(arr[2:7:2])
7 print(arr[::-1])
```

Output

```
['c' 'd' 'e']
['a' 'b' 'c' 'd' 'e']
['f' 'g' 'h']
['c' 'e' 'g']
['h' 'g' 'f' 'e' 'd' 'c']
['b' 'a']
```

Array Slicing Example

	C-0	C-1	C-2	C-3	C-4
R-0	1	2	3	4	5
R-1	6	7	8	9	10
a = R-2	11	12	13	14	15
R-3	16	17	18	19	20
R-4	21	22	23	24	25

► Example :

► `a[2][3]` =

► `a[2,3]` =

► `a[2]` =

► `a[0:2]` =

► `a[0:2:2]` =

► `a[::-1]` =

► `a[1:3,1:3]` =

► `a[3,::3]` =

► `a[:,::-1]` =

Slicing multi-dimensional array

- ▶ Slicing multi-dimensional array would be same as single dimensional array with the help of single bracket notation we learn earlier, lets see an example.

numpyslice1d.py

```
1 arr =  
2 np.array([[ 'a', 'b', 'c'], [ 'd', 'e', 'f'], [ 'g', 'h',  
   'i']])  
3 print(arr[0:2 , 0:2]) #first two rows and cols  
4 print(arr[::-1]) #reversed rows  
5 print(arr[:, ::-1]) #reversed cols  
6 print(arr[::-1, ::-1]) #complete reverse
```

Output

```
[[ 'a'  'b']  
 [ 'd'  'e']]  
[[ 'g'  'h'  'i']  
 [ 'd'  'e'  'f']  
 [ 'a'  'b'  'c']]  
[[ 'c'  'b'  'a']  
 [ 'f'  'e'  'd']  
 [ 'i'  'h'  'g']]  
[[ 'i'  'h'  'g']  
 [ 'f'  'e'  'd']  
 [ 'c'  'b'  'a']]
```

Warning : Array Slicing is mutable !

- ▶ When we slice an array and apply some operation on them, it will also make changes in original array, as it will not create a copy of a array while slicing.
- ▶ Example,

numpyslice1d.py

```
1 import numpy as np
2 arr = np.array([1,2,3,4,5])
3 arrsliced = arr[0:3]
4
5 arrsliced[:] = 2 # Broadcasting
6
7 print('Original Array = ', arr)
8 print('Sliced Array = ',arrsliced)
```

Output

```
Original Array = [2 2 2 4 5]
Sliced Array = [2 2 2]
```

NumPy Arithmetic Operations

numpyop.py

```
1 import numpy as np
2 arr1 = np.array([[1,2,3],[1,2,3],[1,2,3]])
3 arr2 = np.array([[4,5,6],[4,5,6],[4,5,6]])
4
5 arradd1 = arr1 + 2 # addition of matrix with scalar
6 arradd2 = arr1 + arr2 # addition of two matrices
7 print('Addition Scalar = ', arradd1)
8 print('Addition Matrix = ', arradd2)
9
10 arrsub1 = arr1 - 2 # subtraction of matrix with
   scalar
11 arrsub2 = arr1 - arr2 # subtraction of two matrices
12 print('Subtraction Scalar = ', arrsub1)
13 print('Subtraction Matrix = ', arrsub2)
14 arrdiv1 = arr1 / 2 # subtraction of matrix with
   scalar
15 arrdiv2 = arr1 / arr2 # subtraction of two matrices
16 print('Division Scalar = ', arrdiv1)
17 print('Division Matrix = ', arrdiv2)
```

Output

```
Addition Scalar =  [[3 4 5]
 [3 4 5]
 [3 4 5]]
Addition Matrix =  [[5 7 9]
 [5 7 9]
 [5 7 9]]
Substraction Scalar =  [[-1  0  1]
 [-1  0  1]
 [-1  0  1]]
Substraction Matrix =  [[-3 -3 -3]
 [-3 -3 -3]
 [-3 -3 -3]]
Division Scalar =  [[0.5 1.  1.5]
 [0.5 1.  1.5]
 [0.5 1.  1.5]]
Division Matrix =  [[0.25 0.4  0.5
 [0.25 0.4  0.5 ]
 [0.25 0.4  0.5 ]]
```

NumPy Arithmetic Operations (Cont.)

numpyop.py

```
1 import numpy as np
2 arrmul1 = arr1 * 2 # multiply matrix with scalar
3 arrmul2 = arr1 * arr2 # multiply two matrices
4 print('Multiply Scalar = ', arrmul1)
5 #Note : its not metrix multiplication*
6 print('Multiply Matrix = ', arrmul2)
7 # In order to do matrix multiplication
8 arrmatmul = np.matmul(arr1,arr2)
9 print('Matrix Multiplication = ',arrmatmul)
10 # OR
    arrdot = arr1.dot(arr2)
11 print('Dot = ',arrdot)
12 # OR
13 arrpy3dot5plus = arr1 @ arr2
14 print('Python 3.5+ support = ',arrpy3dot5plus)
```

Output

