Array Computations:

NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis. It is the foundation on which nearly all of the higher-level tools are built.

Key highlight:

```
1. ndarray, a fast and space-efficient multidimensional array providing
vectorized arithmetic operations and  sophisticated broadcasting
capabilities.
2. Standard mathematical functions for fast operations on entire arrays of
data without having to write loops
3. Tools for reading / writing array data to disk and working with memory-
mapped files
4. Linear algebra, random number generation, and Fourier transform
capabilities
5. Tools for integrating code written in C, C++, and Fortran
```

It is very easy to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays.

While NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools like pandas much more effectively.

For most data analysis applications, the main areas of functionality

```
1. Fast vectorized array operations for data munging and cleaning,
subsetting and filtering, transformation, and any other kinds of
computations
2. Common array algorithms like sorting, unique, and set operations
3. Efficient descriptive statistics and aggregating/summarizing data
4. Data alignment and relational data manipulations for merging and joining
together heterogeneous data sets
5. Expressing conditional logic as array expressions instead of loops with
if-elifelse branches.
6. Group-wise data manipulations (aggregation, transformation, function
application).
# The NumPy ndarray: A Multidimensional Array Object

#import package
import numpy as np

#Creating ndarray :
#The easiest way to create an array is to use the array function. This
accepts any sequence- like object
#(including other arrays) and produces a new NumPy array containingthe
passed data
data1 = [6, 7, 8, 0, 1]
arr1 = np.array(data1)
print("My First Array:", arr1)

##Nested Sequence, like a list of equal length list, will be converted to
multidimensional array
```

```
data2 = [[1,2,3,4],[5,6,7,8]]
arr2 = np.array(data2)
print("My First Multidimensional Array:\n", arr2)

data3 = [[1,2,3],[5,6,7],[5,6,7]]

arr3 = np.array(data3)
print("My First Multidimensional Array:\n", arr3)
print(arr3.ndim)
My First Array: [6 7 8 0 1]
My First Multidimensional Array:
 [[1 2 3 4]
 [5 6 7 8]]
My First Multidimensional Array:
 [[1 2 3]
 [5 6 7]
 [5 6 7]]
2
data = np.array([[ 0.9526, -0.246 , -0.8856],
[ 0.5639, 0.2379, 0.9104]])

print("2 D Array: \n",data)

# Every array has shape - a tuple - indicating the size of ech dimension
print("Shape of an Array : ",data.shape)


#Every array has dtype, an object describing the data type of an array
#Unless explicitly specified, np.array tries to infer a good data type for
the array that it creates.
print("Data TYpe of an Array :",data.dtype)

#Every array has ndim, an object describing the dimension of an array
print("Data TYpe of an Array :",data.ndim)
#In addition to np.array, there are a number of other functions for
creating new arrays.

#zeros(): Create array of zeros
print("1D zero Matrix: \n",np.zeros(10))

print("2D zero Matrix: \n",np.zeros((3, 6)))

#ones(): Create array of ones
print("2D ones Matrix: \n",np.ones((2,3)))

#empty creates an array withouf initializing it's values to any particular
values.

print("3D empty Matrix: \n",np.empty((2, 3, 2)))
#It’s not safe to assume that np.empty will return an array of all zeros.
#In many cases, it will return uninitialized garbage values.
print(np.empty((2, 3, 2)).ndim)



#np.arange(): arange is an array-valued version of the built-in Python
range function
print(np.arange(15))
1D zero Matrix:
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
2D zero Matrix:
 [[ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
2D ones Matrix:
 [[ 1.  1.  1.]
 [ 1.  1.  1.]]
3D empty Matrix:
 [[[ 0.  0.]
  [ 0.  0.]
  [ 0.  0.]]

 [[ 0.  0.]
  [ 0.  0.]
  [ 0.  0.]]]
3
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
print("3D empty Matrix: \n",np.zeros((2, 3, 2)))
print("3D empty Matrix: \n",np.empty((4, 2, 3)))
3D empty Matrix:
 [[[0. 0.]
  [0. 0.]
  [0. 0.]]

 [[0. 0.]
  [0. 0.]
  [0. 0.]]]
3D empty Matrix:
 [[[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]]
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14]])
# Create a 3x3 identity matrix
np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

NumPy Array Attributes

Each array has attributes ndim (the number of dimensions), shape (the size of each dimension), and size (the total size of the array):

