CHENNAI INSTITUTE OF TECHNOLOGY

(Affiliated to Anna University, Approved by AICTE, Accredited by NAAC & NBA)
Sarathy Nagar, Kundrathur, Chennai – 600069, India.

Lecture Notes Unit IV

DEPARTMENT OF INFORMATION TECHNOLOGY

Subject: CS 3353-Foundations of Data Science

Dr.A.R.Kavitha

II Year IT/III SEMESTER

UNIT IV PYTHON LIBRARIES FOR DATA WRANGLING

Basics of Numpy arrays –aggregations –computations on arrays –comparisons, masks, boolean logic – fancy indexing – structured arrays – Data manipulation with Pandas – data indexing and selection – operating on data – missing data – Hierarchical indexing – combining datasets – aggregation and grouping – pivot tables

Python

- Python is a general purpose, dynamic, high-level, and interpreted programming language
- Python is easy to learn yet powerful and versatile scripting language, which makes it attractivefor Application Development.
- Python's syntax and dynamic typing with its interpreted nature make it an ideal language for scripting and rapid application development.
- Python supports multiple programming pattern, including object-oriented, imperative,and functional or procedural programming styles.
- Python is not intended to work in a particular area, such as web programming. It isknown is multipurpose programming language because it can be used with web, enterprise, 3D CAD, etc.

Built-in Data Types

Variables can store data of different types, and different types can do different things.
Python has the following data types built-in by default, in these categories:

Text Type:	str		
Numeric Types:	int, float, complex	Sequence Types:	list, tuple, range
Mapping Type:	dict		
Set Types:	set, frozenset		
Boolean Type:	bool		
Binary Types:	bytes, bytearray, memoryview	None Type:	NoneType

Getting the Data Type

Get the data type of any object by using the type() functionExample: Print the data type of the variable x:

```
x = 5
print(type(x))
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable.

Python Arrays

- An array is defined as a collection of items that are stored at contiguous memory locations.
- It is a container which can hold a fixed number of items, and these items should be of the same type.

Creation of Arrays

The Array can be created in Python by importing the array module to the python program.from

```
array import *
arrayName = array(typecode, [initializers])
```

Accessing array elements

Access the array elements using the respective indices of those elements.

```
import array as arr
a = arr.array('i', [2, 4, 6, 8]) print("First element:", a[0]) //2
print("Second element:", a[1])  //4
print("Second last element:", a[-1]) // 8
```

change or add elements

```
import array as arr
numbers = arr.array('i', [1, 2, 3, 5, 7, 10])
# changing first element
numbers[0] = 0
print(numbers)
# Output: array('i', [0, 2, 3, 5, 7, 10])
# changing 3rd to 5th element
numbers[2:5] = arr.array('i', [4, 6, 8])
print(numbers)
# Output: array('i', [0, 2, 4, 6, 8, 10])
```

Delete elements from an array

The elements can be deleted from an array using Python's del statement. If we want to delete anyvalue from the array, we can do that by using the indices of a particular element.

```
import array as arr
number = arr.array('i', [1, 2, 3, 3, 4])
del number[2]
# removing third element
print(number)
# Output: array('i', [1, 2, 3, 4])
```

The **length of an array** is defined as the number of elements present in an array. It returns an integervalue that is equal to the total number of the elements present in that array.

Syntax len(array_name)

Python - 2-D Array

Two dimensional array is an array within an array. It is an array of arrays. In this type of array the position of an data element is referred by two indices instead of one. So it represents a tablewith rowsand columns of data.

```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
print(T[0])  // [11,12,5,2]
print(T[1][2]) //10
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
for r in T:
for c in r:
print(c,end = " ")
print()
```

Output:

```
11 12 5 2
15 6 10
10  8 12 5
12 15 8 6
```

Inserting Values

Insert new data elements at specific position by using the insert() method and specifying the index.Example - a new data element is inserted at index position 2.

```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
T.insert(2, [0,5,11,13,6])
for r in T: for c in r:
    print(c,end = " ")print()
```

Updating Values

We can update the entire inner array or some specific data elements of the inner array by reassigningthe values using the array index.

```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
T[2] = [11,9]
T[0][3] = 7
for r in T:
for c in r:
print(c,end = " ")
print()
```

Deleting the Values

We can delete the entire inner array or some specific data elements of the inner array by reassigning the values using the del() method with index. But in case you need to remove specific data elements in one of the inner arrays, then use the update process described above.

```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
del T[3] for r in T:
for c in r:
  print(c,end = " ")
  print()
```

Output:

```
11 12 5 2
15 6 10
10 8 12 5
```

To reverse the order of the items in the array:

```
from array import *
a = array('i', [1, 3, 5, 3, 7, 1, 9, 3])
print("Original array: "+str(a))
a.reverse()
print("Reverse the order of the items:")
print(str(a))
```

To insert a new item before the second element in an existing array:

```
from array import *
a = array('i', [1, 3, 5, 7, 9])
print("Original array: "+str(a)) print("Insert new value 4 before 3:")
a.insert(1, 4)
print("New array: "+str(array_num))
```

NumPy

- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant.
- It is an open source project and you can use it freely.
- NumPy stands for Numerical Python. The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.
- NumPy Faster Than Lists

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Numpy arrays	**Description**
np.array([1,2,3])	1d array
np.array([(1,2,3),(4,5,6)])	2d array
np.arange(start,stop,step)	range array

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray. We can create aNumPy ndarray object by using the array() function.

Example

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

Use a tuple to create a NumPy array:

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

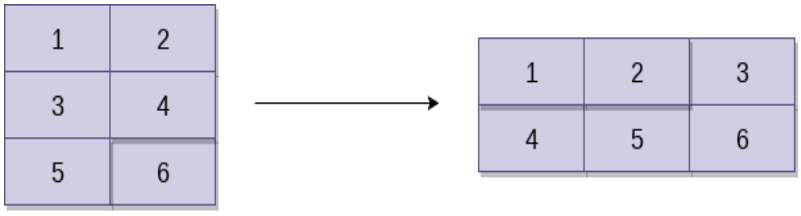
#finding the size of each item in the array

```
import numpy as np a = np.array([[1,2,3]])
print("Each item contains",a.itemsize,"bytes
```

#finding the data type of each array item
import numpy as np a = np.array([[1,2,3]])
print("Each item is of the type",a.dtype) ")

#Finding the shape and size of the array
import numpy as np
a = np.array([[1,2,3,4,5,6,7]])
print("Array Size:",a.size)print("Shape:",a.shape) **Output:**
Array Size: 7
Shape: (1, 7)

Reshaping the array objects
The shape of the array means the number of rows and columns of a multi-dimensional array. The numpy module provides us the way to reshape the array by changing the number of rows and columnsof the multi-dimensional array.
The reshape() function associated with the ndarray object is used to reshape the array. It accepts the two parameters indicating the row and columns of the new shape of the array.



2 X 3 **3 X 2**

Reshape 3x2 array into 2x3 array:
import numpy as np
a = np.array([[1,2],[3,4],[5,6]])
print("printing the original array..")
print(a)
a=a.reshape(2,3)
print("printing the reshaped array..")
print(a)
Output:
printing the original array..[[1 2]
[3 4]
[5 6]]
printing the reshaped array[[1 2 3]
[4 5 6]]

Linspace
The linspace() function returns the evenly spaced values over the given interval. The following examplereturns the 10 evenly separated values over the given interval 5-15
Example
import numpy as np
a=np.linspace(5,15,10) #prints 10 values which are evenly spaced over the given interval 5-15
print(a)
Output:
[5. 6.11111111 7.22222222 8.33333333 9.44444444 10.55555556
11.66666667 12.77777778 13.88888889 15.]

Finding the maximum, minimum, and sum of the array elements

The NumPy provides the max(), min(), and sum() functions which are used to find the maximum,minimum, and sum of the array elements respectively.

Example

```
import numpy as np
a = np.array([1,2,3,10,15,4])
print("The array:",a)
print("The maximum element:",a.max())
print("The minimum element:",a.min())
print("The sum of the elements:",a.sum())
```

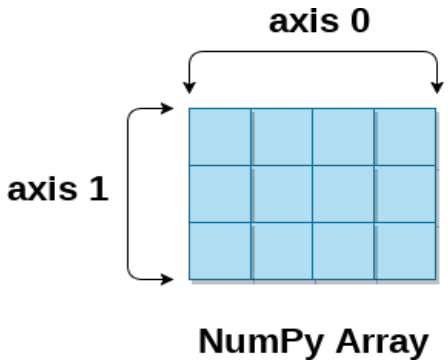
Output:

The array: [1 2 3 10 15 4] The maximum element: 15 The minimum element: 1 The sum of the elements: 35

NumPy Array Axis

A NumPy multi-dimensional array is represented by the axis where axis-0 represents the columns andaxis-1 represents the rows.

We can mention the axis to perform row-level or column-level calculations like the addition of row orcolumn elements.



To calculate the maximum element among each column, the minimum element among each row, andthe addition of all the row elements, consider the following example.

Example

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
print("The array:",a)
print("The maximum elements of columns:",a.max(axis = 0))print("The minimum element of rows",a.min(axis = 1)) print("The sum of all rows",a.sum(axis = 1))
```

Output:

The array: [[1 2 30]
[10 15 4]]
The maximum elements of columns: [10 15 30]The minimum element of rows [1 4]
The sum of all rows [33 29]

Finding square root and standard deviation

The sqrt() and std() functions associated with the numpy array are used to find the square root andstandard deviation of the array elements respectively.