```
Multiply Scalar =  [[2 4 6]
 [2 4 6]
 [2 4 6]]
Multiply Matrix =  [[ 4 10 18]
 [ 4 10 18]
 [ 4 10 18]]
Matrix Multiplication =  [[24 30
 36]
 [24 30 36]
 [24 30 36]]
Dot =  [[24 30 36]
 [24 30 36]
 [24 30 36]]
Python 3.5+ support =  [[24 30 36]
 [24 30 36]
 [24 30 36]]
```

Sorting Array

- ▶ The `sort()` function returns a sorted copy of the input array.

syntax

```
import numpy as np
# arr = our ndarray
np.sort(arr,axis,kind,order)
# OR arr.sort()
```

Parameters

arr = array to sort (inplace)
axis = axis to sort (default=0)
kind = kind of algo to use ('quicksort' <- default, 'mergesort', 'heapsort')
order = on which field we want to sort (if multiple fields)

- ▶ Example :

numpysort.py

```
1 import numpy as np
2 arr =
  np.array(['MBIT', 'Anand', 'College', 'of', 'En
    gineering'])
3 print("Before Sorting = ", arr)
4 arr.sort() # or np.sort(arr)
5 print("After Sorting = ",arr)
```

Output

```
Before Sorting = ['MBIT'
'Anand' 'College' 'of'
'Engineering']
After Sorting = []
```

Sort Array Example

numpysort2.py

```
1 import numpy as np
2 dt = np.dtype([('name', 'S10'),('age', int)])
3 arr2 =
  np.array([('MBIT',200),('ABC',300),('XYZ',100)
  ],dtype=dt)
4 arr2.sort(order='name')
5 print(arr2)
```

Output

```
[(b'ABC', 300) (b'MBIT', 200)
 (b'XYZ', 100)]
```

Conditional Selection

- ▶ Similar to arithmetic operations when we apply any comparison operator to Numpy Array, then it will be applied to each element in the array and a new bool Numpy Array will be created with values True or False.

numpycond1.py

```
1 import numpy as np
2 arr = np.random.randint(1,100,10)
3 print(arr)
4 boolArr = arr > 50
5 print(boolArr)
```

Output

```
[25 17 24 15 17 97 42 10 67
22]
[False False False False
False  True False False  True
False]
```

numpycond2.py

```
1 import numpy as np
2 arr = np.random.randint(1,100,10)
3 print("All = ",arr)
4 boolArr = arr > 50
5 print("Filtered = ", arr[boolArr])
```

Output

```
All = [31 94 25 70 23  9 11
77 48 11]
Filtered = [94 70 77]
```

Unit-03.02

Lets Learn

Pandas

Pandas

- ▶ Pandas is an open source library built on top of NumPy.
 - ▶ It allows for fast data cleaning, preparation and analysis.
 - ▶ It excels in performance and productivity.
 - ▶ It also has built-in visualization features.
 - ▶ It can work with the data from wide variety of sources.
 - ▶ Install :
 - ↳ conda install pandas
- OR ↳ pip install pandas



Outline (Pandas)

- ✓ Series
- ✓ Data Frames
- ✓ Accessing text, CSV, Excel files using pandas
- ✓ Accessing SQL Database
- ✓ Missing Data
- ✓ Group By
- ✓ Merging, Joining & Concatenating
- ✓ Operations

Series

- ▶ Series is an one-dimensional* array with axis labels.
- ▶ It supports both integer and label-based index but index must be of hashable type.
- ▶ If we do not specify index it will assign integer zero-based index.

syntax

```
import pandas as pd  
s = pd.Series(data,index,dtype,copy=False)
```

Parameters

data = array like Iterable
index = array like index
dtype = data-type
copy = bool, default is False

pandasSeries.py

```
1 import pandas as pd  
2 s = pd.Series([1, 3, 5, 7, 9, 11])  
3 print(s)
```

Output

```
0      1  
1      3  
2      5  
3      7  
4      9  
5     11  
dtype: int64
```

Series (Cont.)

- ▶ We can then access the elements inside Series just like array using square brackets notation.

pdSeriesEle.py

```
1 import pandas as pd
2 s = pd.Series([1, 3, 5, 7, 9, 11])
3 print("S[0] = ", s[0])
4 b = s[0] + s[1]
5 print("Sum = ", b)
```

Output

```
S[0] = 1
Sum = 4
```

- ▶ We can specify the data type of Series using **dtype** parameter

pdSeriesdtype.py

```
1 import pandas as pd
2 s = pd.Series([1, 3, 5, 7, 9, 11], dtype='str')
3 print("S[0] = ", s[0])
4 b = s[0] + s[1]
5 print("Sum = ", b)
```

Output

```
S[0] = 1
Sum = 13
```

Series (Cont.)

- ▶ We can specify index to Series with the help of **index** parameter

pdSeriesdtype.py

```
1 import numpy as np
2 import pandas as pd
3 i = ['name', 'address', 'phone', 'email', 'website']
4 d = ['mbit', 'rj', 123, 'd@d.com', 'mbit.edu.in']
5 s = pd.Series(data=d, index=i)
6 print(s)
```

Output

```
name          mbit
address       rj
phone         123
email         d@d.com
website      mbit.edu.in
dtype: object
```

Creating Time Series

- ▶ We can use some of pandas inbuilt date functions to create a time series.

pdSeriesEle.py