```
import numpy as np
np.random.seed(0)  # seed for reproducibility
```

```
x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
print(x1)
print(x3)
x3[2,3,2]
[5 0 3 3 7 9]
[[[8 1 5 9 8]
  [9 4 3 0 3]
  [5 0 2 3 8]
  [1 3 3 3 7]]

 [[0 1 9 9 0]
  [4 7 3 2 7]
  [2 0 0 4 5]
  [5 6 8 4 1]]

 [[4 9 8 1 1]
  [7 9 9 3 6]
  [7 2 0 3 5]
  [9 4 4 6 4]]]
4
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Other attributes include itemsize, which lists the size (in bytes) of each array element, and nbytes, which lists the total size (in bytes) of the array:

```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
itemsize: 4 bytes
nbytes: 240 bytes
```

Data Types:

Dtypes are part of what make NumPy so powerful and flexible. In most cases they map directly onto an underlying machine representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like float or int, followed by a number indicating the number of bits per element. A standard double-precision floating point value takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as float64.

```
## Datatype of ndarray
#The data type or dtype is a special object containing the information the
ndarray needs
#to interpret a chunk of memory as a particular type of data:

arr1 = np.array([1, 2, 3], dtype=np.float64)
arr2 = np.array([1.5, 2.3, 3.2], dtype=np.int32)

print("Data Type of arr1:\n",arr1.dtype)
print("Data Type of arr2:\n",arr2.dtype)
```

```
##You can explicitly convert of cast an array from one type to another
using ndarray's as type method:

arr = np.array([1,2,3,4,5])
print("Original Datatype : ",arr.dtype)

float_arr = arr.astype(np.float64)
print("Data type After Type Casting : ",float_arr.dtype)

##If you cast some floating point numbers to be of integer dtype, the
decimal part will be truncated

arr = np.array([1.2,2.1,3.4,4,4])
print("Original Data TYpe : ",arr.dtype)

arr_int = arr.astype(np.int32)
print("Type Casted Data Type:\n",arr_int)


##String array also you can convert to numeric form

numeric_strings = np.array(['1.25','-9.6','42'],dtype=np.string_)
print(numeric_strings.astype(np.float64))
print(numeric_strings.astype(np.float64).dtype)
Data Type of arr1:
 float64
Data Type of arr2:
 int32
Original Datatype :  int32
Data type After Type Casting :  float64
Original Data TYpe :  float64
Type Casted Data Type:
 [1 2 3 4 4]
[  1.25  -9.6   42.  ]
float64
##If casting were to fail for some reason, a TypeError will be raised

import numpy as np
numeric_strings = np.array(['Hello','-9.6','42'],dtype=np.string_)
print(numeric_strings.astype(float))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-9aaaff704404> in <module>()
      3 import numpy as np
      4 numeric_strings = np.array(['Hello','-9.6','42'],dtype=np.string_)
----> 5 print(numeric_strings.astype(float))

ValueError: could not convert string to float: 'Hello'
# NumPy is smart enough to alias the Python types to the equivalent dtypes
import numpy as np
numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
print(numeric_strings.astype(float))
print(numeric_strings.astype(float).dtype)
print(numeric_strings.astype('float32').dtype)
print(numeric_strings.astype('float64').dtype)
[ 1.25 -9.6  42.  ]
float64
float32
float64
```

Points to Remember:

1. Calling astype always creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

```
##Operations

#Arrays are important because they enable you to express batch operations
on data
#without writing any for loops. This is usually called vectorization. Any
arithmetic operations
#between equal-size arrays applies the operation elementwise:

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

#Addition
print("#########Addition#########")
arr1 = arr + arr
print(arr1)

#Subtraction
print("#########Subtraction#########")
arr2 = arr - arr
print(arr2)

#Multiplication
print("#########Multiplicaton#########")
arr3 = arr * arr
print(arr3)

#Division
print("#########Division#########")
arr4 = arr / arr
print(arr4)
#########Addition#########
[[  2.   4.   6.]
 [  8.  10.  12.]]
#########Subtraction#########
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
#########Multiplicaton#########
[[  1.   4.   9.]
 [ 16.  25.  36.]]
#########Division#########
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
## Arithmetic Operations with Scalar

##Addition

print("Scalar Addition")
print(arr+1)

print("Scalar Subtraction")
print(arr-1)

print("Scalar Multiplication")
print(arr*2)

print("Scalar Division")
print(1/arr)
print(arr/2)


##Operations between differently sized arrays is called broadcasting
Scalar Addition
[[ 2.  3.  4.]
 [ 5.  6.  7.]]
Scalar Subtraction
[[ 0.  1.  2.]
 [ 3.  4.  5.]]
Scalar Multiplication
[[  2.   4.   6.]
 [  8.  10.  12.]]
Scalar Division
[[ 1.          0.5         0.33333333]
 [ 0.25        0.2         0.16666667]]
[[ 0.5  1.   1.5]
 [ 2.   2.5  3. ]]
```

Data Processing Using Array

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as vectorization.

Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
arr = np.arange(10)

print("My Array is:")
print(arr)

print("Extract 5th index from the array")
print(arr[5])

print("Slice 5th to 7th index in array")
print(arr[5:8])
My Array is:
[0 1 2 3 4 5 6 7 8 9]
Extract 5th element from the array
5
Slice 5th to 7th elements in array
[5 6 7]
lst1 = [0,1,2,3,4,5,6,7,8,9]
print(lst1)
print(lst1[5:8])
lst1[5:8] = 12
print(lst1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 6, 7]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-17-476d6553111c> in <module>()
      2 print(lst1)
      3 print(lst1[5:8])
----> 4 lst1[5:8] = 12
      5 print(lst1)

TypeError: can only assign an iterable
arr1 = np.arange(10)
arr1[5:8] = 12
print(arr1)
[ 0  1  2  3  4 12 12 12  8  9]
lst1 = [1,2,3,4,5]
lst1[1] = 12
lst1
[1, 12, 3, 4, 5]
#In a multi-dimensional array, items can be accessed using a comma-
separated tuple of indices:
import numpy as np
np.random.seed(0)
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array

print("Original Array: \n",x2)

print("Slice :",x2[(0, 0)])

print("Slice :",x2[(0, -1)])
Original Array:
 [[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
Slice : 5
Slice : 3
3
import numpy as np
np.random.seed(0)
x2 = np.random.randint(10, size=(3, 4,2))  # Two-dimensional array

print("Original Array: \n",x2)

print("Slice :",x2[(0, 0,0)])

print("Slice :",x2[(0, -1,-1)])
Original Array:
 [[[5 0]
  [3 3]
  [7 9]
  [3 5]]

 [[2 4]
  [7 6]
  [8 8]
  [1 6]]

 [[7 7]
  [8 1]
  [5 9]
  [8 9]]]
Slice : 5
Slice : 5
#Update the Slice which is different from List
```

```
#Array slices are views on the original array.
#This means that the data is not copied, and any modifications to
#the view will be reflected in the source array
arr1 = np.arange(10)
arr_slice = arr1[5:8]
arr_slice[1] = 12345
print(arr1)

#Not true in case of list
lst1 = [0,1,2,3,4,5,6,7,8,9]
lst_slice = lst1[5:8]
lst_slice[1] = 12345
print(lst1)
print(lst_slice)

##As NumPy has been designed with large data use cases in mind,
#you could imagine performance and
##memory problems if NumPy insisted on copying data left and right.
[    0     1     2     3     4     5 12345     7     8     9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 12345, 7]
```

If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array; for example arr[5:8].copy().

```
#Keep in mind that, unlike Python lists, NumPy arrays have a fixed type.
This means, for example,
#that if you attempt to insert a floating-point value to an integer array,
the value will be silently truncated.
x1 = np.arange(10)
x1[0] = 3.14159  # this will be truncated!
x1
array([3, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Array Slicing: Accessing Subarrays

ust as we can use square brackets to access individual array elements, we can also use them to access subarrays with the slice notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

```
# x[start:stop:step]

#they default to the values start=0, stop=size of dimension, step=1

x = np.arange(10)
print("Original Array:",x)

print("First 5 elements :", x[:5])
print("Elements after index 5 :", x[5:])
print("Elements  4th to 6th :", x[4:7])
print("Every Other Element :",x[::2])
print("Every Other Element starting from index 1 :",x[1::2])
Original Array: [0 1 2 3 4 5 6 7 8 9]
First 5 elements : [0 1 2 3 4]
Elements after index 5 : [5 6 7 8 9]
Elements  4th to 6th : [4 5 6]
Every Other Element : [0 2 4 6 8]
Every Other Element starting from index 1 : [1 3 5 7 9]
#Multi-dimensional slices work in the same way, with multiple slices
separated by commas. For example:

print("Origina Array :\n",x2)
print("Two Row and three Column: \n",x2[:2, :3])
print("All rows, every other column :\n",x2[:3, 1::2])
Origina Array :
 [[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
Two Row and three Column:
 [[5 0 3]
 [7 9 3]]
All rows, every other column :
 [[0 3]
 [9 5]
 [4 6]]
#One commonly needed routine is accessing of single rows or columns of an
array.
#This can be done by combining indexing and slicing, using an empty slice
marked by a single colon (:):

print("First Column of x2 :\n",x2[:,0])

print("First row of x2 :\n",x2[0,:])
First Column of x2 :
 [2 9 8]
First row of x2 :
 [2 8 6 6]
```

