Unit-3

NumPy and pandas

Faculty: D Sai Kumar
Dept. of CSE, UCE, OU

NumPy(**Num**erical **Py**thon): A fundamental package for scientific computing with support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions.

pandas: A powerful data manipulation and analysis library that provides data structures and functions for working with structured data.

Difference between Python Lists and NumPy array

Python lists and NumPy arrays are both used to store collections of elements, but they have key differences:

1. Data Type Consistency
2. Performance
3. Functionality
4. Memory Efficiency
5. Operations

Understanding NumPy arrays

- NumPy arrays are a series of homogenous items. Homogenous means the array will have all the elements of the same data type.
- They allow efficient computation and manipulation of numerical data.

- You can create an array using the array() function with a list of items.
- Users can also fix the data type of an array.
- Possible data types are bool, int, float, long, double, and long double.

```
# Creating an array
import numpy as np
a = np.array([2,4,6,8,10])
print(a)
```

Output:

```
[ 2 4 6 8 10]
```

- **Creating an Integer Array**

import numpy as np

arr = np.array([1, 2, 3, 4, 5], dtype=np.int32) # Specifying integer type print(arr)

print("Data type:", arr.dtype)

Output:

[1 2 3 4 5]

Data type: int32

- **Creating a Float Array**

arr = np.array([1.2, 2.3, 3.4], dtype=np.float64) # Specifying float type print(arr)

print("Data type:", arr.dtype)

Output:

[1.2 2.3 3.4]

Data type: float64

- Another way to create a NumPy array is with arange().
- It creates an evenly spaced NumPy array.
- Three values – start, stop, and step – can be passed to the arange(start,[stop],step) function
- The start is the initial value of the range, the stop is the last value of the range, and the step is the increment in that range. The stop parameter is compulsory.

```
import numpy as np
arr = np.arange(1, 10, 2) # Start at 1, go up to 10 (exclusive), step size of 2
print(arr)
```

Output:

[1 3 5 7 9]

```
# Creating an array using arange()
import numpy as np
a = np.arange(1,11)
print(a)
```

Output:

```
[ 1 2 3 4 5 6 7 8 9 10]
```

- Apart from the array() and arange() functions, there are other options, such as zeros(), ones(), full(), eye(), and random(), which can also be used to create a NumPy array, as these functions are initial placeholders.

Here is a detailed description of each function:

- zeros(): The zeros()function creates an array for a given dimension with all zeroes.
- ones(): The ones() function creates an array for a given dimension with all ones.
- fulls(): The full() function generates an array with constant values.
- eyes(): The eye() function creates an identity matrix.
- random(): The random() function creates an array with any given dimension.

```
import numpy as np
# Create an array of all zeros
p = np.zeros((3,3))
print(p)
```

Output:

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

```
# Create an array of all ones
q = np.ones((2,2))
print(q)
```

Output:

```
[[1. 1.]
[1. 1.]]
```

# Create a constant array
r = np.full((2,2), 4)
print(r)
Output:
[[4 4]
[4 4]]

# Create a 4x4 identity matrix
s = np.eye(4)
print(s)
Output:
[[1. 0. 0. 0.]
[0. 1. 0. 0.]
[0. 0. 1. 0.]
[0. 0. 0. 1.]]

# Create an array filled with random values
t = np.random.random((3,3))
print(t)
Output:
[[0.16681892 0.00398631 0.61954178]
[0.52461924 0.30234715 0.58848138]
[0.75172385 0.17752708 0.12665832]]

Let's make an array using the arange() function, as we did in the previous section, and let's check its data type:

```
# Creating an array using arange()
import numpy as np
a = np.arange(1,11)
Print(a)
print(type(a))
print(a.dtype)
print(a.shape)
```

Output:

```
[ 1  2  3  4  5  6  7  8  9 10]
<class 'numpy.ndarray'>
Int64
(10,)
```

- One-dimensional NumPy arrays are also known as vectors.

Selecting array elements

- Let's see an example of a 2*2 matrix:

a = np.array([[5,6],[7,8]])

print(a)

Output:

[[5 6]

[7 8]]

We will now select each item of the matrix one by one as shown in the following code:

print(a[0,0])

Output: 5

print(a[0,1])

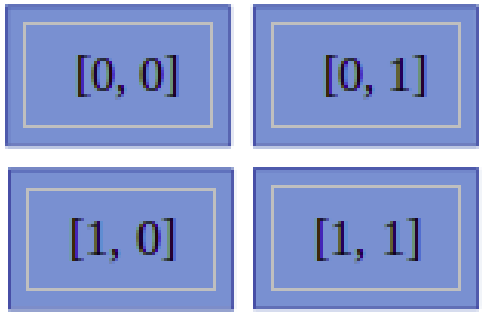
Output: 6

printa([1,0])

Output: 7

printa([1,1])

Output: 8



NumPy array numerical data types

- Python offers three types of numerical data types: integer type, float type, and complex type

Data Type	Details
`bool`	This is a Boolean type that stores a bit and takes `True` or `False` values.
`inti`	Platform integers can be either `int32` or `int64`.
`int8`	Byte store values range from `-128` to `127`.
`int16`	This stores integers ranging from `-32768` to `32767`.
`int32`	This stores integers ranging from `-2 ** 31` to `2 ** 31 -1`.
`int64`	This stores integers ranging from `-2 ** 63` to `2 ** 63 -1`.
`uint8`	This stores unsigned integers ranging from `0` to `255`.
`uint16`	This stores unsigned integers ranging from `0` to `65535`.
`uint32`	This stores unsigned integers ranging from `0` to `2 ** 32 – 1`.
`uint64`	This stores unsigned integers ranging from `0` to `2 ** 64 – 1`.
`float16`	Half-precision float; sign bit with 5 bits exponent and 10 bits mantissa.
`float32`	Single-precision float; sign bit with 8 bits exponent and 23 bits mantissa.
`float64` or `float`	Double-precision float; sign bit with 11 bits exponent and 52 bits mantissa.
`complex64`	Complex number stores two 32-bit floats: real and imaginary number.
`complex128` or `complex`	Complex number stores two 64-bit floats: real and imaginary number.