```
1 import numpy as np
2 import pandas as pd
3 dates = pd.to_datetime("27th of July, 2020")
4 i = dates + pd.to_timedelta(np.arange(5),
5                             unit='D')
5 d = [50,53,25,70,60]
6 time_series = pd.Series(data=d,index=i)
7 print(time_series)
```

Output

2020-07-27	50
2020-07-28	53
2020-07-29	25
2020-07-30	70
2020-07-31	60
dtype: int64	

Data Frames

- ▶ Data frames are two dimensional data structure, i.e. data is aligned in a tabular format in rows and columns.
- ▶ Data frame also contains labelled axes on rows and columns.
- ▶ Features of Data Frame :
 - It is size-mutable
 - Has labelled axes
 - Columns can be of different data types
 - We can perform arithmetic operations on rows and columns.

- ▶ Structure :

	PDS	Algo	SE	INS
101				
102				
103				
...				
160				

Data Frames (Cont.)

► Syntax :

syntax

```
import pandas as pd
df = pd.DataFrame(data, index, columns, dtype, copy=False)
```

Parameters

data = array like Iterable
index = array like row index
columns = array like col index
dtype = data-type
copy = bool, default is False

► Example :

pdDataFrame.py

```
1 import numpy as np
2 import pandas as pd
3 randArr = np.random.randint(0,100,20).reshape(5,4)
4 df =
  pd.DataFrame(randArr,np.arange(101,106,1),[ 'PDS', 'A
lgo', 'SE', 'INS' ])
5 print(df)
```

Output

	PDS	Algo	SE	INS
101	0	23	93	46
102	85	47	31	12
103	35	34	6	89
104	66	83	70	50
105	65	88	87	87

Data Frames (Cont.)

- Grabbing the column

dfGrabCol.py

```
1 import numpy as np
2 import pandas as pd
3 randArr = np.random.randint(0,100,20).reshape(5,4)
4 df =
  pd.DataFrame(randArr,np.arange(101,106,1),[ 'PDS', 'A
lgo', 'SE', 'INS' ])
5 print(df[ 'PDS' ])
```

Output

101	0
102	85
103	35
104	66
105	65

Name: PDS, dtype: int32

- Grabbing the multiple column

dfGrabMulCol.py

```
1 print(df[ 'PDS', 'SE' ])
```

Output

	PDS	SE
101	0	93
102	85	31
103	35	6
104	66	70
105	65	87

Data Frames (Cont.)

▶ Grabbing a row

dfGrabRow.py

```
1 print(df.loc[101]) # using labels
2 #OR
3 print(df.iloc[0]) # using zero based index
```

Output

```
PDS      0
Algo     23
SE       93
INS      46
Name: 101, dtype: int32
```

▶ Grabbing Single Value

dfGrabSingle.py

```
1 print(df.loc[101, 'PDS']) # using labels
```

Output

```
0
```

▶ Deleting Row

dfDelCol.py

```
1 df.drop('103', inplace=True)
2 print(df)
```

Output

	PDS	Algo	SE	INS
101	0	23	93	46
102	85	47	31	12
104	66	83	70	50
105	65	88	87	87

Data Frames (Cont.)

► Creating new column

dfCreateCol.py

```
1 df['total'] = df['PDS'] + df['Algo'] +  
  df['SE'] + df['INS']  
2 print(df)
```

Output

	PDS	Algo	SE	INS	total
101	0	23	93	46	162
102	85	47	31	12	175
103	35	34	6	89	164
104	66	83	70	50	269
105	65	88	87	87	327

► Deleting Column and Row

dfDelCol.py

```
1 df.drop('total',axis=1,inplace=True)  
2 print(df)
```

Output

	PDS	Algo	SE	INS
101	0	23	93	46
102	85	47	31	12
103	35	34	6	89
104	66	83	70	50
105	65	88	87	87

Data Frames (Cont.)

▶ Getting Subset of Data Frame

dfGrabSubSet.py

```
1 print(df.loc[[101,104], [['PDS','INS']]])
```

Output

	PDS	INS
101	0	46
104	66	50

▶ Selecting all cols except one

dfGrabExcept.py

```
1 print(df.loc[:, df.columns != 'Algo' ])
```

Output

	PDS	SE	INS
101	0	93	46
102	85	31	12
103	35	6	89
104	66	70	50
105	65	87	87

Conditional Selection

- ▶ Similar to NumPy we can do conditional selection in pandas.

dfCondSel.py

```
1 import numpy as np
2 import pandas as pd
3 np.random.seed(121)
4 randArr =
  np.random.randint(0,100,20).reshape(5,4)
5 df =
  pd.DataFrame(randArr,np.arange(101,106,1)
  ,['PDS','Algo','SE','INS'])
6 print(df)
7 print(df>50)
```

Output

	PDS	Algo	SE	INS
101	66	85	8	95
102	65	52	83	96
103	46	34	52	60
104	54	3	94	52
105	57	75	88	39
	PDS	Algo	SE	INS
101	True	True	False	True
102	True	True	True	True
103	False	False	True	True
104	True	False	True	True
105	True	True	True	False

- ▶ Note : we have used np.random.seed() method and set seed to be 121, so that when you generate random number it matches with the random number I have generated.

Conditional Selection (Cont.)

- ▶ We can then use this boolean DataFrame to get associated values.

dfCondSel.py

