→ Course contents

1. Programming
2. DataBase Management System
3. Descriptive statistics
4. Inferential statistics
5. Predictive statistics
6. Modeling

→ Types of statistics

1. Descriptive statistics - news and all, indicating variances or central tendencies
2. Inferential statistics - making inferences using different tests, regressions, correlations to take decisions
3. Predictive statistics - predictions based on previous data using regessions, classifications, associations or clustering
4. Diagnostic Analytics - examines data or content tofind the root cause of an event happening, includes techniques such as drill-down, data discovery, data mining and correlations

→ Libraries to study

1. NumPy (package for scientific computing) https://pypi.org/project/numpy/
2. Pandas (fast, flexible, and expressive data structures) https://pypi.org/project/pandas/
3. OpenPyXL library (to export/import data from xlsx / csv file) https://pypi.org/project/openpyxl/

NumPy

- arrays in NumPy are called n-D array (ndarray)

```
import numpy as np
```

np.arange()

```
x = np.arange(7)
# to generate ndarray with even spacing of 7 numbers
x
```

```
array([0, 1, 2, 3, 4, 5, 6])
```

```
x = np.arange(3, 7)
# to generate ndarray between 3, 7 with default step=1
x
```

```
array([3, 4, 5, 6])
```

```
x = np.arange(3, 17, 4)
# to generate ndarray between 3,17 with step=4
x
```

```
array([ 3,  7, 11, 15])
```

```
x = np.arange(40, 6, -2)
# to generate ndarray between 40,6 with reverse step=4
x
```

```
array([40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10,  8])
```

np.array()

```
x = np.array([3, 5, 4, 7]) # creating ndarray
type(x)
```

```
numpy.ndarray
```

- direct operations on ndarray is possible, known as broadcasting

```
x+3
# broadcasting operation returns another ndarray
```

```
array([ 6,  8,  7, 10])
```

```
x
# actual value of ndarray is same
```

```
array([3, 5, 4, 7])
```

```
y = [2, 8, 7, 6]    # creating a list
y = np.array(y)     # casting that list to ndarray
y
```

```
array([2, 8, 7, 6])
```

```
y = y + 3           # direct operation on ndarray
y
```

```
array([ 5, 11, 10,  9])
```

```
y = list(y)         # casting that processed ndarray back to list
y                   # printing list
```

```
[5, 11, 10, 9]
```

```
arr2d = np.array([3, 7, 5, 8]) # creating ndarray
arr2d
```

```
array([3, 7, 5, 8])
```

```
arr2d = np.array([[3, 7, 5, 8], [1, 4, 3, 6]])
# creating 2-D ndarray with two rows
arr2d
```

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

```
arr2d = np.array([[3, 7, 5, 8], [1, 4, 3, 6], [1, 9, 7, 4]])
# creating 2-D ndarray with three rows
arr2d
```

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

np.reshape(r, c)

```
arr3d = np.array([ [[3, 7, 5, 8], [1, 4, 3, 6], [1, 9, 7, 4]], [[4, 5, 6, 8], [8, 7, 5, 6], [1, 4, 3, 7]] ])
# creating 3-D ndarray with two 2-D ndarrays of three row each
arr3d
```

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

```
          [1, 9, 7, 4]],

         [[4, 5, 6, 8],
          [8, 7, 5, 6],
          [1, 4, 3, 7]]])
```

```
t = [2, 4, 3, 6, 5, 7, 8, 9, 1, 2, 3, 5]
type(t) # checking type of t
```

```
list
```

```
len(t)
# printing length of list t
```

```
12
```

```
t = np.array(t)
# casting list t into ndarray t
t
```

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

```
arr2d = t.reshape(3, 4)
# reshaping 1-D ndarray to 2-D ndarray by specifying
# rows & columns to reshape(r,c) method
arr2d
```

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

```
arr2d = t.reshape(6, 2)
# again reshaping a 2-D ndarray to another 2-D ndarray by
# specifying rows & columns to reshape(r,c) method
arr2d
```

