Modules

- A module comprises of various functions which can be created and imported whenever required.

Creating Module

We can create modules in Jupyter Notebook with steps below:

1. Create a new Jupyter notebook
2. define required functions in it
3. Go to File > Download as > Python(.py)
4. Save it in the same working directory

Working with Module

We can import module by using below options.

1. import module_name (We can call function as "module_name.function())
2. import module_name as md (We can call function as md.function())
3. from module_name import function (We can call the function as function())

Downloading a Module

Modules can be downloaded with pip option. We can check the source on google and use pip as below:

- !pip install pandas

Note- To see all functions in module, write "module_name." and press tab key

In [1]:
```
import calculator
import calculator as cal
from calculator import div
```

In [2]:
```
calculator.add(5,2)
cal.add(5,2)
div(5,2)
```

Out[2]:
```
2.5
```

pwd (print working directory) command returns the current directory

In [3]:
```
pwd
```

Out[3]:
```
'C:\\Users\\srbhk\\Downloads\\Python'
```

Modules to learn

1. **Numpy (Numerical Python)** - Mathematical Operations
2. **Pandas** - Data Manipulation
3. **Matplotlib** - Data Visualization
4. **Scikitlearn** - Machine Learning algorithms
5. **Scipy (Scientic Operations)** - Advanced Mathematical/Scientific Operations
6. **Seaborn** - Advanced Visualization

Numpy

- It is an open source module
- It is programmed in C so it is fast
- Adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays like One-Dimensional Numpy, Two-Dimensional Numpy
- Helpful in linear algebra, fourier transformations, integrating databases/C/C++

In [4]:
```
import numpy as np
```

In [5]:
```
list1 = [1,2,3] #each element in lists take 14 bytes of memory
arr1 = np.array([1,2,3]) #each element in array takes 4 byte of memory so faste
```

- **numpy.full(shape, fill_value, dtype = None, order = 'C')** - Return a new array with the same shape and type as a given array filled with a fill_value, dtype is float(by Default)
- **np.zeros(shape)** or **np.ones(shape)** - Returns arrays of 0s/1s with given shape. We can also give dtype as additional argument

In [6]:
```
np.full(3,5)
np.full((3,3),5, dtype=int)
np.zeros(3) #gives 1d arrary of zero values.
np.zeros((3,4)) #gives 2d (3x4) arrary of zero values.
np.ones(3) #gives 1d arrary of 1 values.
np.ones((3,4)) #gives 2d (3x4) arrary of 1 values.
```

Out[6]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

- **np.random.function(args)** returns array of random values as per the function and arguments
- **random(size = num)** or **rand(num)** - returns array of random float values. returns 1 value if no argument given
- **randint(highest, size = num)** returns random int values between 0 and highest of given size.

- **choice(array, size = num)** method takes an array as a parameter and randomly returns one of the values if no size given. If size is given, it returns array of random elements from the input array

In [7]:
```
np.random.random((3,3))
np.random.randint(100)
np.random.randint(100, size=(5))
np.random.randint(100, size=(3, 5))
np.random.rand()
np.random.rand(3, 5)
np.random.choice([3, 5, 7, 9])
np.random.choice([3, 5, 7, 9], size=(3, 5))
```

Out[7]:
```
array([[7, 5, 3, 3, 3],
       [9, 5, 3, 7, 5],
       [3, 5, 9, 9, 9]])
```

- **np.linspace(start, stop, num = 50, endpoint = True, retstep = False, dtype = None)** - Start is 0 by default, restep : If True, return (samples, step/gap). By default restep = False
- **np.arange(start, stop, step, dtype)** : Returns an array with evenly spaced elements as per the interval. Stop is not included. step : step size of interval. By default step size = 1
- **arange can be combined with reshape**
- **np.arange(start,stop,step,dtype).reshape(size)** gives array as per the reshape size given

Note - np.linspace allows you to define how many values you get including the specified min and max value. It infers the stepsize **whereas** np.arange allows you to define the stepsize and infers the number of steps(the number of values you get).

In [8]:
```
np.linspace(0,10, num = 11)
np.linspace(0,10, num = 11, retstep = True)
```

Out[8]:
```
(array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.]), 1.0)
```

In [9]:
```
np.arange(4, 10)
np.arange(4, 20, 3)
np.arange(1,2,0.1)
np.arange(4).reshape(2, 2)
```

Out[9]:
```
array([[0, 1],
       [2, 3]])
```

- **np.concatenate(array1, array2)** flatens the arrays and merges them along required axis. The axis along which the arrays will be joined. If axis is None, arrays are flattened before use. Default is 0.
- **np.vstack(tup)** is used to stack the sequence of input arrays vertically to make a single array.
- **np.hstack(tup)** is used to stack the sequence of input arrays horizontally (i.e. column wise) to make a single array.
- **np.column_stack(tup)**