```
1 dfBool = df > 50
2 print(df[dfBool])
```

- ▶ Note : It will set NaN (Not a Number) in case of False

Output

	PDS	Algo	SE	INS
101	66	85	NaN	95
102	65	52	83	96
103	NaN	NaN	52	60
104	54	NaN	94	52
105	57	75	88	NaN

- ▶ We can apply condition on specific column.

dfCondSel.py

```
1 dfBool = df['PDS'] > 50
2 print(df[dfBool])
```

Output

	PDS	Algo	SE	INS
101	66	85	8	95
102	65	52	83	96
104	54	3	94	52
105	57	75	88	39

Setting/Resetting index

- ▶ In our previous example we have seen our index does not have name, if we want to specify name to our index we can specify it using **DataFrame.index.name** property.

dfCondSel.py

```
1 df.index.name('RollNo')
```

Note: We have name to our index now

Output

	PDS	Algo	SE	INS
RollNo				
101	66	85	8	95
102	65	52	83	96
103	46	34	52	60
104	54	3	94	52
105	57	75	88	39

- ▶ We can use pandas built-in methods to set or reset the index
 - ➔ `pd.set_index('NewColumn',inplace=True)`, will set new column as index,
 - ➔ `pd.reset_index()`, will reset index to zero based numeric index.

Setting/Resetting index (Cont.)

► set_index(new_index)

dfCondSel.py

```
1 df.set_index('PDS') #inplace=True
```

Note: We have PDS as our index now

Output

	Algo	SE	INS
PDS			
66	85	8	95
65	52	83	96
46	34	52	60
54	3	94	52
57	75	88	39

► reset_index()

dfCondSel.py

```
1 df.reset_index()
```

Note: Our **RollNo(index)** become new column, and we now have zero based numeric index

Output

	RollNo	PDS	Algo	SE	INS
0	101	66	85	8	95
1	102	65	52	83	96
2	103	46	34	52	60
3	104	54	3	94	52
4	105	57	75	88	39

Multi-Index DataFrame

- ▶ Hierarchical indexes (AKA multiindexes) help us to organize, find, and aggregate information faster at almost no cost.
- ▶ Example where we need Hierarchical indexes

Numeric Index/Single Index							
	Col	Dep	Sem	RN	S1	S2	S3
0	ABC	CE	5	101	50	60	70
1	ABC	CE	5	102	48	70	25
2	ABC	CE	7	101	58	59	51
3	ABC	ME	5	101	30	35	39
4	ABC	ME	5	102	50	90	48
5	MBIT	CE	5	101	88	99	77
6	MBIT	CE	5	102	99	84	76
7	MBIT	CE	7	101	88	77	99
8	MBIT	ME	5	101	44	88	99

Multi Index						
Col	Dep	Sem	RN	S1	S2	S3
ABC	CE	5	101	50	60	70
		5	102	48	70	25
		7	101	58	59	51
	ME	5	101	30	35	39
		5	102	50	90	48
		5	102	99	84	76
MBIT	CE	5	101	88	99	77
		5	102	99	84	76
		7	101	88	77	99
	ME	5	101	44	88	99

Multi-Index DataFrame (Cont.)

- ▶ Creating multiindexes is as simple as creating single index using `set_index` method, only difference is in case of multiindexes we need to provide list of indexes instead of a single string index, lets see an example for that

dfMultiIndex.py

```
1 dfMulti =  
  pd.read_csv('MultiIndexDemo.csv')  
2 dfMulti.set_index(['Col', 'Dep', 'Sem'],  
  inplace=True)  
3 print(dfMulti)
```

Output

			RN	S1	S2	S3
Col	Dep	Sem				
ABC	CE	5	101	50	60	70
		5	102	48	70	25
		7	101	58	59	51
	ME	5	101	30	35	39
		5	102	50	90	48
		7	101	88	99	77
MBIT	CE	5	101	88	99	77
		5	102	99	84	76
		7	101	88	77	99
	ME	5	101	44	88	99

Multi-Index DataFrame (Cont.)

► Now we have multi-indexed DataFrame from which we can access data using multiple index

► For Example

→ Sub DataFrame for all the students of MBIT

dfGrab MBIT Stu.py

```
1 print(dfMulti.loc[' MBIT '])
```

→ Sub DataFrame for Computer Engineering students from MBIT

dfGrab MBIT CESTu.py

```
1 print(dfMulti.loc[' MBIT ', 'CE'])
```

Output (MBIT)

		RN	S1	S2	S3
Dep	Sem				
CE	5	101	88	99	77
	5	102	99	84	76
	7	101	88	77	99
ME	5	101	44	88	99

Output (MBIT ->CE)

		RN	S1	S2	S3
Sem					
5		101	88	99	77
5		102	99	84	76
7		101	88	77	99

Reading in Multiindexed DataFrame directly from CSV

- ▶ `read_csv` function of pandas provides easy way to create multi-indexed DataFrame directly while fetching the CSV file.

dfMultiIndex.py