For each data type, there exists a matching conversion function:

```
print(np.float64(21))
```

Output: 21.0

```
print(np.int8(21.0))
```

Output: 42

```
print(np.bool(21))
```

Output: True

```
print(np.bool(0))
```

Output: False

```
print(np.bool(21.0))
```

Output: True

```
print(np.float(True))
```

Output: 1.0

```
print(np.float(False))
```

Output: 0.0

Many functions have a data type argument, which is frequently optional:

```
arr=np.arange(1,11, dtype= np.float32)
print(arr)
```

Output:

[1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]

- We are not allowed to change a complex number into an integer. If you try to convert complex data types into integers, then you will get TypeError.
- Let's see the following example:

np.int(42.0 + 1.j)

- This results in the following output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-61a3a50e24b1> in <module>
----> 1 np.int(42.0 + 1.j)

TypeError: can't convert complex to int
```

We can convert float values into complex numbers by setting individual pieces. We can also pull out the pieces using the real and imag attributes.

Let's see that using the following example:

```
c= complex(42, 1)
print(c)
```

Output: (42+1j)

```
print(c.real,c.imag)
```

Output: 42.0 1.0

dtype objects

- The dtype tells us the type of individual elements of an array. NumPy array elements have the same data type, which means that all elements have the same dtype. dtype objects are instances of the numpy.dtype class:

```
# Creating an array
import numpy as np
a = np.array([2,4,6,8,10])
print(a.dtype)
```

Output: 'int64'

dtype objects also tell us the size of the data type in bytes using the itemsize property:

```
print(a.dtype.itemsize)
```

Output:8

Data type character codes

- Character codes are included for backward compatibility with Numeric. Numeric is the predecessor of NumPy.

Type	Character Code
Integer	i
Unsigned integer	u
Single-precision float	f
Double-precision float	d
Bool	b
Complex	D
String	S
Unicode	U
Void	V

Let's take a look at the following code to produce an array of single-precision floats:

```
# Create numpy array using arange() function
var1=np.arange(1,11, dtype='f’)
print(var1)
```

Output:

```
[ 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.]
```

Likewise, the following code creates an array of complex numbers:

```
print(np.arange(1,6, dtype='D’))
```

Output:

```
[1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j]
```

dtype constructors

- There are lots of ways to create data types using constructors. Constructors are used to instantiate or assign a value to an object.
- To try out a general Python float, use the following:

```
print(np.dtype(float))
```

Output: float64

- To try out a single-precision float with a character code, use the following:

```
print(np.dtype('f’))
```

Output: float32

- To try out a double-precision float with a character code, use the following:

```
print(np.dtype('d’))
```

Output: float64

dtype attributes

- The dtype class offers several useful attributes.
- For example, we can get information about the character code of a data type using the dtype attribute:

```
# Create numpy array
var2=np.array([1,2,3],dtype='float64’)
print(var2.dtype.char)
```

Output: 'd’

- The type attribute corresponds to the type of object of the array elements:

```
print(var2.dtype.type)
```

Output: <class 'numpy.float64'>

Manipulating array shapes

- Let's learn some new Python functions of NumPy, such as reshape(), flatten(), ravel(), transpose(), and resize():
- reshape() will change the shape of the array:

```
# Create an array
arr = np.arange(12)
print(arr)
```

Output: [0 1 2 3 4 5 6 7 8 9 10 11]

```
# Reshape the array dimension
new_arr=arr.reshape(4,3)
print(new_arr)
```

Output: [[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11]]

```
# Reshape the array dimension
new_arr2=arr.reshape(3,4)
print(new_arr2)
```

Output:
array([[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11]])

- flatten() transforms an n-dimensional array into a one-dimensional array:

```
# Create an array
arr=np.arange(1,10).reshape(3,3)
print(arr)
```

Output:

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

```
print(arr.flatten())
```

Output:

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

- The ravel() function is similar to the flatten() function. It also transforms an n-dimensional array into a one-dimensional array.

The main difference is that flatten() returns the actual array while ravel() returns the reference of the original array.
The ravel() function is faster than the flatten() function because it does not occupy extra memory:

```
print(arr.ravel())
```

Output:

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

- The transpose() function is a linear algebraic function that transposes the given two-dimensional matrix. The word transpose means converting rows into columns and columns into rows:

```
# Transpose the matrix
print(arr.transpose())
```

Output:

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

- The resize() function changes the size of the NumPy array. It is similar to reshape() but it changes the shape of the original array:

```
# resize the matrix
arr.resize(1,9)
print(arr)
```

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

The stacking of NumPy arrays

- NumPy offers a stack of arrays. Stacking means joining the same dimensional arrays along with a new axis. Stacking can be done horizontally, vertically, column-wise, row-wise, or depth-wise:
- **Horizontal stacking**: In horizontal stacking, the same dimensional arrays are joined along with a horizontal axis using the hstack() and concatenate() functions. Let's see the following example:

```
arr1 = np.arange(1,10).reshape(3,3)
print(arr1)
```

Output:

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

```
arr2 = 2*arr1
print(arr2)
```

Output:

```
[[ 2 4 6]
[ 8 10 12]
[14 16 18]]
```

```
# Horizontal Stacking
arr3=np.hstack((arr1, arr2))
print(arr3)
```

Output:

```
[[ 1 2 3 2 4 6]
[ 4 5 6 8 10 12]
[ 7 8 9 14 16 18]]
```

Using concatenate() function

```
# Horizontal stacking using concatenate() function
arr4=np.concatenate((arr1, arr2), axis=1)
print(arr4)
```

Output:

```
[[ 1 2 3 2 4 6]
[ 4 5 6 8 10 12]
[ 7 8 9 14 16 18]]
```

- **Vertical stacking**: In vertical stacking, the same dimensional arrays are joined along with a vertical axis using the vstack() and concatenate() functions.

Let's see the following example:

```
# Vertical stacking
arr5=np.vstack((arr1, arr2))
print(arr5)
```

Output:

```
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[ 2 4 6]
[ 8 10 12]
[14 16 18]]
```

The concatenate() function can also be used to generate vertical stacking with axis parameter value 0:

```
arr6=np.concatenate((arr1, arr2), axis=0)
print(arr6)
```

Output:

```
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[ 2 4 6]
[ 8 10 12]
[14 16 18]]
```

- **Depth stacking**: In depth stacking, the same dimensional arrays are joined along with a third axis (depth) using the dstack() function. Let's see the following
- example:

```
arr7=np.dstack((arr1, arr2))

print(arr7)
```

Output:

```
[[[ 1 2]
[ 2 4]
[ 3 6]]
[[ 4 8]
[ 5 10]
[ 6 12]]
[[ 7 14]
[ 8 16]
[ 9 18]]]
```

- **Column stacking**: Column stacking stacks multiple sequence one-dimensional arrays as columns into a single two-dimensional array. Let's see an example of

```
# Create 1-D array
arr1 = np.arange(4,7)
print(arr1)
```

Output: [4, 5, 6]

```
# Create 1-D array
arr2 = 2 * arr1
print(arr2)
```

Output: [8, 10, 12]

```
# Create column stack
arr_col_stack=np.column_stack((arr1,arr2))
print(arr_col_stack)
```

Output:

```
[[ 4 8]
[ 5 10]
[ 6 12]]
```

- **Row stacking**: Row stacking stacks multiple sequence one-dimensional arrays as rows into a single two-dimensional arrays. Let's see an example of row stacking:# Create row stack

```
arr_row_stack = np.row_stack((arr1,arr2))

print(arr_row_stack)
```

Output:

```
[[ 4 5 6]

[ 8 10 12]]
```

Partitioning NumPy arrays

- NumPy arrays can be partitioned into multiple sub-arrays.
- NumPy offers three types of split functionality: vertical, horizontal, and depth-wise.
- All the split functions by default split into the same size arrays but we can also specify the split location.
- **Horizontal splitting**: In horizontal split, the given array is divided into *N* equal sub-arrays along the horizontal axis using the hsplit() function. Let's see how to split an array:

```
# Create an array
arr=np.arange(1,10).reshape(3,3)
print(arr)
```

Output:

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

```
# Peroform horizontal splitting
arr_hor_split=np.hsplit(arr, 3)
print(arr_hor_split)
```

Output:

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

- **Vertical splitting**: In vertical split, the given array is divided into *N* equal subarrays along the vertical axis using the vsplit() and split() functions. The split function with axis=0 performs the same operation as the vsplit() function:

```
# vertical split
arr_ver_split=np.vsplit(arr, 3)
print(arr_ver_split)
```

Output:

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

- Let's see another function,split(), which can be utilized as a vertical and horizontal split, in the followingexample:

```
# split with axis=0
arr_split=np.split(arr,3,axis=0)
print(arr_split)
```

Output:

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

```
# split with axis=1
arr_split = np.split(arr,3,axis=1)
```

Output:

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

Changing the data type of NumPy arrays

- The astype() function converts the data type of the array.

```
# Create an array
arr=np.arange(1,10).reshape(3,3)
print("Integer Array:",arr)


print("actual Datatype:", arr.dtype)
# Change datatype of array
arr=arr.astype(float)


# print array
print("Float Array:", arr)


# Check new data type of array
print("Changed Datatype:", arr.dtype)
```

Output:

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

actual Datatype: int32

Float Array: [[1. 2. 3.]
[4. 5. 6.]
[7. 8. 9.]]
Changed Datatype: float64
```

- The tolist() function converts a NumPy array into a Python list. Let's see an example of the tolist() function:

```
# Create an array
arr=np.arange(1,10)
print(arr)
# Convert NumPy array to Python List
list1=arr.tolist()
print(list1)
```

Output:

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

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

Creating NumPy views and copies

- Some of the Python functions return either a copy or a view of the input array.
- A Python copy stores the array in another location while a view uses the same memory content.
- This means copies are separate objects and treated as a deep copy in Python. Views are the original base array and are treated as a shallow copy.
- Here are some properties of copies and views:

1. Modifications in a view affect the original data whereas modifications in a copy do not affect the original array.
2. Views use the concept of shared memory.
3. Copies require extra space compared to views. Copies are slower than views.

- Let's understand the concept of copy and view using the following example:

```
# Create NumPy Array
arr = np.arange(1,5).reshape(2,2)
print(arr)
```

Output:

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

- let's perform object copy operations:

```
# Create no copy only assignment
arr_no_copy=arr

# Create Deep Copy
arr_copy=arr.copy()

# Create shallow copy using View
 arr_view=arr.view()
print("Original Array: ",id(arr))
 print("Assignment: ",id(arr_no_copy))
print("Deep Copy: ",id(arr_copy))
print("Shallow Copy(View): ",id(arr_view))
```

Output:

```
Original Array: 140426327484256
Assignment: 140426327484256
Deep Copy: 140426327483856
Shallow Copy(View): 140426327484496
```

- Let's continue with this example and update the values of the original array and check its impact on views and copies:

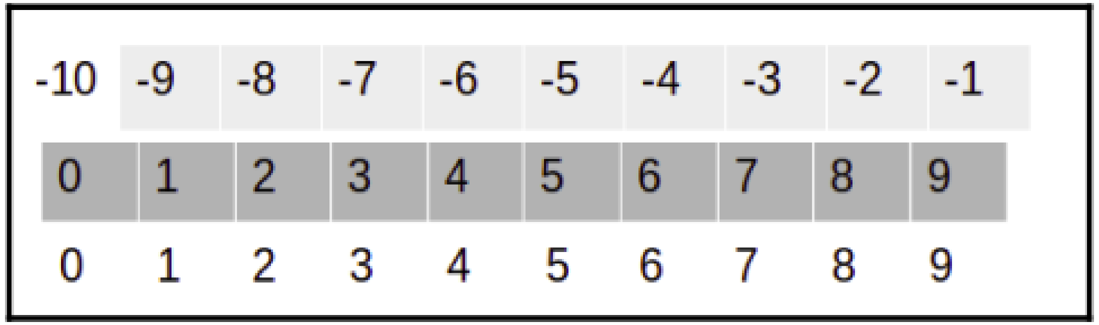
```
# Update the values of original array
arr[1]=[99,89]

# Check values of array view
print("View Array:\n", arr_view)

 # Check values of array copy
print("Copied Array:\n", arr_copy)
```

Output:

```
View Array: [[ 1 2]
[99 89]]
Copied Array: [[1 2]
[3 4]]
```

Slicing NumPy arrays

- Slicing in NumPy is similar to Python lists. Indexing prefers to select a single value while slicing is used to select multiple values from an array.
- NumPy arrays also support negative indexing and slicing. Here, the negative sign indicates the opposite direction and indexing starts from the right-hand side with a starting value of -1:



- Let's check this out using the following code:

```
# Create NumPy Array
arr = np.arange(0,10)
print(arr)
```