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

```
arr2d = t.reshape(2, 6)
# again reshaping a 2-D ndarray to another 2-D ndarray by
# swapping rows & columns count
arr2d
```

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

```
arr1d = np.array([3, 7, 6, 4, 8])
# creating 1-D ndarray
arr1d
```

```
array([3, 7, 6, 4, 8])
```

```
len(arr1d)
```

```
5
```

accessing elements in 1-D ndarray

```
arr1d[3]
# accessing 3rd element from 1-D ndarray
```

```
4
```

```
arr1d[3] = 7
# updating 3rd element from 1-D ndarray
arr1d
```

```
array([3, 7, 6, 7, 8])
```

np.append(ndarr, ele)

```
arr1d = np.append(arr1d, 10)
# np.append(ndarr, ele) appends ele to last of ndarray
arr1d
```

```
array([ 3,  7,  6,  7,  8, 10])
```

np.insert(ndarray, index, ele)

```
arr1d = np.insert(arr1d, 3, 5)
# np.insert(ndarray, pos, ele) inserts ele at index in ndarray
arr1d
```

```
array([ 3,  7,  6,  5,  7,  8, 10])
```

np.delete(ndarray, index) in 1-D ndarray

```
arr1d = np.delete(arr1d, 4)
# np.delete(ndarray, index) deletes element at specified index in ndarray
arr1d
```

```
array([ 3,  7,  6,  5,  8, 10])
```

np.array([list1, list2])

```
list1 = [2, 5, 4, 6]
list2 = [3, 9, 8, 1]
```

```
arr2d = np.array([list1, list2])
# using two lists as rows in 2-D ndarray
arr2d
```

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

np.array([list1, list2])

```
list1 = [2, 5, 4, 6]
list2 = [3, 9, 8, 1]
list3 = [4, 8, 7, 2]
```

```
arr2d = np.array([list1, list2, list3])
# using three lists as rows in 2-D ndarray
arr2d
```

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

accessing elements in 2-D ndarray

```
arr2d[1]
# to access 1st row of 2-D ndarray
```

```
array([3, 9, 8, 1])
```

```
arr2d[0]
# to access zeroth row of 2-D ndarray
```

```
array([2, 5, 4, 6])
```

```
arr2d[ : , 2]
# to access  only 2nd column element from all of the rows r[ : ] of 2-D ndarray
# [r[ : ],c]
```

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

```
arr2d[1][3]
# to access element at 1st row, 3rd column
```

```
1
```

```
arr2d[1][3] = 5
# to update element at 1st row, 3rd column
arr2d
```

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

vstack(ndarray1, ndarray2 or list)

- adding newrow to the existing 2D array, using vstack() method
- vstack will grow array in vertical dimension

```
list1 = [2, 5, 4, 6]
list2 = [3, 9, 8, 1]
list3 = [4, 8, 7, 2]
arr2d = np.array([list1, list2, list3])
# using three lists as rows in 2-D ndarray
arr2d
```

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

```
newrow = [3, 7, 1, 4]
# creating a horizontal list / row
newrow
```

```
[3, 7, 1, 4]
```

```
arr2d = np.vstack([arr2d, newrow])
# vertical stacking a horizontal list into 2-D ndarray
# using vstack(ndarray1, ndarray2 or list)
arr2d
```

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

hstack([ndarray1, ndarray2])

```
newcol = np.array([3, 8, 7, 2])
# creating a horiizontal list / row
newcol
```

```
array([3, 8, 7, 2])
```

```
newcol = newcol.reshape(4, 1)
# reshaping to convert horizontal ndarray/row INTO vertical ndarray/ column
newcol
```

```
array([[3],
       [8],
       [7],
       [2]])
```

```
arr2d = np.hstack([arr2d, newcol])
# using vertical ndarray / column to horizontally stack into 2-D ndarray
# hstack takes only vertical column to stack into 2-D ndarray
# because list cannot be vertical
arr2d
```