```
1 dfMultiCSV =  
  pd.read_csv('MultiIndexDemo.csv',  
             ,index_col=[0,1,2])  
  #for multi-index in cols we can  
  use header parameter  
2 print(dfMultiCSV)
```

Output

			RN	S1	S2	S3
Col	Dep	Sem				
ABC	CE	5	101	50	60	70
		5	102	48	70	25
		7	101	58	59	51
	ME	5	101	30	35	39
		5	102	50	90	48
		7	101	88	99	77
MBIT	CE	5	101	88	99	77
		5	102	99	84	76
		7	101	88	77	99
	ME	5	101	44	88	99

Cross Sections in DataFrame

- ▶ The `xs()` function is used to get cross-section from the Series/DataFrame.
- ▶ This method takes a `key` argument to select data at a particular level of a MultiIndex.
- ▶ Syntax :

syntax

```
DataFrame.xs(key, axis=0, level=None, drop_level=True)
```

- ▶ Example :

dfMultiIndex.py

```
1 dfMultiCSV =  
  pd.read_csv('MultiIndexDemo.csv',  
    index_col=[0,1,2])  
2 print(dfMultiCSV)  
3 print(dfMultiCSV.xs('CE',axis=0,level='Dep'))
```

=== Parameters ===
key : label
axis : Axis to retrieve cross section
level : level of key
drop_level : False if you want to preserve the level

Output

RN	S1	S2	S3
Col	Sem						
ABC	5	101	50	60	70		
	5	102	48	70	25		
	7	101	58	59	51		
MBIT	5	101	88	99	77		
	5	102	99	84	76		
	7	101	88	77	99		

	7	101	88	77	99		
ME	5	101	44	88	99		

Dealing with Missing Data

- ▶ There are many methods by which we can deal with the missing data, some of most commons are listed below,
 - ➔ `dropna`, will drop (delete) the missing data (rows/cols)
 - ➔ `fillna`, will fill specified values in place of missing data
 - ➔ `interpolate`, will interpolate missing data and fill interpolated value in place of missing data.

Groupby in Pandas

- ▶ Any groupby operation involves one of the following operations on the original object. They are
 - ➔ **Splitting** the Object
 - ➔ **Applying** a function
 - ➔ **Combining** the results
- ▶ In many situations, we split the data into sets and we apply some functionality on each subset.
- ▶ we can perform the following operations
 - ➔ **Aggregation** - computing a summary statistic
 - ➔ **Transformation** - perform some group-specific operation
 - ➔ **Filtration** - discarding the data with some condition
- ▶ Basic ways to use of groupby method
 - ➔ `df.groupby('key')`
 - ➔ `df.groupby(['key1', 'key2'])`
 - ➔ `df.groupby(key, axis=1)`

College	Enno	CPI
MBIT	123	8.9
MBIT	124	9.2
MBIT	125	7.8
MBIT	128	8.7
ABC	211	5.6
ABC	212	6.2
ABC	215	3.2
ABC	218	4.2
XYZ	312	5.2
XYZ	315	6.5
XYZ	315	5.8

College	Mean CPI
MBIT	8.65
ABC	4.8
XYZ	5.83

Groupby in Pandas (Cont.)

► Example : Listing all the groups

dfGroup.py

```
1 dfIPL = pd.read_csv('IPLDataSet.csv')
2 print(dfIPL.groupby('Year').groups)
```

Output

```
{2014: Int64Index([0, 2, 4, 9],
dtype='int64'),
2015: Int64Index([1, 3, 5, 10],
dtype='int64'),
2016: Int64Index([6, 8],
dtype='int64'),
2017: Int64Index([7, 11],
dtype='int64')}
```


Groupby in Pandas (Cont.)

► Example : Group by multiple columns

dfGroupMul.py

```
1 dfIPL = pd.read_csv('IPLDataSet.csv')
2 print(dfIPL.groupby(['Year','Team']).groups)
```

Output

```
{(2014, 'Devils'): Int64Index([2],
dtype='int64'),
 (2014, 'Kings'): Int64Index([4],
dtype='int64'),
 (2014, 'Riders'): Int64Index([0],
dtype='int64'),
 .....
 .....
 (2016, 'Riders'): Int64Index([8],
dtype='int64'),
 (2017, 'Kings'): Int64Index([7],
dtype='int64'),
 (2017, 'Riders'): Int64Index([11],
dtype='int64')}
```

Groupby in Pandas (Cont.)

► Example : Iterating through groups

dfGrouplter.py

```
1 dfIPL = pd.read_csv('IPLDataSet.csv')
2 groupIPL = dfIPL.groupby('Year')
3 for name, group in groupIPL :
4     print(name)
5     print(group)
```

Output

2014

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

2015

	Team	Rank	Year	Points
1	Riders	2	2015	789
3	Devils	3	2015	673
5	kings	4	2015	812
10	Royals	1	2015	804

2016

	Team	Rank	Year	Points
6	Kings	1	2016	756
8	Riders	2	2016	694

2017

	Team	Rank	Year	Points
7	Kings	1	2017	788
11	Riders	2	2017	690

Groupby in Pandas (Cont.)

► Example : Aggregating groups

dfGroupAgg.py

```
1 dfSales = pd.read_csv('SalesDataSet.csv')
2 print(dfSales.groupby(['YEAR_ID']).count(
3     )['QUANTITYORDERED'])
4 print(dfSales.groupby(['YEAR_ID']).sum()['QUANTITYORDERED'])
5 print(dfSales.groupby(['YEAR_ID']).mean()['QUANTITYORDERED'])
```

Output

