Experiment-1

NumPy Installation using different scientific python distributions(Anaconda, Python(x,y), WinPython, Pyzo)

Installing via Python Package index pip

python -m pip install --user numpy scipy matplotlib ipython jupyter pandas sympy nose

Ubuntu

sudo apt-get install python-numpy python-scipy python-matplotlib ipython ipython-notebook python-pandas

Experiment-2

NumPy Basics (np.array, np.arrange, np.linespace, np.zeros, np.ones, np.random.random, np.empty)

Difference between list, tuple and array

```
from array import array

# List
fruits_list = ["Apple", "Banana", "Orange"]

# Tuple
colors_tuple = ("Red", "Green", "Blue")

# Array
numbers_array = array('i', [1, 2, 3, 4, 5])

# Modifying the list
fruits_list.append("Grapes")
fruits_list[0] = "Cherry"  # Valid modification

# Attempting to modify the tuple (which is not allowed for tuples)
try:
    colors_tuple[0] = "Yellow"  # This line will raise a TypeError
except TypeError as e:
    print(f"Error with Tuple: {e}")

# Modifying the array
numbers_array.append(6)
numbers_array[0] = 0  # Valid modification

# Displaying the collections after modifications
print("List of Fruits:", fruits_list)
print("Tuple of Colors:", colors_tuple)  # This line won't execute due to the error
print("Array of Numbers:", numbers_array)
```

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

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

O/P: [1 2 3 4 5]
<class 'numpy.ndarray'>

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray

Use a tuple to create a NumPy array:

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

O/P: [1 2 3 4 5]

Dimensions in Arrays
A dimension in arrays is one level of array depth (nested arrays).
nested array: are arrays that have arrays as their elements.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example
Create a 0-D array with value 42

```
import numpy as np
arr = np.array(42)
print(arr)
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
These are the most common and basic arrays.

Example
Create a 1-D array containing the values 1,2,3,4,5:

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

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.
These are often used to represent matrix or 2nd order tensors.
NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

Example
Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.
These are often used to represent a 3rd order tensor.

Example
Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

Higher Dimensional Arrays

An array can have any number of dimensions.
When the array is created, you can define the number of 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)
```

Check how many dimensions the arrays have:

```
import numpy as np
a = 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)
```

Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example

Get third and fourth elements from the following array and add them.

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[2] + arr[3])
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the row represents the dimension and the index represents the column.

Example

Access the element on the 2nd row, 5th column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(': ', arr[1, 4])
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

Example

Access the third element of the second array of the first array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

Negative Indexing

Use negative indexing to access an array from the end.

Example

Print the last element from the 2nd dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [*start*:*end*].

We can also define the step, like this: [*start*:*end*:*step*].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

Example

Slice elements from index 1 to index 5 from the following array:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

Negative Slicing

Use the minus operator to refer to an index from the end:

Example

Slice from the index 3 from the end to index 1 from the end:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-3:-1])
```

np.arrange()

NumPy arange() is one of the array creation routines based on numerical ranges. It creates an instance of ndarray with *evenly spaced values* and returns the reference to it.

```
numpy.arange([start, ]stop, [step, ], dtype=None) -> numpy.ndarray
```

1. **start** is the number (integer or decimal) that defines the first value in the array.
2. **stop** is the number that defines the end of the array and isn't included in the array.
3. **step** is the number that defines the spacing (difference) between each two consecutive values in the array and defaults to 1.
4. **dtype** is the type of the elements of the output array and defaults to None.

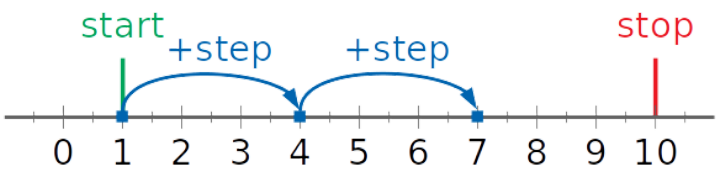
step can't be zero. Otherwise, you'll get a **ZeroDivisionError**. You can't move away anywhere from start if the increment or decrement is 0.

```
import numpy as np
np.arange(start=1, stop=10, step=3)
o/p: array([1, 4, 7])
```

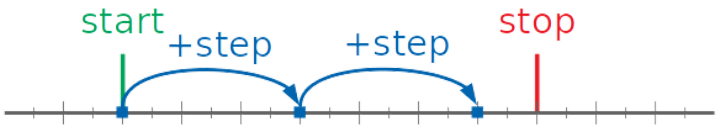
In this example, start is 1. Therefore, the first element of the obtained array is 1. step is 3, which is why your second value is 1+3, that is 4, while the third value in the array is 4+3, which equals 7.

```
np.arange(1, 10.1, 3)
O/P:
```

```
>>> np.arange(1, 10, 3)
array([1, 4, 7])
```




```
>>> np.arange(1, 8, 3)
array([1, 4, 7])
```



```
array([ 1.,  4.,  7., 10.])
>>> np.arange(start=0, stop=10, step=1)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(0, 10, 1)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(start=0, stop=10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(0, 10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Providing Negative Arguments

If you provide negative values for start or both start and stop, and have a positive step, then arange() will work the same way as with all positive arguments:

```
>>>
```

```
>>> np.arange(-5, -1)
array([-5, -4, -3, -2])
>>> np.arange(-8, -2, 2)
array([-8, -6, -4])
>>> np.arange(-5, 6, 4)
array([-5, -1,  3])
```

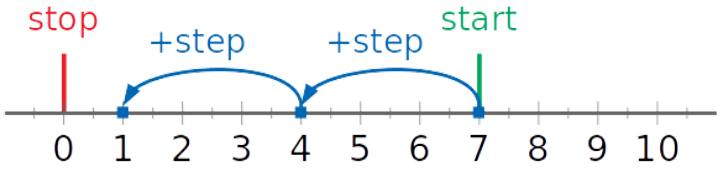
Counting Backwards

Sometimes you'll want an array with the values decrementing from left to right. In such cases, you can use arange() with a negative value for step, and with a start greater than stop:

```
>>>
```

```
>>> np.arange(5, 1, -1)
array([5, 4, 3, 2])
>>> np.arange(7, 0, -3)
array([7, 4, 1])
```

```
>>> np.arange(7, 0, -3)
array([1, 4, 7])
```



Getting Empty Arrays

There are several edge cases where you can obtain empty NumPy arrays with arange(). These are regular instances of numpy.ndarray without any elements.

If you provide equal values for start and stop, then you'll get an empty array:

```
>>>
```

```
>>> np.arange(2, 2)
array([], dtype=int64)
```

np.linespace

numpy.linspace(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, *dtype=None*, *axis=0*)[source]

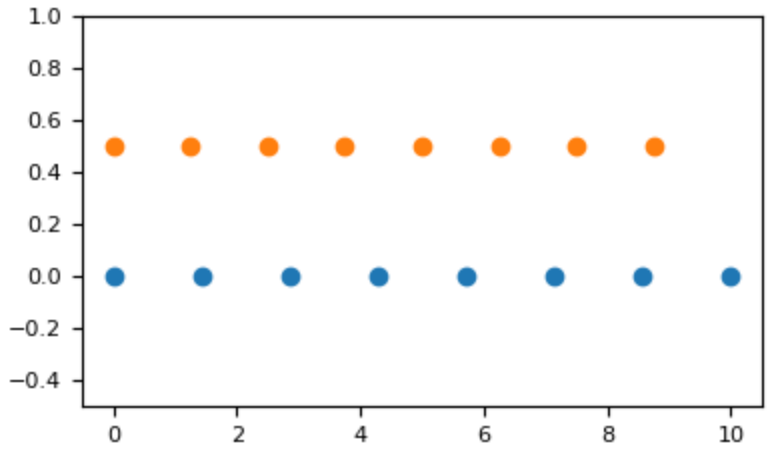
Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

```
np.linspace(2.0, 3.0, num=5)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. ,  2.2,  2.4,  2.6,  2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

```
import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
```



numpy.zeros()

The **numpy.zeros()** function returns a new array of given shape and type, with zeros.
Syntax:
numpy.zeros(shape, dtype = None, order = 'C')

Parameters :
shape : integer or sequence of integers
order : C_contiguous or F_contiguous
C-contiguous order in memory(last index varies the fastest)
C order means that operating row-rise on the array will be slightly quicker
FORTRAN-contiguous order in memory (first index varies the fastest).
F order means that column-wise operations will be faster.
dtype : [optional, float(byDeafult)] Data type of returned array.
Returns :

ndarray of zeros having given shape, order and datatype.

```
import numpy as np

b = np.zeros(2, dtype = int)
print("Matrix b : \n", b)

a = np.zeros([2, 2], dtype = int)
print("\nMatrix a : \n", a)

c = np.zeros([3, 3])
print("\nMatrix c : \n", c)
```

O/P

Matrix b :

[0 0]

Matrix a :

[[0 0]

[0 0]]

Matrix c :

[[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]]

Code 2 : Manipulating data types

```
import numpy as np

# manipulation with data-types
b = np.zeros((2,), dtype=[('x', 'float'), ('y', 'int')])
print(b)
```

Output :
[(0.0, 0) (0.0, 0)]

numpy.ones()

The **numpy.ones()** function returns a new array of given shape and type, with ones.

Syntax: **numpy.ones(shape, dtype = None, order = 'C')**

```
import numpy as np

b = np.ones(2, dtype = int)
print("Matrix b : \n", b)

a = np.ones([2, 2], dtype = int)
print("\nMatrix a : \n", a)

c = np.ones([3, 3])
print("\nMatrix c : \n", c)
```

Output :

Matrix b :

[1 1]

Matrix a :

[[1 1]

[1 1]]

Matrix c :

[[1. 1. 1.]

[1. 1. 1.]

[1. 1. 1.]]

numpy.random.random

random.random(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

numpy.empty().

The numpy module of Python provides a function called **numpy.empty()**. This function is used to create an array without initializing the entries of given shape and type.

Syntax

numpy.empty(shape, dtype=**float**, order='C')

shape: int or tuple of ints

This parameter defines the shape of the empty array, such as (3, 2) or (3, 3).

dtype: data-type(optional)

This parameter defines the data type, which is desired for the output array.

order: {'C', 'F'}(optional)

This parameter defines the order in which the multi-dimensional array is going to be stored either in **row-major** or **column-major**. By default, the order parameter is set to **'C'**

Returns:

This function returns the array of uninitialized data that have the shape, dtype, and order defined in the function.

Example 1:

```
1. import numpy as np
2. x = np.empty([3, 2])
3. x
```

Output:

```
array([[7.56544226e-316, 2.07617768e-316],
       [2.02322570e-316, 1.93432036e-316],
       [1.93431918e-316, 1.93431799e-316]])
```

In the above code

- We have imported numpy with alias name np.
- We have declared the variable **'x'** and assigned the returned value of the **np.empty()** function.
- We have passed the shape in the function.
- Lastly, we tried to print the value of **'x'** and the difference between elements.

Example 2:

```
1. import numpy as np
2. x = np.empty([3, 3], dtype=float)
3. x
```

```
array([[ 2.94197848e+120, -2.70534020e+252, -4.25371363e+003],
       [ 1.44429964e-088,  3.12897830e-053,  1.11313317e+253],
       [-2.28920735e+294, -5.11507284e+039,  0.00000000e+000]])
```

Example 3:

```
1. import numpy as np
2. x = np.empty([3, 3], dtype=float, order='C')
3. x
```

Output:

```
array([[ 2.94197848e+120, -2.70534020e+252, -4.25371363e+003],
       [ 1.44429964e-088,  3.12897830e-053,  1.11313317e+253],
       [-2.28920735e+294, -5.11507284e+039,  0.00000000e+000]])
```

In the above code

- We have imported numpy with alias name np.
- We have declared the variable **'x'** and assigned the returned value of the **np.empty()** function.
- We have passed the shape, data-type, and order in the function.
- Lastly, we tried to print the value of **'x'** and the difference between elements.

In the output, it shows an array of uninitialized values of defined shape, data type, and order.

Example 4:

```
1. import numpy as np
2. x = np.empty([3, 3], dtype=float, order='F')
3. x
```

Output:

```
array([[ 2.94197848e+120,  1.44429964e-088, -2.28920735e+294],
       [-2.70534020e+252,  3.12897830e-053, -5.11507284e+039],
       [-4.25371363e+003,  1.11313317e+253,  0.00000000e+000]])
```

Experiment-3

Arrays (array.shape, len(array), array.ndim, array.dtype, array.astype(type), type(array))

array.shape

Shape of an Array

The shape of an array is the number of elements in each dimension.

NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

Print the shape of a 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)
```

Create an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4:

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('shape of array :', arr.shape)
```

What does the shape tuple represent?

Integers at every index tells about the number of elements the corresponding dimension has.

In the example above at index-4 we have value 4, so we can say that 5th (4 + 1 th) dimension has 4 elements.

Reshaping arrays

The word "**reshape**" simply indicates changing the shape and that is what this function is used for

- The reshape() function in the NumPy library is mainly used to **change the shape of the array** without changing its original data.

- Thus reshape() function helps in **providing new shape to an array**, which can be useful baed on your usecase.
- np. reshape(x, m, n): This creates , X matrices with M rows and N colums
- np. reshape(2, 3, 2): This creates , 2 matrices with 3 rows and 2 colums

Syntax of reshape():

The syntax required to use this function is as follows:

numpy.reshape(a, newshape, order='C')

```
import numpy as np



a = np.arange(12)

print("The Original array : \n", a)



# shaping the array with 2 rows and 4 columns

a1= np.arange(12).reshape(2, 6)

print("\n The reshaped array with 2 rows and 6 columns : \n", a1)



# shaping the array with 4 rows and 2 columns

a2 = np.arange(12).reshape(6,2)

print("\n The reshaped array with 6 rows and 2 columns : \n", a2)



# Construction of a 3D array
```

```
The Original array :
 [ 0  1  2  3  4  5  6  7  8  9 10 11]

 The reshaped array with 2 rows and 6 columns :
 [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]

 The reshaped array with 6 rows and 2 columns :
 [[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
```

```
a3 = np.arange(12).reshape(2, 3, 2)

print("\nAfter reshaping the original array to 3D : \n", a3)
```

```
import numpy as np
x = np.arange(12)
print("The array is :\n",x)
y = np.reshape(x, (4, 3), order='F')
print("Reshaping the original array using F-like index ordering")print(y)
```

```
The array is :
 [ 0  1  2  3  4  5  6  7  8  9 10 11]
Reshaping the original array using F-like index ordering
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

```
import numpy as np
x = np.arange(12)
print("The array is :\n",x)
y = np.reshape(x, (4, 3), order='C')
print("Reshaping the original array using C-like index ordering")
print(y)
```

```
The array is :
 [ 0  1  2  3  4  5  6  7  8  9 10 11]
Reshaping the original array using C-like index ordering
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

Reshape From 1-D to 2-D

Example

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
```

array.ndim

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Example

Check how many dimensions the arrays have:

```
import numpy as np

a = 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)
```

Data Types in Python

By default Python have these data types:

- strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- integer - used to represent integer numbers. e.g. -1, -2, -3
- float - used to represent real numbers. e.g. 1.2, 42.42
- boolean - used to represent True or False.
- complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- **i - integer**
- **b - boolean**
- **u - unsigned integer**
- **f - float**
- **c - complex float**
- **m - timedelta**
- **M - datetime**
- **O - object**
- **S - string**
- **U - unicode string**
- **V - fixed chunk of memory for other type (void)**
- **Checking the Data Type of an Array**

Note: The NumPy array object has a property called dtype that returns the data type of the array.

Example

Get the data type of an array object:

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

Get the data type of an array containing strings:

```
import numpy as np
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

Creating Arrays With a Defined Data Type

We use the array() function to create arrays, this function can take an optional argument: dtype that allows us to define the expected data type of the array elements:

Example

Create an array with data type string:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

For i, u, f, S and U we can define size as well.

Example

Create an array with data type 4 bytes integer:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)
print(arr.dtype)
```

What if a Value Can Not Be Converted?

If a type is given in which elements can't be casted then NumPy will raise a ValueError.

ValueError: In Python ValueError is raised when the type of passed argument to a function is unexpected/incorrect.

A non integer string like 'a' can not be converted to integer (will raise an error):

```
import numpy as np

arr = np.array(['a', '2', '3'], dtype='i')
```

o/p:

```
Traceback (most recent call last):
  File "./prog.py", line 3, in
ValueError: invalid literal for int() with base 10: 'a'
```

astype()

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method.

The astype() function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

Change data type from float to integer by using 'i' as parameter value:

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')
print(newarr)
print(newarr.dtype)
```

output

```
[1 2 3]
int32
```

Change data type from float to integer by using int as parameter value:

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype(int)
print(newarr)
print(newarr.dtype)
```

output

```
[1 2 3]
int64
```

Change data type from integer to boolean:

```
import numpy as np
arr = np.array([1, 0, 3])
newarr = arr.astype(bool)
print(newarr)
print(newarr.dtype)
```

O/P:
[TRUE,FALSE,TRUE]

Type(array) versus dtype(array), len(array)

Python len() method enables us to find the total number of elements in the array. That is, it returns the count of the elements in the array.

Syntax:

```
len(array)
```

```
import numpy as np
arr = np.array([1.1, 2.6, 3.7, 4.7, 5.5])
x = arr.copy()
arr[0] = 42
```

```
a1=np.array(['divyagdgfd'],dtype='S')
a2=np.array(['divyagdgfd'])
print(type(arr))
print(arr.dtype)
print(type(a1))
print(a1.dtype)
print(type(a2))
print(a2.dtype)
print(len(arr))
```

O/P:

```
<class 'numpy.ndarray'>
float64
<class 'numpy.ndarray'>
|S10
<class 'numpy.ndarray'>
<U10
5
```

Experiment-4

Array Manipulation (np.append, np.insert, np.resize, np.delete, np.concatenate, np.vstack, np.hstack)

Array Manipulations

Several routines are available in NumPy package for manipulation of elements in ndarray object.

Adding / Removing Elements

Sr.No.	Element & Description
1	resize Returns a new array with the specified shape
2	append Appends the values to the end of an array
3	insert Inserts the values along the given axis before the given indices
4	delete Returns a new array with sub-arrays along an axis deleted
5	unique Finds the unique elements of an array

1. Resize( )

This function returns a new array with the specified size. If the new size is greater than the original, the repeated copies of entries in the original are contained. The function takes the following parameters.

numpy.resize(arr, shape)

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])

print( 'First array:' )
print(a)
print('\n')

print ('The shape of first array:' )
```

```
print (a.shape )
print ('\n'  )
b = np.resize(a, (3,2))

print('Second array:' )
print(b)
print('\n')

print('The shape of second array:' )
print(b.shape)
print('\n'  )
# Observe that first row of a is repeated in b since size is bigger

Print('Resize the second array:' )
b = np.resize(a,(3,3))
print(b)
```

The above program will produce the following output −

```
First array:
[[1 2 3]
 [4 5 6]]

The shape of first array:
(2, 3)

Second array:
[[1 2]
 [3 4]
 [5 6]]

The shape of second array:
(3, 2)

Resize the second array:
[[1 2 3]
 [4 5 6]
 [1 2 3]]
```

2. np.append( )

This function adds values at the end of an input array. The append operation is not inplace, a new array is allocated. Also the dimensions of the input arrays must match otherwise ValueError will be generated.

The function takes the following parameters.

numpy.append(arr, values, axis)

values

To be appended to arr. It must be of the same shape as of arr (excluding axis of appending)

axis

The axis along which append operation is to be done. If not given, both parameters are flattened

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])

print('First array:')
print(a)
print('\n')

print('Append elements to array:')
print(np.append(a, [7,8,9]) )
print('\n')

print('Append elements along axis 0:')
print(np.append(a, [[7,8,9]],axis = 0))
print('\n')

print('Append elements along axis 1:')
print(np.append(a, [[5,5,5],[7,8,9]],axis = 1))
```

```
o/p:
First array:
[[1 2 3]
 [4 5 6]]

Append elements to array:
[1 2 3 4 5 6 7 8 9]

Append elements along axis 0:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Append elements along axis 1:
[[1 2 3 5 5 5]
 [4 5 6 7 8 9]]
```

3. *np.insert( )*

This function inserts values in the input array along the given axis and before the given index. If the type of values is converted to be inserted, it is different from the input array. Insertion is not done in place and the function returns a new array. Also, if the axis is not mentioned, the input array is flattened.

The insert() function takes the following parameters −
numpy.insert(arr, obj, values, axis)

obj- The index before which insertion is to be made

values- The array of values to be inserted

axis- The axis along which to insert. If not given, the input array is flattened

```
import numpy as np

a = np.array([[1,2],[3,4],[5,6]])

print('First array:')

print(a)

print( '\n' )

print ('Axis parameter not passed. The input array is flattened before insertion.')

print (np.insert(a,3,[11,12]))

print( '\n' )

 print('Axis parameter passed. The values array is broadcast to match input array.')

print ('Broadcast along axis 0:')

print (np.insert(a,2,[11],axis = 0) )

print( '\n' )

 print('Broadcast along axis 1:' )

print (np.insert(a,2,11,axis = 1))
```

```
First array:
[[1 2]
```

```
[3 4]
[5 6]]
```

Axis parameter not passed. The input array is flattened before insertion.

```
[ 1  2  3 11 12  4  5  6]
```

Axis parameter passed. The values array is broadcast to match input array.
Broadcast along axis 0:

```
[[ 1  2]
 [ 3  4]
 [11 11]
 [ 5  6]]
```

Broadcast along axis 1:

```
[[ 1  2 11]
 [ 3  4 11]
 [ 5  6 11]]
```

4. np.delete()

This function returns a new array with the specified subarray deleted from the input array. As in case of insert() function, if the axis parameter is not used, the input array is flattened. The function takes the following parameters −

Numpy.delete(arr, obj, axis)

Obj- Can be a slice, an integer or array of integers, indicating the subarray to be deleted from the input array

Axis- The axis along which to delete the given subarray. If not given, arr is flattened

```
import numpy as np
a = np.arange(12).reshape(3,4)
print('First array:')
print(a)
print( '\n' )
print('Array flattened before delete operation as axis not used:' )
print( np.delete(a,5) )
print( '\n' )

print ('Column 2 deleted:'  )
print(np.delete(a,1,axis = 1))
print( '\n' )
print ('Column 2 deleted:'  )
print(np.delete(a,1,axis = 0))
print( '\n' )
print ('A slice containing alternate values from array deleted:' )
```

```
a = np.array([1,2,3,4,5,6,7,8,9,10])
print (np.delete(a, np.s_[::2]))
```

```
First array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Array flattened before delete operation as axis not used:
[ 0  1  2  3  4  6  7  8  9 10 11]

Column 2 deleted:
[[ 0  2  3]
 [ 4  6  7]
 [ 8 10 11]]

Column 2 deleted:
[[ 0  1  2  3]
 [ 8  9 10 11]]

A slice containing alternate values from array deleted:
[ 2  4  6  8 10]
```

5. numpy.unique( )

This function returns an array of unique elements in the input array. The function can be able to return a tuple of array of unique vales and an array of associated indices. Nature of the indices depend upon the type of return parameter in the function call.

numpy.unique(arr, return_index, return_inverse, return_counts)

```
import numpy as np
a = np.array([5,2,6,2,7,5,6,8,2,9])
print('First array:')
print(a)
print( '\n' )
print ('Unique values of first array:' )
u = np.unique(a)
print (u)
print( '\n' )
print ('Unique array and Indices array:' )
u,indices = np.unique(a, return_index = True)
```

```
print (indices)
print( '\n' )
```

First array:
[5 2 6 2 7 5 6 8 2 9]

Unique values of first array:
[2 5 6 7 8 9]

Unique array and Indices array:
[1 0 2 4 7 9]

Splitting Arrays

Sr.No.	Array & Description
1	split Splits an array into multiple sub-arrays
2	hsplit Splits an array into multiple sub-arrays horizontally (column-wise)
3	vsplit Splits an array into multiple sub-arrays vertically (row-wise)

1. numpy.split()

This function divides the array into subarrays along a specified axis. The function takes three parameters.

numpy.split(ary, indices_or_sections, axis)

ary -Input array to be split

indices_or_sections-Can be an integer, indicating the number of equal sized subarrays to be created from the input array. If this parameter is a 1-D array, the entries indicate the points at which a new subarray is to be created.

