Programming with Python & Java (CS 29008)

Lab 2

Programming using Built-in Modules in Python

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)

Deemed to be University U/S 3 of UGC Act, 1956

School of Electronics Engineering
KIIT Deemed to be University
Bhubeneswar, Odisha

Contents

1 Introduction

Python is a popular and versatile programming language that can be used for various applications, such as web development, automation, machine learning, and data science. Python has a rich set of libraries or modules that provide useful functionalities and tools for working with data. In this lab manual, we will introduce some of the most commonly used Python modules for data science, such as Numpy, Pandas, matplotlib, and others. We will also demonstrate how to use these modules to perform basic data manipulation, analysis, and visualization tasks.

2 Numpy

Numpy (Numerical Python) is a module that provides support for **numerical computing** in Python. It offers a powerful **array** object that can store and manipulate multidimensional data efficiently. Numpy also provides various mathematical functions and operations that can be applied to arrays, such as linear algebra, statistics, trigonometry, etc. Numpy is the foundation for many other Python modules that deal with data, such as Pandas and Scipy. Please refer to `https://numpy.org/devdocs/index.html` for more details.

2.1 Numpy array creation and its attributes

The main object in NumPy is the homogeneous multidimensional array, which is a grid of elements (typically numbers) that have the same data type and are accessed by a pair of non-negative integers. NumPy's array class is called ndarray The array's dimensions are called axes in NumPy. For instance, the array [1,2,3] has one axis with three elements, while the array [[1,2,3],[4,5,7]] has two axes, where the first axis has two elements and the second axis has three elements. The important attributes of Numpy Array are shown in Table. 1.

```
>> import numpy as the alias np.
>> dir(np) # check all the methods in the numpy module
>> A = np.array([1,2,3]) # Creation of an Numpy Array (Vector)
>> type(A) # numpy.ndarray
>> A.size # 3 elements
>> A.shape # (3,)  #
>> A.ndim # 1
>> A.itemsize # 4
>> A.dtype # default dtype('int32')
# Note that if we put dot after each element , it changes to float.
# Change Data Type
```

Table 1: Important attributes of an ndarray object

Numpy Array Attribute	Function
ndarray.ndim	Number of axes (dimensions) of the array.
ndarray.shape	Shape of an array is a tuple of non-negative integers that specify the number of elements along each dimension.
ndarray.size	Total number of elements of the array
ndarray.dtype	An object describing the type of the elements in the array (numpy.int32, numpy.int16, and numpy.float64)
ndarray.itemsize	Size in bytes of each element of the array.

```
>> A = np.array([1,2,3],dtype='float32') # User defined datatype
# Altenatively astype() method
>> A.astype((np.float64)
# Creation of multidimensional Array
>> A = np.array([[1,2],[2,4]])
# or
>> B = np.array([(7.8, 9, 10), (45, 5, 6)])
# Alternatively
>> A = np.array(([1,2],[2,3]),ndmin=2)
# Creation of 3 dimensional Matrix
>> A = np.array([[[1,2,3],[2,3,4],[4,5,6]],
                 [[7,8,9],[12,13,14],[14,15,16]],
                 [[27,28,29],[22,23,24],[24,25,26]]])
# Data Types for Strings
>> A =np.array(["hello", "world citizen"])
>> A.dype # dtype('<U13') 13 characters
# Unique items in an array 7 Counts
>> A = np.array([11, 11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 14, 18,
                                    19, 20])
>> unique_values = np.unique(A)
>> print(unique_values)
>> unique_values, indices_list = np.unique(A, return_index=True)
>> print(indices_list) # an array of first index positions of unique
                                    values
>> unique_values, occurrence_count = np.unique(a, return_counts=True)
>> print(occurrence_count) #frequency count of unique values in a
                                    NumPy array
```

2.1.1 Creation of Numpy Array using functions

```
>> A = np.arange(3) # array([0, 1, 2])
>> A = np.arange(1,5) # start to Stop-1 \& array([1, 2, 3, 4])
>> A = np.arange(1,10,2) # start to Stop-1 with step \& array([1, 3,
                                    5, 7, 9])
```

```
>> A = np.linspace(4, 20, 6) #array([ 4. ,  7.2, 10.4, 13.6, 16.8, 20
                                 . ])
>> A = np.array(range(4)) #  array([0, 1, 2, 3])
>> z = np.array([i % 4 for i in range(14)]) # Using List
                                 Comprehension
>> A = np.zeros(4) # array([0., 0., 0., 0.])
>> A = np.zeros((3,4)) #array([[0., 0., 0., 0.],[0., 0., 0., 0.],[0.,
                                 0., 0., 0.]])
>> A = np.zeros((3,4,3)) #array([[0., 0., 0., 0.],[0., 0., 0., 0.],[0
                                 ., 0., 0., 0.]])
>> A = np.ones((2, 3, 4), dtype=np.int16)
>> A = np.empty((2, 3)) # Random elements
>> A = np.eye(3) #array([[1., 0., 0.], [0., 1., 0.],[0., 0., 1.]])
>> A = np.diag([1, 2, 3]) # array([[1, 0, 0],[0, 2, 0],[0, 0, 3]])
#Vandermonde matrix useful for Least square
>> A = np.vander(np.linspace(0, 2, 5), 2)
>> B =  np.vander([1, 2, 3, 4], 2)
>> C= np.vander((1, 2, 3, 4), 4)
# Random Matrix
>> rand_gen = np.random.default_rng(42) #instance of default random
                                 number generator
>> A=rand_gen.random((2,3)) # array([[0.77395605, 0.43887844, 0.
                                 85859792],[0.69736803, 0.09417735,
                                  0.97562235]])
```