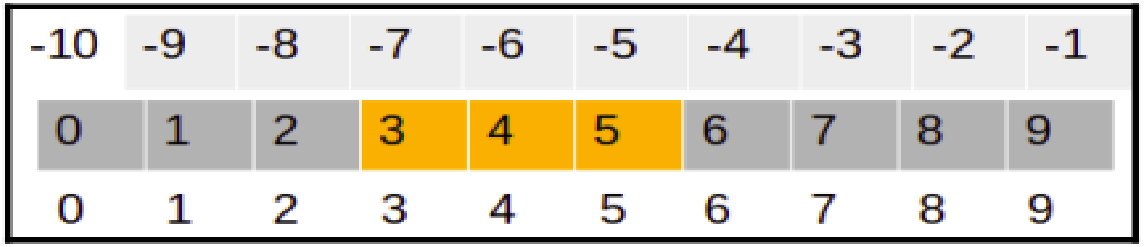
Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

- In the slice operation, we use the colon symbol to select the collection of values.
- Slicing takes three values: start, stop, and step:

print(arr[3:6])

Output: [3, 4, 5]

This can be represented as follows:

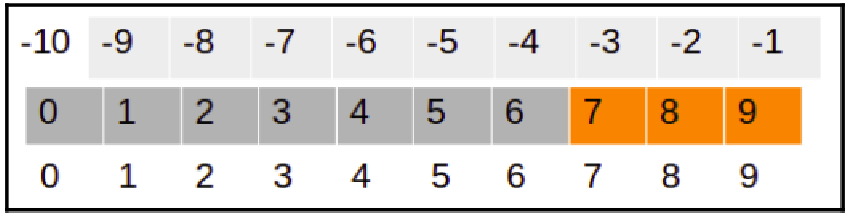


print(arr[3:])

Output: array([3, 4, 5, 6, 7, 8, 9])

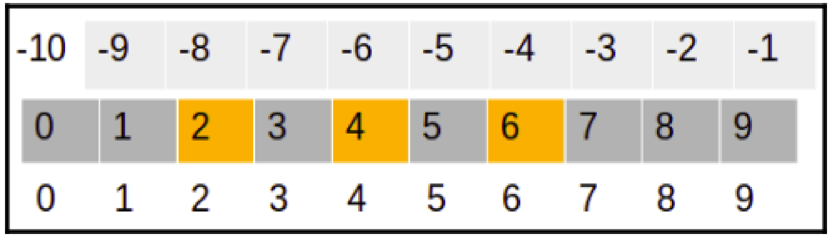
print(arr[-3:])

Output: array([7, 8, 9])



print(arr[2:7:2])

Output: array([2, 4,6])



Boolean and fancy indexing

- Indexing techniques help us to select and filter elements from a NumPy array.
- Boolean indexing uses a Boolean expression in the place of indexes (in square brackets) to filter the NumPy array. This indexing returns elements that have a true value for the Boolean expression:

```
# Create NumPy Array
 arr = np.arange(21,41,2)
print("Orignial Array:\n",arr)


# Boolean Indexing
 print("After Boolean Condition:",arr[arr>30])
```

Output:

Orignial Array: [21 23 25 27 29 31 33 35 37 39]

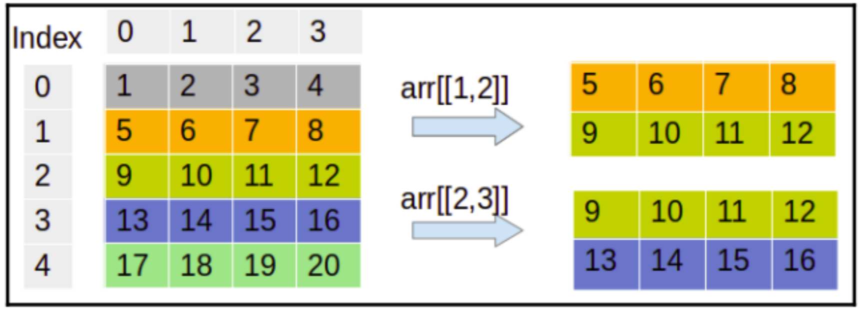
After Boolean Condition: [31 33 35 37 39]

- Fancy indexing is a special type of indexing in which elements of an array are selected by an array of indices.
- This means we pass the array of indices in brackets.
- Fancy indexing also supports multi-dimensional arrays.
- This will help us to easily select and modify a complex multi-dimensional set of arrays.

```
# Create NumPy Array
 arr = np.arange(1,21).reshape(5,4)
 print("Orignial Array:\n",arr)


 # Selecting 2nd and 3rd row
indices = [1,2]
print("Selected 1st and 2nd Row:\n", arr[indices])


# Selecting 3nd and 4th row
indices = [2,3]
print("Selected 3rd and 4th Row:\n", arr[indices])
```

Output:

```
 Orignial Array: [[ 1 2 3 4]
[ 5 6 7 8]
[ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]]

Selected 1st and 2nd Row:
[[ 5 6 7 8]
[ 9 10 11 12]]

Selected 3rd and 4th Row:
 [[ 9 10 11 12]
[13 14 15 16]]
```



Broadcasting arrays

- Python lists do not support direct vectorizing arithmetic operations.
- NumPy offers a faster vectorized array operation compared to Python list loop-based operations.
- Broadcasting functionality checks a set of rules for applying binary functions, such as addition, subtraction, and multiplication, on different shapes of an array.

```
# Create NumPy Array
arr1 = np.arange(1,5).reshape(2,2)
print(arr1)
```

Output:

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

```
# Create another NumPy Array
arr2 = np.arange(5,9).reshape(2,2)
print(arr2)
```

Output:

```
[[5 6]
 [7 8]]
```

```
# Add two matrices
print(arr1+arr2)
```

Output: [[6 8]
 [10 12]]

In all three preceding examples, we can see the addition of two arrays of the same size. This concept is known as broadcasting:

# Multiply two matrices
print(arr1*arr2)
Output: [[5 12]
[21 32]]

# Add a scaler value
print(arr1 + 3)
Output: [[4 5]
 [6 7]]

 # Multiply with a scalar value
print(arr1 * 3)
Output:
 [[3 6]
[9 12]]

Creating pandas Data Frames

- The pandas library is designed to work with a panel or tabular data.
- pandas is a fast, highly efficient, and productive tool for manipulating and analyzing string, numeric, datetime, and time-series data.
- pandas provides data structures such as DataFrames and Series.
- A pandas DataFrame is a tabular, two-dimensional labeled and indexed data structure with a grid of rows and columns.
- Its columns are heterogeneous types.
- It has the capability to work with different types of objects, carry out grouping and joining operations, handle missing values, create pivot tables, and deal with dates.