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

np.delete(ndarray, element, axis=0) in 2-D ndarray

- Axis 0: Row
- Axis 1: Column
- in reshape command, we follow `reshape(row, column)`

```
np.delete(arr2d, 2, axis=0)
# will delete 2nd element as per axis
```

```
# axis=0 will delete 2nd row
```

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

```
np.delete(arr2d, 2, axis=1)
# will delete 1st element as per axis
# axis=1 will delete 2nd column
```

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

slicing in ndarray

```
arr1d = np.array([3, 6, 5, 2, 7, 6, 9, 1])
# creating 1-D ndarray using list
```

```
arr1d[:4]
# accesing initial/zeroth to 3rd element
```

```
array([3, 6, 5, 2])
```

```
arr1d[5:]
# accesing 5th to last element
```

```
array([6, 9, 1])
```

```
arr1d[3:7]
# accesing 3rd to 6th (7-1=6) element
```

```
array([2, 7, 6, 9])
```

```
arr1d[2:6]
# accesing 2nd to 5th element
```

```
array([5, 2, 7, 6])
```

```
arr1d[2:5:3]
# accesing 2nd to 4th (51=4) element with step 3
# only 2nd element
```

```
array([5])
```

operational difference between array & list

1. list can't be operated on the go, it needs to be accessed with loop to operate on. While in array, array can be operated on the go, no need for using loop (broadcasting is possible, meaning you can operate on-the-go)
2. you can access mutiple non-adjacent index in array, but you can't in list
3. array can be multi-dimensional but list is always one-dimensional

sorted(ndarray)

```
sorted_arr1d = sorted(arr1d)
# by default sorted(ndarray) sorts in ascending order
sorted_arr1d
```

```
[1, 2, 3, 5, 6, 6, 7, 9]
```

```
sorted_arr1d = sorted(arr1d, reverse=True)
# sorted(ndarray) sorts in descending order
```

```
sorted_arr1d
```

```
[9, 7, 6, 6, 5, 3, 2, 1]
```

Questions in 1-D ndarray:

1. count of elements which are >= 4
2. print value of elements which are >= 4
3. print indices of elements which are >= 4

```
arr1d = np.array([3, 6, 5, 2, 7, 6, 9, 1])
# creating 1-D ndarray using list
arr1d
```

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

index = np.where(condition)

- input of `np.where(condition)` is condition
- output of `np.where(condition)` is index where condition is True

```
(arr1d >= 4)
# condition marks True for each of the element of ndarray for which
# condition is true, shows broadcasting behavior
```

```
array([False,  True,  True, False,  True,  True,  True, False])
```

```
ind = np.where(arr1d >= 4)
# check for indices of elements satisfying condition
# returns tuple of ndarrays containing indices of Trues returned from condition
ind
```

```
(array([1, 2, 4, 5, 6]),)
```

```
len(ind)
# returns 1, because length of tuple is one
```

```
1
```

```
len(ind[0])
# Q1 soln
# returns 5, because length of zeroth element
# (which is ndarray of indexes satisfying condition) in the tuple is five
```

```
5
```

```
ind[0]
# Q3 soln
# accesses zeroth element
# (which is ndarray of indexes satisfying condition) of tuple
```

```
array([1, 2, 4, 5, 6])
```

```
type(ind[0])
```

```
numpy.ndarray
```

```
type(ind)
```

```
tuple
```

(condition).sum()

```
(arr1d>=4).sum()
# Q1 soln
# sums all the True values found in ndarray from satisfying condition
# which is actually the count of elements satisfying condition >= 4
```

```
5
```

```
arr1d[ind]
# Q2 soln
# passing all the indxes where condition is satidfied using ind vaiable
# to ndarray to get value of elements which satisfied the condition
```

```
array([6, 5, 7, 6, 9])
```

Questions in 2-D ndarray:

1. count of elements which are >= 4
2. print value of elements which are >= 4
3. print indices of elements which are >= 4

```
list1 = [2, 5, 4, 6, 3]
list2 = [3, 9, 8, 1, 8]
list3 = [4, 8, 7, 2, 7]
list4 = [3, 7, 1, 4, 2]
arr2d = np.array([list1, list2, list3, list4])
# creating 2-D ndarray using four lists as rows
arr2d
```

