NumPy

export & import ndarray

```
import os # it allows you to work with files and directories, and other OS resources
import numpy as np # importing NumPy library (Numerical Python) to do
# Numerical operations using nd arrays
```

os.getcwd()

```
os.getcwd()
# shows current workng directory
```

```
'/content'
```

files.upload() - Google Colab

```
# from google.colab import files
# uploaded=files.upload()
# Students.xlsx
# to be used with google colab

# os.chdir(r'D:/advanced-analytics-files/day02')
# to change current working directory to specified path
# to be used while running on local system
```

os.chdir()

```
# os.chdir(r'D:/advanced-analytics-files/day02')
# to change current working directory to specified path
```

```
x = np.array([3, 5, 4, 5, 7 ,6, 8, 9, 7, 2, 1, 4])
# creating ndarray from list
x
```

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

```
arr2d = x.reshape(4, 3)
# reshaping 1-D ndarray to 2-D ndarray by specifying (r,c) to reshape
arr2d
```

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

np.save(fileName, ndarray)

```
np.save('myarray', arr2d)
# exports array2d into myarray.npy to cwd / file system
```

np.load('fileName.npy')

```
newarray = np.load('myarray.npy')
# imports myarray.npy to ndarray
newarray
```

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

```
os.getcwd()
```

```
'/content'
```

```
if newarray in arr2d:
    print("True")
# to check if exported ndarray and imported ndarray are same
```

```
True
```

Matrix multiplication

m1's column = m2's column and resultant matrix

- For multiplying two matrices, the `number of columns of m1 = number of rows of m2`
- `order of resultant matrix` = `no. of rows of m1` X `no. of columns of m2`

```
x = np.array([[3, 7, 5], [1, 2, 3]])
# creating 2-D ndarray to represent 2rx3c matrix
x
```

```
array([[3, 7, 5],
       [1, 2, 3]])
```

```
y = np.array([[1, 7], [2, 4], [8, 1]])
# creating 2-D ndarray to represent 3rx2c matrix
y
```

```
array([[1, 7],
       [2, 4],
       [8, 1]])
```

ndarray1.dot(ndarray2)

```
x.dot(y)
# matrix multiplication using dot() method
```

```
array([[57, 54],
       [29, 18]])
```

np.matmul(ndarray1, ndarray2)

```
np.matmul(x, y)
# matrix multiplication using matmul(method)
```

```
array([[57, 54],
       [29, 18]])
```

numpy.linalg package

- the NumPy linalg module is a part of the NumPy package and set of functions for linear algebra operations
- it is used for performing various mathematical operations, such as matrix multiplication, matrix inversion, eigenvalues, and eigenvectors computations, solving linear systems, etc.

```
from numpy import linalg
from numpy.linalg import inv
```

```
#  x +  y +  z =   6
#      2y + 5z =  -4
# 2x + 5y -  z =  27
```

```
A = np.array([[1, 1, 1], [0, 2, 5], [2, 5, -1]])
# creating ndarray to represent LHS of equations as matrix 'A'
A
```

```
array([[ 1,  1,  1],
       [ 0,  2,  5],
       [ 2,  5, -1]])
```

```
B = np.array([6, -4, 27])
# creating ndarray to represent RHS of equations as matrix
# product/result matrix B in horizontal ndarray
B
```

```
array([ 6, -4, 27])
```

```
B = B.reshape(3, 1)
# reshaping product matrix as vertical ndarray to calculate
# upon matrix A & Ainv
# product matrix in actual shape
B
```

```
array([[ 6],
       [-4],
       [27]])
```

```
Ainv = np.linalg.inv(A)
# calculating multiplicative inverse matrix 'Ainv' for matrix 'A'
Ainv
```

```
array([[ 1.28571429, -0.28571429, -0.14285714],
       [-0.47619048,  0.14285714,  0.23809524],
       [ 0.19047619,  0.14285714, -0.0952381 ]])
```

```
Ainv.dot(B)
# using property to find variable matrix
#    A.X = B
```

```
# => X = B.(1/A)
# => X=B.Ainv
```

```
array([[ 5.],
       [ 3.],
       [-2.]])
```