2.2 Adding, removing, and sorting elements

```
>> A = np.array([2, 1, 5, 3, 7, 4, 6, 8])
>> np.sort(A) # other functions are argsort,lexsort,searchsorted \&
                                 Partition
>> A = np.array([1, 2, 3, 4])
>> B = np.array([5, 6, 7, 8])
>> print(np.concatenate((A, B))
>> x = np.array([[1, 2], [3, 4]])
>> y = np.array([[5, 6]])
#Concatenate along axis = 0
>> np.concatenate((x, y), axis=0)
```

2.3 Indexing and slicing

```
>> A = np.arange(5)**2 # array([ 0,  1,  4,  9, 16])
>> B = A[1:4] # array([1, 4, 9])
>> A[:5] # array([ 0,  1,  4,  9, 16])
>> A[:len(A)+1] #array([ 0,  1,  4,  9, 16])
>> A[0:]  # array([ 0,  1,  4,  9, 16])
>> A[:-1] #array([0, 1, 4, 9])
```

```
>> A[::-1]
>> A[0:5:2]
# Slicing 2D Numpy Array
>> A = np.array([[4,5,6],[7,8,9],[10,11,12]])
>> A[:,1] # array([ 5,  8, 11]) All rows 2nd column
>> A[1,:] # array([7, 8, 9]) All columns  2nd  row
>> A[0:3,0:2] #array([[ 4,  5], [ 7,  8],[10, 11]])
>> A[-1]   # the last row. Equivalent to A[-1, :]
```

2.4 Vector and Matrix calculus with Numpy Arrays

```
# Vector Calculus
>> A  = np.arange(0, 7)
>> B = np.arange(7, 0, -1)
>> print(A + B)
>> print(A * B) # Element wise Multiplication
>> print(A ** B) # Element wise exponentiation
>> print(np.dot(A,B)) # Dot product of two vectors
>> print(A.dot(B)) # Another Dot product of two vectors
# Matrix Calculus
>> A = np.array([[1, 1],
                 [0, 1]])
>> B = np.array([[2, 0],
                 [3, 4]])
>> A * B     # element-wise product
>> A @ B     # matrix product
>>> A.dot(B)  # another matrix product
# Iterator over all element in an array
>> for element in A.flat
        print(element)
# Inverse of Matrix
>> inv_A = np.linalg.inv(A) # Use linalg method
```

2.5 Array's Shape Manipulation and Stacking

```
# Create a random Matrix
>> rg=np.random.default_rng(42)
>> A = np.floor(10 * rg.random((3, 4)))
>> A.ravel()  # returns the array, flattened but a reference to the
                                   parent array (i.e., a "view")
# Or alternatively
>> A.flatten() # flatten, changes to your new array  w o n t  change
                                    the parent array.
>> A.reshape(6, 2)  # returns the array with a modified shape
>> A.T # returns the array, transposed
>> A = np.arange(30)
```

```
>> B = A.reshape((2, -1, 3))  # -1 means "whatever is needed"
>> B.shape # Its shape is (2, 5, 3) i.e  two 5 by 3 matrix
# The reshape function returns its argument with a modified shape,
#whereas the ndarray.resize method modifies the array itself.
>> A.resize(6,2)
# Stacking together different arrays
>> A = np.array([[2,3],[4,5]])
>> B = np.array([[7,8],[9,10]])
>> print(np.vstack((A, B))) # Vertical Stacking stacks along their
                                first axes
>> print(np.hstack((A, B))) # Horizontal Stacking stacks along their
                                second axes
# Column stacking by converting row into column vector & then stacks
                                along column
>> A = np.array([6, 7])
>> B = np.array([8, 9])
>> print(np.column_stack((A, B)))  # returns a 2D array
# Or
>> print(np.vstack((A.T,B.T)))
# Use of Newaxis & increasing the dimensions from 1D to 2D
>> A = np.array([6, 7])
>> A.shape # Its shape is (2,)
>> B = A[:, np.newaxis]  # view A as a 2D column vector
>> B.shape # Its shape is 2 by 1
>> B = A[np.newaxis,:]
>> B.shape # Its shape is 1 by 2
# Or Use np.expand_dims to add an axis at index position 1 with:
>> B = np.expand_dims(A, axis=1)
>> B.shape
# Fliping an Array
>> A = np.array([1, 2, 3, 4, 5, 6, 7, 8])
>> reversed_A = np.flip(A)
# For Matrix A
>> reversed_arr_rows = np.flip(A, axis=0)
>> reversed_arr_columns = np.flip(A, axis=1)
```