```
YEAR_ID
2003      1000
2004      1345
2005       478
Name: QUANTITYORDERED, dtype:
int64
YEAR_ID
2003      34612
2004      46824
2005      17631
Name: QUANTITYORDERED, dtype:
int64
YEAR_ID
2003      34.612000
2004      34.813383
2005      36.884937
Name: QUANTITYORDERED, dtype:
float64
```

Groupby in Pandas (Cont.)

► Example : Describe details

dfGroupDesc.py

```
1 dfIPL =  
  pd.read_csv('IPLDataSet.csv')  
2 print(dfIPL.groupby('Year').describe()[ 'Points' ])
```

Output

	count	mean	std	min	max
	25%	50%	75%		
Year					
2014	4.0	795.25	87.439026	701.0	731.0
		802.0	866.25	876.0	
2015	4.0	769.50	65.035888	673.0	760.0
		796.5	806.00	812.0	
2016	2.0	725.00	43.840620	694.0	709.5
		725.0	740.50	756.0	
2017	2.0	739.00	69.296465	690.0	714.5
		739.0	763.50	788.0	

Concatenation in Pandas

- ▶ Concatenation basically glues together DataFrames.
- ▶ Keep in mind that dimensions should match along the axis you are concatenating on.
- ▶ You can use **pd.concat** and pass in a list of DataFrames to concatenate together:

dfConcat.py

```
1 dfCX = pd.read_csv('CX_Marks.csv',index_col=0)
2 dfCY = pd.read_csv('CY_Marks.csv',index_col=0)
3 dfCZ = pd.read_csv('CZ_Marks.csv',index_col=0)
4 dfAllStudent = pd.concat([dfCX,dfCY,dfCZ])
5 print(dfAllStudent)
```

Output

	PDS	Algo	SE
101	50	55	60
102	70	80	61
103	55	89	70
104	58	96	85
201	77	96	63
202	44	78	32
203	55	85	21
204	69	66	54
301	11	75	88
302	22	48	77
303	33	59	68
304	44	55	62

- ▶ Note : We can use `axis=1` parameter to concat columns.

Join in Pandas

- ▶ `df.join()` method will efficiently join multiple DataFrame objects by **index**(or column specified) .
- ▶ some of important **Parameters** :
 - ➞ **dfOther** : Right Data Frame
 - ➞ **on** (Not recommended) : specify the column on which we want to join (Default is index)
 - ➞ **how** : How to handle the operation of the two objects.
 - **left**: use calling frame's index (*Default*).
 - **right**: use dfOther index.
 - **outer**: form union of calling frame's index with other's index (or column if on is specified), and sort it. lexicographically.
 - **inner**: form intersection of calling frame's index (or column if on is specified) with other's index, preserving the order of the calling's one.

Join in Pandas (Example)

dfJoin.py

```
1 dfINS =  
  pd.read_csv('INS_Marks.csv',index_col=0)  
2 dfLeftJoin = allStudent.join(dfINS)  
3 print(dfLeftJoin)  
4 dfRightJoin =  
  allStudent.join(dfINS,how='right')  
5 print(dfRightJoin)
```

Output - 1

	PDS	Algo	SE	INS
101	50	55	60	55.0
102	70	80	61	66.0
103	55	89	70	77.0
104	58	96	85	88.0
201	77	96	63	66.0
202	44	78	32	NaN
203	55	85	21	78.0
204	69	66	54	85.0
301	11	75	88	11.0
302	22	48	77	22.0
303	33	59	68	33.0
304	44	55	62	44.0

Output - 2

	PDS	Algo	SE	INS
301	11	75	88	11
302	22	48	77	22
303	33	59	68	33
304	44	55	62	44
101	50	55	60	55
102	70	80	61	66
103	55	89	70	77
104	58	96	85	88
201	77	96	63	66
203	55	85	21	78
204	69	66	54	85

Merge in Pandas

- ▶ Merge DataFrame or named Series objects with a database-style join.
- ▶ Similar to join method, but used when we want to join/merge with the columns instead of index.
- ▶ some of important **Parameters** :
 - ➔ **dfOther** : Right Data Frame
 - ➔ **on** : specify the column on which we want to join (Default is index)
 - ➔ **left_on** : specify the column of left Dataframe
 - ➔ **right_on** : specify the column of right Dataframe
 - ➔ **how** : How to handle the operation of the two objects.
 - **left**: use calling frame's index (*Default*).
 - **right**: use dfOther index.
 - **outer**: form union of calling frame's index with other's index (or column if on is specified), and sort it. lexicographically.
 - **inner**: form intersection of calling frame's index (or column if on is specified) with other's index, preserving the order of the calling's one.

Merge in Pandas (Example)

dfMerge.py