`random.rand()` & `random.randint()`

```
from numpy import random
```

```
x = np.random.rand(10)
# generates ndarray of specified shape with
# random float values between 0-1 by default
x
```

```
array([0.04681776, 0.30342902, 0.61709683, 0.4835337 , 0.18198171,
       0.22964861, 0.97829009, 0.54960681, 0.4795286 , 0.88730672])
```

```
y = np.round(x, 3)
# rounds off each element of ndarray to 3 decimal value
y
```

```
array([0.047, 0.303, 0.617, 0.484, 0.182, 0.23 , 0.978, 0.55 , 0.48 ,
       0.887])
```

```
np.random.randint(20, 50)
# generates ndarray with random int value between specified values inclusively
# with size=1, means it generates only one value
```

```
34
```

```
np.random.randint(20, 50, 5)
# generates ndarray with random int value between 20-50 inclusively
# with size=5, means it will generate five values
```

```
array([20, 30, 35, 39, 33])
```

```
x = np.random.rand(5)
# generates ndarray with 5 random float values between 0-1
x
```

```
array([0.13353443, 0.72190237, 0.84794376, 0.94348106, 0.03195847])
```

```
y = np.round(x, 3)
# rounding those 5 floating values to three decimals in ndarray
y
```

```
array([0.134, 0.722, 0.848, 0.943, 0.032])
```

```
z = np.random.randint(20, 50, 5)
# grnerates nd array of 5 integer values between 20-50 inclusively
z + y
# print addition of ndarrays with floating ndarray & int ndarray
# as another ndarray
```

```
array([34.134, 41.722, 34.848, 22.943, 42.032])
```

OpenPyXL

- openPyXL is a python library that allows you to work with Excel files
- it provides functionality to R/W, Create, modify & save Excel Files using python

reading from xlsx

```
import openpyxl as op
import os
# importing OpenPyXl library to work on excel files
```

```
os.getcwd()
# shows current workng directory
```

```
'/content'
```

```
from google.colab import files
uploaded=files.upload()
# Students.xlsx
# to be used with google colab

# os.chdir(r'D:/advanced-analytics-files/day02')
# to change current working directory to specified path
# to be used while running on local system
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

Saving Students.xlsx to Students.xlsx

op.load_workbook(file.xlsx)

```
wb = op.load_workbook('Students.xlsx')
# to load a work book using openPyXL into a workbook object
# workbook is now accessible using object wb
```

```
type(wb)
# type of loaded workbook is Workbook
```

```
openpyxl.workbook.workbook.Workbook
```

wb['WorkSheetName']

```
sht1 = wb['Kharghar']
# creating a sheet object by
# acessing sheet name as index in workbook object
```

```
type(sht1)
# type of loaded sheet is Worksheet
```

```
openpyxl.worksheet.worksheet.Worksheet
```

workSheet[cellName].value

```
x = sht1['A3'].value
x
# printing value of cell using sheet object referring with cell name
```

```
'Rakesh'
```

WorkSheet.cell(r,c).value

```
sht1.cell(3,1).value
# printing value of cell using cell method
```

```
'Rakesh'
```

WorkSheet.max_row

```
sht1.max_row
```

```
5
```

WorkSheet.max_column

```
sht1.max_column
```

```
3
```

```
students = []
# creating empty list to store student names read from excel file
type(students)
```

```
list
```

```
len(students)
```

```
0
```

```
for ctr in range(2, 6):
    x = sht1.cell(ctr, 1).value
    # reading value from cells using sheet.cell(r.c).value method
    students.append(x)
    # appending read value to students list

# student names to be read from col#1
# reading of data should start from row#2 and finish at row#5
students
# printing values being read from column#1
```

```
['Sudeep', 'Rakesh', 'Dhruv', 'Ruchika']
```

```
print(x)
# printing the last read value from student column in excel
```

```
Ruchika
```

```
marks = []
# creating empty list to store student names read from excel file
```

```
for ctr in range(2, 6):
    x = sht1.cell(ctr, 3).value
    # reading value from cells using sheet.cell(r.c).value method
    marks.append(x)
    # appending read value to students list

# marks names to be read from col#3
# reading of data should start from row#2 and finish at row#5
marks
# creating empty list to store student names read from excel file
```

```
[68, 84, 73, 82]
```

writing array into xlsx

```
pl_name = ['Virat', 'Sachin', 'Dhoni', 'Rohit']
pl_score = [34534, 868964, 46464, 87564]
# creating two lists with data to write into two columns in excel sheet
```

op.Workbook()