Transposing:

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the reshape method

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the reshape method will use a no-copy view of the initial array

Transposing is a special form of reshaping which similarly returns a view on the underlying data without copying anything. Arrays have the transpose method and also the special T attribute

```
arr = np.arange(15).reshape((3, 5))
print(arr)


print("Transposed Array\n",arr.T)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
Transposed Array
 [[ 0  5 10]
 [ 1  6 11]
 [ 2  7 12]
 [ 3  8 13]
 [ 4  9 14]]
```

```
#Simple transposing with .T is just a special case of swapping axes.
ndarray has the method
#swapaxes which takes a pair of axis numbers:

arr1 = np.arange(24).reshape(2, 3, 4)

print(arr1)

print(arr1.swapaxes(0, 2))

#swapaxes similarly returns a view on the data without making a copy.
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
[[[ 0 12]
  [ 4 16]
  [ 8 20]]

 [[ 1 13]
  [ 5 17]
  [ 9 21]]

 [[ 2 14]
  [ 6 18]
  [10 22]]

 [[ 3 15]
  [ 7 19]
  [11 23]]]
arr1.T
array([[[ 0, 12],
        [ 4, 16],
        [ 8, 20]],

       [[ 1, 13],
        [ 5, 17],
        [ 9, 21]],

       [[ 2, 14],
        [ 6, 18],
        [10, 22]],

       [[ 3, 15],
        [ 7, 19],
        [11, 23]]])
#Universal Functions: Fast Element-wise Array Functions

arr = np.arange(10)
print("Original Array :\n",arr)
#x = randn(8)
print("Sqrt Array :\n",np.sqrt(arr))

print("Exp Array: \n",np.exp(arr))
Original Array :
 [0 1 2 3 4 5 6 7 8 9]
Sqrt Array :
 [ 0.          1.          1.41421356  1.73205081  2.          2.23606798
```

```
  2.44948974  2.64575131  2.82842712  3.        ]
Exp Array:
 [  1.00000000e+00   2.71828183e+00   7.38905610e+00   2.00855369e+01
   5.45981500e+01   1.48413159e+02   4.03428793e+02   1.09663316e+03
   2.98095799e+03   8.10308393e+03]
#add or maximum, take 2 arrays and return a single array as the result:
import numpy as np
x = np.random.randn(8)
y = np.random.randn(8)

print("First Matrix:\n",x)
print(x)

print("Second Matrix:\n",y)
print(y)


print("Add both Matrix:\n",np.add(x,y))
print("Find Maximum Elmentwise in Both Matrix",np.maximum(x,y))
First Matrix:
 [-0.75610969  0.69656958 -0.87344017  2.156342    1.10866695  0.00945711
  0.13371106 -0.77212832]
[-0.75610969  0.69656958 -0.87344017  2.156342    1.10866695  0.00945711
  0.13371106 -0.77212832]
Second Matrix:
 [ 0.24607568  2.11883655  0.77434516 -1.58902625  1.02611714 -0.54678633
  0.12309212  1.05902309]
[ 0.24607568  2.11883655  0.77434516 -1.58902625  1.02611714 -0.54678633
  0.12309212  1.05902309]
Add both Matrix:
 [-0.51003401  2.81540613 -0.09909501  0.56731575  2.1347841  -0.53732922
  0.25680319  0.28689477]
Find Maximum Elmentwise in Both Matrix [ 0.24607568  2.11883655  0.77434516
2.156342    1.10866695  0.00945711
  0.13371106  1.05902309]
```

Mathematical and Statistical Methods

A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods. Aggregations (often called reductions) like sum, mean, and standard deviation std can either be used by calling the array instance method:

```
import numpy as np
arr = np.random.randn(5, 4) # normally-distributed data
print("Array is:\n",arr)

print("Mean is: ",arr.mean())
print("Mean is:", np.mean(arr))

print("Sum is:", arr.sum())
print("Sum is:", np.sum(arr))
Array is:
 [[-0.61203566  1.06892144  0.22197301  0.46956584]
 [-0.36731883 -0.30786135  0.6577473  -0.78640214]
 [-0.44283905 -1.14007558  0.46196163 -0.0830546 ]
 [-0.65335901  0.17495488 -0.13372751 -0.53136   ]
 [-0.90283751 -0.17647121 -0.78778754  0.55121405]]
Mean is:  -0.16593959050508397
Mean is: -0.16593959050508397
Sum is: -3.3187918101016796
Sum is: -3.3187918101016796
```