Axis- Default is 0

```
import numpy as np
a = np.arange(9)
print('First array:')
print(a)
print( '\n' )
print ('Split the array in 3 equal-sized subarrays:' )
b = np.split(a,3)
```

```
print(b)
print( '\n' )
print( 'Split the array at positions indicated in 1-D array:' )
b = np.split(a,[4,7])
print(b)
```

```
First array:
[0 1 2 3 4 5 6 7 8]

Split the array in 3 equal-sized subarrays:
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]

Split the array at positions indicated in 1-D array:
[array([0, 1, 2, 3]), array([4, 5, 6]), array([7, 8])]
```

The numpy.hsplit is a special case of split() function where axis is 0 indicating a horizontal split regardless of the dimension of the input array.

numpy.vsplit is a special case of split() function where axis is 1 indicating a vertical split regardless of the dimension of the input array. The following example makes this clear.

```
import numpy as np
a = np.arange(16).reshape(4,4)
print('First array:')
print(a)
print( '\n' )

print('Horizontal splitting:' )
b = np.hsplit(a,2)
print(b)
print( '\n' )

print('VERTICAL splitting:' )
c = np.vsplit(a,2)
print(c)
print( '\n' )
```

```
First array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
Horizontal splitting:
[array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]]), array([[ 2,  3],
       [ 6,  7],
```

```
      [10, 11],
      [14, 15]])]
VERTICAL splitting:
[array([[0, 1, 2, 3],
      [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
      [12, 13, 14, 15]])]
```

Joining Arrays

Sr.No.	Array & Description
1	concatenate Joins a sequence of arrays along an existing axis
2	stack Joins a sequence of arrays along a new axis
3	hstack Stacks arrays in sequence horizontally (column wise)
4	vstack Stacks arrays in sequence vertically (row wise)

Concatenate

Concatenation refers to joining. This function is used to join two or more arrays of the same shape along a specified axis. The function takes the following parameters.

numpy.concatenate((a1, a2, ...), axis)

a1,a2..-Sequence of arrays of the same type

axis- Axis along which arrays have to be joined. Default is 0

```
import numpy as np
a = np.array([[1,2],[3,4]])
print('First array:')
print (a)
print ('\n'  )
b = np.array([[5,6],[7,8]])
print ('Second array:' )
print (b)
print ('\n' )
# both the arrays are of same dimensions
```

```
print ('Joining the two arrays along axis 0:' )
print (np.concatenate((a,b)) )
print ('\n'  )
print ('Joining the two arrays along axis 1:' )
print (np.concatenate((a,b),axis = 1))
```

OUTPUT

```
First array:
[[1 2]
 [3 4]]


Second array:
[[5 6]
 [7 8]]

Joining the two arrays along axis 0:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]

Joining the two arrays along axis 1:
[[ 7 8]1 2 5 6]
 [3 4]
```

Numpy.stack()

This function joins the sequence of arrays along a new axis. This function has been added since NumPy version 1.10.0. Following parameters need to be provided.

Note − This function is available in *version 1.10.0* onwards.

numpy.stack(arrays, axis)

arrays: Sequence of arrays of the same shape

axis: Axis in the resultant array along which the input arrays are stacked

```
import numpy as np
a = np.array([[1,2],[3,4]])
print('First array:')
print (a)
print ('\n'  )
b = np.array([[5,6],[7,8]])
print ('Second array:' )
print (b)
print ('\n' )
print ('Stack the two arrays along axis 0:' )
print (np.stack((a,b),0) )
```

```
print ('\n'  )
print ('Stack the two arrays along axis 1:' )
print (np.stack((a,b),1))
```

OUTPUT

```
First array:
[[1 2]
 [3 4]]
Second array:
[[5 6]
 [7 8]]
Stack the two arrays along axis 0:
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
Stack the two arrays along axis 1:
[[[1 2]
  [5 6]]
 [[3 4]
  [7 8]]]
```

numpy.hstack

Variants of numpy.stack function to stack so as to make a single array horizontally.

```
import numpy as np
a = np.array([[1,2],[3,4]])

print('First array:')
print (a)
print ('\n'  )
b = np.array([[5,6],[7,8]])

print ('Second array:' )
print (b)
print ('\n' )

print ('Horizontal stacking:' )
c = np.hstack((a,b))
print (c)
print ('\n'  )
```

OUTPUT

```
First array:
[[1 2]
 [3 4]]

Second array:
[[5 6]
 [7 8]]

Horizontal stacking:
```

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

numpy.vstack

Variants of numpy.stack function to stack so as to make a single array vertically.

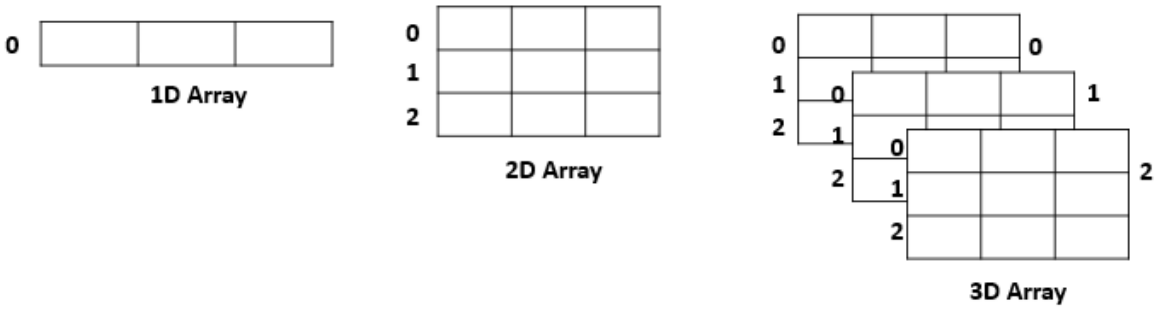
```
import numpy as np
a = np.array([[1,2],[3,4]])
print('First array:')
print (a)
print ('\n'  )
b = np.array([[5,6],[7,8]])

print ('Second array:' )
print (b)
print ('\n' )
print ('vertical stacking:' )
c = np.vstack((a,b))
print (c)
print ('\n'  )
```

OUTPUT

```
First array:
[[1 2]
 [3 4]]
Second array:
[[5 6]
 [7 8]]
Horizontal stacking:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

Ndarray

Ndarray is one of the most important classes in the NumPy python library. It is basically a multidimensional or n-dimensional array of fixed size with homogeneous elements(i.e., the data type of all the elements in the array is the same).



In Numpy, the number of dimensions of the array is given by Rank. Thus, in the above example, the ranks of the array of 1D, 2D, and 3D arrays are 1, 2 and 3 respectively.

***np*.ndarray(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)**

Here, the size and the number of elements present in the array is given by the shape attribute. The data type of the array(elements in particular) is given by the dtype attribute. Buffer attribute is an object exposing the buffer interface. An offset is the offset of the array data in the buffer. Stride attribute specifies the number of locations in the memory between the starting of successive array elements.

It should always be greater or equal to the size of the data type of the elements. Finally, the order attribute is to specify if we want a row-major or column-major order. Among all the above-mentioned attributes, shape and dtype are the compulsory ones. All other attributes are optional and can be specified on the requirement basis.

Working with Ndarray

An array can be created using the following functions :

- **np.ndarray(shape, type):** Creates an array of the given shape with random numbers.

- **np.array(array_object):** Creates an array of the given shape from the list or tuple.
- **np.zeros(shape):** Creates an array of the given shape with all zeros.
- **np.ones(shape):** Creates an array of the given shape with all ones.
- **np.full(shape,array_object, dtype):** Creates an array of the given shape with complex numbers.
- **np.arange(range):** Creates an array with the specified range.

An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

Arrays should be constructed using **array**, **zeros** or **empty** (refer to the See Also section below). The parameters given here refer to a low-level method (*ndarray(...)*) for instantiating an array.

For more information, refer to the **numpy** module and examine the methods and attributes of an array.

Parameters
(for the __new__ method; see Notes below)

- ✔ **shape*tuple of ints* -** Shape of created array.
- ✔ **dtype*data-type, optional-*** Any object that can be interpreted as a numpy data type.
- ✔ **buffer*object exposing buffer interface, optional-*** Used to fill the array with data.
- ✔ **offset*int, optional-*** Offset of array data in buffer.
- ✔ **strides*tuple of ints, optional-*** Strides of data in memory.
- ✔ **order*{'C', 'F'}, optional-*** Row-major (C-style) or column-major (Fortran-style) order.

There are two modes of creating an array using __new__:

1. If *buffer* is None, then only **shape**, **dtype**, and *order* are used.
2. If *buffer* is an object exposing the buffer interface, then all keywords are interpreted.

No __init__ method is needed because the array is fully initialized after the __new__ method.

```
import numpy as np

arr=np.ndarray(shape=(2,2), dtype=float, order='F',buffer=None)

arr=([[1.56, 3.43],[3.6987, 2.5323]])

print(arr)
[[1.56, 3.43], [3.6987, 2.5323]]
```

<class 'list'>

https://www.educba.com/numpy-ndarray/

print(type(arr))

Link: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html

Experiment-5

Mathematical Operations(np.add, np.substract, np.divide, np.multiply, np.sqrt, np.sin, np.cos, np.log, np.dot, np.roots) , Statistical Operations(np.mean, np.median, np.std, array.corrcoef( ))

NumPy - Arithmetic Operations

Input arrays for performing arithmetic operations such as add(), subtract(), multiply(), and divide() must be either of the same shape or should conform to array broadcasting rules.