2.6 Save and load NumPy objects

The ndarray objects can be

- saved to and loaded from the disk files with loadtxt and savetxt functions that handle normal text files
- load and save functions that handle NumPy binary files with a .npy file extension, and a savez function that handles NumPy files with a .npz file extension. The .npy and .npz files store data, shape, dtype, and other information

```
>> A = np.array([1, 2, 3, 4, 5, 6])
>> np.save('filename', A) # Saving it as "filename.npy"
>> B = np.load('filename.npy')
# Save a NumPy array as a plain text file like a .csv or .txt file
                                    with np.savetxt
>> csv_arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
# savetxt() and loadtxt() functions accept additional optional
# parameters such as header, footer, and delimiter.
>> np.savetxt('new_file.csv', csv_arr)
>> np.loadtxt('new_file.csv')
```

2.7 Advanced Concepts in Numpy

2.7.1 Structured arrays in Numpy

Structured arrays are ndarrays whose datatype is a composition of simpler datatypes organized as a sequence of named fields. For example,

```
>> x = np.array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],
             dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f4')]
                                                    )
# Here x is a one-dimensional array of length two whose datatype is
#a structure with three fields:
# 1. A string of length 10 or less named 'name'
# 2. a 32-bit integer named 'age'
# 3. a 32-bit float named 'weight'
>> x[1] # ('Fido', 3, 27.)
>> x['age'] # array([9, 3], dtype=int32)
```

2.7.2 Advanced indexing in Numpy

```
>> A = np.arange(12)**2  # the first 12 square numbers
>> i = np.array([1, 1, 3, 8, 5])  # an array of indices
>> A[i]  # the elements of A at the positions i
# Output - array([ 1,  1,  9, 64, 25])
>> j = np.array([[3, 4], [9, 7]])  # A bidimensional array of indices
>> A[j]  # the same shape as `j`
#  Output - array([[ 9, 16],[81, 49]])
>> A = np.arange(12).reshape(3, 4)
>> i = np.array([[0, 1],[1, 2]]) # indices for the first dim of A
>> j = np.array([[2, 1],[3, 3]]) # indices for the second dim of A
>> a[i, j]  # i and j must have equal shape
# Its is same as A[l] where l = (i,j)
>> a[i, 2]
>> a[:, j]
```

List of some useful NumPy functions and their corresponding categories have been listed in Table 2.

Table 2: List of some useful NumPy functions and methods names ordered in categories.

Category	Functions
Array Creation	arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r_, zeros, zeros_like
Conversions	ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat
Manipulations	array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack
Questions	all, any, nonzero, where
Ordering	argmax, argmin, argsort, max, min, ptp, searchsorted, sort
Operations	choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum
Basic Statistics	cov, mean, std, var
Basic Linear Algebra	cross, dot, outer, linalg.svd, vdot

3 Pandas

Pandas is a module that provides high-level data structures and tools for **data analysis** in Python. It offers two main objects: "Series" and "DataFrame" which are delineated in in Table 3. Pandas also provides various methods and functions that can be used to manipulate, filter, group, aggregate, and transform data. Pandas is widely used for data exploration, cleaning, and processing & its functionality can be accessed from `https://pandas.pydata.org/`.

Table 3: Data structures in Pandas and their analogy. Source :*Harrison, Matt. Effective Pandas: Patterns for Data Manipulation. Matt Harrison, 2021.*

Data Structures	Dimensionality	Spreadsheet Analog	Database Analog	Linear Algebra
Series	1 D	Column	Column	Column Vector
DataFrame	2 D	Single Sheet	Table	Matrix

3.1 Data Structures in Pandas

The Fig. 1 depicts the difference between series and DataFrame.

DataFrame

Axis 1

Axis 0

Index	Age	Height	Weight	Married
0	25	160	60	NO
1	24	170	80	No
2	23	165	65	Yes
3	28	180	90	Yes