Standard deviation means how much each element of the array varies from the mean value of thenumpy array.

Example

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
print(np.sqrt(a))print(np.std(a))
```

Output:

```
[[1.       1.41421356 5.47722558]
[3.16227766 3.87298335 2.          ]]
10.044346115546242
```

Array Concatenation

The numpy provides us with the vertical stacking and horizontal stacking which allows us to concatenate two multi-dimensional arrays vertically or horizontally.

Example

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
b = np.array([[1,2,3],[12, 19, 29]])
print("Arrays vertically concatenated\n",np.vstack((a,b))); print("Arrays horizontally
concatenated\n",np.hstack((a,b)))
```

Output:

```
Arrays vertically concatenated[[ 1 2 30]
[10 15 4]
[ 1  2  3]
[12 19 29]]
Arrays horizontally concatenated[[ 1 2 30 1 2 3]
[10 15 4 12 19 29]]
```

Numpy.empty

As the name specifies, The empty routine is used to create an uninitialized array of specified shape anddata type. The syntax is given below.

numpy.empty(shape, dtype = float, order = 'C')It accepts the following parameters.

Shape: The desired shape of the specified array.

dtype: The data type of the array items. The default is the float.

Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.empty((3,2), dtype = int)print(arr)
```

Output:

```
[[         140482883954664          36917984]
[         140482883954648     140482883954648]
[6497921830368665435  172026472699604272]]
```

NumPy.Zeros

This routine is used to create the numpy array with the specified shape where each numpy array itemis initialized to 0.

The syntax is given below. numpy.zeros(shape, dtype = float, order = 'C')

Shape: The desired shape of the specified array.

dtype: The data type of the array items. The default is the float.
Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.zeros((3,2), dtype = int)print(arr)
```

Output:

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

NumPy.ones

This routine is used to create the numpy array with the specified shape where each numpy array itemis initialized to 1.
The syntax to use this module is given below. numpy.ones(shape, dtype = none, order = 'C') It accepts the following parameters.
Shape: The desired shape of the specified array.dtype: The data type of the array items.
Order: The default order is the c-style row-major order. It can be set to F for FORTRAN-style column-major order.

Example

```
import numpy as np
arr = np.ones((3,2), dtype = int)print(arr)
```

Output:

```
[[1 1]
[1 1]
[1 1]]
```

Numpy array from existing data

NumPy provides us the way to create an array by using the existing data.numpy.asarray
This routine is used to create an array by using the existing data in the form of lists, or tuples. This routine is useful in the scenario where we need to convert a python sequence into the numpy array object.
numpy.asarray(sequence, dtype = None, order = None)It accepts the following parameters.
sequence: It is the python sequence which is to be converted into the python array.dtype: It is the data type of each item of the array.
order: It can be set to C or F. The default is C.

Example: creating numpy array using the list

```
import numpy as npl=[1,2,3,4,5,6,7]
a = np.asarray(l);print(type(a)) print(a)
```

Output:

```
<class 'numpy.ndarray'>[1 2 3 4 5 6 7]
```

Example: creating a numpy array using Tuple

```
import numpy as np
l=(1,2,3,4,5,6,7)
a = np.asarray(l)
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>[1 2 3 4 5 6 7]
```

Example: creating a numpy array using more than one list

```
import numpy as np
 l=[[1,2,3,4,5,6,7],[8,9]]
a = np.asarray(l);print(type(a)) print(a)
```

Output:

```
<class 'numpy.ndarray'>
[list([1, 2, 3, 4, 5, 6, 7]) list([8, 9])]
```

numpy.frombuffer

This function is used to create an array by using the specified buffer.numpy.frombuffer(buffer,

type = float, count = -1, offset = 0)

It accepts the following parameters.
buffer: It represents an object that exposes a buffer interface.
dtype: It represents the data type of the returned data type array. The default value is 0.count: It represents the length of the returned ndarray. The default value is -1.offset: It represents the starting position to read from. The default value is 0.

Example

```
import numpy as npl = b'hello world' print(type(l))
a = np.frombuffer(l, dtype = "S1")
print(a)
print(type(a))
```

Output:

```
<class 'bytes'>
[b'h' b'e' b'l' b'l' b'o' b' ' b'w' b'o' b'r' b'l' b'd']
<class 'numpy.ndarray'>
```

numpy.fromiter

This routine is used to create a ndarray by using an iterable object. It returns a one-dimensional ndarrayobject.

numpy.fromiter(iterable, dtype, count = - 1)It accepts the following parameters.
Iterable: It represents an iterable object.
dtype: It represents the data type of the resultant array items.
count: It represents the number of items to read from the buffer in the array.

Example

```
import numpy as nplist = [0,2,4,6]
it = iter(list)
x = np.fromiter(it, dtype = float)print(x)
print(type(x))
```

Output:

```
[0. 2. 4. 6.]
<class 'numpy.ndarray'>
```

Check how many dimensions the arrays have:

import numpy as npa = np.array(42)

```
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Output

```
0
1
2
3
```

Higher Dimensional Arrays

An array can have any number of dimensions. When the array is created, you can define the numberof dimensions by using the ndmin argument.

Example

Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

Vector

In Python, vector is a **single one-dimension** array of lists and behaves same as a Python list

create vector from a list in python

```
# Importing numpy import numpy as np
# creating list
list1 = [10, 20, 30, 40, 50]
# Creating 1-D Horizontal Arrayvtr = np.array(list1)
print("We create a vector from a list:")
print(vtr)
```

Vector dot product

The vector dot product performs between the two same-length sequential vectors and returns thesingle dot product. The **.dot()** method is used to perform the dot product.

```
list1 = [10,20,30,40,50]
list2 = [5,2,4,3,1]
vtr1 = np.array(list1)vtr2= np.array(list2)
print("We create vector from a list 1:")
print(vtr1)
print("We create a vector from a list 2:")print(vtr2)
vtr_product = vtr1.dot(vtr2)
print("Dot product of two vectors: ",vtr_product)
```

output:

We create vector from a list 1:
[10 20 30 40 50]
We create vector from a list 2:
[5 2 4 3 1]
Dot product of two vectors: 3800

Basic operations that can be performed in vector.

- Arithmetic (vtr1 + vtr2)
- Subtraction
- Multiplication
- Division
- Dot Product
- Scalar Multiplications (vtr1 * slr)

vectorization in Python

Vectorization is a technique of implementing array operations without using for loops. The functions defined by various modules are used, which are highly optimized that reduces therunning and execution time of code.

Fancy indexing in vectorization

Fancy indexing allows you to use one array as an index into another array.
It can be used to pick values from an array, and also to implement table lookup algorithms.

Example:

```
a = np.array([-1.0, 2.0, -3.0, 4.0])
idx = np.array([2, 1, 1, 0, 3])
r = a[idx]
r contains:
[-3.  2.  2. -1. 4.]
```

When using fancy indexing, the shape of the result reflects the shape of the index arrays rather thanthe shape of the array being indexed:

```
import numpy as np
rand = np.random.RandomState(42)
x = rand.randint(100, size=10)
print(x)
[51 92 14 71 60 20 82 86 74 74]
ind = np.array([[3, 7],[4, 5]])
x[ind] array([[71, 86],[60, 20]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:X = np.arange(12).reshape((3, 4))

```
X
array([[ 0,  1,  2, 3],
[ 4, 5,  6,  7],
[ 8, 9, 10, 11]])
```

Like with standard indexing, **the first index refers to the row, and the second to the column:**

row = np.array([0, 1, 2])
col = np.array([2, 1, 3])X[row, col]
array([2, 5, 11])

X[row[:, np.newaxis], col]
array([[2, 1, 3], [6, 5, 7], [10, 9, 11]])
Here, each row value is matched with each column vector, exactly as we saw in broadcasting ofarithmetic operations. For example:
row[:, np.newaxis] * col
Output
array([[0, 0, 0],
[2, 1, 3],
[4, 2, 6]])
It is always important to remember with fancy indexing that the return value reflects the broadcastedshape of the indices, rather than the shape of the array being indexed.