```
# Python program explaining numpy.add() function  when inputs are scalar

import numpy as np
in_num1 = 10
in_num2 = 15

print ("1st Input number : ", in_num1)
print ("2nd Input number : ", in_num2)

out_num = np.add(in_num1, in_num2)
```

```
 Python program explaining arithmetic operations when inputs are not scalar
import numpy as np
a=np.arange(9, dtype=float).reshape(3,3)
print(a)
b=np.array([10,10,10])
print(b)
print(a+b)
print(np.add(a,b))
print(a-b)
print(np.subtract(a,b))
print(a*b)
print(np.multiply(a,b))
print(a/b)
print(np.divide(a,b))
```

Dot Product

This function returns the dot product of two arrays. For 2-D vectors, it is the equivalent to matrix multiplication. For 1-D arrays, it is the inner product of the vectors. For N-dimensional arrays, it is a sum product over the last axis of a and the second-last axis of b.

 # Python program explaining log() and dot () function

```
import numpy as np

print(np.log(2**8))

print(np.log(4**4))

a=np.array([[1,2],[3,4]])

b=np.array([[11,12],[13,14]])

print(np.dot(a,b))

x=10

y=20

print(np.dot(x,y))
```

```
5.545177444479562

5.545177444479562

[[37 40]

 [85 92]]

200
```

Note that the dot product is calculated as −

```
[[1*11+2*13, 1*12+2*14],[3*11+4*13, 3*12+4*14]]
```

numpy.roots()

Syntax: numpy.roots(p)

Parameter
It takes the coefficients of an given polynomial.
Return Value
The function will return the roots of the polynomial.
Let's do some code to understand.
Example 1:
Let us consider an equation: **x2 + 5*x + 6**
The coefficients are 1, 5 and 6.

```
#numpy.roots(p)

import numpy as np

p=[1,5,6]

roots=np.roots(p)

print(roots)
```

output

```
[-3. -2.]
```

Roots of three-degree polynomial

To find the roots of the three-degree polynomial we need to factorise the given polynomial equation first so that we get a linear and quadratic equation. Then, we can easily determine the zeros of the three-degree polynomial. Let us understand with the help of an example.

Example: $2x^3 - x^2 - 7x + 2$

Divide the given polynomial by x – 2 since it is one of the factors.

$2x^3 - x^2 - 7x + 2 = (x - 2)(2x^2 + 3x - 1)$

Now we can get the roots of the above polynomial since we have got one linear equation and one quadratic equation for which we know the formula.

Now let us consider the following polynomial for a cubic equation: x3 – 6 * x2 + 11 * x – 6

The coefficients are 1, -6 , 11 and -6.

```
#Third degree polynomial numpy.roots(p)
import numpy as np
p=[1,-6,11,-6]
roots=np.roots(p)
print(roots)
```

output

```
[3. 2. 1.]
```

Trigonometric Functions

NumPy has standard trigonometric functions which return trigonometric ratios for a given angle in radians.

```
#Trignometric funtions (Given angle in radians)
import numpy as np
a=np.array([0,30,45,60,90])
print("sine values")
print(np.sin(a*np.pi/180))
print("Tan values")
print(np.tan(a*np.pi/180))
print("Cosine values")
print(np.cos(a*np.pi/180))
```

output

```
sine values
[0.         0.5        0.70710678 0.8660254  1.        ]
Tan values
[0.00000000e+00 5.77350269e-01 1.00000000e+00 1.73205081e+00
 1.63312394e+16]
Cosine values
```

[1.00000000e+00 8.66025404e-01 7.07106781e-01 5.00000000e-01

6.12323400e-17]

NumPy - Statistical Functions

numpy.median()

Median is defined as the value separating the higher half of a data sample from the lower half. The numpy.median() function is used as shown in the following program.

Example

```
#numpy.median(--Separates upper half from lower half )

import numpy as np

a=np.array([[30,65,70],[80,95,10],[50,90,60]])

print(a)

print(np.median(a))

print(np.median(a,axis=0))

print(np.median(a,axis=1))
```

output

```
[[30 65 70]

 [80 95 10]

 [50 90 60]]

65.0

[50. 90. 60.]

[65. 80. 60.]
```

numpy.mean()

Arithmetic mean is the sum of elements along an axis divided by the number of elements. The numpy.mean() function returns the arithmetic mean of elements in the array. If the axis is mentioned, it is calculated along it.

Example

```
#numpy.mean(--Sum of elements along axis divided by number of elements )
import numpy as np
a=np.array([[1,2,3],[3,4,5],[4,5,6]])
print(a)
print(np.mean(a))
print(np.mean(a,axis=0))
print(np.mean(a,axis=1))
```

output

```
[[1 2 3]
 [3 4 5]
 [4 5 6]]
3.6666666666666665
[2.66666667 3.66666667 4.66666667]
[2. 4. 5.]
```

numpy.average()

Weighted average is an average resulting from the multiplication of each component by a factor reflecting its importance. The numpy.average() function computes the weighted average of elements in an array according to their respective weight given in another array. The function can have an axis parameter. If the axis is not specified, the array is flattened.

Considering an array [1,2,3,4] and corresponding weights [4,3,2,1], the weighted average is calculated by adding the product of the corresponding elements and dividing the sum by the sum of weights.

Weighted average = (1*4+2*3+3*2+4*1)/(4+3+2+1)

Example

```
#numpy.average() --Weighted average is average resulting from the #multiplication of each componenet by a factor reflecting its importance.

import numpy as np
a=np.array([1,2,3,4])
print(a)
print(np.average(a))
```

```
wt=np.array([4,3,2,1])
print(np.average(a,weights=wt))
wavg=np.average([1,2,3,4],weights=[4,3,2,1],returned=True)
print(wavg)
wavg1=np.average([1,2,3,4],weights=[4,3,2,1],axis=0,returned=True)
print(wavg1)
```

output

```
[1 2 3 4]
2.5
2.0
(2.0, 10.0)
(2.0, 10.0)
```

In a multi-dimensional array, the axis for computation can be specified.

Example

```
#Specifying axis for average in a multidimensional array
import numpy as np
a=np.arange(6).reshape(3,2)
print(a)
wt1=np.array([3,5])
print(np.average(a,axis=1,weights=wt1))
wt2=np.array([1,3,5])
print(np.average(a,axis=0,weights=wt2))
```

output

```
[[0 1]
 [2 3]
 [4 5]]
[0.625 2.625 4.625]
[2.88888889 3.88888889]
```

Standard Deviation

Standard deviation is the square root of the average of squared deviations from mean. The formula for standard deviation is as follows −

```
std = sqrt(mean(abs(x - x.mean())**2))
```

If the array is [1, 2, 3, 4], then its mean is 2.5. Hence the squared deviations are [2.25, 0.25, 0.25, 2.25] and the square root of its mean divided by 4, i.e., sqrt (5/4) is 1.1180339887498949.

Example

```
#Standard Deviation-Square root of average of squared deviations from mean
#std=sqrt(mean(abs(x-x.mean())**2))

import numpy as np
print(np.std([1,2,3,4]))
1.118033988749895
```

Variance

Variance is the average of squared deviations, i.e., mean(abs(x - x.mean())**2). In other words, the standard deviation is the square root of variance.

Example

```
#Varience is average of squared deviations from mean
#std=sqrt(mean(abs(x-x.mean())**2))

import numpy as np
print(np.var([1,2,3,4]))

1.25
```

Pearson Correlation Coefficient

It is most frequently used correlation metrics in machine learning or statistics.

$$\text{Pearson correlation} = \text{covariance}(x,y)/\text{std}(x) \text{ X } \text{std}(y)$$

Covariance is measure of variation between x and y variable. std(x) is standard deviation of variable x and std(y) is standard deviation of variable y.

```
#Import Libraries
import pandas as pd
import seaborn as sns

# Get the dataset from seaborn library
```

```
tips = sns.load_dataset('tips')

# Get pearson correlation coefficient
tip.corr(method='pearson')
# Get spearman correlation coefficient
tip.corr(method='spearman')
```

Experiment-8

NumPy String Operations

The following functions are used to perform vectorized string operations for arrays of dtype numpy.string_ or numpy.unicode_. They are based on the standard string functions in Python's built-in library.

Sr.No.	Function & Description

1	add() Returns element-wise string concatenation for two arrays of str or Unicode
2	multiply() Returns the string with multiple concatenation, element-wise
3	center() Returns a copy of the given string with elements centered in a string of specified length
4	capitalize() Returns a copy of the string with only the first character capitalized
5	title() Returns the element-wise title cased version of the string or unicode
6	lower() Returns an array with the elements converted to lowercase
7	upper() Returns an array with the elements converted to uppercase
8	split() Returns a list of the words in the string, using separatordelimiter

9	splitlines() Returns a list of the lines in the element, breaking at the line boundaries
10	strip() Returns a copy with the leading and trailing characters removed
11	join() Returns a string which is the concatenation of the strings in the sequence
12	replace() Returns a copy of the string with all occurrences of substring replaced by the new string

This function performs elementwise string concatenation.

```
import numpy as np

print 'Concatenate two strings:'
print np.char.add(['hello'],[' xyz'])
print '\n'

print 'Concatenation example:'
print np.char.add(['hello', 'hi'],[' abc', ' xyz'])
```

Its output would be as follows −

```
Concatenate two strings:
['hello xyz']

Concatenation example:
['hello abc' 'hi xyz']
```

This function performs multiple concatenations.

```
import numpy as np
print np.char.multiply('Hello ',3)
```

Its output would be as follows −

```
Hello Hello Hello
```

this function returns an array of the required width so that the input string is centered and padded on the left and right with fillchar.

```
import numpy as np
# np.char.center(arr, width,fillchar)
print np.char.center('hello', 20,fillchar = '*')
```

Here is its output −

```
*******hello********
```

this function returns the copy of the string with the first letter capitalized.

```
import numpy as np
print np.char.capitalize('hello world')
```

Its output would be −

```
Hello world
```

This function returns a title cased version of the input string with the first letter of each word capitalized.

```
import numpy as np
print np.char.title('hello how are you?')
```

Its output would be as follows −

```
Hello How Are You?
```

This function returns an array with elements converted to lowercase. It calls str.lower for each element.

```
import numpy as np
print np.char.lower(['HELLO','WORLD'])
print np.char.lower('HELLO')
```

Its output is as follows −

```
['hello' 'world']
hello
```

This function calls str.upper function on each element in an array to return the uppercase array elements.

```
import numpy as np
print np.char.upper('hello')
print np.char.upper(['hello','world'])
```

Here is its output −

```
HELLO
['HELLO' 'WORLD']
```

This function returns a list of elements in the array, breaking at line boundaries.

```
import numpy as np
print np.char.splitlines('hello\nhow are you?')
print np.char.splitlines('hello\rhow are you?')
```

Its output is as follows −

```
['hello', 'how are you?']
['hello', 'how are you?']
```

'\n', '\r', '\r\n' can be used as line boundaries.

This function returns a copy of array with elements stripped of the specified characters leading and/or trailing in it.

```
import numpy as np
print np.char.strip('ashok arora','a')
print np.char.strip(['arora','admin','java'],'a')
```

Here is its output −

```
shok aror
['ror' 'dmin' 'jav']
```

This method returns a string in which the individual characters are joined by separator character specified.

```
import numpy as np
print np.char.join(':','dmy')
print np.char.join([':','-'],['dmy','ymd'])
```

Its output is as follows −

```
d:m:y
['d:m:y' 'y-m-d']
```

This function returns a new copy of the input string in which all occurrences of the sequence of characters is replaced by another given sequence.

```
import numpy as np
print np.char.replace ('He is a good boy', 'is', 'was')
```

Its output is as follows −

```
He was a good boy
```

Experiment-9
Numpy financial functions

---

fv(rate, nper, pmt, pv[, when])	Compute the future value.
npv(rate, values)	Returns the NPV (Net Present Value) of a cash flow series.

fv(rate, nper, pmt, pv[, when])	Compute the future value.
pmt(rate, nper, pv[, fv, when])	Compute the payment against loan principal plus interest.
ppmt(rate, per, nper, pv[, fv, when])	Compute the payment against the loan principal.
pv(rate, nper, pmt[, fv, when])	Compute the present value.

The fv() function is used to compute the future value.

Notes:

The future value is computed by solving the equation:

fv +
pv*(1+rate)**nper +
pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) == 0

or, when rate == 0:

fv + pv + pmt * nper == 0

NumPy.fv() method Example-1:

What is the future value after 10 years of saving $200 now, with an additional monthly savings of $200. Assume the interest rate is 6% (annually) compounded monthly?