- Let's create an empty DataFrame:

```
# Import pandas library

import pandas as pd
# Create empty DataFrame
df = pd.DataFrame()
# Header of dataframe.
df.head()
```

Output:

_

- Let's create a DataFrame using a dictionary of the list:

```
# Create dictionary of list
data = {'Name': ['Vijay', 'Sundar', 'Satyam', 'Indira'], 'Age': [23, 45,46, 52 ]}
# Create the pandas DataFrame
df = pd.DataFrame(data)
# Header of dataframe.
df.head()
```

Output:

	Name	Age
0	Vijay	23
1	Sundar	45
2	Satyam	46
3	Indira	52

- Let's create a DataFrame using the list of dictionaries:

```
# Pandas DataFrame by lists of dicts.
# Initialise data to lists.
data =[ {'Name': 'Vijay', 'Age': 23},{'Name': 'Sundar', 'Age': 25},{'Name': 'Shankar', 'Age': 26}]
# Creates DataFrame.
df = pd.DataFrame(data,columns=['Name','Age'])
# Print dataframe header
df.head()
```

- Let's create a DataFrame using a list of tuples:

```
# Creating DataFrame using list of tuples.
data = [('Vijay', 23),( 'Sundar', 45), ('Satyam', 46), ('Indira',52)]
# Create dataframe
df = pd.DataFrame(data, columns=['Name','Age'])
# Print dataframe header
df.head()
```

Output:

```
  Name Age
0 Vijay 23
1 Sundar 45
2 Shankar 46
3 Indira    52
```

Understanding pandas Series

- pandas Series is a one-dimensional sequential data structure that is able to handle any type of data, such as string, numeric, datetime, Python lists, and dictionaries with labels and indexes.
- Series is one of the columns of a DataFrame.
- We can create a Series using a Python dictionary, NumPy array, and scalar value.

- **Using a Python dictionary:** Create a dictionary object and pass it to the Series object.

```
# Creating Pandas Series using Dictionary
dict1 = {0 : 'Ajay', 1 : 'Jay', 2 : 'Vijay'}

# Create Pandas Series
series = pd.Series(dict1)

# Show series
series
```

Output:

```
0           Ajay
1           Jay
2           Vijay
dtype: object
```

- **Using a NumPy array:** Create a NumPy array object and pass it to the Series object.

```
#Load Pandas and NumPy libraries
import pandas as pd
import numpy as np

# Create NumPy array
arr = np.array([51,65,48,59, 68])

# Create Pandas Series
series = pd.Series(arr)
series
```

Output:

```
0       51
1       65
2       48
3       59
4        68
dtype: int64
```

- **Using a single scalar value:** To create a pandas Series with a scalar value, pass the scalar value and index list to a Series object:

```
# load Pandas and NumPy
import pandas as pd
import numpy as np
# Create Pandas Series
series = pd.Series(10, index=[0, 1, 2, 3, 4, 5])
Series
```

Output:

```
0        10
1        10
2        10
3        10
4        10
5        10
dtype: int64
```

- We can also create a series by selecting a column, such as country, which happens to be the first column in the datafile. Then, show the type of the object currently in the local scope:

```
# Import pandas
import pandas as pd
# Load data using read_csv()
df = pd.read_csv("WHO_first9cols.csv")
# Show initial 5 records
df.head()
```

]:

	Country	CountryID	Continent	Adolescent fertility rate (%)	Adult literacy rate (%)	Gross national income per capita (PPP international $)	Net primary school enrolment ratio female (%)	Net primary school enrolment ratio male (%)	Population (in thousands) total
0	Afghanistan	1	1	151.0	28.0	NaN	NaN	NaN	26088.0
1	Albania	2	2	27.0	98.7	6000.0	93.0	94.0	3172.0
2	Algeria	3	3	6.0	69.9	5940.0	94.0	96.0	33351.0
3	Andorra	4	2	NaN	NaN	NaN	83.0	83.0	74.0
4	Angola	5	3	146.0	67.4	3890.0	49.0	51.0	16557.0

```
# Select a series
country_series=df['Country']
# check datatype of series
type(country_series)
```

Output:

```
pandas.core.series.Series
```

- The pandas Series data structure shares some of the common attributes of DataFrames and also has a name attribute. Explore these properties as follows:

```
# Show the shape of DataFrame
print("Shape:", df.shape)
```

Output:

```
Shape: (202, 9)
```

- To check the column list of a DataFrame:

```
# Check the column list of DataFrame
print("List of Columns:", df.columns)
```

Output: List of Columns: Index(['Country', 'CountryID', 'Continent',
'Adolescent fertility rate (%)',
'Adult literacy rate (%)',
'Gross national income per capita (PPP international $)',
'Net primary school enrolment ratio female (%)',
'Net primary school enrolment ratio male (%)’,
'Population (in thousands) total'],
dtype='object')

To check the data types of DataFrame columns:

```
# Show the datatypes of columns
print("Data types:", df.dtypes)
```

Output:

```
Data types: Country
object
             CountryID
int64
             Continent
int64
             Adolescent fertility rate (%)
float64
             Adult literacy rate (%)
float64
             Gross national income per capita (PPP international $)
float64
             Net primary school enrolment ratio female (%)
float64
             Net primary school enrolment ratio male (%)
float64
             Population (in thousands) total
float64
             dtype: object
```

- The slicing of a pandas Series:

```
# Pandas Series Slicing
country_series[-5:]
```

Output:

```
197 Vietnam
198 West Bank and Gaza
199 Yemen
200 Zambia
201 Zimbabwe
Name: Country, dtype: object
```

Reading and querying the Quandl data

- Quandl is a Canada-based company that offers commercial and alternative financial data for investment data analyst.
- Quandl understands the need for investment and financial quantitative analysts. It provides data using API, R, Python, or Excel.
- To install the Quandl package using pip:

```
$ pip3 install Quandl
```

- Downloading installers from https:/ / pypi.python. org/ pypi/ Quandl

Describing pandas Data Frames

- The pandas DataFrame has a dozen statistical methods. The following table lists these methods, along with a short description of each:

Method	Description
describes	This method returns a small table with descriptive statistics.
count	This method returns the number of non-NaN items.
mad	This method calculates the mean absolute deviation, which is a robust measure similar to standard deviation.
median	This method returns the median. This is equivalent to the value at the 50th percentile.
min	This method returns the minimum value.
max	This method returns the maximum value.
mode	This method returns the mode, which is the most frequently occurring value.
std	This method returns the standard deviation, which measures dispersion. It is the square root of the variance.
var	This method returns the variance.
skew	This method returns skewness. Skewness is indicative of the distribution symmetry.
kurt	This method returns kurtosis. Kurtosis is indicative of the distribution shape.