In [10]:
```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[7,8,9],[1,5,8]])
print("Concat:\n",np.concatenate((a,b)))
#print("Concat1:\n",np.concatenate((a,b), axis =0)) is same as above statement
print("Concat2:\n",np.concatenate((a,b), axis =1))
print("vstack:\n",np.vstack((a,b)))
print("hstack:\n",np.hstack((a,b)))
print("column_stack:\n",np.column_stack((a,b)))
```

```
Concat:
 [[1 2 3]
 [4 5 6]
 [7 8 9]
 [1 5 8]]
Concat2:
 [[1 2 3 7 8 9]
 [4 5 6 1 5 8]]
vstack:
 [[1 2 3]
 [4 5 6]
 [7 8 9]
 [1 5 8]]
hstack:
 [[1 2 3 7 8 9]
 [4 5 6 1 5 8]]
column_stack:
 [[1 2 3 7 8 9]
 [4 5 6 1 5 8]]
```

- **array.shape -** returns a tuple with each index having the number of corresponding elements.
- **array.ndim -** returns the number of dimensions of array
- **array.size -** returns the total number of elements in array.
- **numpy.size(arr, axis) -** gives the size of array across particluar axis. 0=x axis, 1 = y axis.
- **array.dtype -** returns datatype of the array
- **array.itemsize -** returns the length of one array element in bytes.

In [11]:
```
a = np.array([[1,2,3],[4,5,6]])
print("Shape: ", a.shape)
print("dimension: ", a.ndim)
print("Size: ", a.size)
print("datatype :", a.dtype)
print("a.itemsize: ", a.itemsize)
```

```
Shape:  (2, 3)
dimension:  2
Size:  6
datatype : int32
a.itemsize:  4
```

Note: We can change the shape of array keeping in mind that elements are equally accomodated.

In [12]:
```
print("old array is: \n", a)
print("old shape is: :", a.shape)
a.shape = 3,2
print("new array is: \n", a)
print("new shape is: :", a.shape)
```

```
old array is:
 [[1 2 3]
 [4 5 6]]
old shape is: : (2, 3)
new array is:
 [[1 2]
 [3 4]
 [5 6]]
new shape is: : (3, 2)
```

- **np.equal(arr1, arr2) -** used to compare and return bool values elementwise
- **np.sum(arr, axis, dtype, out) -** axis =0 means along the column, 1 means sum along the row. Out if we want array to be assigned to other array
- **np.subtract(arr1, arr2) -** gives arr1-arr2 elementwise
- Similarly we have **divide, multiply, exponetial (e^n), sqrt etc.**

In [13]:
```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[1,1,1],[2,2,2]])
print("Equal:\n", np.equal(a,b))
print(np.sum([[1,2,3],[4,5,6]]))
print(np.sum([[1,2,3],[4,5,6]], axis = 0))
print(np.sum([[1,2,3],[4,5,6]], axis = 1))
print("Subtract:\n", np.subtract(a,b))
print("Divide:\n", np.divide(a,b))
print("Multiply:\n",np.multiply(a,b))
print("exponential:\n", np.exp(a))
print("root:\n", np.sqrt(a))
```

```
Equal:
 [[ True False False]
 [False False False]]
21
[5 7 9]
[ 6 15]
Subtract:
 [[0 1 2]
 [2 3 4]]
Divide:
 [[1.  2.  3. ]
 [2.  2.5 3. ]]
Multiply:
 [[ 1  2  3]
 [ 8 10 12]]
exponential:
 [[  2.71828183   7.3890561   20.08553692]
 [ 54.59815003 148.4131591  403.42879349]]
root:
 [[1.         1.41421356 1.73205081]
 [2.         2.23606798 2.44948974]]
```

We also have functions to find min, max, mean, median

In [14]:
```
a = np.array([[1,2,3],[4,5,6]])
print(np.min(a))
print(np.max(a))
print(np.mean(a))
print(np.median(a))
print(np.min(a))
print(np.std(a)) #Standard Deviation
print(np.corrcoef(a)) #Correlation Coifficient
```

```
1
6
3.5
3.5
1
1.707825127659933
[[1. 1.]
 [1. 1.]]
```

Indexing and Slicing in Numpy

In [15]:
```
a = np.array([1,2,3,4])
b = np.array([[5,6,7],[8,9,10]])
print(a[0])
print(a[-1]) # or print(a[3])
print(a[0:2]) #prints 1 less than index
print(a[-2:-1])
```

```
1
4
[1 2]
[3]
```

In [16]:
```
print("\n2D arrays:\n", b[0])
print(b[0,2]) # or b[0][2] gives same result)
print(b[-1,-1])
print(b[0:2,1:3])
```

```

2D arrays:
 [5 6 7]
7
10
[[ 6  7]
 [ 9 10]]
```

Create the following NumPy arrays:

a) A 1-D array called arr_zeros having 10 elements and all the elements are set to zero.

b) A 1-D array called arr_vowels having the elements ‘a’, ‘e’, ‘i’, ‘o’ and ‘u’.

c) A 2-D array called arr_ones having 2 rows and 5 columns and all the elements are set to 1 and dtype as int.

d) Use nested Python lists to create a 2-D array called arr_myarray1 having 3 rows and 3 columns and store the following data: 2.7, -2, -19 0, 3.4, 99.9 10.6, 0, 13

e) A 2-D array called arr_myarray2 using arange() having 3 rows and 5 columns with start value = 4, step size 4 and dtype as float.