```
import numpy_financial as npf
a=npf.fv(0.06/12, 10*12, -200, -200)
print(a)
```

output

33139.748708098065

```
import numpy_financial as npf
b=npf.pv(0.06/12, 10*12, -200, 33139.748708098065)
print(b)
```

output

-200.00000000000185

NumPy.fv() method Example-2:

By convention, the negative sign represents cash flow out (i.e. money not available today). Thus, saving $200 a month at 6% annual interest leads to $33,139.75 available to spend in 10 years.

If any input is array_like, returns an array of equal shape. Let's compare different interest rates from the example above.

```
import numpy_financial as npf
import numpy as np
x = np.array((0.04, 0.06, 0.07))/12
a=npf.fv(x, 10*12, -200, -200)
print(a)
b=npf.pv(x, 10*12, -200, 33139.75)
print(b)
```

output

```
[29748.12748158    33139.7487081    35018.89376205]
[-2474.98535444     -200.00071007        735.05492211]
```

npv() function

The npv() function is used to get the NPV (Net Present Value) of a cash flow series.

Syntax:

numpy.npv(rate, values)

values The values of the time series of cash flows. The (fixed) time interval between cash flow "events" must be the same as that for which rate is given (i.e., if rate is per year, then precisely a year is understood to elapse between each cash flow event). By convention, investments or "deposits" are negative, income or "withdrawals" are positive; values must begin with the initial investment, thus values[0] will typically be negative. array_like, shape(M,)

```
import numpy_financial as np
a=np.npv(0.281,[ -200,39, 59, 55, 20])
print(a)
print(np.npv(0.281,[-100, 39, 59, 55, 20]))
```

output

```
-100.00847859163845
-0.00847859163845488
```

```
import numpy_financial as np
#declaring values
values = [-100,10, 10,10]
rate1 = 0.50
rate2 = 0.30
rate3 = 1
#Printing NPV Values
print("NPV value with rate ", rate1, " is: ", np.npv(rate1, values))
print("NPV value with rate ", rate2, " is: ", np.npv(rate2, values))
print("NPV value with rate ", rate3, " is: ", np.npv(rate3, values))
NPV value with rate  0.5  is:  -85.92592592592592
NPV value with rate  0.3  is:  -81.83887118798361
NPV value with rate  1  is:  -91.25
```

pmt() function

The pmt() function is used to compute the payment against loan principal plus interest.

Syntax:

numpy.pmt(rate, nper, pv, fv=0, when='end')
The ppmt() function is used to compute the payment against loan principal.

Syntax:

numpy.ppmt(rate, per, nper, pv, fv=0.0, when='end')

```
import numpy_financial as npf
print(npf.pmt(0.085/12,12*12, 100000))
print( npf.ppmt(0.085/12,1, 12*12, 100000))
```

output

```
-1110.0555643145096
-401.72223098117627
```

Experiment-10
Functional programming in Numpy

Functional programming

apply_along_axis(func1d, axis, arr, *args, ...)	Apply a function to 1-D slices along the given axis.
apply_over_axes(func, a, axes)	Apply a function repeatedly over multiple axes.
vectorize(pyfunc[, otypes, doc, excluded, ...])	Generalized function class.

Examples

```
>>> def my_func(a):
...     """Average first and last element of a 1-D array"""
...     return (a[0] + a[-1]) * 0.5

>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])

>>> np.apply_along_axis(my_func, 0, b)
array([4., 5., 6.])

>>> np.apply_along_axis(my_func, 1, b)
array([2.,  5.,  8.])
```

Sorting by along axis:

```
import numpy as np
y = np.array([[2,4,6], [1,3,5], [9,7,8]])
print(y)
print('\n')
print(np.apply_along_axis(sorted, 1, y))
print('\n')
print(np.apply_along_axis(sorted, 0, y))
[[2 4 6]
 [1 3 5]
 [9 7 8]]

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

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

Indexing

```
import numpy as np
x = np.arange(8).reshape(2,2,2)
```

```
print(x)
print('\n')
a=x[1,1,0]

print(a)

print('\n')

b=x[:,0,0]

print(b)

c=x[1]

print('\n')

print(c)

d=x[1,:,:]
```

output

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

A DataFrame object has two axes: “axis 0” and “axis 1”. “axis 0” represents rows and “axis 1” represents columns. Now it's clear that Series and DataFrame share the same direction for “axis 0” – it goes along rows direction.

```
import numpy as np

y = np.array([[2,4,6], [1,3,5], [9,7,8]])

print(np.apply_along_axis(np.diag, -1, y))

[[[2 0 0]

 [0 4 0]

 [0 0 6]]

[[1 0 0]

 [0 3 0]

 [0 0 5]]



[[9 0 0]

 [0 7 0]

 [0 0 8]]]
```

Adding elements along axis

```
import numpy as np

x = np.arange(6).reshape(3,2)

print(x)

print('\n')

print(np.apply_along_axis(np.sum,0,x))
```

```
print('\n')

print(np.apply_along_axis(np.sum,1,x))

[[0 1]

 [2 3]

 [4 5]]




[6 9]

[1 5 9]
```

Addition over Axis:

```
import numpy as np

x = np.arange(24).reshape(3,2,4)

print(x)

print(np.apply_over_axes(np.sum, x, [0,0]))

print(np.apply_over_axes(np.sum, x, [0,1]))

print(np.apply_over_axes(np.sum, x, [0,2]))

print(np.apply_over_axes(np.sum, x, [2,0]))

print(np.apply_over_axes(np.sum, x, [1,0]))

[[[ 0  1  2  3]

  [ 4  5  6  7]]
```

```
 [[ 8  9 10 11]
  [12 13 14 15]]

 [[16 17 18 19]
  [20 21 22 23]]]
[[[24 27 30 33]
  [36 39 42 45]]]
[[[60 66 72 78]]]
[[[114]
  [162]]]
[[[114]
  [162]]]
[[[60 66 72 78]]]
```

```
import numpy as np
x = np.arange(27).reshape(3,3,3)
print(x)
print(np.apply_over_axes(np.sum, x, [0,0]))
print(np.apply_over_axes(np.sum, x, [0,1]))
print(np.apply_over_axes(np.sum, x, [0,2]))
```

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

 [[ 9 10 11]
 [12 13 14]
 [15 16 17]]

 [[18 19 20]
 [21 22 23]
 [24 25 26]]]
[[[27 30 33]
 [36 39 42]
 [45 48 51]]]
[[[108 117 126]]]
[[[ 90]
 [117]
 [144]]]
```

vectorize(fun):

```
import numpy as np
def fun(a, b):
```

```
  If a > b return a + b,

  else return a - b.

  if a >= b:

     return a + b

  else:

     return a - b

# Create a vectorized version of foo

vecfun = np.vectorize(fun)

print(vecfun(np.arange(5), 5))

print(vecfun(np.arange(5), [1,2,3,4,5]))

print(vecfun([[2,4,6,1]], [1,2,3,4]))

[-5 -4 -3 -2 -1]

[-1 -1 -1 -1 -1]

[[ 3  6  9 -3]]
```

Experiment-11

Write a Python program for the following

- Importing matplotlib,
- Simple Line Plots,
- Adjusting the Plot: Line Colors and Styles, Axes Limits,
- Labeling Plots,
- Simple Scatter Plots,
- Histograms,
- Customizing Plot Legends,
- Choosing Elements for the Legend,
- Multiple Legends,
- Customizing Colorbars,
- Multiple Subplots,
- Text and Annotation,
- Customizing Tick.

Simple Line Plot:

```
# importing matplotlib module
from matplotlib import pyplot as plt

# x-axis values
x = [5, 2, 9, 4, 7]

# Y-axis values
y = [10, 5, 8, 4, 2]

# Function to plot
plt.plot(x, y)

# function to show the plot
plt.show()
```

BAR CHART

```
# importing matplotlib module
from matplotlib import pyplot as plt

# x-axis values
x = [5, 2, 9, 4, 7]
```

```
# Y-axis values
y = [10, 5, 8, 4, 2]

# Function to plot
plt.bar(x, y)

# function to show the plot
plt.show()
```

#HISTOGRAM

```
# importing matplotlib module
from matplotlib import pyplot as plt

# Y-axis values
y = [10, 5, 8, 4, 2]

# Function to plot histogram
plt.hist(y)

# Function to show the plot
plt.show()
```

SCATTER PLOT

```
# importing matplotlib module
from matplotlib import pyplot as plt

# x-axis values
x = [5, 2, 9, 4, 7]

# Y-axis values
y = [10, 5, 8, 4, 2]

# Function to plot scatter
plt.scatter(x, y)

# function to show the plot
plt.show()
```

Labeling Plots

```
# importing matplotlib module
from matplotlib import pyplot as plt

# x-axis values
x = [5, 2, 9, 4, 7]

# Y-axis values
y = [10, 5, 8, 4, 2]

# Function to plot
plt.scatter(x, y)

# Adding Title
```

```
plt.title("Data Analysis and Visualization using PYTHON")

# Labeling the axes
plt.xlabel("Time (hr)")
plt.ylabel("Position (Km)")
plt.xlim(1,10)
plt.ylim(1,10)
# function to show the plot
plt.show()
```

#Colors,Font sizes

```
# importing matplotlib module
from matplotlib import pyplot as plt

# x-axis values
x = [5, 2, 9, 4, 7]

# Y-axis values
y = [10, 5, 8, 4, 2]

# Function to plot
plt.scatter(x, y)
font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}

# Adding Title
plt.title("D", fontdict = font1,loc='left')
plt.title("A", fontdict = font1,loc='center')
plt.title("V", fontdict = font1,loc='right')
# Labeling the axes
plt.xlabel("Time (hr)", fontdict = font2)
plt.ylabel("Position (Km)", fontdict = font2)
plt.xlim(1,10)
plt.ylim(1,10)
# function to show the plot
plt.show()
```

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X

'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o')
plt.show()

#plt.plot(ypoints, marker = '*')
```

You can use also use the *shortcut string notation* parameter to specify the marker.