Combined Indexing
For even more powerful operations, fancy indexing can be combined with the other indexing schemes.X = np.arange(12).reshape((3, 4))
print(X)
[[0 1 2 3]
[4 5 6 7]
[8 9 10 11]]
We can combine fancy and simple indices:X[2, [2, 0, 1]]
array([10, 8, 9])

We can also combine fancy indexing with slicing:X[1:, [2, 0, 1]]
array([[6, 4, 5],
[10, 8, 9]])
And we can combine fancy indexing with masking:mask = np.array([1, 0, 1, 0], dtype=bool)
X[row[:, np.newaxis], mask]
array([[0, 2],
[4, 6],
[8, 10]])

Add, Subtract, Divide and Multiply matrices

Numpy methods for matrix manipulations:

numpy.add() − Add two matrices

numpy.subtract() − Subtract two matrices

numpy.divide() − Divide two matrices
numpy.multiply() − Multiply two matrices
Example:
mx1 = np.array([[5, 10], [15, 20]])

```
mx2 = np.array([[25, 30], [35, 40]])
print("Matrix1 =\n",mx1) print("\nMatrix2 =\n",mx2)
# The addition() is used to add matrices
print ("\nAddition of two matrices: ")
print (np.add(mx1,mx2))
# The subtract() is used to subtract matrices
print ("\nSubtraction of two matrices: ")
print (np.subtract(mx1,mx2))
# The divide() is used to divide matrices
print ("\nMatrix Division: ")
print (np.divide(mx1,mx2))
# The multiply()is used to multiply matrices
print ("\nMultiplication of two matrices: ")
print (np.multiply(mx1,mx2))
```

Summation of matrix elements

The sum() method is used to find the summation

Example

```
import numpy as np
# A matrix
mx = np.array([[5, 10], [15, 20]])
print("Matrix =\n",mx)
print ("\nThe summation of elements=")
print (np.sum(mx))
print ("\nThe column wise summation=")
print (np.sum(mx,axis=0))
print ("\nThe row wise summation=")
print (np.sum(mx,axis=1))
```

Transpose a Matrix

The .T property is used to find the Transpose of a Matrix −

Example

```
import numpy as np
# A matrix
mx = np.array([[5, 10], [15, 20]])
print("Matrix =\n",mx)
print ("\nThe Transpose =")print (mx.T)
```

Comparisons, masks, boolean logic

This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

Comparison Operators as ufuncs

In Computation on NumPy Arrays: Universal Functions we introduced ufuncs, and focused in particular on arithmetic operators. We saw that using +, -, *, /, and others on arrays leads to element-wise operations. NumPy also implements comparison operators such as < (less than) and > (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:

```
x = np.array([1, 2, 3, 4, 5])
x < 3  # less than
array([ True,  True, False, False, False], dtype=bool)
x > 3  # greater than
array([False, False, False,  True,  True], dtype=bool)
x <= 3  # less than or equal
array([ True,  True,  True, False, False], dtype=bool)
x >= 3  # greater than or equal
array([False, False,  True,  True,  True], dtype=bool)
x != 3  # not equal
array([ True,  True, False,  True,  True], dtype=bool)
x == 3  # equal
array([False, False,  True, False, False], dtype=bool)
```

It is also possible to do an element-wise comparison of two arrays, and to include compound expressions:

```
 (2 * x) == (x ** 2)
array([False,  True, False, False, False], dtype=bool)
```

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write x < 3, internally NumPy uses np.less(x, 3).A summary of the comparison operators and their equivalent ufunc is shown here:

Operator	Equivalent ufunc	Operator	Equivalent ufunc
==	np.equal	!=	np.not_equal
<	np.less	<=	np.less_equal
>	np.greater	>=	np.greater_equal

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
x < 6
```

```
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]], dtype=bool)
```

In each case, the result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

Working with Boolean Arrays

Given a Boolean array, there are a host of useful operations you can do. We'll work with `x`, the two-dimensional array we created earlier.

```
print(x)
```

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

Counting entries

To count the number of `True` entries in a Boolean array, `np.count_nonzero` is useful:

```
# how many values less than 6?
np.count_nonzero(x < 6)
```

```
8
```

We see that there are eight array entries that are less than 6. Another way to get at this information is to use `np.sum`; in this case, `False` is interpreted as `0`, and `True` is interpreted as `1`:

```
np.sum(x < 6)
```

```
8
```

The benefit of `sum()` is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

```
# how many values less than 6 in each row?
np.sum(x < 6, axis=1)
```

```
array([4, 2, 2])
```

This counts the number of values less than 6 in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) `np.any` or `np.all`:

```
# are there any values greater than 8?
np.any(x > 8)
```

```
True
```

```
# are there any values less than zero?
np.any(x < 0)
```

```
False
```

```
# are all values less than 10?
np.all(x < 10)
```

```
True
```

 # are all values equal to 6?
np.all(x == 6)
False
np.all and np.any can be used along particular axes as well. For example:
 # are all values in each row less than 8?
np.all(x < 8, axis=1)
array([True, False, True], dtype=bool)
Here all the elements in the first and third rows are less than 8, while this is not the case for the second row.
Finally, a quick warning: as mentioned in Aggregations: Min, Max, and Everything In Between, Python has built-in sum(), any(), and all() functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multidimensional arrays. Be sure that you are using np.sum(), np.any(),and np.all() for these examples!
Boolean Arrays as Masks
In the preceding section we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our x array from before, suppose we want an array of all values in the array that are less than, say, 5:
x
array([[5, 0, 3, 3],
    [7, 9, 3, 5],
    [2, 4, 7, 6]])
We can obtain a Boolean array for this condition easily, as we've already seen:
x < 5
array([[False, True, True, True],
    [False, False, True, False],
    [True, True, False, False]], dtype=bool)
Now to *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:
x[x < 5]
array([0, 3, 3, 3, 2, 4])
What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is True.
We are then free to operate on these values as we wish. For example, we can compute some relevant statistics on our Seattle rain data:
 # construct a mask of all rainy days
rainy = (inches > 0)

 # construct a mask of all summer days (June 21st is the 172nd day)

```
days = np.arange(365)
summer = (days > 172) & (days < 262)

print("Median precip on rainy days in 2014 (inches): ",
    np.median(inches[rainy]))
print("Median precip on summer days in 2014 (inches): ",
    np.median(inches[summer]))
print("Maximum precip on summer days in 2014 (inches): ",
    np.max(inches[summer]))
print("Median precip on non-summer rainy days (inches):",
    np.median(inches[rainy & ~summer]))
```

```
Median precip on rainy days in 2014 (inches):    0.194881889764
Median precip on summer days in 2014 (inches):  0.0
Maximum precip on summer days in 2014 (inches): 0.850393700787
Median precip on non-summer rainy days (inches): 0.200787401575
```

Python Structured Array

Numpy's Structured Array is similar to Struct in C. It is used for grouping data of different types and sizes. Structure array uses data containers called fields. Each data field can contain data of any type and size. Array elements can be accessed with the help of dot notation.

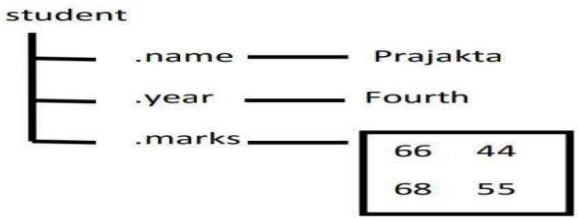
Note: Arrays with named fields that can contain data of various types and sizes.

Properties of Structured array

All structs in array have same number of fields.

- All structs have same fields names.

For example, consider a structured array of student which has different fields like name, year, marks.



Each record in array student has a structure of class Struct. The array of a structure is referred to as struct as adding any new fields for a new struct in the

array, contains the empty array.

Example 1:

```
 # Python program to demonstrate
# Structured array
import numpy as np
 a = np.array([('Sana', 2, 21.0), ('Mansi', 7, 29.0)],
     dtype=[('name', (np.str_, 10)), ('age', np.int32), ('weight', np.float64)])

print(a)
```

Output:

[('Sana', 2, 21.0) ('Mansi', 7, 29.0)]

Data Manipulation with Pandas

Pandas is an open-source library that is used from data manipulation to data analysis & is very powerful, flexible & easy to use tool which can be imported using import pandasas pd. Pandas deal essentially with data in 1-D and 2-D arrays; Although, pandas handles these two differently. In pandas, 1-D arrays are stated as a series & a dataframe is simplya 2-D array. The dataset used here is country_code.csv.

Below are various operations used to manipulate the dataframe:

- First, import the library which is used in data manipulation i.e. pandas then assign and read the dataframe:
- # import module
- import pandas as pd
- # assign dataset
- df = pd.read_csv("country_code.csv")
- # display
- print("Type-", type(df))
- df
- We can read the dataframe by using **head()** function also which is having an argument (n) i.e. number of rows to be displayed.
  df.head(10)
- Counting the rows and columns in DataFrame using **shape().** It returns the no. of rows and columns enclosed in a tuple.
  df.shape
- summary of Statistics of DataFrame using **describe()** method.
  df.describe()
- Dropping the missing values in DataFrame, it can be done using the **dropna()** method, it removes all the NaN values in the dataframe.
  df.dropna()

df.dropna(axis=1)- This will drop all the columns with any missing values.

- Merging DataFrames using **merge()**, arguments passed are the dataframes to be merged along with the column name.