Series

Index	Age
0	25
1	24
2	23
3	28

Index	Height
0	160
1	170
2	165
3	180

Figure 1: Data structures in Pandas

3.1.1 Series

A Series is a one-dimensional array that can store any type of data, such as numbers, strings, or booleans.

```
# Basic Example
>> import pandas as pd
>> S = pd.Series([6, 9, 10, -3])
>> print(S)
>> S.dtypes
>> type(S) # pandas.core.series.Series
>> S.array # <PandasArray>[6, 9, 10, -3]Length: 4, dtype: int64
>> S.values  # Numpy Array - array([ 6,  9, 10, -3])
>> S.index # RangeIndex(start=0, stop=4, step=1)

#--- Creation of Series using function
>> S = pd.Series(range(10))
>> S =  pd.Series(range(3,10,2))

#--------------- Creation using Other data type ------------
# Dictionary to Pandas Series
>> A = {"Delhi": 45000, "Mumbai": 75000, "Kolkata": 27000, "Banguluru
                                    ": 80000}
>> S = pd.Series(A)
# Back to a dictionary
>> B = S.to_dict()
```

```
#--  List to Pandas Series
>> A = list(range(2,10,2))
>> S = pd.Series(A)
# Back to a List
>> B = S.to_list()

#--  Numpy Array to Pandas Series
>> A = np.array([1, 2])
>> S = pd.Series(A)
# Back to a Numpy Array
>> B = S.values

# ---- Change index annotations--------
>> S = pd.Series([6, 9, 10, -3], index = ["d", "b", "a", "c"])
# Alternatively
>> indices = ["d", "b", "a", "c"]
>> S = pd.Series([6, 9, 10, -3], index = indices)

#-- Accessing values using index labels -----------
>> S["d"]
>> S[["d","c","b"]
>> S[["d","c","b"]].values # array([ 6, -3,  9], dtype=int64)
# Using iloc method
>> S = pd.Series(range(1,10))
>> S.iloc[5]

#--  Filtering with a Boolean array \& Scalar multiplication
>> S[S > 0]
>> S **2

# -- Using Functions to Manipulate ----------
>> import numpy as np
>> np.exp(A)

# ----------- Locating  "missing," "NA," or "null" values
>> pd.isna(S) # output is boolean values
# Or Alternatively
>> S.isna()
>> pd.notna(S) # Checking Not NA Values

# Dropping a Rows
>> df = pd.Series(np.arange(7.), index=["a", "b", "c", "d", "e","f","
                                        g"])
>> df.drop("a") or df.drop(["a"]
>> df.drop(["a","c"])
>> df.drop(index = ["a","c"])

# Filtering Missing Data
>> data = pd.Series([1, np.nan, 3.5, np.nan, 7])
```

```
>> data.dropna()
# or
>> data[data.notna()]

# Arithmetic
>> s1 = pd.Series([6, 7, 8, 9], index=["a", "b", "c", "d"])
>> s2 = pd.Series([16, 17, 8, 11], index=["a", "b", "c", "d"])
>> s1+s2 # Output Only if columns labels are same other wise NaN
                                    values
```