This parameter is also called fmt, and is written with this syntax:

marker*|*line*|*color

'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

#plt.plot(ypoints, marker = 'o')
#plt.plot(ypoints, '*:r')
plt.plot(ypoints,'o-b')
plt.show()
```

'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Marker Size

You can use the keyword argument markersize or the shorter version, ms to set the size of the markers:

Set the EDGE color to red:

```
import matplotlib.pyplot as plt
import numpy as np
```

```
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r')
plt.show()
```

Try various options in this link: https://www.w3schools.com/python/matplotlib_markers.asp

```
import numpy as np
import matplotlib.pyplot as plt

# X-axis values
x = [1, 2, 3, 4, 5]

# Y-axis values
y = [1, 4, 9, 16, 25]

# Function to plot
plt.plot(x, y)

# Function add a legend
plt.legend(['single element'])

# function to show the plot
plt.show()

# importing modules
import numpy as np
import matplotlib.pyplot as plt

# Y-axis values
y1 = [2, 3, 4.5]

# Y-axis values
y2 = [1, 1.5, 5]

# Function to plot
plt.plot(y1)
plt.plot(y2)

# Function add a legend
plt.legend(["blue", "green"], loc ="lower right")

# function to show the plot
plt.show()


import numpy as np
import matplotlib.pyplot as plt

# X-axis values
x = np.arange(5)
```

```
# Y-axis values
y1 = [1, 2, 3, 4, 5]

# Y-axis values
y2 = [1, 4, 9, 16, 25]

# Function to plot
plt.plot(x, y1, label ='Numbers')
plt.plot(x, y2, label ='Square of numbers')

# Function add a legend
plt.legend()

# function to show the plot
plt.show()


# importing modules
import numpy as np
import matplotlib.pyplot as plt

# X-axis values
x = [0, 1, 2, 3, 4, 5, 6, 7, 8]

# Y-axis values
y1 = [0, 3, 6, 9, 12, 15, 18, 21, 24]
# Y-axis values
y2 = [0, 1, 2, 3, 4, 5, 6, 7, 8]

# Function to plot
plt.plot(y1, label ="y = x")
plt.plot(y2, label ="y = 3x")

# Function add a legend
plt.legend(bbox_to_anchor =(0.75, 1.15), ncol = 2)

# function to show the plot
plt.show()
```

#Multiple Legends

```
from matplotlib import pyplot as plt


plt.rcParams["figure.figsize"] = [7.50, 3.50]
plt.rcParams["figure.autolayout"] = True


line1, = plt.plot([1, 2, 3], label="Line 1", linestyle='--')
line2, = plt.plot([3, 2, 1], label="Line 2", linewidth=4)
```

```
line3, = plt.plot([5, 3, 1], label="Line 3", linewidth=9,linestyle=':')
first_legend = plt.legend(handles=[line1], loc='upper right')

plt.gca().add_artist(first_legend)
second_legend = plt.legend(handles=[line1], loc='center')

plt.gca().add_artist(second_legend)
plt.legend(handles=[line2], loc='lower right')
#plt.legend(handles=[line3], loc='Middle right')
plt.show()
```

Custom Legends with Matplotlib

In this , you learn to customize the legend in matplotlib. matplotlib is a popular data visualization library. It is a plotting library in python and has its numerical extension NumPy.

Legend is an area of the graph describing each part of the graph. A graph can be simple as it is. But adding the title, X label, Y label and legend will be more clear. By seeing the names we can easily guess what the graph is representing and what type of data it is representing.

Let us first see how to create a legend in matplotlib.

> **syntax:** legend(*args, **kwargs)

This can be called as follows,

> **legend() ->** automatically detects which element to show. It does this by displaying all plots that have been labeled with the label keyword argument.
>
> **legend(labels) ->** Name of X and name of Y that is displayed on the legend
>
> **legend(handles, labels) ->** A list of lines that should be added to the legend. Using handles and labels together can give full control of what should be displayed in the legend. The length of the legend and handles should be the same.

```
# importing library
import matplotlib.pyplot as plt

# plotting values
a = [1, 2, 3, 4]
b = [1, 4, 9, 16]

# PLotting using matplotlib
plt.plot(a, label="A")
plt.plot(b, label="B")

# Creating legend
plt.legend()
```

Output:

legend

Customizing the legend

Legend adds meaning to the graph plots. Adding the font, location, and many more, make the legend more legible and easily recognizable. Location

Sometimes the legend may or may not be in the appropriate place. In matplotlib, we can also add the location where we want to place it. With this flexibility, we can place the legend somewhere where it does not overlay the plots, and hence the plots will look much cleaner and tidier.

Syntax: legend(loc=”)

It can be passed as follows,

‘upper left’, ‘upper right’, ‘lower left’, ‘lower right’ -> It is placed on the corresponding corner of the plot.

‘upper center’, ‘lower center’, ‘center left’, ‘center right’ -> It is placed on the center of corresponding edge.

‘center’ -> It is placed exact center of the plot.

‘best’ -> It is placed without the overlapping of the artists

For example,

```
# importing library
import matplotlib.pyplot as plt

# plotting values
a = [1, 2, 3, 4]
b = [1, 4, 9, 16]

# PLotting using matplotlib
plt.plot(a, label="A")
plt.plot(b, label="B")

# Creating legend
# Adding the location
plt.legend(loc='center left')
```

Output:

center left

Font size

To make the legend more appealing we can also change the font size of the legend, by passing the parameter font size to the function we can change the fontsize inside the legend box just like the plot titles.

Syntax: legend(fontsize=”)

It can be passed as, ‘xx-small’, ‘x-small’, ‘small’, ‘medium’, ‘large’, ‘x-large’, ‘xx-large’

Example:

```
# importing library
import matplotlib.pyplot as plt

# plotting values
a = [1, 2, 3, 4]
b = [1, 4, 9, 16]

# PLotting using matplotlib
plt.plot(a, label="label1")
plt.plot(b, label="label2")

# Creating legend
plt.legend(fontsize='xx-large')
```

Output:

Changing font size

Color of the legend

Sometimes we can feel that it would be great if the legend box was filled with some color to make it more attractive and makes the legends stand out from the plots. Matplotlib also covers this by letting us change the theme of the legend by changing the background, text, and even the edge color of the legend.-+

Syntax:

legend(labelcolor=”)

labelcolor is used to change the color of the text.

legend(facecolor=”)

facecolor is used to change background color of the legend.

legend(edgecolor=”)

edgecolor is used to change the edge color of the legend

Example:

```
# importing library
import matplotlib.pyplot as plt

# plotting values
a = [1, 2, 3, 4]
b = [1, 4, 9, 16]

# PLotting using matplotlib
plt.plot(a, label="label1")
plt.plot(b, label="label2")

# Creating legend
# Adding color to the legend
plt.legend(labelcolor='white', facecolor='black',
        edgecolor='red', fontsize='xx-large')
```

Experiment-12

A) Pandas DataSeries:

1) Write a Pandas program to create and display a one-dimensional array-like object containing an array of data using Pandas module.

Pandas deals with the following three data structures.
1)series
2)Data frames
3)Panel These three data structures are faster than numpy array.

Series: It is a one-dimensional array like structure with homogeneous data.For example the following series is a collection of integers

```
10   23   56   17   52   61   73   90   14
```

Dataframe: Dataframe is a two dimensional array with heterogeneous data.For ex,

Name	Age	Gender	Rating
steve	32	male	3.45
Lia	28	female	4.6
Vin	45	male	3.9
katie	38	female	2.78

Panel: Panel is a three dimensional data structure with heterogeneous data . A panel is a container of dataframe.

Series: A Series can be created using various inputs like
1)Array
2)Dict
3)Scalar value or constant

Create an empty Series: A basic series which can be created is an empty series.

```
import pandas as pd
s=pd.Series()
print(s)
```

Series([], dtype: float64)

Creating a series from ndarray:

If data is an ndarray then index passed must be of the same length.If no index is passed ,then by default index will be range(n)

```
import pandas as pd
import numpy as np
data=np.array(['a','b','c','d'])  # no index is passed.So index ranges from 0 to 3
s=pd.Series(data)
print(s)
```

```
0    a
1    b
2    c
3    d
dtype: object
```

```
import pandas as pd
import numpy as np
data=np.array(['a','b','c','d'])
s=pd.Series(data,index=[100,101,102,103])  # index is passed.
print(s)
```

```
100    a
101    b
102    c
103    d
dtype: object
```

Create a Series from Dict: A dict can be passed as input and if no index is specified,then the dict keys are taken in a sorted order to construct index.If index is passed the values in data corresponding to the labels in the index will be pulled out.

```
data={'a':0,'b':1,'c':2}
s=pd.Series(data)
print(s)
```

```
a    0
b    1
c    2
dtype: int64
```

```
data={'a':0,'b':1,'c':2}
s=pd.Series(data,index=['b','c','d','a'])
print(s)
```

```
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Creating a Series from scalar: If data is a scalar value, an index must be provided. The value will be repeated to match the length of index.

```
s=pd.Series(5,index=[0,1,2])
print(s)
```

```
0    5
1    5
2    5
dtype: int64
```

Accessing data from series with position: counting strats from zero for the array, which means that the first element is stored at zeroth position and so on.

```
s=pd.Series([1,2,3,4],index=['a','b','c','d'])
print(s['a'])
print(s[:3])
print(s[-3:])
```

```
1
a    1
b    2
c    3
```

```
dtype: int64
b    2
c    3
d    4
dtype: int64
```

Retrieve data using label(index):

```
s=pd.Series([1,2,3,4],index=['a','b','c','d'])
print(s['a']) # retrive a single element using index label value
print(s[['a','b','d']])  #retrive  multiple elements using index label value
```

```
1
a    1
b    2
d    4
dtype: int64
```

2) Write a Pandas program to convert a Panda module Series to Python list and it's type.

```
import pandas as pd
import numpy as np
data=np.array(['a','b','c','d'])  # no index is passed.So index ranges from 0 to 3
s=pd.Series(data)
print(s)
l1=list(s)
print(type(l1))
```

output:

```
0    a
1    b
2    c
3    d
dtype: object
<class 'list'>
```

```
import numpy as np
s1=pd.Series(np.random.rand(4),index=['a','b','c','d'])
s2=pd.Series(np.arange(4),index=['a','b','c','d'])

print(s1)
print(s2)
```

```
a    0.492583
b    0.322283
c    0.722065
d    0.467811
dtype: float64
a    0
b    1
c    2
d    3
dtype: int32
```

```
#Missing data: The library isnull() is used to detect missing data.
s=pd.Series({'001':'Nam','002':'mary','003':'peter'},
          index=['002','001','024','065'])
print(s)
pd.isnull(s)
002    mary
001     Nam
024     NaN
065     NaN
dtype: object
```

Out[10]:

```
002    False
001    False
024     True
065     True
dtype: bool
```

3) Write a Pandas program to add, subtract, multiple and divide two Pandas Series.

```
import pandas as pd
ds1 = pd.Series([2, 4, 6, 8, 10])
ds2 = pd.Series([1, 3, 5, 7, 9])
ds = ds1 + ds2
print("Add two Series:")
print(ds)
print("Subtract two Series:")
ds = ds1 - ds2
print(ds)
print("Multiply two Series:")
ds = ds1 * ds2
```

```
print(ds)
print("Divide Series1 by Series2:")
ds = ds1 / ds2
print(ds)
```

```
Add two Series:
0     3
1     7
2    11
3    15
4    19
dtype: int64
Subtract two Series:
0    1
1    1
2    1
3    1
4    1
dtype: int64
Multiply two Series:
0     2
1    12
2    30
3    56
4    90
dtype: int64
Divide Series1 by Series2:
0    2.000000
1    1.333333
2    1.200000
3    1.142857
4    1.111111
dtype: float64
```

Dataframe: A dataframe is a two dimensionals data structure. A pandas Dataframe can be created using the following constructor.

pandas.DataFrame(data,index,columns,dtype,copy)

here, data-data takes various forms like,ndaray,series,map,lists,dict,constants and also another dataframe.

index-for the row labels,the index to be used for the resulting frame is optional default. np.arrange(n) if no index is passed.

columns- for columns labels the optional default syntax is np.arange(n) if no index is passed.

dtype- datatype of each column

copy-This command is used for copying of data,default is false.