```
df1 = pd.read_csv("country_code.csv")
merged_col = pd.merge(df, df1, on='Name')
merged_col
```

Renaming the columns of dataframe using **rename()**, arguments passed are the columns to be renamed & inplace.

```
country_code = df.rename(columns={'Name': 'CountryName',
                                  'Code': 'CountryCode'},
                         inplace=False)
country_code
```

Creating a dataframe manually:

```
student = pd.DataFrame({'Name': ['Rohan', 'Rahul', 'Gaurav',
                                 'Ananya', 'Vinay', 'Rohan',
                                 'Vivek', 'Vinay'],

                        'Score': [76, 69, 70, 88, 79, 64, 62, 57]})
 # Reading Dataframe
Student
```

- Sorting the DataFrame using **sort_values()** method.

```
student.sort_values(by=['Score'], ascending=True)
```

- Sorting the DataFrame using multiple columns:

```
student.sort_values(by=['Name', 'Score'],
                    ascending=[True, False])
```

- Creating another column in DataFrame, Here we will create column name percentage which will calculate the percentage of student score by using aggregate function sum().

```
student['Percentage'] = (student['Score'] / student['Score'].sum()) * 100
student
```

- Selecting DataFrame rows using logical operators:

```
# Selecting rows where score is
# greater than 70
print(student[student.Score>70])
# Selecting rows where score is greater than 60
# OR less than 70
print(student[(student.Score>60) | (student.Score<70)])
```

- **Indexing & Slicing :**
  Here **.loc** is label base & **.iloc** is integer position based methods used for slicing & indexing of data.

```
# Printing five rows with name column only
# i.e. printing first 5 student names.
print(student.loc[0:4, 'Name'])

# Printing all the rows with score column
# only i.e. printing score of all the
# students
print(student.loc[:, 'Score'])

# Printing only first rows having name,
# score columns i.e. print first student
# name & their score.
print(student.iloc[0, 0:2])

# Printing first 3 rows having name,score &
# percentage columns i.e. printing first three
# student name,score & percentage.
print(student.iloc[0:3, 0:3])

# Printing all rows having name & score
# columns i.e. printing all student
# name & their score.
print(student.iloc[:, 0:2])
```

- Apply Functions, this function is used to apply a function along an axis of dataframe whether it can be row (axis=0) or column (axis=1).

```
# explicit function
def double(a):
   return 2*a
student['Score'] = student['Score'].apply(double)

# Reading Dataframe
student
```

Hierarchical indexing

For enhancing the capabilities of Data Processing, we have to use some indexing that helps to sort the data based on the labels. So, Hierarchical indexing is comes into the picture and defined as an essential feature of pandas that helps us to use the multiple index levels.

Creating multiple index

In Hierarchical indexing, we have to create multiple indexes for the data. This example creates a series with multiple indexes.

```
import pandas as pd
info = pd.Series([11, 14, 17, 24, 19, 32, 34, 27],
index = [['x', 'x', 'x', 'x', 'y', 'y', 'y', 'y'],
['obj1', 'obj2', 'obj3', 'obj4', 'obj1', 'obj2', 'obj3', 'obj4']])
data
```

We have taken two level of index here i.e. **(a, b)** and **(obj1,..., obj4)** and can see the index by using '**index**' command.

```
info.index
```

Output:

```
MultiIndex(levels=[['x', 'y'], ['obj1', 'obj2', 'obj3', 'obj4']],
labels=[[0, 0, 0, 0, 1, 1, 1, 1], [0, 1, 2, 3, 0, 1, 2, 3]])
```

Partial indexing

Partial indexing can be defined as a way to choose the particular index from a hierarchical indexing.

Below code is extracting 'b' from the data,

```
import pandas as pd
info = pd.Series([11, 14, 17, 24, 19, 32, 34, 27],
index = [['x', 'x', 'x', 'x', 'y', 'y', 'y', 'y'],
['obj1', 'obj2', 'obj3', 'obj4', 'obj1', 'obj2', 'obj3', 'obj4']])
info['b']
```

Combining Datasets with Pandas

With pandas, you can **merge**, **join**, and **concatenate** your datasets, allowing you to unify and better understand your data as you analyze it.

you'll learn how and when to combine your data in pandas with:

(n) **merge()** for combining data on common columns or indices

(o) **.join()** for combining data on a key column or an index

(p) **concat()** for combining DataFrames across rows or columns

If you have some experience using DataFrame and Series objects in pandas and you're ready to learn how to combine them, then this tutorial will help you do exactly that. If you're feeling a bit rusty, then you can watch a quick refresher on DataFrames beforeproceeding.

You can follow along with the examples in this tutorial using the interactive Jupyter Notebook and data files available at the link below:

Download the notebook and data set: Click here to get the Jupyter Notebook and CSV data set you'll use to learn about Pandas merge(), .join(), and concat()

pandas merge(): Combining Data on Common Columns or Indices

The first technique that you'll learn is merge(). You can use merge() anytime you want functionality similar to a database's join operations. It's the most flexible of the three operations that you'll learn.

When you want to combine data objects based on one or more keys, similar to what you'd do in a relational database, merge() is the tool you need. More specifically, merge() is most useful when you want to combine rows that share data.

You can achieve both **many-to-one** and **many-to-many** joins with merge(). In a many-to-one join, one of your datasets will have many rows in the merge column that repeat the same values. For example, the values could be 1, 1, 3, 5, and 5. At the same time, the merge column in the other dataset won't have repeated values. Take 1, 3, and 5 as an example.

As you might have guessed, in a many-to-many join, both of your merge columns will have repeated values. These merges are more complex and result in the Cartesian product of the joined rows.

This means that, after the merge, you'll have every combination of rows that share the same value in the key column. You'll see this in action in the examples below.

What makes merge() so flexible is the sheer number of options for defining the behavior of your merge. While the list can seem daunting, with practice you'll be able to expertly merge datasets of all kinds.

When you use merge(), you'll provide two required arguments:

1. The left DataFrame
2. The right DataFrame

After that, you can provide a number of optional arguments to define how your datasets are merged:

(q) **how** defines what kind of merge to make. It defaults to 'inner', but other possible options include 'outer', 'left', and 'right'.

(r) **on** tells merge() which columns or indices, also called **key columns** or **key indices**, you want to join on. This is optional. If it isn't specified, and left_index and right_index (covered below) are False, then columns from the two DataFrames that share names will be used as join keys. If you use on, then the column or index that you specify must be present in both objects.

(s) **left_on** and **right_on** specify a column or index that's present only in the left or right object that you're merging. Both default to None.

(t) **left_index** and **right_index** both default to False, but if you want to use the index of the left or right object to be merged, then you can set the relevant argument to True.

(u) **suffixes** is a tuple of strings to append to identical column names that aren't merge keys. This allows you to keep track of the origins of columns with the same name.

These are some of the most important parameters to pass to merge(). For the full list, see the pandas documentation.

Note: In this tutorial, you'll see that examples always use on to specify which column(s) to join on. This is the safest way to merge your data because you and anyone reading your code will know exactly what to expect when calling merge(). If you don't specify the merge column(s) with on, then pandas will use any columns with the same name as the merge keys.

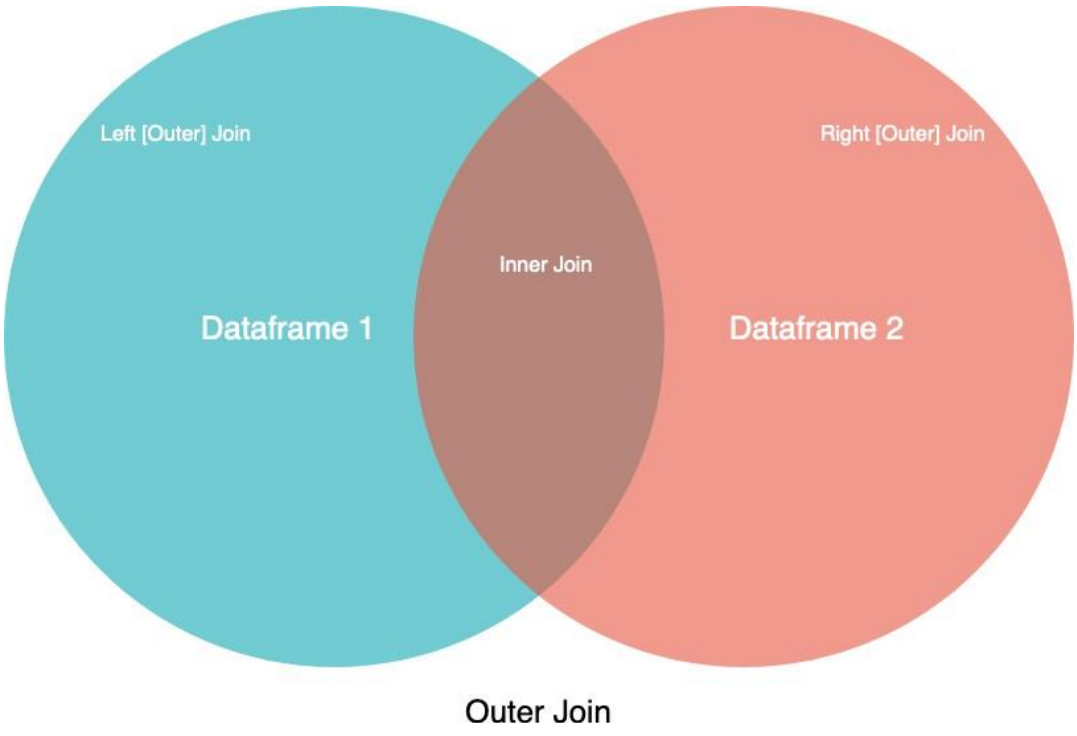
How to Use merge()

Before getting into the details of how to use merge(), you should first understand the various forms of joins:

(v) inner
(w) outer
(x) left
(y) right

Note: Even though you're learning about merging, you'll see inner, outer, left, and right also referred to as join operations. For this tutorial, you can consider the terms **merge** and **join** equivalent.

You'll learn about these different joins in detail below, but first take a look at this visual representation of them:



Visual Representation of Join Types

In this image, the two circles are your two datasets, and the labels point to which part or parts of the datasets you can expect to see. While this diagram doesn't cover all the nuance, it can be a handy guide for visual learners.

If you have an SQL background, then you may recognize the merge operation names from the JOIN syntax. Except for inner, all of these techniques are types of **outer joins**. With outer joins, you'll merge your data based on all the keys in the left object, the right object,or both. For keys that only exist in one object, unmatched columns in the other object will be filled in with NaN, which stands for *Not a Number*.

You can also see a visual explanation of the various joins in an SQL context on Coding Horror. Now take a look at the different joins in action.

Examples

Many pandas tutorials provide very simple DataFrames to illustrate the concepts that they are trying to explain. This approach can be confusing since you can't relate the data to anything concrete. So, for this tutorial, you'll use two real-world datasets as the DataFrames to be merged:

1. Climate normals for California (temperatures)
2. Climate normals for California (precipitation)

You can explore these datasets and follow along with the examples below using the interactive Jupyter Notebook and climate data CSVs:

Download the notebook and data set: Click here to get the Jupyter Notebook and CSV data set you'll use to learn about Pandas merge(), .join(), and concat() in this tutorial.

If you'd like to learn how to use Jupyter Notebooks, then check out Jupyter Notebook: An Introduction.

These two datasets are from the National Oceanic and Atmospheric Administration (NOAA) and were derived from the NOAA public data repository. First, load the datasets into separate DataFrames:

```
>>>
>>> import pandas as pd
>>> climate_temp = pd.read_csv("climate_temp.csv")
>>> climate_precip = pd.read_csv("climate_precip.csv")
```

In the code above, you used pandas' read_csv() to conveniently load your source CSV files into DataFrame objects. You can then look at the headers and first few rows of the loaded DataFrames with .head():

```
>>>
>>> climate_temp.head()
         STATION         STATION_NAME  ... DLY-HTDD-BASE60 DLY-HTDD-NORMAL
0  GHCND:USC00049099  TWENTYNINE PALMS CA US  ...          10             15
1  GHCND:USC00049099  TWENTYNINE PALMS CA US  ...          10             15
2  GHCND:USC00049099  TWENTYNINE PALMS CA US  ...          10             15
3  GHCND:USC00049099  TWENTYNINE PALMS CA US  ...          10             15
```

```
4  GHCND:USC00049099  TWENTYNINE PALMS CA US  ...          10          15

>>> climate_precip.head()
         STATION  ... DLY-SNOW-PCTALL-GE050TI
0  GHCND:USC00049099  ...               -9999
1  GHCND:USC00049099  ...               -9999
2  GHCND:USC00049099  ...               -9999
3  GHCND:USC00049099  ...                   0
4  GHCND:USC00049099  ...                   0
```

Here, you used .head() to get the first five rows of each DataFrame. Make sure to try this on your own, either with the interactive Jupyter Notebook or in your console, so that you can explore the data in greater depth.

Next, take a quick look at the dimensions of the two DataFrames:

```
>>>
>>> climate_temp.shape
(127020, 21)
>>> climate_precip.shape
(151110, 29)
```

Note that .shape is a property of DataFrame objects that tells you the dimensions of the DataFrame. For climate_temp, the output of .shape says that the DataFrame has 127,020 rows and 21 columns.

Inner Join

In this example, you'll use merge() with its default arguments, which will result in an inner join. Remember that in an inner join, you'll lose rows that don't have a match in the other **DataFrame's** key column**.**

With the two datasets loaded into DataFrame objects, you'll select a small slice of the precipitation dataset and then use a plain merge() call to do an inner join. This will result in a smaller, more focused dataset:

```
>>>
>>> precip_one_station = climate_precip.query("STATION == 'GHCND:USC00045721'")
>>> precip_one_station.head()
           STATION  ... DLY-SNOW-PCTALL-GE050TI
1460  GHCND:USC00045721  ...               -9999
1461  GHCND:USC00045721  ...               -9999
1462  GHCND:USC00045721  ...               -9999
1463  GHCND:USC00045721  ...               -9999
1464  GHCND:USC00045721  ...               -9999
```

Here you've created a new DataFrame called precip_one_station from the climate_precip DataFrame, selecting only rows in which the STATION field is "GHCND:USC00045721".

If you check the shape attribute, then you'll see that it has 365 rows. When you do the merge, how many rows do you think you'll get in the merged DataFrame? Remember that you'll be doing an inner join:

```
>>>
>>> inner_merged = pd.merge(precip_one_station, climate_temp)
>>> inner_merged.head()
         STATION          STATION_NAME ... DLY-HTDD-BASE60  DLY-HTDD-NORMAL
0  GHCND:USC00045721  MITCHELL CAVERNS CA US  ...            14            19
1  GHCND:USC00045721  MITCHELL CAVERNS CA US  ...            14            19
2  GHCND:USC00045721  MITCHELL CAVERNS CA US  ...            14            19
3  GHCND:USC00045721  MITCHELL CAVERNS CA US  ...            14            19
4  GHCND:USC00045721  MITCHELL CAVERNS CA US  ...            14            19

>>> inner_merged.shape
(365, 47)
```

If you guessed 365 rows, then you were correct! This is because merge() defaults to an inner join, and an inner join will discard only those rows that don't match. Because all of your rows had a match, none were lost. You should also notice that there are many more columns now: 47 to be exact.

With merge(), you also have control over which column(s) to join on. Let's say that you want to merge both entire datasets, but only on Station and Date since the combination of the two will yield a unique value for each row. To do so, you can use the on parameter:

```
>>>
>>> inner_merged_total = pd.merge(
...     climate_temp, climate_precip, on=["STATION", "DATE"]
... )
>>> inner_merged_total.shape
(123005, 48)
```

You can specify a single key column with a string or multiple key columns with a list. This results in a DataFrame with 123,005 rows and 48 columns.

Why 48 columns instead of 47? Because you specified the key columns to join on, pandas doesn't try to merge all mergeable columns. This can result in "duplicate" column names, which may or may not have different values.

"Duplicate" is in quotation marks because the column names will not be an exact match. By default, they are appended with _x and _y. You can also use the suffixes parameter to control what's appended to the column names.

To prevent surprises, all the following examples will use the on parameter to specify the column or columns on which to join.

Outer Join

Here, you'll specify an outer join with the how parameter. Remember from the diagrams above that in an outer join—also known as a **full outer join**—all rows from both DataFrames will be present in the new DataFrame.

If a row doesn't have a match in the other DataFrame based on the key column(s), then you won't lose the row like you would with an inner join. Instead, the row will be in the merged DataFrame, with NaN values filled in where appropriate.

This is best illustrated in an example:

```
>>>
>>> outer_merged = pd.merge(
...     precip_one_station, climate_temp, how="outer", on=["STATION", "DATE"]
... )
>>> outer_merged.shape
(127020, 48)
```

If you remember from when you checked the .shape attribute of climate_temp, then you'll see that the number of rows in outer_merged is the same. With an outer join, you can expect to have the same number of rows as the larger DataFrame. That's because no rows are lost in an outer join, even when they don't have a match in the other DataFrame.

Left Join

In this example, you'll specify a left join—also known as a **left outer join**—withthe how parameter. Using a left outer join will leave your new merged DataFrame with all rows from the left DataFrame, while discarding rows from the right DataFrame that don't have a match in the key column of the left DataFrame.

You can think of this as a half-outer, half-inner merge. The example below shows you this in action:

```
>>>
>>> left_merged = pd.merge(
...     climate_temp, precip_one_station, how="left", on=["STATION", "DATE"]
... )
>>> left_merged.shape
(127020, 48)
```

left_merged has 127,020 rows, matching the number of rows in the left DataFrame, climate_temp. To prove that this only holds for the left DataFrame, run the same code, but change the position of precip_one_station and climate_temp:

```
>>>
>>> left_merged_reversed = pd.merge(
...     precip_one_station, climate_temp, how="left", on=["STATION", "DATE"]
... )
>>> left_merged_reversed.shape
(365, 48)
```

This results in a DataFrame with 365 rows, matching the number of rowsin precip_one_station.

Right Join

The right join, or **right outer join**, is the mirror-image version of the left join. With this join, all rows from the right DataFrame will be retained, while rows in the left DataFrame without a match in the key column of the right DataFrame will be discarded.

To demonstrate how right and left joins are mirror images of each other, in the example below you'll recreate the left_merged DataFrame from above, only this time using a right join:

>>>

```
>>> right_merged = pd.merge(
...     precip_one_station, climate_temp, how="right", on=["STATION", "DATE"]
... )
>>> right_merged.shape
(127020, 48)
```

Here, you simply flipped the positions of the input DataFrames and specified a right join. When you inspect right_merged, you might notice that it's not exactly the same as left_merged. The only difference between the two is the order of the columns: the first input's columns will always be the first in the newly formed DataFrame.

merge() is the most complex of the pandas data combination tools. It's also the foundation on which the other tools are built. Its complexity is its greatest strength, allowing you to combine datasets in every which way and to generate new insights into your data.

On the other hand, this complexity makes merge() difficult to use without an intuitive grasp of set theory and database operations. In this section, you've learned about the various data merging techniques, as well as many-to-one and many-to-many merges, which ultimately come from set theory. For more information on set theory, check out Sets in Python.

Now, you'll look at .join(), a simplified version of merge().

pandas .join(): Combining Data on a Column or Index

While merge() is a **module function**, .join() is an **instance method** that lives on your DataFrame. This enables you to specify only one DataFrame, which will join the DataFrame you call .join() on.

Under the hood, .join() uses merge(), but it provides a more efficient way to joinDataFrames than a fully specified merge() call. Before diving into the options available to you, take a look at this short example:

>>>

```
>>> precip_one_station.join(
...     climate_temp, lsuffix="_left", rsuffix="_right"
... ).shape
(365, 50)
```

With the indices visible, you can see a left join happening here, with precip_one_station being the left DataFrame. You might notice that this example provides the parameters lsuffix and rsuffix. Because .join() joins on indices and doesn't directly merge DataFrames, all columns—even those with matching names—are retainedin the resulting DataFrame.

Now flip the previous example around and instead call .join() on the larger DataFrame:

>>>

```
>>> climate_temp.join(
...     precip_one_station, lsuffix="_left", rsuffix="_right"
... ).shape
(127020, 50)
```

Notice that the DataFrame is larger, but data that doesn't exist in the smaller DataFrame, precip_one_station, is filled in with NaN values.

How to Use .join()

By default, .join() will attempt to do a left join on indices. If you want to join on columns like you would with merge(), then you'll need to set the columns as indices.

Like merge(), .join() has a few parameters that give you more flexibility in your joins.

However, with .join(), the list of parameters is relatively short:

(z) **other** is the only required parameter. It defines the other DataFrame to join. You can also specify a list of DataFrames here, allowing you to combine a number of datasets in a single .join() call.

(aa) **on** specifies an optional column or index name for the left DataFrame (climate_temp in the previous example) to join the other DataFrame's index. If it's set to None, which is the default, then you'll get an index-on-index join.

(bb) **how** has the same options as how from merge(). The difference is that it's index-based unless you also specify columns with on.

(cc) **lsuffix** and **rsuffix** are similar to suffixes in merge(). They specify a suffix to add to any overlapping columns but have no effect when passing a list of other DataFrames.

(dd) **sort** can be enabled to sort the resulting DataFrame by the join key.Examples

In this section, you'll see examples showing a few different use cases for .join(). Some will be simplifications of merge() calls. Others will be features that set .join() apart from the more verbose merge() calls.

Since you already saw a short .join() call, in this first example you'll attempt to recreate a merge() call with .join(). What will this require? Take a second to think about a possible solution, and then look at the proposed solution below:

>>>

```
>>> inner_merged_total = pd.merge(
...     climate_temp, climate_precip, on=["STATION", "DATE"]
```

```
... )
>>> inner_merged_total.shape
(123005, 48)

>>> inner_joined_total = climate_temp.join(
...     climate_precip.set_index(["STATION", "DATE"]),
...     on=["STATION", "DATE"],
...     how="inner",
...     lsuffix="_x",
...     rsuffix="_y",
... )
>>> inner_joined_total.shape
(123005, 48)
```

Because .join() works on indices, if you want to recreate merge() from before, then you must set indices on the join columns that you specify. In this example, you used .set_index() to set your indices to the key columns within the join. Notethat .join() does a left join by default so you need to explictly use how to do an inner join.

With this, the connection between merge() and .join() should be clearer.

Below you'll see a .join() call that's almost bare. Because there are overlapping columns, you'll need to specify a suffix with lsuffix, rsuffix, or both, but this example will demonstrate the more typical behavior of .join():

```
>>>
>>> climate_temp.join(climate_precip, lsuffix="_left").shape
(127020, 50)
```

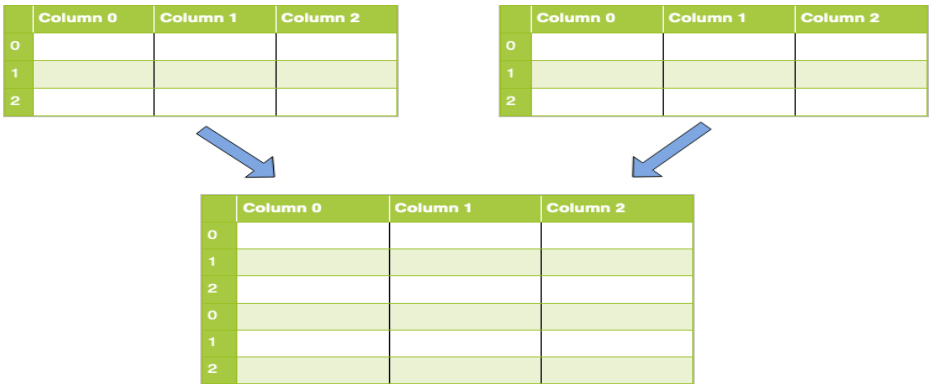
This example should be reminiscent of what you saw in the introduction to .join() earlier. The call is the same, resulting in a left join that produces a DataFrame with the same number of rows as climate_temp.

In this section, you've learned about .join() and its parameters and uses. You've also learned about how .join() works under the hood, and you've recreated a merge() call with .join() to better understand the connection between the two techniques.

pandas concat(): Combining Data Across Rows or Columns

Concatenation is a bit different from the merging techniques that you saw above. With merging, you can expect the resulting dataset to have rows from the parent datasets mixed in together, often based on some commonality. Depending on the type of merge, you might also lose rows that don't have matches in the other dataset.

With concatenation, your datasets are just stitched together along an **axis** — either the **row axis** or **column axis**. Visually, a concatenation with no parameters along rows would look like this:

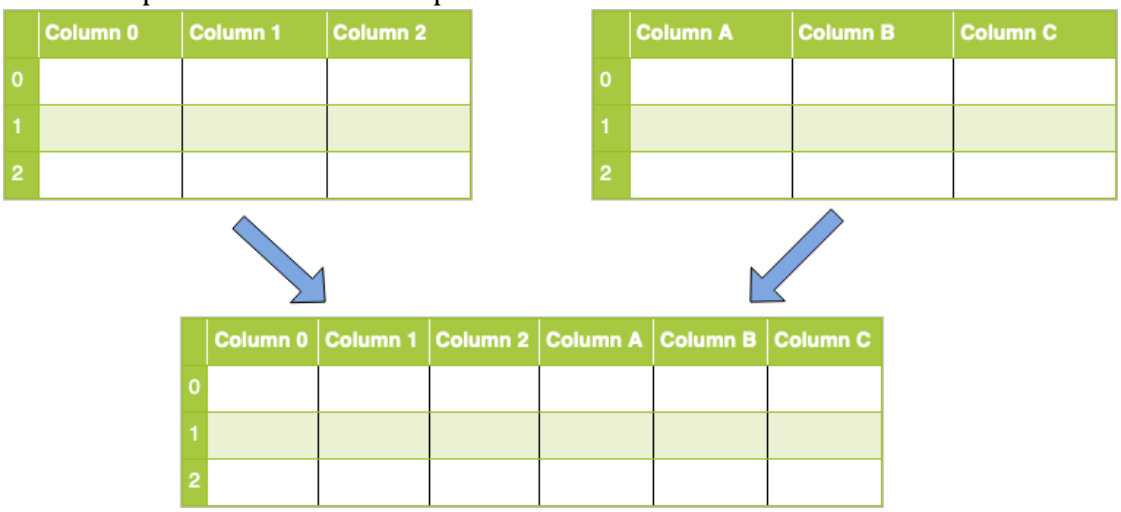


To implement this in code, you'll use concat() and pass it a list of DataFrames that you want to concatenate. Code for this task would look like this:

```
concatenated = pandas.concat([df1, df2])
```

Note: This example assumes that your column names are the same. If your column names are different while concatenating along rows (axis 0), then by default the columns will also be added, and NaN values will be filled in as applicable.

What if you wanted to perform a concatenation along columns instead? First, take a look at a visual representation of this operation:



To accomplish this, you'll use a concat() call like you did above, but you'll also need to pass the axis parameter with a value of 1 or "columns":

```
concatenated = pandas.concat([df1, df2], axis="columns")
```

Note: This example assumes that your indices are the same between datasets. If they're different while concatenating along columns (axis 1), then by default the extra indices (rows) will also be added, and NaN values will be filled in as applicable.

You'll learn more about the parameters for concat() in the section below.

How to Use concat()

As you can see, concatenation is a simpler way to combine datasets. It's often used to form a single, larger set to do additional operations on.

Note: When you call concat(), a copy of all the data that you're concatenating is made. You should be careful with multiple concat() calls, as the many copies that are made may negatively affect performance. Alternatively, you can set the optional copy parameterto False

When you concatenate datasets, you can specify the axis along which you'll concatenate. But what happens with the other axis?

Nothing. By default, a concatenation results in a **set union**, where all data is preserved. You've seen this with merge() and .join() as an outer join, and you can specify this with the join parameter.

If you use this parameter, then the default is outer, but you also have the inner option, which will perform an inner join, or **set intersection**.

As with the other inner joins you saw earlier, some data loss can occur when you do an inner join with concat(). Only where the axis labels match will you preserve rows or columns.

Note: Remember, the join parameter only specifies how to handle the axes that you're *not* concatenating along.

Since you learned about the join parameter, here are some of the other parameters that concat() takes:

(ee) **objs** takes any sequence—typically a list—of Series or DataFrame objects to be concatenated. You can also provide a dictionary. In this case, the keys will be used to construct a hierarchical index.

(ff) **axis** represents the axis that you'll concatenate along. The default value is 0, which concatenates along the index, or row axis. Alternatively, a value of 1 will concatenate vertically, along columns. You can also use the string values "index" or "columns".

(gg) **join** is similar to the how parameter in the other techniques, but it only accepts the values inner or outer. The default value is outer, which preserves data, while inner would eliminate data that doesn't have a match in the other dataset.

(hh) **ignore_index** takes a Boolean True or False value. It defaults to False. If True, then the new combined dataset won't preserve the original index values in the axis specified in the axis parameter. This lets you have entirely new index values.

(ii) **keys** allows you to construct a hierarchical index. One common use case is to have a new index while preserving the original indices so that you can tell which rows, for example, come from which original dataset.

(jj) **copy** specifies whether you want to copy the source data. The default value is True. If the value is set to False, then pandas won't make copies of the source data.

This list isn't exhaustive. You can find the complete, up-to-date list of parameters in the pandas documentation.

Examples

First, you'll do a basic concatenation along the default axis using the DataFrames that you've been playing with throughout this tutorial:

```
>>>
>>> double_precip = pd.concat([precip_one_station, precip_one_station])
>>> double_precip.shape
(730, 29)
```
This one is very simple by design. Here, you created a DataFrame that is a double of a small DataFrame that was made earlier. One thing to notice is that the indices repeat. If you want a fresh, 0-based index, then you can use the ignore_index parameter:
```
>>>
>>> reindexed = pd.concat(
...     [precip_one_station, precip_one_station], ignore_index=True
... )
>>> reindexed.index
RangeIndex(start=0, stop=730, step=1)
```
As noted before, if you concatenate along axis 0 (rows) but have labels in axis 1 (columns) that don't match, then those columns will be added and filled in with NaN values. This results in an outer join:
```
>>>
>>> outer_joined = pd.concat([climate_precip, climate_temp])
>>> outer_joined.shape
(278130, 47)
```
With these two DataFrames, since you're just concatenating along rows, very few columns have the same name. That means you'll see a lot of columns with NaN values.

To instead drop columns that have any missing data, use the join parameter with the value "inner" to do an inner join:
```
>>>
>>> inner_joined = pd.concat([climate_temp, climate_precip], join="inner")
>>> inner_joined.shape
(278130, 3)
```
Using the inner join, you'll be left with only those columns that the original DataFrames have in common: STATION, STATION_NAME, and DATE.

You can also flip this by setting the axis parameter:
```
>>>
>>> inner_joined_cols = pd.concat(
...     [climate_temp, climate_precip], axis="columns", join="inner"
... )
>>> inner_joined_cols.shape
(127020, 50)
```
Now you have only the rows that have data for all columns in both DataFrames. It's no coincidence that the number of rows corresponds with that of the smaller DataFrame.

Another useful trick for concatenation is using the keys parameter to create hierarchical axis labels. This is useful if you want to preserve the indices or column names of the original datasets but also want to add new ones:

```
>>>
>>> hierarchical_keys = pd.concat(
...     [climate_temp, climate_precip], keys=["temp", "precip"]
... )
>>> hierarchical_keys.index
MultiIndex([(  'temp',     0),
            (  'temp',     1),
            ...
            ('precip', 151108),
            ('precip', 151109)],
           length=278130)
```

If you check on the original DataFrames, then you can verify whether the higher-level axis labels temp and precip were added to the appropriate rows.

You've now learned the three most important techniques for combining data in pandas:

1. **merge()** for combining data on common columns or indices
2. **.join()** for combining data on a key column or an index
3. **concat()** for combining DataFrames across rows or columns

In addition to learning how to use these techniques, you also learned about set logic by experimenting with the different ways to join your datasets. Additionally, you learnedabout the most common parameters to each of the above techniques, and what arguments you can pass to customize their output.

You saw these techniques in action on a real dataset obtained from the NOAA, whichshowed you not only how to combine your data but also the benefits of doing so with pandas' built-in techniques. If you haven't downloaded the project files yet, you can get them here:

Download the notebook and data set: Click here to get the Jupyter Notebook and CSV data set you'll use to learn about Pandas merge(), .join(), and concat() in this tutorial.

Did you learn something new? Figure out a creative way to solve a problem by combining complex datasets? Let us know in the comments below!

Grouping and Aggregating with Pandas

We are going to see grouping and aggregating using pandas. Grouping and aggregating will help to achieve data analysis easily using various functions. These methods will help us to the group and summarize our data and make complex analysis comparatively easy.

Creating a sample dataset of marks of various subjects.

```
# import module
import pandas as pd
```

```
# Creating our dataset
df = pd.DataFrame([[9, 4, 8, 9],
                   [8, 10, 7, 6],
                   [7, 6, 8, 5]],
                  columns=['Maths', 'English',
                           'Science', 'History'])

# display dataset
print(df)
```

output:

	Maths	English	Science	History
0	9	4	8	9
1	8	10	7	6
2	7	6	8	5

Aggregation in Pandas

Aggregation in pandas provides various functions that perform a mathematical or logical operation on our dataset and returns a summary of that function. Aggregation can be used to get a summary of columns in our dataset like getting sum, minimum, maximum, etc. from a particular column of our dataset. The function used for aggregation is agg(), the parameter is the function we want to perform.

Some functions used in the aggregation are:

Function Description:

- *sum() :Compute sum of column values*
- *min() :Compute min of column values*
- *max() :Compute max of column values*
- *mean() :Compute mean of column*
- *size() :Compute column sizes*
- *describe() :Generates descriptive statistics*
- *first() :Compute first of group values*
- *last() :Compute last of group values*
- *count() :Compute count of column values*
- *std() :Standard deviation of column*
- *var() :Compute variance of column*
- *sem() :Standard error of the mean of column*

- The sum() function is used to calculate the sum of every value.

```
df.sum()
```

output:

```
Maths      24
English    20
Science    23
History    20
dtype: int64
```

- The describe() function is used to get a summary of our dataset

df.describe()

output:

	Maths	English	Science	History
count	3.0	3.000000	3.000000	3.000000
mean	8.0	6.666667	7.666667	6.666667
std	1.0	3.055050	0.577350	2.081666
min	7.0	4.000000	7.000000	5.000000
25%	7.5	5.000000	7.500000	5.500000
50%	8.0	6.000000	8.000000	6.000000
75%	8.5	8.000000	8.000000	7.500000
max	9.0	10.000000	8.000000	9.000000

- We used agg() function to calculate the sum, min, and max of each column in our dataset.

df.agg(['sum', 'min', 'max'])

	Maths	English	Science	History
sum	24	20	23	20
min	7	4	7	5
max	9	10	8	9

Grouping in Pandas

Grouping is used to group data using some criteria from our dataset. It is used as split-apply-combine strategy.

- Splitting the data into groups based on some criteria.
- Applying a function to each group independently.
- Combining the results into a data structure.

Examples:

We use groupby() function to group the data on “Maths” value. It returns the object as result.

df.groupby(by=['Maths'])

Applying groupby() function to group the data on “Maths” value. To view result of formed groups use first() function.

a = df.groupby('Maths')

a.first()

output:

Maths	English	Science	History
7	6	8	5
8	10	7	6
9	4	8	9

First grouping based on “Maths” within each team we are grouping based on “Science”

b = df.groupby(['Maths', 'Science'])

b.first()

output:

		English	History
Maths	**Science**		
7	**8**	6	5
8	**7**	10	6
9	**8**	4	9

Implementation on a Dataset

Here we are using a dataset of diamond information.