3.1.2 DataFrame

A DataFrame is a two-dimensional table that can store heterogeneous data in rows and columns.

```
# --------------- Creation of Data Frame----------------
# Using List of Lists
>> df =pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]])
# Using Methods
>> df = pd.DataFrame(np.arange(25).reshape((5, 5)))
>> df = pd.DataFrame(np.arange(25).reshape((5, 5)),columns=["one", "
                                    two", "three", "four","five"])
>> df = pd.DataFrame(np.arange(25).reshape((5, 5)),index=["Rohan", "
                                    Mohan", "shyam", "Peter","John"],
                                    columns=["one", "two", "three", "
                                    four","five"])
# Using Dictionary
>> data = {"City": ["Delhi", "Kolkata", "Mumbai", "Chennai", "
                                    Bengaluru", "Gurgaon"],
"year": [2021, 2021, 2021, 2021, 2021, 2020],
"pop": [30, 15, 21, 11, 12, 3.2]}
>> df = pd.DataFrame(data)
# Using List of Dictionary
>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
            {'a': 100, 'b': 200, 'c': 300, 'd': 400},
            {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>> df.dtype


# ------- Rearrangement of columns
>> df = pd.DataFrame(data, columns=["year", "state", "pop"])
# Passing a columns with no data
>> df2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])
#Axis Indexes with Duplicate Labels
>> df = pd.DataFrame(np.random.standard_normal((5, 3)),index=["a", "a
                                    ", "b", "b", "c"])
>> df.index.is_unique
```

```
#-------Descriptive Statistics---------------------
# sum,mean,cumulative sum (cumsum)
>> df = pd.DataFrame([[3, np.nan],[8, -4.8],[np.nan, np.nan], [2.75,
                                    -3.5]],index=["a", "b", "c", "d"],
                                    columns=["one", "two"])
# It skips the NaN values
>> df.sum() # sum along rows or
>> df.sum(axis=1)
or
>> df.sum(axis="columns")
>> df.mean()
>> df.cumsum()

# Do not skip nan values
>> df.sum(axis="index", skipna=False)
>> df.idxmax()

# Compute a histogram on multiple related columns in a DataFrame
>> df = pd.DataFrame({"A": [1, 3, 4, 3, 4],"B": [2, 3, 1, 2, 3],"C":
                                    [1, 5, 2, 4, 4]})

# compute the value counts for a single column
>> result = df.apply(pd.value_counts).fillna(0)

# --------------- Conversion-----------
# Numpy to Dataframe
>> A =np.array(range(25)).reshape(5,5)
>> df=pd.DataFrame(A)
# Dataframe to Numpy
>> df = pd.DataFrame(np.arange(25).reshape((5, 5)),index=["Rohan", "
                                    Mohan", "shyam", "Peter","John"],
                                    columns=["one", "two", "three", "
                                    four","five"])
>> df.to_numpy()
# or
>> df.values
# Dataframe to dictionary
>> df = pd.DataFrame(np.arange(25).reshape((5, 5)),index=["Rohan", "
                                    Mohan", "shyam", "Peter","John"],
                                    columns=["one", "two", "three", "
                                    four","five"])
>> df.to_dict()

# ---------------- Accessing the columns-------------------
>> data = {"City": ["Delhi", "Kolkata", "Mumbai", "Chennai", "
                                    Bengaluru", "Gurgaon"],
"year": [2021, 2021, 2021, 2021, 2021, 2020],
"pop": [30, 15, 21, 11, 12, 3.2]}
>> df = pd.DataFrame(data)
>> df.columns # Index(['City', 'year', 'pop'], dtype='object')
```

```
# Accessing a particular column or Multiple columns
>> df["city"]
# by using the dot attribute notation
>> df.city # works only when the column name is a valid
# Accessing Multiple Columns
>> df[['City',"pop"]]
# Accessing using integer indexing
>>  data[:2]

# Accessing DataFrame with loc (label-based) and iloc (integer-based)
>> df = pd.DataFrame(np.arange(25).reshape((5, 5)),index=["Rohan", "
                                    Mohan", "shyam", "Peter","John"],
                                    columns=["one", "two", "three", "
                                    four","five"])
>> df.loc["Rohan"] #1st Row
>> df.iloc[0] # 1st Row
>> df.loc[["Rohan","Mohan"]] # Selecting two rows
>> df.iloc[[0,2]] # Selecting two rows
# combine both row and column selection
>> df.loc['Rohan', 'two']
>> df.loc["Mohan", ["two", "three"]]
>> df.iloc[2, [3, 0, 1]]
>> df.iloc[[0, 2], [1, 3]]
>> df.iloc[1:3, 0:3]
>> df.iloc[[1, 2], [3, 0, 1]]
>> df.iloc[:, :3][df.three > 5]
>> df.loc[df.three >= 2]
#  indexing functions  with slices
>> df.loc[:"Shyam","two"]
# Accessing using functions
>> df.iloc[lambda x: x.index % 2 == 0]
>> df[df["three"] > 5]
>>


# Modified the columns or rows by assignment.
>> df["one"]=5
>> df[["one","two"]]=5
>> df[["one","two"]]= (5,6)
>> df["one"] = np.arange(5.)
# Adding extra columns with some values
>> df["debt"]=np.arange(5,) # Debt column is added
# Passing a series to Dataframe
>> val = pd.Series([4, 5, 7], index=["Rohan", "Mohan", "John"])
>>df["debt"]=val  # Some values are NaN
# deleting a particular column
>> del df["debt"]

# ------------Sorting and Ranking -----------
# DataFrame.sort_index() sorts by an axis:
```

```
>> df.sort_index() # # Default Ascending order for index in rows
>> df.sort_index(ascending=False) # Descending order
>> df.sort_index(axis=1) # Sorting along index of columns
or
>> df.sort_index(axis="columns")
# DataFrame.sort_values() sorts by values:
>> df.sort_values(by="B")

# ----- Viewing the DataFrame---------------------
>> df.head(), # the head method selects the first five rows
>> df.tail()  # tail returns the last five rows:
>> df.describe() # Complete statistical information


# ------ Dropping Entries from a data frame------------
>> df1 = pd.Series(np.arange(7.), index=["a", "b", "c", "d", "e","f",
                                        "g"])
>> df2 = pd.DataFrame(np.arange(25).reshape((5, 5)),index=["Rohan", "
                                       Mohan", "shyam", "Peter","John"],
                                       columns=["one", "two", "three", "
                                       four","five"])
>> df2.drop(columns=["three","four"])
or
>> df.drop("two", axis=1)
or
>> df.drop("two", axis="columns")
# Using dropna method to drop NA values
>> df = pd.DataFrame([[7, 7.6, 5.8], [6, np.nan, np.nan],[np.nan, np.
                                       nan, np.nan], [np.nan, 7.8, 4.]])
>> df.dropna()
# drop only rows that are all NA
>> data.dropna(how="all")


# ---------- Filling In Missing Data---------
>> df.fillna(0) #replaces missing values with 0
>> df.fillna({1: 0.5, 2: 0}) # Replaces values in column 1 \& 2
>> df.fillna(method="ffill") # use interpolation to fill
>> df.fillna(df.mean()) # use the mean of non-nan values in column


# Arithmetic and Data Alignment
>> df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("
                                       bcd"),index=["Indore", "Rohtak", "
                                       Kasol"])
>> df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("
                                       bcd"),index=["Panipat", "
                                       Kurukshetra", "Indore", "Rohtak"])
>> df1+df2
>> df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("
```

```
                                    bcd"),index=["Indore", "Rohtak", "
                                    Kasol"])
>> df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("
                                    bde"),index=["Indore", "Rohtak", "
                                    Kasol","Panipat"])
>> df1+df2

# Arithmetic methods with fill values
>> df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("
                                    bcd"),index=["Indore", "Rohtak", "
                                    Kasol"])
>> df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("
                                    bcd"),index=["Panipat", "
                                    Kurukshetra", "Indore", "Rohtak"])
>> df =df1+df2
>> df[df.isna()]=0
or
# Using the add method on df1, I pass df2 and an argument to fill
                                    value
>>  df1.add(df2, fill_value=0)
#  reindexing a Series or DataFrame & Fill value
>> df1.reindex(columns=df2.columns, fill_value=0)
# Division
>> 1 / df1 or df1.rdiv(1) # r is for reversed


# Data Transformation
>> df= pd.DataFrame({"t1": ["five", "six"] * 3 + ["six"],"t2": [1, 1,
                                     2, 3, 3, 4, 4]})
>> df.duplicated() # Return Bool whether each row is a duplicate
>> df.drop_duplicates() # Drop duplicates row
# Add another column
>> df["v"] = range(7)
# Dropping values by subset of columns
>> df.drop_duplicates(subset=["t1"])
# Keep Last means keep only last duplicate else default is first
>> df.drop_duplicates(["t1", "t2"], keep="last")

# Transforming Data Using a Function or Mapping
>> df = pd.DataFrame({"food": ["bacon", "pulled pork", "bacon","
                                    pastrami", "corned beef", "bacon",
                                    "pastrami", "honey ham", "nova lox
                                    "],"ounces": [4, 3, 12, 6, 7.5, 8,
                                     3, 5, 6]})

df = pd.DataFrame({"food": ["Matar Paneer", "Aloo Dam", "Matar Paneer
                                    ","pasta", "Daal", "Matar Paneer",
                                    "pasta", "honey chilly potato", "
                                    rajma"],"plate": [4, 3, 12, 6, 7.5
                                    , 8, 3, 5, 6]})
```

```
# Add a column indicating the type of animal
>> party_list = {
 "Matar Paneer": "mohan",
 "Aloo Dam": "mohan",
 "pasta": "sohan",
 "Daal": "sohan",
 "honey chilly potato": "mohan",
 "rajma": "timmy"}
>>  df["party"] = data["food"].map(party_list)
# Or
>> def get_list(x):
        return party_list[x]
>> data["food"].map(get_list)

# Replacing Value
>> df = pd.Series([1., -222., 2., -222., -222., 4.])
>> df.replace(-999, np.nan)
>> df.replace([-999, -1000], np.nan)
>> df.replace([-999, -1000], [np.nan, 0])
>> df.replace({-999: np.nan, -1000: 0})

# Discretization & Binning
>> ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
>> bins = [18, 25, 35, 60, 100]
>> age_categories = pd.cut(ages, bins)
>> pd.cut(ages, bins, right=False) # Open the right interval
>> age_categories.codes # Bin number of each element
>> age_categories.categories # bin Interval
>> pd.value_counts(age_categories)
>> group_names = ["Youth", "YoungAdult", "MiddleAged", "Senior"]
>> pd.cut(ages, bins, labels=group_names)
# Automated Discretization & Binning
>> df = np.random.uniform(size=20)
>> pd.cut(data, 4, precision=2)
>> df = np.random.standard_normal(1000)
>> quartiles = pd.qcut(df, 4, precision=2)
>> pd.value_counts(quartiles)
>> pd.qcut(df, [0, 0.1, 0.5, 0.9, 1.]).value_counts()

# Detecting and Filtering Outliers
>> df = pd.DataFrame(np.random.standard_normal((1000, 4)))
>> col = df[2] # Select columns
>> col[col.abs() > 3]
# Select all rows having a value exceeding 3 or -3
>> df[(df.abs() > 3).any(axis="columns")]
>> df[df.abs() > 3] = np.sign(df) * 3

# Permutation and Random Sampling
>> df = pd.DataFrame(np.arange(5 * 7).reshape((5, 7)))
```

```
>> sampler = np.random.permutation(5)
>> df.take(sampler)
>> df.iloc[sampler]
# Random sampling along columns
>> sampler = np.random.permutation(5)
>> df.take(column_sampler, axis="columns")
# To select a random subset without replacement
>> df.sample(n=3)
# To select a random subset with replacement
>> df = pd.Series([5, 7, -1, 6, 4])
>> df.sample(n=10, replace=True)
```