```
1 m1 = pd.read_csv('Merge1.csv')
2 print(m1)
3 m2 = pd.read_csv('Merge2.csv')
4 print(m2)
5 m3 = m1.merge(m2,on='EnNo')
6 print(m3)
```

Output

	RollNo	EnNo	Name
0	101	11112222	Abc
1	102	11113333	Xyz
2	103	22224444	Def

	EnNo	PDS	INS
0	11112222	50	60
1	11113333	60	70

	RollNo	EnNo	Name	PDS	INS
0	101	11112222	Abc	50	60
1	102	11113333	Xyz	60	70

Read CSV in Pandas

- ▶ `read_csv()` is used to read Comma Separated Values (CSV) file into a pandas DataFrame.
- ▶ some of important **Parameters** :
 - ➔ **filePath** : str, path object, or file-like object
 - ➔ **sep** : separator (Default is comma)
 - ➔ **header**: Row number(s) to use as the column names.
 - ➔ **index_col** : index column(s) of the data frame.

readCSV.py

```
1 dfINS = pd.read_csv('Marks.csv', index_col=0, header=0)
2 print(dfINS)
```

Output

	PDS	Algo	SE	INS
101	50	55	60	55.0
102	70	80	61	66.0
103	55	89	70	77.0
104	58	96	85	88.0
201	77	96	63	66.0

Read Excel in Pandas

- ▶ Read an Excel file into a pandas DataFrame.
- ▶ Supports *xls*, *xlsx*, *xlsm*, *xlsb*, *odf*, *ods* and *odt* file extensions read from a local filesystem or URL. Supports an option to read a single sheet or a list of sheets.
- ▶ some of important **Parameters** :
 - **excelFile** : str, bytes, ExcelFile, xlrd.Book, path object, or file-like object
 - **sheet_name** : sheet no in integer or the name of the sheet, can have list of sheets.
 - **index_col** : index column of the data frame.

Read from MySQL Database

- ▶ We need two libraries for that,
 - ➔ conda install **sqlalchemy**
 - ➔ conda install **pymysql**
- ▶ After installing both the libraries, import create_engine from sqlalchemy and import pymysql

importsForDB.py

```
1 from sqlalchemy import create_engine
2 import pymysql
```

- ▶ Then, create a database connection string and create engine using it.

createEngine.py

```
1 db_connection_str = 'mysql+pymysql://username:password@host/dbname'
2 db_connection = create_engine(db_connection_str)
```

Read from MySQL Database (Cont.)

- ▶ After getting the engine, we can fire any sql query using `pd.read_sql` method.
- ▶ `read_sql` is a generic method which can be used to read from any sql (MySQL, MSSQL, Oracle etc...)

readSQLDemo.py

```
1 df = pd.read_sql('SELECT * FROM cities', con=db_connection)
2 print(df)
```

Output

	CityID	CityName	CityDescription	CityCode
0	1	Rajkot	Rajkot Description here	RJT
1	2	Ahemdabad	Ahemdabad Description here	ADI
2	3	Surat	Surat Description here	SRT

What Is Web Scraping?

- ▶ **Web scraping** is the process of gathering information from the Internet. Even copying and pasting the lyrics of your favorite song is a form of web scraping! However, the words “web scraping” usually refer to a process that involves automation. Some websites don't like it when automatic scrapers gather their data, while others don't mind.
- ▶ Challenges of Web Scraping
- ▶ The Web has grown organically out of many sources. It combines many different technologies, styles, and personalities, and it continues to grow to this day. In other words, the Web is a hot mess! Because of this, you'll run into some challenges when scraping the Web:
 - **Variety:** Every website is different. While you'll encounter general structures that repeat themselves, each website is unique and will need personal treatment if you want to extract the relevant information.
 - **Durability:** Websites constantly change. Say you've built a shiny new web scraper that automatically cherry-picks what you want from your resource of interest. The first time you run your script, it works flawlessly. But when you run the same script only a short while later, you run into a discouraging and lengthy stack of tracebacks!

An Alternative to Web Scraping: APIs

- ▶ Some website providers offer application programming interfaces (APIs) that allow you to access their data in a predefined manner. With APIs, you can avoid parsing HTML. Instead, you can access the data directly using formats like JSON and XML. HTML is primarily a way to present content to users visually.
- ▶ When you use an API, the process is generally more stable than gathering the data through web scraping. That's because developers create APIs to be consumed by programs rather than by human eyes.
- ▶ The front-end presentation of a site might change often, but such a change in the website's design doesn't affect its API structure. The structure of an API is usually more permanent, which means it's a more reliable source of the site's data.
- ▶ **Parse HTML Code With BeautifulSoup**
- ▶ Beautiful Soup is a Python library for **parsing structured data**. It allows you to interact with HTML in a similar way to how you interact with a web page using developer tools. The library exposes a couple of intuitive functions you can use to explore the HTML you received. To get started, use your terminal to install BeautifulSoup:

Step 1: Inspect Your Data Source

- ▶ Before you write any Python code, you need to get to know the website that you want to scrape. That should be your first step for any web scraping project you want to tackle. You'll need to understand the site structure to extract the information that's relevant for you. Start by opening [the site you want to scrape](#) with your favorite browser.

- ▶ **Step 2: Scrape HTML Content From a Page**

- ▶ Now that you have an idea of what you're working with, it's time to start using Python. First, you'll want to get the site's HTML code into your Python script so that you can interact with it. For this task, you'll use Python's requests library.

- ▶ `$ python -m pip install requests`

```
import requests
```

```
URL = "https://realpython.github.io/fake-jobs/"
```

```
page = requests.get(URL)
```

```
print(page.text)
```


- ▶ This code issues an HTTP GET request to the given URL. It retrieves the HTML data that the server sends back and stores that data in a Python object.
- ▶ If you print the `.text` attribute of `page`, then you'll notice that it looks just like the HTML that you inspected earlier with your browser's developer tools. You successfully fetched the static site content from the Internet! You now have access to the site's HTML from within your Python script.

Static Websites

- ▶ The website that you're scraping in this tutorial serves **static HTML content**. In this scenario, the server that hosts the site sends back HTML documents that already contain all the data that you'll get to see as a user.