```
# import module
import numpy as np
import pandas as pd
# reading csv file
dataset = pd.read_csv("diamonds.csv")
# printing first 5 rows
print(dataset.head(5))
```

Output:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

- We group by using cut and get the sum of all columns.

dataset.groupby('cut').sum()

Output:

cut	carat	depth	table	price	x	y	z
Fair	1684.28	103107.1	95076.6	7017600	10057.50	9954.07	6412.26
Good	4166.10	305967.0	287955.9	19275009	28645.08	28703.75	17855.42
Ideal	15146.84	1329899.3	1205814.4	74513487	118691.07	118963.24	73304.61
Premium	12300.95	844901.1	810167.4	63221498	82385.88	81985.82	50297.49
Very Good	9742.70	746888.4	700226.2	48107623	69359.09	69713.45	43009.52

- Here we are grouping using cut and color and getting minimum value for all other groups.

dataset.groupby(['cut', 'color']).agg('min')

Output:

cut	color	carat	clarity	depth	table	price	x	y	z
Fair	D	0.25	I1	52.2	52.0	536	4.09	4.11	2.49
	E	0.22	I1	51.0	49.0	337	3.87	3.78	2.33
	F	0.25	I1	52.3	50.0	496	4.19	4.15	2.32
	G	0.23	I1	43.0	53.0	369	0.00	0.00	0.00
	H	0.33	I1	52.7	50.0	659	4.40	4.32	2.84
	I	0.41	I1	50.8	49.0	735	4.62	4.66	2.93
	J	0.30	I1	55.0	52.0	416	4.24	4.16	2.72
Good	D	0.23	I1	54.3	52.0	361	3.83	3.85	2.37
	E	0.23	I1	56.3	53.0	327	3.83	3.85	2.31
	F	0.23	I1	56.2	52.0	357	0.00	0.00	0.00
	G	0.23	I1	56.2	53.0	394	3.94	3.90	0.00
	H	0.25	I1	56.0	51.0	368	4.04	4.06	2.46
	I	0.30	I1	56.1	51.0	351	4.19	4.19	2.67
	J	0.28	I1	56.2	52.0	335	4.22	4.23	2.51
Ideal	D	0.20	I1	58.5	52.0	367	3.81	3.77	2.33
	E	0.20	I1	58.3	52.0	326	3.76	3.73	2.06
	F	0.23	I1	58.0	52.4	408	0.00	3.92	0.00
	G	0.23	I1	58.8	52.0	361	0.00	0.00	0.00
	H	0.23	I1	58.3	52.0	357	3.94	3.97	1.41
	I	0.23	I1	58.4	43.0	348	3.94	3.90	1.53
	J	0.23	I1	43.0	53.0	340	3.93	3.90	2.46
Premium	D	0.20	I1	58.0	52.0	367	0.00	0.00	0.00
	E	0.20	I1	58.0	52.0	326	3.79	3.75	2.24
	F	0.20	I1	58.0	51.0	342	3.73	3.71	0.00
	G	0.23	I1	58.0	52.0	382	3.95	3.92	0.00

	H	0.23	I1	58.0	51.0	368	0.00	0.00	0.00
	I	0.23	I1	58.0	52.0	334	3.97	3.94	0.00
	J	0.30	I1	58.0	54.0	363	4.22	4.21	2.59
Very Good	D	0.23	I1	57.5	52.0	357	3.86	3.85	2.35
	E	0.20	I1	57.7	44.0	352	3.74	3.71	2.25
	F	0.23	I1	56.9	52.0	357	3.84	3.86	2.36
	G	0.23	I1	57.1	52.0	354	3.88	3.92	2.36
	H	0.23	I1	56.8	52.0	337	0.00	0.00	0.00
	I	0.24	I1	57.5	52.0	336	3.95	3.98	2.46
	J	0.24	I1	57.6	51.6	336	3.94	3.96	2.48

- Here we are grouping using color and getting aggregate values like sum, mean, min, etc. for the price group.

```
# dictionary having key as group name of price and
# value as list of aggregation function
# we want to perform on group price
agg_functions = {
   'price':
   ['sum', 'mean', 'median', 'min', 'max', 'prod']
}
 dataset.groupby(['color']).agg(agg_functions)
```

Output:

	price					
	sum	mean	median	min	max	prod
color						
D	21476439	3169.954096	1838.0	357	18693	inf
E	30142944	3076.752475	1739.0	326	18731	inf
F	35542866	3724.886397	2343.5	342	18791	inf
G	45158240	3999.135671	2242.0	354	18818	inf
H	37257301	4486.669196	3460.0	337	18803	inf
I	27608146	5091.874954	3730.0	334	18823	inf
J	14949281	5323.818020	4234.0	335	18710	inf

We can see that in the prod(product i.e. multiplication) column all values are inf, inf is the result of a numerical calculation that is mathematically infinite.

Pivot Table

Pandas.pivot_table()

pandas.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All') create a spreadsheet-style pivot table as a DataFrame.

Levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters:

data : *DataFrame*

values : *column to aggregate, optional*

index: *column, Grouper, array, or list of the previous*

columns: *column, Grouper, array, or list of the previous*

aggfunc: *function, list of functions, dict, default numpy.mean*

-> *If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names.*

-> *If dict is passed, the key is column to aggregate and value is function or list of functions*

fill_value[scalar, default None] : *Value to replace missing values with*

margins[boolean, default False] : *Add all row / columns (e.g. for subtotal / grand totals)*

dropna[boolean, default True] : *Do not include columns whose entries are all NaN*

margins_name[string, default 'All'] : *Name of the row / column that will contain the totals when margins is True.*

Returns: *DataFrame*

```
# Create a simple dataframe
# importing pandas as pd
import pandas as pd
import numpy as np
# creating a dataframe
df = pd.DataFrame({'A': ['John', 'Boby', 'Mina', 'Peter', 'Nicky'],
    'B': ['Masters', 'Graduate', 'Graduate', 'Masters', 'Graduate'],
    'C': [27, 23, 21, 23, 24]})
df
```

Out[1]:

	A	B	C
0	John	Masters	27
1	Boby	Graduate	23
2	Mina	Graduate	21
3	Peter	Masters	23
4	Nicky	Graduate	24

```
# Simplest pivot table must have a dataframe
# and an index/list of index.
table = pd.pivot_table(df, index =['A', 'B'])
table
```

Out[2]:

		C
A	B	
Boby	Graduate	23
John	Masters	27
Mina	Graduate	21
Nicky	Graduate	24
Peter	Masters	23

```
# Creates a pivot table dataframe
table = pd.pivot_table(df, values ='A', index =['B', 'C'],
                       columns =['B'], aggfunc = np.sum)
 table
```

Out[3]:

	B	Graduate	Masters
B	C		
Graduate	21	Mina	NaN
	23	Boby	NaN
	24	Nicky	NaN
Masters	23	NaN	Peter
	27	NaN	John

Question bank

Part - A

1. How to create an empty and a full NumPy array?
2. Create a Numpy array filled with all zeros.
3. How to Remove rows in Numpy array that contains non-numeric values?
4. How to compare two NumPy arrays?
5. Change data type of given numpy array
6. Reverse a numpy array.
7. Define the Pandas/Python pandas?
8. Mention the different types of Data Structures in Pandas?
9. Define Series in Pandas?
10. How can we calculate the standard deviation from the Series?
11. Define DataFrame in Pandas?
12. What are the significant features of the pandas Library?
13. Explain Reindexing in pandas?
14. What is the name of Pandas library tools used to create a scatter plot matrix?
15. Define the different ways a DataFrame can be created in pandas?
16. Explain Categorical data in Pandas?
17. How will you add a column to a pandas DataFrame?

Part – B

1. Describe the advanced vectorization in Numpy with examples.
2. Write a Python program to find the eigenvalues and eigenvectors of a square matrix.
3. Write a Python program to remove all duplicate elements from a given array and returns a new array.
4. Explain standard deviation and interquartile range and write python code to compute standard deviation and interquartile range.
5. Explain in details about aggregation and grouping using Pandas with suitable programs.
6. Discuss in detail about data manipulation using Pandas with suitable program.
7. Write a python program for summation and transpose of a matrix using numpy.