```
wkbook = op.Workbook()
# create workbook object
```

WorkBook.create_sheet('SheetName')

```
sht1 = wkbook.create_sheet('IPL')
# create WorkSheet object
```

WorkSheet['cellName'] = value

```
sht1['A1'] = 'Name'
sht1['B1'] = 'Score'
# setting value for header in excel sheet
```

WorkBook.save('fileName.xlsx')

```
wkbook.save("Cricket.xlsx")
# saving the Workbook to current directory as excel file
```

```
len(pl_name)
```

```
4
```

```
a = len(pl_name)
# storing length of lists to be written into
# excel WorkBook file to use in loop
```

```
for ctr in range(2, a+2):
    x = pl_name[ctr-2]
    #reading list from 0...length of list
    sht1.cell(ctr, 1).value = x
    # writing the value from list to cell in WorkSheet

# player names to be written to col#1
# writing of data should start from row#2 and finish at row#(length of list + 2)
```

```
for ctr in range(2, a+2):
    x = pl_score[ctr-2]
    #reading list from 0...length of list
    sht1.cell(ctr, 2).value = x
    # writing the value from list to cell in WorkSheet

# scores to be written to col#2
# writing of data should start from row#2 and finish at row#(length of list + 2)
```

```
wkbook.save("Cricket.xlsx")
```

Pandas

- open sourced, fast and efficient `DataFrame` object for data manipulation with integrated indexing
- Tools for `reading and writing data` between in-memory data structures and different formats: CSV and text files, Microsoft Excel, SQL databases, and the fast HDF5 format
- Intelligent `data alignment` and integrated handling of `missing data`: gain automatic label-based alignment in computations and easily manipulate messy data into an orderly form
- Flexible `reshaping` and pivoting of data sets
- Intelligent label-based `slicing, fancy indexing, and subsetting` of large data sets
- Columns can be inserted and deleted from data structures for `size mutability`
- Aggregating or transforming data with a powerful `group by` engine allowing split-apply-combine operations on data sets
- High performance `merging and joining` of data sets
- `Hierarchical axis indexing` provides an intuitive way of working with high-dimensional data in a lower-dimensional data structure
- `Time series`-functionality: date range generation and frequency conversion, moving window statistics, date shifting and lagging. Even create domain-specific time offsets and join time series without losing data
- Highly `optimized for performance`, with critical code paths written in Cython or C
- Python with pandas is in use in a wide variety of `academic and commercial` domains, including Finance, Neuroscience, Economics, Statistics, Advertising, Web Analytics, and more

- DS in pandas
    1. Series (single-Dimensional)
    2. DataFrames (Multi-Dimensional)

pandas Series

```
1 import pandas as pd
2 # importing pandas library
```

```
1 x = [3, 7, 8, 1, 2]
2 # creating a list
3 x
```

```
[3, 7, 8, 1, 2]
```

```
x[2]
# accessing 2nd index of list
```

```
8
```

pd.Series(iterable)

```
ser1 = pd.Series(x)
# creating series using the list preated previously
ser1
```

```
0    3
1    7
2    8
3    1
4    2
dtype: int64
```

```
print(type(ser1))
#checking type of pandas Series
```

```
<class 'pandas.core.series.Series'>
```

```
ser1[2]
# accesing value at 2nd index of series
```

```
8
```

pd.Series(iterable, index=ArrayLikeOrIndex)

```
ser1 = pd.Series(x, index=['A', 'B', 'C', 'D', 'E'])
# creates a series with default index like 0, 1, 2, 3...
# but then reindexes as per index argument list

# so this Series can be accessed by both default indices like
# 0, 1, 2, ... or by custom indices from index argument
ser1
```

```
A    3
B    7
C    8
D    1
```

```
E    2
dtype: int64
```

```
ser1[2]
# accesing Series element using default index
```

```
8
```

```
ser1['C']
# accesing Series element using custom index
```

```
8
```

pd.Series(dict)

```
x = {'Sudeep':'Blore', 'Sachin':'Mumbai', 'Virat':'Delhi', 'Ashwin':'Chennai'}
# creating a dict with names:location
```