```
<div class="card">

<div class="card-content">

  <div class="media">

    <div class="media-left">

      <figure class="image is-48x48">

      </figure>

    </div>

    <div class="media-content">

      <h2 class="title is-5">Senior Python Developer</h2>

      <h3 class="subtitle is-6 company">Payne, Roberts and Davis</h3>

    </div>

  </div>

</div>
```

```
<div class="content">

  <p class="location">Stewartbury, AA</p>

  <p class="is-small has-text-grey">

    <time datetime="2021-04-08">2021-04-08</time>

  </p>

</div>
```

```
<footer class="card-footer">

  <a

    href="https://www.realpython.com"

    target="_blank"
```

- ▶ The HTML you'll encounter will sometimes be confusing. Luckily, the HTML of this job board has descriptive class names on the elements that you're interested in:
- ▶ `class="title is-5"` contains the title of the job posting.
- ▶ `class="subtitle is-6 company"` contains the name of the company that offers the position.
- ▶ `class="location"` contains the location where you'd be working.
- ▶ In case you ever get lost in a large pile of HTML, remember that you can always go back to your browser and use the developer tools to further explore the HTML structure interactively.

Step 3: Parse HTML Code With BeautifulSoup

- ▶ You've successfully scraped some HTML from the Internet, but when you look at it, it just seems like a huge mess. There are tons of HTML elements here and there, thousands of attributes scattered around—and wasn't there some JavaScript mixed in as well? It's time to parse this lengthy code response with the help of Python to make it more accessible and pick out the data you want.
- ▶ BeautifulSoup is a Python library for parsing structured data. It allows you to interact with HTML in a similar way to how you interact with a web page using developer tools. The library exposes a couple of intuitive functions you can use to explore the HTML you received. To get started, use your terminal to install BeautifulSoup:

```
$ python -m pip install beautifulsoup4
```

Then, import the library in your Python script and create a Beautiful Soup object:

```
import requests
```

```
from bs4 import BeautifulSoup
```

```
URL = "https://realpython.github.io/fake-jobs/"
```

```
page = requests.get(URL)
```

```
soup = BeautifulSoup(page.content, "html.parser")
```

When you add the two highlighted lines of code, you create a Beautiful Soup object that takes `page.content`, which is the HTML content you scraped earlier, as its input.

Find Elements by ID

- ▶ In an HTML web page, every element can have an id attribute assigned. As the name already suggests, that id attribute makes the element uniquely identifiable on the page. You can begin to parse your page by selecting a specific element by its ID.
- ▶ Switch back to developer tools and identify the HTML object that contains all the job postings. Explore by hovering over parts of the page and using right-click to Inspect.

The element you're looking for is a `<div>` with an id attribute that has the value "ResultsContainer". It has some other attributes as well, but below is the gist of what you're looking for:

```
<div id="ResultsContainer">  
  <!-- all the job listings -->  
</div>
```

Beautiful Soup allows you to find that specific HTML element by its ID:

```
results = soup.find(id="ResultsContainer")
```

For easier viewing, you can prettify any BeautifulSoup object when you print it out. If you call `.prettify()` on the `results` variable that you just assigned above, then you'll see all the HTML contained within the `<div>`:

```
print(results.prettify())
```

When you use the element's ID, you can pick out one element from among the rest of the HTML. Now you can work with only this specific part of the page's HTML. It looks like the soup just got a little thinner! However, it's still quite dense.

Find Elements by HTML Class Name

- ▶ You've seen that every job posting is wrapped in a `<div>` element with the class `card-content`. Now you can work with your new object called `results` and select only the job postings in it. These are, after all, the parts of the HTML that you're interested in! You can do this in one line of code:

```
job_elements = results.find_all("div", class_="card-content")
```

Here, you call `.find_all()` on a BeautifulSoup object, which returns an iterable containing all the HTML for all the job listings displayed on that page.

```
for job_element in job_elements:
```

```
    title_element = job_element.find("h2", class_="title")
```

```
    company_element = job_element.find("h3", class_="company")
```

```
    location_element = job_element.find("p", class_="location")
```

```
    print(title_element)
```

```
    print(company_element)
```

```
    print(location_element)
```

```
    print()
```


- ▶ Each `job_element` is another `BeautifulSoup()` object. Therefore, you can use the same methods on it as you did on its parent element, `results`.
- ▶ With this code snippet, you're getting closer and closer to the data that you're actually interested in. Still, there's a lot going on with all those HTML tags and attributes floating around:

```
<h2 class="title is-5">Senior Python Developer</h2>
```

```
<h3 class="subtitle is-6 company">Payne, Roberts and Davis</h3>
```

```
<p class="location">Stewartbury, AA</p>
```