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

index = np.where(condition) in 2-D ndarray

```
ind2 = np.where(arr2d>=4)
# check for indices of elements satisfying condition
ind2
# Q3 soln
```

```
(array([0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3]),
 array([1, 2, 3, 1, 2, 4, 0, 1, 2, 4, 1, 3]))
```

```
len(ind2)
```

```
2
```

```
len(ind2[0])
# Q1 soln
# count of elements which are >= 4
```

```
12
```

(condition).sum() in 2-D ndarray

```
(arr2d>=4).sum()
# Q1 soln
# count of elements which are >= 4
```

```
12
```

```
arr2d[ind2]
# Q2 solns
# print value of elements which are >= 4
```

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

np.concatenate((ndarray1, ndarray2))

```
arr1d = np.array([3, 6, 5, 2, 7, 6, 9, 1, 3])
# creating 1-D ndarray using list
arr1d
```

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

```
x = np.array([3, 5, 7, 3, 3, 5, 9, 1, 3])
x
```

```
array([3, 5, 7, 3, 3, 5, 9, 1, 3])
```

```
arr1d = np.concatenate((arr1d, x))
arr1d
```

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

np.unique(ndarray)

```
np.unique(arr1d)
# returns sorted list of unique values from array
```

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

```
4 in np.unique(arr1d)
# checks if 4 exists in elements of ndarray arr1d
```

```
False
```

1-D ndarray operations

np.in1d(ndarray1, ndarray2 or list)

```
np.in1d(arr1d, [2, 7])
# condition to check if each element of 1st ndarray is in 2nd ndarray / list
# returns ndarray of True/False based on if condition is satisfied for each element
```

```
array([False, False, False,  True,  True, False, False, False, False,
       False, False,  True, False, False, False, False, False, False])
```

```
np.where(np.in1d(arr1d, [2, 7]))
# using np.in1d() as a condition to np.where() to get indeces of elements
# satisfying condition
```

```
(array([ 3,  4, 11]),)
```

- input of `np.where(condition)` is condition
- output of `np.where(condition)` is index where condition is True

```
x = np.array([4, 7, 6, 3, 9])
y = np.array([1, 4, 5, 7, 8])
# creating two 1-D ndarrays
```

np.union1d(ndarray1, ndarray2)

```
np.union1d(x, y)
# returns sorted ndarray of union of two ndarrays
# all unique elements from both ndarrays
```

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

```
np.union1d(y, x)
# np.union1d() is not affected by order of ndarrays in argument
```

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

np.intersection(ndarray1, ndarray2)

```
np.intersect1d(x, y)
# np.intersection() returns sorted ndarray of common elements
```

```
array([4, 7])
```

```
np.intersect1d(y, x)
# np.intersection() is not affected by order of ndarrays in argument
```

```
array([4, 7])
```

```
np.in1d(x, y)
# means x in y
```

```
array([ True,  True, False, False, False])
```

```
np.in1d(y, x)
# means y in x
# checks if all elements of ndarray1 are in ndarray2
```

```
array([False,  True, False,  True, False])
```

```
x
```

```
array([4, 7, 6, 3, 9])
```

```
y
```

```
array([1, 4, 5, 7, 8])
```

np.setdiff1d(ndarray1d1, ndarray1d2)

```
np.setdiff1d(x, y)
# shows exclusive elements of x which are not in y
```

```
array([3, 6, 9])
```

```
np.setdiff1d(y, x)
# shows exclusive elements of y which are not in x
# result will vary if order of ndarrays is changed in argument list
```

```
array([1, 5, 8])
```

np.setxor1d(ndarray1d1, ndarray1d2)

```
np.setxor1d(x, y)
# exclusive elements of x or exclusive elements of y
# from either x or y but not the common ones
```

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

```
np.setxor1d(y, x)
# exclusive elements of x or exclusive elements of y
# from either x or y but not the common ones
# result will not vary if order of ndarrays is changed in argument list
```

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