3.2 Input/Output file handling

3.2.1 Text and binary data loading functions in pandas

The various functions in Pandas are listed in Table 4.

Table 4: Functions used for file handling in pandas

Function	Function Description
read_csv	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
read_fwf	Read data in fixed-width column format (i.e., no delimiters)
read_clipboard	Variation of read_csv that reads data from the clipboard; useful for converting tables from web pages
read_excel	Read tabular data from an Excel XLS or XLSX file
read_hdf	Read HDF5 files written by pandas
read_html	Read all tables found in the given HTML document
read_json	Read data from a JSON (JavaScript Object Notation) string representation, file, URL, or file-like object
read_feather	Read the Feather binary file format
read_orc	Read the Apache ORC binary file format
read_parquet	Read the Apache Parquet binary file format
read_pickle	Read an object stored by pandas using the Python pickle format
read_sas	Read a SAS dataset stored in one of the SAS system's custom storage formats
read_spss	Read a data file created by SPSS
read_sql	Read the results of a SQL query (using SQLAlchemy)
read_sql_table	Read a whole SQL table (using SQLAlchemy); equivalent to using a query that selects everything in taht table using read_sql
read_stata	Read a dataset from Stata file format
read_xml	Read a table of data from an XML file

```
>> df = pd.read_csv("E:\Python Manual\Test_data.csv")
>> df = pd.read_csv("E:\Python Manual\Test_data.csv",header=None)
# Assigning column names
>> df =pd.read_csv("E:\Python Manual\Test_data.csv",names=["Name","
                                    Height","Weight","Age"])
# skipping the rows
>> df =pd.read_csv("E:\Python Manual\Test_data.csv",names=["Name","
                                    Height","Weight","Age"],skiprows=[
                                    0,2,3])
 # Note: ``index_col=False`` can be used to force pandas to *not* use
                                      the first
 # column as the index, e.g. when you have a malformed file with
                                     delimiters at
 # the end of each line.
```

```
>> df = pd.read_csv("E:\Python Manual\Test_data.csv", names=names,
                                    index_col="id")
# Checking Missing values
>> pd.isna(df) # Return boolean where values are missing
#-------- Customized Missing values
# Use of  na_values
>> df = pd.read_csv("E:\Python Manual\Test_data.csv", na_values=["
                                    NULL"])
# Keeping the disabled NA values in pandas with the keep_default_na
>> df2 = pd.read_csv("Test_data.csv", keep_default_na=False)
>> pd.isna(df2)
# Use of NA sentinels passing as a dictionary
>> sentinels = {"Age": [28, 24], "Name": ["Timmy"]}
>> df = pd.read_csv("E:\Python Manual\Test_data.csv",names=["Name","
                                    Height","Weight","Age"],
na_values=sentinels,keep_default_na=False)

# Writing Data
>> df= pd.read_csv("Test_data.csv")
>> df.to_csv("out.csv")
>> df.to_csv("out.csv", index=False, columns=["a", "b", "c"])
```