Functions like mean and sum take an optional axis argument which computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
print(arr)
arr.mean(axis=1)
[[-0.61203566  1.06892144  0.22197301  0.46956584]
 [-0.36731883 -0.30786135  0.6577473  -0.78640214]
 [-0.44283905 -1.14007558  0.46196163 -0.0830546 ]
 [-0.65335901  0.17495488 -0.13372751 -0.53136   ]
 [-0.90283751 -0.17647121 -0.78778754  0.55121405]]
array([ 0.28710616, -0.20095875, -0.3010019 , -0.28587291, -0.32897055])
np.mean(arr,axis=0)
array([-0.59567801, -0.07610636,  0.08403338, -0.07600737])
```

Other methods like cumsum and cumprod do not aggregate, instead producing an array of the intermediate results:

```
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

print("Array is :\n",arr)

print("Cumulative Sum Rowwise: \n",arr.cumsum(0))

print("Cumulative Sum Columwise: \n",arr.cumsum(1))

print("Cumulative Product Rowwise: \n",arr.cumprod(0))

print("Cumulative Product Columwise: \n",arr.cumprod(1))
Array is :
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Cumulative Sum Rowwise:
 [[ 0  1  2]
 [ 3  5  7]
 [ 9 12 15]]
Cumulative Sum Columwise:
 [[ 0  1  3]
 [ 3  7 12]
 [ 6 13 21]]
Cumulative Product Rowwise:
 [[ 0  1  2]
 [ 0  4 10]
 [ 0 28 80]]
Cumulative Product Columwise:
 [[  0   0   0]
 [  3  12  60]
 [  6  42 336]]
#Sorting:
import numpy as np
arr = np.random.randn(5,3)

print("Original Array: \n",arr)
#np.sort returns a sorted copy of an array
arr.sort(0)
```

```
print("Sortd Array Row wise: \n",arr)
arr.sort(1)
print("Sortd Array Column wise: \n",arr)
Original Array:
 [[ 0.38263521 -1.32558675  0.9638272 ]
 [ 0.73020209  0.47968292  0.85165092]
 [ 0.01410554  0.64176615  0.16994591]
 [-0.58405723 -0.26258071 -0.75043188]
 [ 1.35291584 -0.52301173  1.20502556]]
Sortd Array Row wise:
 [[-0.58405723 -1.32558675 -0.75043188]
 [ 0.01410554 -0.52301173  0.16994591]
 [ 0.38263521 -0.26258071  0.85165092]
 [ 0.73020209  0.47968292  0.9638272 ]
 [ 1.35291584  0.64176615  1.20502556]]
Sortd Array Column wise:
 [[-1.32558675 -0.75043188 -0.58405723]
 [-0.52301173  0.01410554  0.16994591]
 [-0.26258071  0.38263521  0.85165092]
 [ 0.47968292  0.73020209  0.9638272 ]
 [ 0.64176615  1.20502556  1.35291584]]
#NumPy has some basic set operations for one-dimensional ndarrays. Probably
the most
#commonly used one is np.unique, which returns the sorted unique values in
an array

names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
print("Unique Values in Array (Sorted Order) :",np.unique(names))
Unique Values in Array (Sorted Order) : ['Bob' 'Joe' 'Will']
```

Random Number Generation

The numpy.random module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from the standard differennt distributions

```
rand: Draw samples from a uniform distribution
randint: Draw random integers from a given low-to-high range
randn: Draw samples from a normal distribution
binomial: Draw samples a binomial distribution
normal: Draw samples from a normal (Gaussian) distribution
chi-square: Draw samples from a chi-square distribution
uniform: Draw samples from a uniform [0, 1) distribution
samples = np.random.normal(size=(4, 4))
print(samples)
[[-2.13772189  0.55799732 -0.12064481 -0.39563065]
 [-0.88607616  1.54565784  1.54784569 -1.48324663]
 [ 1.33135235 -1.34664601  0.5987885   0.59592675]
 [-0.29389159 -0.94253054 -0.32973574 -3.38952094]]
```