Create Dataframe: A pandas dataframe can be created using various inputs like
1)lists 2) Dict 3) Series 4) Numpy ndarrays. 5) Another dataframe

Create an empty DataFrame: A basic Dataframe,which can be created is an empty Dataframe.

```
import pandas as pd
df=pd.DataFrame()
print(df)
```

```
Empty DataFrame
Columns: []
Index: []
```

```
Creating dataframe from lists:
        The dataframe canbe created using a single list or a list of lists
import pandas  as pd
data=[1,2,3,4,5]
df=pd.DataFrame(data,index=[4,5,6,7,8])
print(df)
```

```
   0
4  1
5  2
6  3
7  4
8  5
```

```
import pandas  as pd
data=[1,2,3,4,5]
df=pd.DataFrame(data,index=['a','b','c','d','e'],columns=['x'])
print(df)
```

```
   x
a  1
b  2
c  3
d  4
```

e 5

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print(df)
```

```
    Name  Age
0   Alex   10
1    Bob   12
2 Clarke   13
```

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print(df)
```

```
    Name   Age
0   Alex  10.0
1    Bob  12.0
2 Clarke  13.0
```

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,index=['s1','s2','s3'],columns=['Name','Age'])
print(df)
```

```
     Name  Age
s1   Alex   10
s2    Bob   12
s3 Clarke   13
```

Creating a DataFrame from Dict of ndarray/Lists:

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
```

```
df = pd.DataFrame(data)
print(df)
```

```
   Name  Age
0   Tom   28
1  Jack   34
2 Steve   29
3 Ricky   42
```

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data,index=['rank1','rank2','rank3','rank4'])
print(df)
```

```
       Name  Age
rank1   Tom   28
rank2  Jack   34
rank3 Steve   29
rank4 Ricky   42
```

Create a DataFrame from list of Dicts: List of Dictionries can be passed as input data to create a DataFrame. The Dictionary keys are by default taken as column names

```
data=[{'a':1,'b':2},{'a':5,'b':10,'c':20}]
df=pd.DataFrame(data)
print(df)
```

```
   a   b     c
0  1   2   NaN
1  5  10  20.0
```

```
data=[{'a':1,'b':2},{'a':5,'b':10,'c':20}]
df=pd.DataFrame(data,index=['first','second'])
print(df)
print(df.index)   #gives index labels
print(df.c)  #or df['c'] gives particular label
```

```
        a   b     c
first   1   2   NaN
second  5  10  20.0
```

```
Index(['first', 'second'], dtype='object')
first      NaN
second    20.0
Name: c, dtype: float64
```

```
data=[{'a':1,'b':2},{'a':5,'b':10,'c':20}]
df1=pd.DataFrame(data,index=['first','second'],columns=['a','b'])
df2=pd.DataFrame(data,index=['first','second'],columns=['a','b1'])
print(df1)
print(".....")
print(df2)
```

```
        a   b
first   1   2
second  5  10
.....
        a  b1
first   1 NaN
second  5 NaN
```

4) Write a Pandas program to convert a NumPy array to a Pandas series.
Sample Series:

```
import numpy as np
import pandas as pd
np_array = np.array([10, 20, 30, 40, 50])
 print("NumPy array:")
print(np_array)
 new_series = pd.Series(np_array) print("Converted Pandas series:")
print(new_series)
```

```
NumPy array:
[10 20 30 40 50]

Converted Pandas series:
0 10
1 20
```

```
2 30
3 40
4 50
dtype: int64
```

B) Pandas DataFrames:

Consider Sample Python dictionary data and list labels:

```
exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',
'Matthew', 'Laura', 'Kevin', 'Jonas'],
'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'  qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

1) Write a Pandas program to create and display a DataFrame from a specified dictionary data which has the index labels.

```
import pandas as pd
ed = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',
'Matthew', 'Laura', 'Kevin', 'Jonas'],
'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}

df=pd.DataFrame(ed,index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'])

print(df)
```

```
      name  score  attempts qualify
a  Anastasia   12.5         1     yes
b       Dima    9.0         3      no
c  Katherine   16.5         2     yes
d      James    NaN         3      no
e      Emily    9.0         2      no
f    Michael   20.0         3     yes
g    Matthew   14.5         1     yes
h      Laura    NaN         1      no
i      Kevin    8.0         2      no
j      Jonas   19.0         1     yes
```

2) Write a Pandas program to change the name 'James' to 'Suresh' in name column of the DataFrame.

```
import pandas as pd
ed = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',
'Matthew', 'Laura', 'Kevin', 'Jonas'],
'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
```

```
'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}

df=pd.DataFrame(ed,index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'])

print(df)
print("\nChange the name 'James' to 'Suresh':")
df['name'] = df['name'].replace('James', 'Suresh')
print(df)
```

3) Write a Pandas program to insert a new column in existing DataFrame.

```
import pandas as pd
import numpy as np
exam_data  = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew',
'Laura', 'Kevin', 'Jonas'],
'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19], 'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
df = pd.DataFrame(exam_data , index=labels)
print("Original rows:")
print(df)
color = ['Red','Blue','Orange','Red','White','White','Blue','Green','Green','Red']
df['color'] = color
print("\nNew DataFrame after inserting the 'color' column")
print(df)
```

```
New DataFrame after inserting the 'color' column
   attempts      name qualify  score   color
a         1  Anastasia     yes   12.5     Red
b         3       Dima      no    9.0    Blue
c         2  Katherine     yes   16.5  Orange
d         3      James      no    NaN     Red
e         2      Emily      no    9.0   White
f         3    Michael     yes   20.0   White
g         1    Matthew     yes   14.5    Blue
h         1      Laura      no    NaN   Green
i         2      Kevin      no    8.0   Green
j         1      Jonas     yes   19.0     Red
```

4) Write a Pandas program to get list from DataFrame column headers.

```
import pandas as pd
import numpy as np
exam_data  = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew',
'Laura', 'Kevin', 'Jonas'],
'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19], 'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
```

```
'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
df = pd.DataFrame(exam_data , index=labels)
print("Original rows:")
print(df)
color = ['Red','Blue','Orange','Red','White','White','Blue','Green','Green','Red']
df['color'] = color
print("\nNew DataFrame after inserting the 'color' column")
print(df)
print(list(df.columns.values))
print(list(df.index.values))
print(list(df.color.values))
```

C) Pandas Index:

1) Write a Pandas program to display the default index and set a column as an Index in a given dataframe.

```
import pandas as pd
df = pd.DataFrame({
   'school_code': ['s001','s002','s003','s001','s002','s004'],
   'class': ['V', 'V', 'VI', 'VI', 'V', 'VI'],
   'name': ['Alberto Franco','Gino Mcneill','Ryan Parkes', 'Eesha Hinton', 'Gino Mcneill', 'David Parkes'],
   'date_Of_Birth': ['15/05/2002','17/05/2002','16/02/1999','25/09/1998','11/05/2002','15/09/1997'],
   'weight': [35, 32, 33, 30, 31, 32],
   'address': ['street1', 'street2', 'street3', 'street1', 'street2', 'street4'],
   't_id':['t1', 't2', 't3', 't4', 't5', 't6']})
print("Default Index:")
print(df.head(10))
print("\nt_id as new Index:")
df1 = df.set_index('t_id')
print(df1)
print("\nReset the index:")
df2 = df1.reset_index(inplace=False)
print(df2)
```

2) Write a Pandas program to create an index labels by using 64-bit integers, using floating-point numbers in a given dataframe

```
import pandas as pd
print("Create an Int64Index:")
df_i64 = pd.DataFrame({
   'school_code': ['s001','s002','s003','s001','s002','s004'],
   'class': ['V', 'V', 'VI', 'VI', 'V', 'VI'],
   'name': ['Alberto Franco','Gino Mcneill','Ryan Parkes', 'Eesha Hinton', 'Gino Mcneill', 'David Parkes'],
```

```
   'date_Of_Birth':
['15/05/2002','17/05/2002','16/02/1999','25/09/1998','11/05/2002','15/09/1997'],
   'weight': [35, 32, 33, 30, 31, 32],
   'address': ['street1', 'street2', 'street3', 'street1', 'street2', 'street4']},
   index=[1, 2, 3, 4, 5, 6])
print(df_i64)
print("\nView the Index:")
print(df_i64.index)

print("\nFloating-point labels using Float64Index:")
df_f64 = pd.DataFrame({
   'school_code': ['s001','s002','s003','s001','s002','s004'],
   'class': ['V', 'V', 'VI', 'VI', 'V', 'VI'],
   'name': ['Alberto Franco','Gino Mcneill','Ryan Parkes', 'Eesha Hinton', 'Gino Mcneill',
'David Parkes'],
   'date_Of_Birth ':
['15/05/2002','17/05/2002','16/02/1999','25/09/1998','11/05/2002','15/09/1997'],
   'weight': [35, 32, 33, 30, 31, 32],
   'address': ['street1', 'street2', 'street3', 'street1', 'street2', 'street4']},
   index=[.1, .2, .3, .4, .5, .6])
print(df_f64)
print("\nView the Index:")
print(df_f64.index)
```

D) Pandas String and Regular Expressions:

1) Write a Pandas program to convert all the string values to upper, lower cases in agivenpandas series. Also find the length of the string values.

```
import pandas as pd
import numpy as np
s = pd.Series(['X', 'Y', 'Z', 'Aaba', 'Baca', np.nan, 'CABA', None, 'bird', 'horse', 'dog'])
print("Original series:")
print(s)
print("\nConvert all string values of the said Series to upper case:")
print(s.str.upper())
print("\nConvert all string values of the said Series to lower case:")
print(s.str.lower())
print("\nLength of the string values of the said Series:")
print(s.str.len())
```

2) Write a Pandas program to remove whitespaces, left sided whitespaces and right sided whitespaces of the string values of a given pandas series.

```
import pandas as pd
color1 = pd.Index(['     Green', 'Black ', ' Red ', 'White', ' Pink      '])
print("Original series:")
print(color1)
print("\nRemove whitespace")
```

```
print(color1.str.strip())
print("\nRemove left sided whitespace")
print(color1.str.lstrip())
print("\nRemove Right sided whitespace")
print(color1.str.rstrip())
```

3) Write a Pandas program to count of occurrence of a specified substring in a DataFrame column.

```
import pandas as pd
df = pd.DataFrame({
   'name_code': ['c001','c002','c022', 'c2002', 'c2222'],
   'date_of_birth ': ['12/05/2002','16/02/1999','25/09/1998','12/02/2022','15/09/1997'],
   'age': [18.5, 21.2, 22.5, 22, 23]
})
print("Original DataFrame:")
print(df)
print("\nCount occurrence of 2 in date_of_birth column:")
df['count'] = list(map(lambda x: x.count("2"), df['name_code']))
print(df)
```

4)Write a Pandas program to swap the cases of a specified character column in a givenDataFrame.

```
import pandas as pd
df = pd.DataFrame({
   'company_code': ['Abcd','EFGF', 'zefsalf', 'sdfslew', 'zekfsdf'],
   'date_of_sale': ['12/05/2002','16/02/1999','25/09/1998','12/02/2022','15/09/1997'],
   'sale_amount': [12348.5, 233331.2, 22.5, 2566552.0, 23.0]
})
print("Original DataFrame:")
print(df)
print("\nSwapp cases in comapny_code:")
df['swapped_company_code'] = list(map(lambda x: x.swapcase(), df['company_code']))
print(df)
```

Pandas joining and merging DataFrame:

a) **Write a Pandas program to join the two given dataframes along rows and assign all data.**

Test Data:

```
student_data1:
  student_id            name  marks
0        S1  Danniella Fenton    200
```

```
1        S2     Ryder Storey    210
2        S3     Bryce Jensen    190
3        S4       Ed Bernal    222
4        S5      Kwame Morin    199

student_data2:
  student_id            name  marks
0        S4  Scarlette Fisher    201
1        S5  Carla Williamson    200
2        S6      Dante Morse    198
3        S7    Kaiser William    219
4        S8   Madeeha Preston    201
```

```
import pandas as pd

student_data1 = pd.DataFrame({
        'student_id': ['S1', 'S2', 'S3', 'S4', 'S5'],
         'name': ['Danniella Fenton', 'Ryder Storey', 'Bryce Jensen', 'Ed Bernal', 'Kwame
Morin'],
        'marks': [200, 210, 190, 222, 199]})

student_data2 = pd.DataFrame({
        'student_id': ['S4', 'S5', 'S6', 'S7', 'S8'],
        'name': ['Scarlette Fisher', 'Carla Williamson', 'Dante Morse', 'Kaiser William',
'Madeeha Preston'],
        'marks': [201, 200, 198, 219, 201]})

print("Original DataFrames:")
print(student_data1)
print("-----------------------------------")
print(student_data2)
print("\nJoin the said two dataframes along rows:")
result_data = pd.concat([student_data1, student_data2])
print(result_data)
```

b) Write a Pandas program to append a list of dictionaries or series to a existing DataFrame and display the combined data.