3.3 Advanced Concept in Pandas

3.3.1 Creation of Pandas Data structure

```
# Nested dictionary of dictionaries
>> pops = {"Mumbai": {2010: 4, 2011: 5, 2012: 6},"Indore": {2010: 4,
                                    2012: 7}}
>> df = pd.DataFrame(pops)
# Outer key values as columns while inner key values as index of rows
```

3.3.2 Function Mapping

```
>> df = pd.DataFrame(np.random.standard_normal((4, 3)),columns=list("
                                    bde"),index=["Utah", "Ohio", "
                                    Texas", "Oregon"])
>> np.abs(df)
# X is series
>> def f1(x):
    return x.max() - x.min()
>> df.apply(f1)
>> df.apply(f1, axis="columns")

>> def f2(x):
    return pd.Series([x.min(), x.max()], index=["min", "max"])
```

```
>> df.apply(f2)
>> def my_format(x):
     return f"{x:.2f}
>> df.applymap(my_format)

\subsubsection{Computing Indicator/Dummy Variables}
#converting a categorical variable into a dummy or indicator matrix.
#If a column in a DataFrame has k distinct values, you would derive a
#matrix or DataFrame with k columns containing all 1s and 0s.
# pandas has a pandas.get_dummies function
>> df = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"], "data1":
                                    range(6)})
# Add prefix to output columns
>> dummies = pd.get_dummies(df["key"], prefix="key")
>> df_with_dummy = df[["data1"]].join(dummies)
```

4 Matplotlib

Matplotlib is a module that provides "data visualization" capabilities in Python. It offers a low-level interface that allows users to create and customize various types of plots, such as line plots, bar charts, scatter plots, histograms, pie charts, etc. Matplotlib also supports interactive features, such as zooming, panning, and saving figures. matplotlib is the most widely used plotting library in Python, and it can be integrated with other modules, such as Numpy and Pandas, to create powerful and informative visualizations.

Importance and Usage

- Visualizing data trends and patterns.
- Exploring relationships between variables.
- Communicating insights effectively through graphical representation.
- Matplotlib is widely used in fields like data science, machine learning, finance, and more.

Objectives

- Basic plotting functions: plot, scatter, bar, hist, pie.
- Customization options: colors, styles, labels, annotations.
- Advanced features: subplots, axes customization, saving plots.

TO use funcionality of Matplot Library

```
>> import matplotlib.pyplot as plt
```

4.1 Plotting a Simple Line Graph

To create a basic line plot, you can use the plot() function.

```
# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
# Plotting
plt.plot(x, y)
plt.show()
```

4.2 Customizing Plot Appearance

You can customize the appearance of your plot using various parameters like color, line style, and marker.

4.2.1 Adding Titles and Labels

It's essential to label your plot properly for clarity. Use xlabel(), ylabel(), and title() functions for this purpose.

```
# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
plt.plot(x, y, color='red', linestyle='--', marker='o', markersize=8)
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Customized Line Plot')
plt.grid(True)
plt.show()
```

4.2.2 Saving Plots to Files

You can save your plot as an image file using the savefig() function.

```
plt.savefig('plot.png')
```

4.2.3 Basic Types of Plots

Matplotlib supports various types of plots for different kinds of data visualization. **Scatter Plot**:To create a scatter plot, use the scatter() function.

```
plt.scatter(x, y, color='blue', marker='x')
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
```

```
plt.title('Scatter Plot')
plt.show()
```

Bar Plot: For creating a bar plot, use the bar() function.

```
categories = ['A', 'B', 'C', 'D']
values = [10, 20, 15, 25]
plt.bar(categories, values, color='green')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot')
plt.show()
```

Histogram:For generating a histogram, use the hist() function.

```
data = [1, 1, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 6, 7, 7, 7, 8, 9, 9]
plt.hist(data, bins=5, color='orange')
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram')
plt.show()
```