```
ser2 = pd.Series(x)
# using dict to create series
# first Series is created with default indices like 0, 1, 2, ... and
# values from dict
# then Series is reindexed as per keys from dict
# so this Series is accessible by both default indices and keys from dict
ser2
```

```
Sudeep      Blore
Sachin     Mumbai
Virat       Delhi
Ashwin    Chennai
dtype: object
```

```
ser2['Sudeep'] = 'Mumbai'
# updating value in series by refering element using key
ser2
```

```
Sudeep     Mumbai
Sachin     Mumbai
Virat       Delhi
Ashwin    Chennai
dtype: object
```

```
ser2[0] = 'Blore'
# updating value in series by refering element using default index
ser2
```

```
Sudeep      Blore
Sachin     Mumbai
Virat       Delhi
Ashwin    Chennai
dtype: object
```

```
new_pl = pd.Series(['Blore','Kolkata'], index = ['ABD', 'Sehwag'])
# creating a new series using list as for elements and
# providing custom indices in index argument
new_pl
```

```
ABD         Blore
Sehwag    Kolkata
dtype: object
```

Series1.append(Series2)

```
ser2 = ser2.append(new_pl)
# appending more players using Series new_pl to previous Series ser2
ser2
```

```
<ipython-input-71-206af7a04b9e>:1: FutureWarning: The series.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat inst
  ser2 = ser2.append(new_pl)
Sudeep      Blore
Sachin     Mumbai
Virat       Delhi
Ashwin    Chennai
ABD         Blore
Sehwag    Kolkata
dtype: object
```

Series1.drop('Key' or index)

```
ser2 = ser2.drop(['ABD'])
# dropping record with index 'ABD'
ser2
```

```
Sudeep      Blore
Sachin     Mumbai
```

```
Virat       Delhi
Ashwin    Chennai
Sehwag    Kolkata
dtype: object
```

pandas DataFrame

```
stu_info = [ ['Sudeep', 'Blore', 78], ['Rakesh', 'Chennai', 57], ['Mayur', 'Mumbai', 85], ['Charu', 'Kharghar', 80] ]
# creating a 2-D list
```

```
len(stu_info)
# printing length of 2-D list
```

```
4
```

```
len(stu_info[0])
# printing length of first list in 2-D List
```

```
3
```

pd.DataFrame(2-D List)

```
stu_df = pd.DataFrame(stu_info, columns=['Name', 'Location', 'Marks'])
# creating a DataFrame using 2-D List and columns attribute
stu_df
```

	Name	Location	Marks
0	Sudeep	Blore	78
1	Rakesh	Chennai	57
2	Mayur	Mumbai	85
3	Charu	Kharghar	80

players in DataFrame

- appending pl_df into new_pl with different column names

pd.DataFrame(dict)

```
pl_info = {'Name':['Sachin', 'Virat', 'Rohit', 'Ashwin'], 'Score':[3463, 64563, 342, 6467], 'Grade':['A', 'B', 'A', 'C']}
# creating a dict with keys as 'Name', 'Score', 'Grade'
```

```
pl_df = pd.DataFrame(pl_info)
# using dict to create a DataFrame
pl_df
```

	Name	Score	Grade
0	Sachin	3463	A
1	Virat	64563	B
2	Rohit	342	A
3	Ashwin	6467	C

DataFrame.at[index, 'ColumnName'] = value

```
pl_df.at[2, 'Grade'] = 'B'
# updating value of column by specifying index and ColumnName
# pl_df.at[index, 'ColumnName'] = 'value'
pl_df
```

	Name	Score	Grade
0	Sachin	3463	A
1	Virat	64563	B
2	Rohit	342	B
3	Ashwin	6467	C

DataFrame(index, 'ColumnNumber') = value

```
pl_df.at[3, 2] = 'B'
# updating value of column by specifying index and ColumnNumber
# pl_df.at[index, 'ColumnNumber'] = 'value'
pl_df
```

	Name	Score	Grade
0	Sachin	3463	A
1	Virat	64563	B
2	Rohit	342	B
3	Ashwin	6467	B

appending one DataFrame into another with Different column names

```
new_pl = pd.DataFrame([['Jadeja', 34534, 'B'], ['Dhoni', 5346, 'A']], columns = ['Player', 'Runs', 'Grade'])
# creating a new DataFrame with some different column names as previous one
new_pl
```

	Player	Runs	Grade
0	Jadeja	34534	B
1	Dhoni	5346	A