Describe the dataset

df.describe()

	CountryID	Continent	Adolescent fertility rate (%)	Adult literacy rate (%)	Gross national income per capita (PPP international $)	Net primary school enrolment ratio female (%)	Net primary school enrolment ratio male (%)	Population (in thousands) total
count	202.000000	202.000000	177.000000	131.000000	178.000000	179.000000	179.000000	1.890000e+02
mean	101.500000	3.579208	59.457627	78.871756	11250.112360	84.033520	85.698324	3.409964e+04
std	58.456537	1.808263	49.105286	20.415760	12586.753417	17.788047	15.451212	1.318377e+05
min	1.000000	1.000000	0.000000	23.600000	260.000000	6.000000	11.000000	2.000000e+00
25%	51.250000	2.000000	19.000000	68.400000	2112.500000	79.000000	79.500000	1.328000e+03
50%	101.500000	3.000000	46.000000	86.500000	6175.000000	90.000000	90.000000	6.640000e+03
75%	151.750000	5.000000	91.000000	95.300000	14502.500000	96.000000	96.000000	2.097100e+04
max	202.000000	7.000000	199.000000	99.800000	60870.000000	100.000000	100.000000	1.328474e+06

Count number of observation

df.count()

```
Country                                              202
CountryID                                            202
Continent                                            202
Adolescent fertility rate (%)                        177
Adult literacy rate (%)                              131
Gross national income per capita (PPP international $)    178
Net primary school enrolment ratio female (%)        179
Net primary school enrolment ratio male (%)          179
Population (in thousands) total                      189
dtype: int64
```

Compute median of all the columns

df.median()

```
CountryID                                               101.5
Continent                                                 3.0
Adolescent fertility rate (%)                            46.0
Adult literacy rate (%)                                  86.5
Gross national income per capita (PPP international $)   6175.0
Net primary school enrolment ratio female (%)            90.0
Net primary school enrolment ratio male (%)              90.0
Population (in thousands) total                        6640.0
dtype: float64
```

Compute the standard deviation of all the columns

df.std()

```
CountryID                                                58.456537
Continent                                                 1.808263
Adolescent fertility rate (%)                            49.105286
Adult literacy rate (%)                                  20.415760
Gross national income per capita (PPP international $)   12586.753417
Net primary school enrolment ratio female (%)            17.788047
Net primary school enrolment ratio male (%)              15.451212
Population (in thousands) total                         131837.708677
dtype: float64
```

Grouping and joining pandas Data Frame

- Grouping is a kind of data aggregation operation.
- The grouping term is taken from a relational database.
- Relational database software uses the `group by` keyword to group similar kinds of values in a column.
- We can apply aggregate functions on groups such as mean, min, max, count, and sum.
- The `pandas` DataFrame also offers similar kinds of capabilities. Grouping operations are based on the split-apply-combine strategy.
- It first divides data into groups and applies the aggregate operation, such as mean, min, max, count, and sum, on each group and combines results from each group:

Working with missing values

- Most real-world datasets are messy and noisy. Due to their messiness and noise, lots of values are either faulty or missing. pandas offers lots of built-in functions to deal with missing values in DataFrames:
- **Check missing values in a DataFrame**: pandas' isnull() function checks for the existence of null values and returns True or False, where True is for null and False is for not-null values. The sum() function will sum all the True values and returns the count of missing values.

```
# Count missing values in DataFrame

pd.isnull(df).sum()
```

```
Country                                                0
CountryID                                              0
Continent                                              0
Adolescent fertility rate (%)                         25
Adult literacy rate (%)                               71
Gross national income per capita (PPP international $)    24
Net primary school enrolment ratio female (%)         23
Net primary school enrolment ratio male (%)           23
Population (in thousands) total                       13
dtype: int64
```

- **Drop missing values**: A very naive approach to deal with missing values is to drop them for analysis purposes. pandas has the dropna() function to drop or delete such observations from the DataFrame.
- Here, the inplace=True attribute makes the changes in the original DataFrame:

```
# Drop all the missing values
df.dropna(inplace=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 118 entries, 1 to 200
Data columns (total 9 columns):
Country                                         118 non-null object
CountryID                                       118 non-null int64
Continent                                       118 non-null int64
Adolescent fertility rate (%)                   118 non-null float64
Adult literacy rate (%)                         118 non-null float64
Gross national income per capita (PPP international $)    118 non-null float64
Net primary school enrolment ratio female (%)   118 non-null float64
Net primary school enrolment ratio male (%)     118 non-null float64
Population (in thousands) total                 118 non-null float64
dtypes: float64(6), int64(2), object(1)
memory usage: 9.2+ KB
```

- **Fill the missing values**: Another approach is to fill the missing values with zero, mean, median, or constant values:

```
# Fill missing values with 0
df.fillna(0,inplace=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 202 entries, 0 to 201
Data columns (total 9 columns):
Country                                         202 non-null object
CountryID                                       202 non-null int64
Continent                                       202 non-null int64
Adolescent fertility rate (%)                   202 non-null float64
Adult literacy rate (%)                         202 non-null float64
Gross national income per capita (PPP international $)    202 non-null float64
Net primary school enrolment ratio female (%)   202 non-null float64
Net primary school enrolment ratio male (%)     202 non-null float64
Population (in thousands) total                 202 non-null float64
dtypes: float64(6), int64(2), object(1)
memory usage: 14.3+ KB
```

Creating pivot tables

- A pivot table is a summary table. It is the most popular concept in Excel.
- Most data analysts use it as a handy tool to summarize their results.
- pandas offers the pivot_table() function to summarize DataFrames.
- A DataFrame is summarized using an aggregate function, such as mean, min, max, or sum.

```
# Import pandas
import pandas as pd
# Load data using read_csv()
purchase = pd.read_csv("purchase.csv")
# Show initial 10 records
purchase.head(10)
```

	Weather	Food	Price	Number
0	cold	soup	3.745401	8
1	hot	soup	9.507143	8
2	cold	icecream	7.319939	8
3	hot	chocolate	5.986585	8
4	cold	icecream	1.560186	8
5	hot	icecream	1.559945	8
6	cold	soup	0.580836	8

- # Summarise dataframe using pivot table

```
pd.pivot_table(purchase,values='Number', index=['Weather',],columns=['Food'], aggfunc=np.sum)
```

Food	chocolate	icecream	soup
Weather			
cold	NaN	16.0	16.0
hot	8.0	8.0	8.0

Dealing with dates

- Dealing with dates is messy and complicated. You can recall the Y2K bug, the upcoming 2038 problem, and time zones dealing with different problems.
- In time-series datasets, we come across dates.
- pandas offers date ranges, resamples time-series data, and performs date arithmetic operations.