In [17]:
```
#a
arr_zeros = np.full(10, 0) #or np.zeros(10)
#print(arr_zeros)

#b
arr_vowels = np.array(["a","e","i","o","u"])
#print(arr_vowels)

#c
arr_ones = np.full((2, 5),1, dtype=np.int32) #or np.ones((2,5), dtype = np.int
#print(arr_ones)

#d
list1 = [[2.7, -2, -19], [0, 3.4, 99.9], [10.6, 0, 13]]
arr_myarray1 = np.array(list1)
#print(arr_myarray1)

#e
temp_myarray2 = np.arange(4,64,4,float)
arr_myarray2 = temp_myarray2.reshape(3,5)
print(arr_myarray2)
```

```
[[ 4.   8. 12. 16. 20.]
 [24. 28. 32. 36. 40.]
 [44. 48. 52. 56. 60.]]
```

Using the arrays created in question 1 above, write NumPy commands for the following:

a) Find the dimensions, shape, size, data type of the items and itemsize of arrays arr_zeros, arr_vowels, arr_ones, arr_myarray1 and arr_myarray2.

b) Reshape the array arr_ones to have all the 10 elements in a single row.

c) Display the 2nd and 3rd element of the array arr_vowels.

d) Display all elements in the 2nd and 3rd row of the array arr_myarray1.

e) Display the elements in the 1st and 2nd column of the array arr_myarray1.

f) Display the elements in the 1st column of the 2nd and 3rd row of the array arr_myarray1.

g) Reverse the array of arr_vowels.

In [18]:
```
#a
np.ndim(arr_zeros)
np.shape(arr_zeros)
np.size(arr_zeros)
arr_zeros.dtype
arr_zeros.itemsize

#b
arr_ones.reshape(10,)

#c
arr_vowels[1:3]

#d
arr_myarray1[1:3,]

#e
arr_myarray1[:,0:2]

#f
arr_myarray1[1:3,0]

#g
arr_vowels[: : -1] # Check or arr_vowels[range(4,-1,-1)]
```

Out[18]:
```
array(['u', 'o', 'i', 'e', 'a'], dtype='<U1')
```

Using the arrays created in question1 above, write NumPy commands for the following:

a) Divide all elements of array arr_ones by 3.

b) Add the arrays arr_myarray1 and arr_myarray2.

c) Subtract arr_myarray1 from arr_myarray2 and store the result in a new array.

d) Multiply arr_myarray1 and arr_myarray2 elementwise.

e) Do the matrix multiplication of arr_myarray1 and arr_myarray2 and store the result in a new array arr_myarray3.

f) Divide arr_myarray1 by arr_myarray2.

g) Find the cube of all elements of arr_myarray1 and divide the resulting array by 2.

h) Find the square root of all elements of arr_myarray2 and divide the resulting array by 2. The result should be rounded to two places of decimals.

In [19]:
```
#a
arr_ones/3
#arr = arr_myarray1 + arr_myarray2
#np.add(arr_myarray1, arr_myarray2)

#g
(arr_myarray1**3)/2

#h
arr = (arr_myarray2**0.5)/2
np.around(arr, decimals =2)
```

Out[19]:
```
array([[1.  , 1.41, 1.73, 2.  , 2.24],
       [2.45, 2.65, 2.83, 3.  , 3.16],
       [3.32, 3.46, 3.61, 3.74, 3.87]])
```

Using the arrays created in Question 1 above, write NumPy commands for the following:

a) Find the transpose of arr_ones and arr_myarray2.

b) Sort the array arr_vowels in reverse.

c) Sort the array arr_myarray1 such that it brings the lowest value of the column in the first row and so on.

In [20]:
```
#a
arr_ones.T
arr_myarray2.T

#b
arr_vowels[::-1]
print("old array is:\n",arr_myarray1)
#c
print("\nSorted array is:")
np.sort(arr_myarray1, axis =0)
```

```
old array is:
 [[  2.7  -2.  -19. ]
 [  0.    3.4  99.9]
 [ 10.6   0.   13. ]]

Sorted array is:
```

Out[20]:
```
array([[  0. ,  -2. , -19. ],
       [  2.7,   0. ,  13. ],
       [ 10.6,   3.4,  99.9]])
```