```
pl_df
# printing previous DataFrame
```

	Name	Score	Grade
0	Sachin	3463	A
1	Virat	64563	B
2	Rohit	342	B
3	Ashwin	6467	B

```
pl_df = pl_df.append(new_pl)
# appending new dataFrame into previous one, but with different column names
pl_df
# printing dataFrame with appended values
# column name does not match so, it'll place NaN values in non-matching values
```

```
<ipython-input-83-890c2d82e3d1>:1: FutureWarning: The frame.append method is deprecated an
  pl_df = pl_df.append(new_pl)
```

	Name	Score	Grade	Player	Runs
0	Sachin	3463.0	A	NaN	NaN
1	Virat	64563.0	B	NaN	NaN
2	Rohit	342.0	B	NaN	NaN
3	Ashwin	6467.0	B	NaN	NaN
0	NaN	NaN	B	Jadeja	34534.0
1	NaN	NaN	A	Dhoni	5346.0

appending one DataFrame into another with same column names

- appending `pl_df` into `new_pl` with same column names

```
pl_info = {'Name':['Sachin', 'Virat', 'Rohit', 'Ashwin'], 'Score':[3463, 64563, 342, 6467], 'Grade':['A', 'B', 'A', 'C']}
# creating dict with keys 'Name', 'Score', 'Grade'
```

```
pl_df = pd.DataFrame(pl_info)
# using dict to create first DataFrame with columns 'Name', 'Score', 'Grade'
pl_df
```

	Name	Score	Grade
0	Sachin	3463	A
1	Virat	64563	B
2	Rohit	342	A
3	Ashwin	6467	C

```
new_pl = pd.DataFrame([['Jadeja', 34534, 'B'], ['Dhoni', 5346, 'A']], columns = ['Name', 'Score', 'Grade'])
# creating second DataFrame with same columns 'Name', 'Score', 'Grade'
new_pl
```

```
Name  Score  Grade
```

```
pl_df = pl_df.append(new_pl)
# appending second DataFrame to First one, but with same columns
pl_df
# note that indices are not proper in resultant DataFrame
```

```
<ipython-input-87-69d84f99d1bf>:1: FutureWarning: The frame.append method is deprecated an
  pl_df = pl_df.append(new_pl)
```

	Name	Score	Grade
0	Sachin	3463	A
1	Virat	64563	B
2	Rohit	342	A
3	Ashwin	6467	C
0	Jadeja	34534	B
1	Dhoni	5346	A

DataFrame1.shape

```
pl_df.shape
# checking the shape of DataFrame
```

```
(6, 3)
```

```
pl_df.shape[0]
# prints row count for Dataframe
```

```
6
```

DataFrame1.index

```
pl_df.index = range(pl_df.shape[0])
# updating the DataFrame index attribute with number of rows
pl_df
# note that indices are now properly ordered
```

	Name	Score	Grade
0	Sachin	3463	A
1	Virat	64563	B
2	Rohit	342	A
3	Ashwin	6467	C
4	Jadeja	34534	B

DataFrame1[newColumnName] = list1 or Iterator

- adds a new column to DataFrame

```
y = ['MI', 'RCB', 'MI', 'RR', 'CSK', 'CSK']
# creating a 1-D list of teams
y
```

```
['MI', 'RCB', 'MI', 'RR', 'CSK', 'CSK']
```

```
len(y)
# length of list with team names
```

```
6
```

```
pl_df['Team'] = y
# ading a new column to existing DataFrame
# DataFrame1[NewColumnName'] = List or Iterator
pl_df
```

	Name	Score	Grade	Team
0	Sachin	3463	A	MI
1	Virat	64563	B	RCB
2	Rohit	342	A	MI
3	Ashwin	6467	C	RR
4	Jadeja	34534	B	CSK
5	Dhoni	5346	A	CSK

Accessing a particular column of DataFrame

```
pl_df['Score']
```

```
0     3463
1    64563
2      342
3     6467
4    34534
5     5346
Name: Score, dtype: int64
```

```
pl_df.Score
# Alternate method, not recommended
```

```
0     3463
1    64563
2      342
3     6467
4    34534
5     5346
Name: Score, dtype: int64
```

```
1
```