Create a range of dates starting from January 1, 2020, lasting for 45 days, as follows:

```
# Date range function
pd.date_range('01-01-2000', periods=45, freq='D')
```

Output:

```
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04','2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
'2000-01-09', '2000-01-10', '2000-01-11', '2000-01-12','2000-01-13', '2000-01-14', '2000-01-15', '2000-01-16',
'2000-01-17', '2000-01-18', '2000-01-19', '2000-01-20','2000-01-21', '2000-01-22', '2000-01-23', '2000-01-24',
'2000-01-25', '2000-01-26', '2000-01-27', '2000-01-28','2000-01-29', '2000-01-30', '2000-01-31', '2000-02-01',
'2000-02-02', '2000-02-03', '2000-02-04', '2000-02-05','2000-02-06', '2000-02-07', '2000-02-08', '2000-02-09',
'2000-02-10', '2000-02-11', '2000-02-12', '2000-02-13','2000-02-14'],
dtype='datetime64[ns]', freq='D')
```

to_datetime(): to_datetime() converts a timestamp string into datetime:

Convert argument to datetime

pd.to_datetime('1/1/1970')

Output: Timestamp('1970-01-01 00:00:00’)

- We can convert a timestamp string into a datetime object in the specified format:

Convert argument to datetime in specified format

pd.to_datetime(['20200101', '20200102'], format='%Y%m%d')

Output:

DatetimeIndex(['2020-01-01', '2020-01-02'], dtype='datetime64[ns]',freq=None)

- **Handling an unknown format string**: Unknown input format can cause value errors.
- We can handle this by using an errors parameter with coerce. coerce will set invalid strings to NaT

```
# Value Error
pd.to_datetime(['20200101', 'not a date'])
```

Output:

ValueError: ('Unknown string format:', 'not a date’)

```
# Handle value error
pd.to_datetime(['20200101', 'not a date'], errors='coerce')
```

Output:

DatetimeIndex(['2020-01-01', 'NaT'], dtype='datetime64[ns]',freq=None)

linear algebra

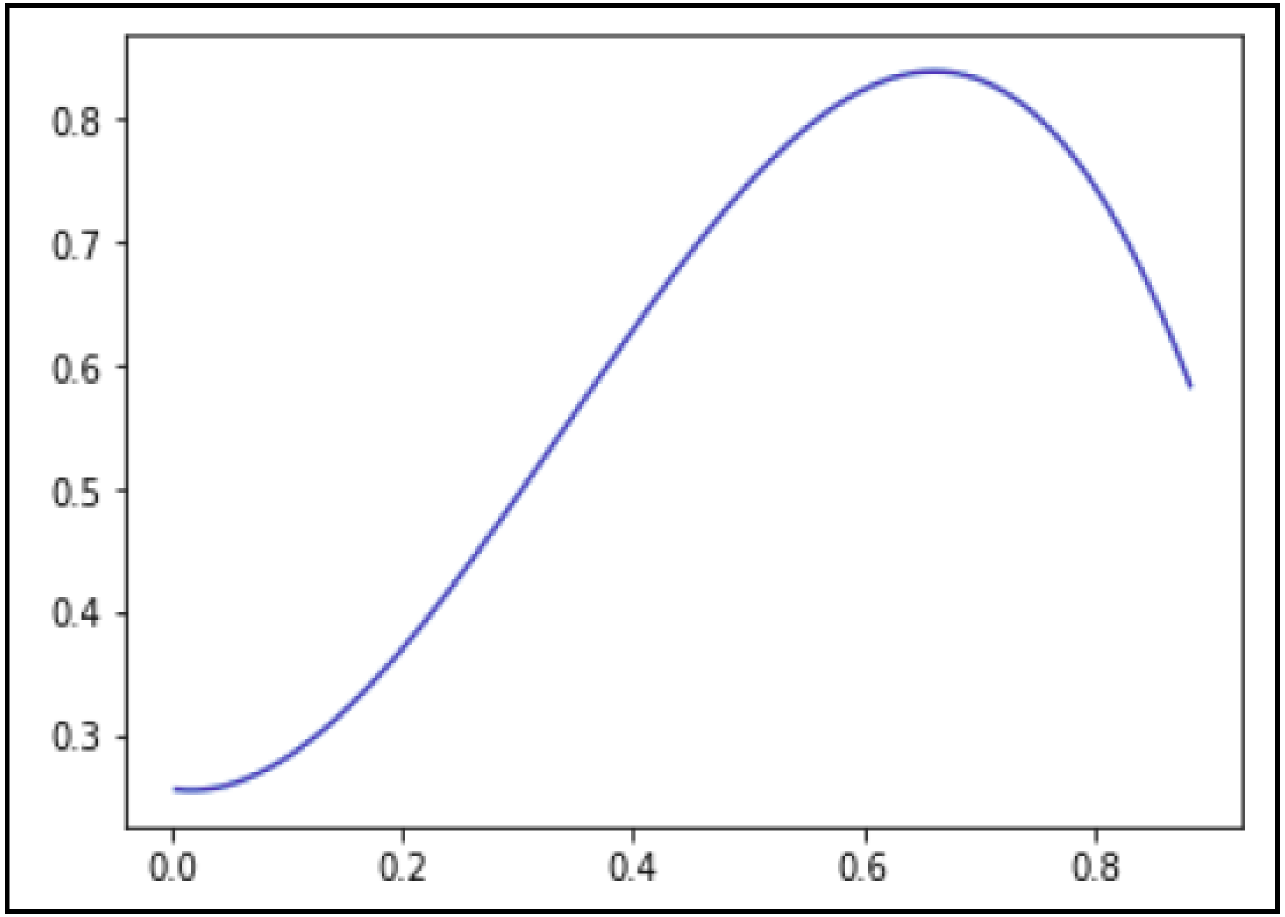
- linear algebra is one of the fundamental mathematical subjects that is the core foundation for any data professional.
- Linear algebra is useful for working with vectors and matrices.
- A strong understanding of linear algebra enables data analysts and data scientists to grasp the inner workings of machine learning and deep learning algorithms, allowing them to adapt and tailor the algorithms to meet specific business requirements.