Pie Chart: To create a pie chart, use the pie() function.

```
sizes = [30, 20, 25, 15, 10]
labels = ['A', 'B', 'C', 'D', 'E']
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=['red', '
                                    green', 'blue', 'orange', 'purple'
                                    ])
plt.title('Pie Chart')
plt.show()
```

Adding Multiple Plots to a Figure: You can create multiple plots within the same figure using subplots().

```
sizes = [30, 20, 25, 15, 10]
labels = ['A', 'B', 'C', 'D', 'E']
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=['red', '
                                    green', 'blue', 'orange', 'purple'
                                    ])
plt.title('Pie Chart')
plt.show()
```

```
fig, ax = plt.subplots(2, 2)
ax[0, 0].plot(x, y)
ax[0, 1].scatter(x, y)
ax[1, 0].bar(categories, values)
ax[1, 1].pie(sizes, labels=labels, autopct='%1.1f%%')
plt.show()
```

4.3 Use of Axis Method

```
# Plots in matplotlib reside within a Figure object.
>> fig = plt.figure() #
# plt.figure has a number of options; notably, figsize
# use of Axis Methods
>> ax1 = fig.add_subplot(2, 2, 1)
>> ax2 = fig.add_subplot(2, 2, 2)
>> ax3 = fig.add_subplot(2, 2, 3)
>> ax3.plot(np.random.standard_normal(50).cumsum(), color="black",
                                    linestyle="dashed")
>> ax1.hist(np.random.standard_normal(100), bins=20, color="black",
                                    alpha=0.3)
>> ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.
                                    standard_normal(30))
```

Setting the title, axis labels, ticks, and tick labels

```
fig, ax = plt.subplots()
ax.plot(np.random.standard_normal(1000).cumsum())
ticks = ax.set_xticks([0, 250, 500, 750, 1000])
labels = ax.set_xticklabels(["one", "two", "three", "four", "five"],
                                    rotation=30, fontsize=8)
ax.set_xlabel("Stages")
ax.set_title("My first matplotlib plot")
ax.set(title="My first matplotlib plot", xlabel="Stages")
# Adding Legends
fig = plt.figure()
ax = fig.add_subplot()
data = np.random.standard_normal(30).cumsum()
ax.plot(data, color="black", linestyle="dashed", label="Default");
ax.plot(data, color="black", linestyle="dashed",drawstyle="steps-post
                                    ", label="steps-post");
ax.legend()
```

Lab 2 Exercises

Lab Objectives
The students will be able to

- Understand the difference between one-, two- and n-dimensional arrays in NumPy
- Understand how to apply some linear algebra operations to n-dimensional arrays without using for-loops
- Understand axis and shape properties for n-dimensional arrays.
- acquire proficiency in data manipulation, analysis, and visualization using pandas and matplotlib.
- Understand techniques for cleaning and preprocessing data, integrating multiple data sources, and conducting statistical analysis using pandas module.

Lab 2 Assignments Questions

1. Generate the following matrix in numpy

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{1}$$

   and modify the entries to

$$B = \begin{bmatrix} 1 & 999 & 999 \\ 4 & 999 & 999 \\ 7 & 8 & 9 \end{bmatrix} \tag{2}$$

2. Consider the system of equations

$$\begin{aligned} 3x + 2y &= 5 \\ 10x - y &= 3 \end{aligned} \tag{3}$$

   Find the unique solution to these equations using matrix calculus in Numpy array.

3. Generate a 5 by 5 random matrix and find the maximum, minimum, average (mean) & variance values along all axes.

4. Generate a 5 by 5 random integer matrix whose values are between 2 & 100. Find the another matrix which produces boolean output as true where Values of random matrix are between 20 & 70.

5. Find the mean square error between two array A = [1,2,3,4,5,6] & B = [7,8,9,10,12,17] using function in Numpy module.

6. Create a pandas data structure (series) X using a dictionary D = "Rohan": 176,"John": 175,"Harvinder": 180,"Hasan": 174 . Then pass other list ["Tinku","John","Ramesh","Hasan"] as index into series X . Comment on the resultant output & locate the index of NAN values.

7. Create a Normalized random variable of size 20 using Numpy and find out each quartile & their corresponding counts.

8. Draw four line plots using the subplot() function.
   Use the following data:
   For line 1: x = [0, 1, 2, 3, 4, 5] and y = [0, 100, 200, 300, 400, 500]
   For line 2: x = [0, 1, 2, 3, 4, 5] and y = [50, 20, 40, 20, 60, 70]
   For line 3: x = [0, 1, 2, 3, 4, 5] and y = [10, 20, 30, 40, 50, 60]
   For line 4: x = [0, 1, 2, 3, 4, 5] and y = [200, 350, 250, 550, 450, 150]

9. Using the following data plot a bar plot and a horizontal bar plot.
   company = ["Apple", "Microsoft", "Google", "AMD"]
   profit = [3000, 8000, 1000, 10000]

10. Using the following data to plot a box plot.
    import numpy as np
    box1 = np.random.normal(100, 10, 200)
    box2 = np.random.normal(90, 20, 200)