Test Data:

```
 student_id            name  marks
0        S1  Danniella Fenton    200
1        S2     Ryder Storey    210
2        S3     Bryce Jensen    190
```

```
3        S4        Ed Bernal    222
4        S5      Kwame Morin    199

Dictionary:
[{'student_id': 'S6', 'name': 'Scarlette Fisher', 'marks': 203},
        {'student_id': 'S7', 'name': 'Bryce Jensen', 'marks': 207}]
dtype: object

dicts = [{'student_id': 'S6', 'name': 'Scarlette Fisher', 'marks': 203},
        {'student_id': 'S7', 'name': 'Bryce Jensen', 'marks': 207}]

print("Original DataFrames:")
print(student_data1)
print("\nDictionary:")
print(dicts)
combined_data =  student_data1.append(dicts, ignore_index=True, sort=False)
print("\nCombined Data:")
print(combined_data)
```

c) **Write a Pandas program to join the two dataframes with matching records from both sides where available.**

Test Data:

```
student_data1:
  student_id            name  marks
0        S1  Danniella Fenton    200
1        S2      Ryder Storey    210
2        S3      Bryce Jensen    190
3        S4        Ed Bernal    222
4        S5       Kwame Morin    199

student_data2:
  student_id            name  marks
0        S4  Scarlette Fisher    201
1        S5  Carla Williamson    200
2        S6       Dante Morse    198
3        S7    Kaiser William    219
4        S8   Madeeha Preston    201

import pandas as pd
student_data1 = pd.DataFrame({
       'student_id': ['S1', 'S2', 'S3', 'S4', 'S5'],
```

```
    'name': ['Danniella Fenton', 'Ryder Storey', 'Bryce Jensen', 'Ed Bernal', 'Kwame Morin'],
    'marks': [200, 210, 190, 222, 199]})

student_data2 = pd.DataFrame({
    'student_id': ['S4', 'S5', 'S6', 'S7', 'S8'],
    'name': ['Scarlette Fisher', 'Carla Williamson', 'Dante Morse', 'Kaiser William', 'Madeeha Preston'],
    'marks': [201, 200, 198, 219, 201]})

print("Original DataFrames:")
print(student_data1)
print(student_data2)
merged_data = pd.merge(student_data1, student_data2, on='student_id', how='outer')
merged_data1 = pd.merge(student_data1, student_data2, on='student_id', how='right')
merged_data2 = pd.merge(student_data1, student_data2, on='student_id', how='left')
merged_data3 = pd.merge(student_data1, student_data2, on='student_id', how='inner')
print("Merged data (outer join):")
print(merged_data)
print(merged_data1)
print(merged_data2)
print(merged_data2)
```

F) Pandas Time Series:

1. Write a Pandas program to create
a) Datetime object for Jan 15 2012.
b) Specific date and time of 9:20 pm.
c) Local date and time.
d) A date without time.
e) Current date.
f) Time from a datetime.
g) Current local time.

```
import datetime
from datetime import datetime
print("Datetime object for Jun 22 2022:")
print(datetime(2022, 6, 22))
print("\nSpecific date and time of 9:20 pm")
print(datetime(2022, 6, 22, 21, 20))
print("\nLocal date and time:")
print(datetime.now())
print("\nA date without time: ")
print(datetime.date(datetime(2022, 6, 22)))
print("\nCurrent date:")
```

```
print(datetime.now().date())
print("\nTime from a datetime:")
print(datetime.time(datetime(2022, 6, 22, 19, 15)))
print("\nCurrent local time:")
print(datetime.now().time())
```

color highlight

```
df = pd.DataFrame({"A" : [14, 4, 5, 4, 1],
                   "B" : [5, 2, 54, 3, 2],
                   "C" : [20, 20, 7, 3, 8],
                   "D" : [14, 3, 6, 2, 6]})


print("Original DataFrame :")
display(df)
def highlight_cols(s):
        color = 'red' if s < 6 else 'blue'
        return 'background-color: % s' % color
display(df.style.applymap(highlight_cols))
```

H) Pandas Styling:

1) Create a dataframe of ten rows, four columns with random values. Write a Pandas program to highlight the negative numbers red and positive numbers black.

The numpy.random.randn() function creates an array of specified shape and fills it with random values as per standard normal distribution.
If positive arguments are provided, randn generates an array of shape (d0, d1, …, dn), filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
#linspace(start,stop,num) num-->equal partitions
```

```
print(df)
df = pd.concat([df, pd.DataFrame(np.random.randn(10, 4), columns=list('BCDE'))],
          axis=1)
#random.randn(10,4)--> 10 rows X 4 columns
print("Original array:")
print(df)
def color_negative_red(val):
    color = 'red' if val < 0 else 'black'
    return 'color: %s' % color
print("\nNegative numbers red and positive numbers black:")
df.style.applymap(color_negative_red)
```

Create a dataframe of ten rows, four columns with random values. Write a Pandas program to highlight the maximum value in each column.

The iloc property gets, or sets, the value(s) of the specified indexes.

Specify both row and column with an index.

To access more than one row, use double brackets and specify the indexes, separated by commas:

```
df.iloc[[0, 2]]
```

Specify columns by including their indexes in another list:

```
df.iloc[[0, 2], [0, 1]]
```

In pandas s is very often Series (column in DataFrame).
So you compare all values in Series with max value of Series and get boolean mask. Output is in is_max. And then set style 'background-color: yellow' only to cell of table where is True value - where is max value.
Sample:

```
s = pd.Series([1,2,3])
 print (s)
0 1
1 2
2 3
dtype: int64
 is_max = s == s.max()
print (is_max)
 0 False
 1 False
 2 True
 dtype: bool
```

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
df = pd.concat([df, pd.DataFrame(np.random.randn(10, 4), columns=list('BCDE'))],
               axis=1)
df.iloc[0, 2] = np.nan
df.iloc[3, 3] = np.nan
df.iloc[4, 1] = np.nan
df.iloc[9, 4] = np.nan
print("Original array:")
print(df)
def highlight_max(s):
    '''
    highlight the maximum in a Series green.
    '''
    is_max = s == s.max()
    return ['background-color: green' if v else '' for v in is_max]

print("\nHighlight the maximum value in each column:")
df.style.apply(highlight_max,subset=pd.IndexSlice[:, ['B', 'C', 'D', 'E']])
```

Create a dataframe of ten rows, four columns with random values. Write a Pandas program to highlight dataframe's specific columns.

```
import pandas as pd
import numpy as np
np.random.seed(24)
df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
df = pd.concat([df, pd.DataFrame(np.random.randn(10, 4), columns=list('BCDE'))],
               axis=1)
df.iloc[0, 2] = np.nan
df.iloc[3, 3] = np.nan
df.iloc[4, 1] = np.nan
df.iloc[9, 4] = np.nan
print("Original array:")
print(df)
def highlight_cols(s):
    color = 'pink'
    return 'background-color: %s' % color
print("\nHighlight specific columns:")
df.style.applymap(highlight_cols, subset=pd.IndexSlice[:, ['B', 'C','E']])
```

I)Excel:

1)Write a Pandas program to import excel data into a Pandas dataframe.

Create an excel file and copy its location to the program.Place r before the specified path.

```
#Desktop
import pandas as pd
import numpy as np
df = pd.read_excel(r"C:\Users\Admin\Desktop\CSE.xlsx")
df.tail(n=5)
```

```
#Desktop
import pandas as pd
import numpy as np
df = pd.read_excel(r"C:\Users\Admin\Desktop\CSE.xlsx")
df.head(n=5)
```

2)Write a Pandas program to find the sum, mean, max, min value of a column of file.

#Marks and no's are column names in excel file. Store the excel file in .xlsx format.

```
import pandas as pd
import numpy as np
df = pd.read_excel(r"C:\Users\Admin\Desktop\CSE1.xlsx")
print("Sum: ",df["marks"].sum())
print("Mean: ",df["no"].mean())
print("Maximum: ",df["no"].max())
print("Minimum: ",df["no"].min())
```

```
Sum:  588
Mean:  5.5
Maximum:  10
Minimum:  1
```

Plotting

Write a Pandas program to create a horizontal stacked bar plot of opening stock prices of any stock dataset between two specific dates.

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv(r"C:\Users\Admin\Desktop\alpha.csv")
start_date = pd.to_datetime('2022-4-1')
end_date = pd.to_datetime('2022-4-30')
df['Date'] = pd.to_datetime(df['Date'])
new_df = (df['Date']>= start_date) & (df['Date']<= end_date)
df1 = df.loc[new_df]
df2 = df1[['Date', 'Open', 'Close']]
```

```
df3 = df2.set_index('Date')
plt.figure(figsize=(20,20))
df3.plot.barh(stacked=True)
plt.legend(bbox_to_anchor =(0.75, 1.25), ncol = 2)
plt.suptitle('Opening/Closing stock prices\n01-04-2022 to 30-04-2022', fontsize=12,
color='black')
plt.show()
```

2) Write a Pandas program to create a histograms plot of opening, closing, high, low stock prices of stock dataset between two specific dates.

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv(r"C:\Users\Admin\Desktop\alpha.csv")
start_date = pd.to_datetime('2022-4-1')
end_date = pd.to_datetime('2022-4-30')
df['Date'] = pd.to_datetime(df['Date'])
new_df = (df['Date']>= start_date) & (df['Date']<= end_date)
df1 = df.loc[new_df]
df2 = df1[['Open','Close','High','Low']]
plt.figure(figsize=(20,20))
df2.plot.hist(alpha=0.5)
#plt.legend(bbox_to_anchor =(0.75, 1.25), ncol = 2)
plt.suptitle('Opening/Closing/HIGH/LOW stock prices\n01-04-2022 to 30-04-2022',
fontsize=12, color='black')
plt.show()
```

3) Write a Pandas program to create a stacked histograms plot of opening, closing, high,low stock prices of stock dataset between two specific dates with more bins.

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv(r"C:\Users\Admin\Desktop\alpha.csv")
start_date = pd.to_datetime('2022-4-1')
end_date = pd.to_datetime('2022-4-30')
df['Date'] = pd.to_datetime(df['Date'])
new_df = (df['Date']>= start_date) & (df['Date']<= end_date)
df1 = df.loc[new_df]
df2 = df1[['Open','Close','High','Low']]
plt.figure(figsize=(25,25))
df2.plot.hist(stacked=True, bins=200)
#plt.legend(bbox_to_anchor =(0.75, 1.25), ncol = 2)
```

```
plt.suptitle('Opening/Closing/HIGH/LOW stock prices\n01-04-2022 to 30-04-2022',
fontsize=12, color='black')
plt.show()
```

Pandas SQL Query:

1) Write a Pandas program to display all the records of a any file.

#In this example we have taken stock file and displayed all the contents of stock file, this is similar to select * from alpha;

```
import pandas as pd
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
stock = pd.read_csv(r"C:\Users\Admin\Desktop\alpha.csv")
print(stock)
```

2) Write a Pandas program to select distinct department id from department file.

```
import pandas as pd
departments = pd.read_csv(r"C:\Users\Admin\Desktop\dept.csv")
print("Distinct department_id:")
print(departments.DEPARTMENT_ID.unique())
```