Fitting to polynomials with numpy

- Polynomials are mathematical expressions with non-negative strategies.
- Examples of polynomial functions are linear, quadratic, cubic, and quartic functions.
- NumPy offers the polyfit() function to generate polynomials using least squares. This function takes x-coordinate, y-coordinate, and degree as parameters, and returns a list of polynomial coefficients.
- NumPy also offers polyval() to evaluate the polynomial at given values. This function takes coefficients of polynomials and arrays of points and returns resultant values of polynomials.
- Another function is linspace(), which generates a sequence of equally separated values. It takes the start, stop, and the number of values between the start-stop range and returns equally separated values in the closed interval.

```
# Import required libraries NumPy, polynomial and matplotlib
import numpy as np
import matplotlib.pyplot as plt
# Generate two random vectors
v1=np.random.rand(10)
v2=np.random.rand(10)
# Creates a sequence of equally separated values
sequence = np.linspace(v1.min(),v1.max(), num=len(v1)*10)
# Fit the data to polynomial fit data with 4 degrees of the polynomial
coefs = np.polyfit(v1, v2, 3)
# Evaluate polynomial on given sequence
polynomial_sequence = np.polyval(coefs,sequence)
# plot the polynomial curve
plt.plot(sequence, polynomial_sequence)
# Show plot
plt.show()
```

This results in the following output:



Determinant

- It is a scalar value that is calculated from a square matrix.
- The determinant is a fundamental operation that helps us in the inverse matrix and in solving linear equations.
- Determinants are only calculated for square matrices. A square matrix has an equal number of rows and columns.
- The numpy.linalg subpackage provides the det() function for calculating the determinant of a given input matrix.

```python
# Import numpy
import numpy as np
# Create matrix using NumPy
mat=np.mat([[2,4],[5,7]])
print("Matrix:\n",mat)
# Calculate determinant
print("Determinant:",np.linalg.det(mat))
```

Output:

```
Matrix:
[[2 4]
[5 7]]
Determinant: -5.999999999999998
```

finding the rank of a matrix

- The rank of a matrix represents the amount of information that is kept in the matrix.
- A lower rank means less information, and a higher rank means a high amount of information.
- Rank can be defined as the number of independent rows or columns of a matrix.
- The numpy.linalg subpackage provides the matrix_rank() function.

```
# import required libraries
import numpy as np
from numpy.linalg import matrix_rank
# Create a matrix
mat=np.array([[5, 3, 1],[5, 3, 1],[1, 0, 5]])
# Compute rank of matrix
print("Matrix: \n", mat)
print("Rank:",matrix_rank(mat))
```

Output:

```
Matrix:
[[5 3 1]
[5 3 1]
[1 0 5]]
Rank: 2
```

Matrix inverse using numpy

- A matrix is a rectangular sequence of numbers, expressions, and symbols organized in rows and columns.
- The multiplication of a square matrix and its inverse is equal to the identity matrix I.
- We can write it using the following equation:

$$AA^{-1} = I$$

- The numpy.linalg subpackage provides a function for an inverse operation: the inv() function.

```
# Import numpy
import numpy as np
# Create matrix using NumPy
mat=np.mat([[2,4],[5,7]])
print("Input Matrix:\n",mat)
# Find matrix inverse
inverse = np.linalg.inv(mat)
print("Inverse:\n",inverse)
```

Output:

```
Input Matrix:
[[2 4]
[5 7]]
Inverse:
[[-1.16666667 0.66666667]
[ 0.83333333 -0.33333333]]
```

solving linear equations using numpy

- Matrix operations can transform one vector into another vector.
- These operations will help us to find the solution for linear equations. NumPy provides the solve() function to solve linear equations in the form of Ax=B.
- Here, A is the n*n matrix, B is a one-dimensional array and x is the unknown one-dimensional vector.
- We will also use the dot() function to compute the dot product of two floating-point number arrays.

1. Create matrix A and array B for a given equation, like this:

$$x1+x2 = 200$$
$$3x1+2x2 = 450$$

```
# Create matrix A and Vector B using NumPy
A=np.mat([[1,1],[3,2]])
print("Matrix A:\n",A)
B = np.array([200,450])
print("Vector B:", B)
```

Output:

```
Matrix A:
[[1 1]
[3 2]]
Vector B: [200 450]
```

2. # Solve linear equations

```
solution = np.linalg.solve(A, B)
print("Solution vector x:", solution)
```

Output:

Solution vector x: [50. 150.]

3. Check the solution using the dot() function, like this:

```
# Check the solution
print("Result:",np.dot(A,solution))
```

Output:

Result: [[200. 450.]]

Decomposing a matrix using svd

- Matrix decomposition is the process of splitting a matrix into parts. It is also known as matrix factorization.
- There are lots of matrix decomposition methods available such as

1. lower-upper (LU) decomposition
2. QR decomposition (where Q is orthogonal and R is upper-triangular)
3. Cholesky decomposition, and
4. SVD.

- Eigenanalysis decomposes a matrix into vectors and values.
- SVD decomposes a matrix into the following parts: singular vectors and singular values.
- SVD is widely used in signal processing, computer vision, **natural language processing** (**NLP**), and machine learning—for example, topic modeling and recommender systems where SVD is widely accepted and implemented in real-life business solutions.

$$A = U\Sigma V^T$$

Here, A is a m x n left singular matrix, Σ is a n x n diagonal matrix, V is a m x n right singular matrix, and V^T is the transpose of the V. The numpy.linalg subpackage offers the svd() function to decompose a matrix.

```
# import required libraries
import numpy as np
from scipy.linalg import svd
# Create a matrix
mat=np.array([[5, 3, 1],[5, 3, 0],[1, 0, 5]])
# Perform matrix decomposition using SVD
U, Sigma, V_transpose = svd(mat)
print("Left Singular Matrix:",U)
print("Diagonal Matrix: ", Sigma)
print("Right Singular Matrix:", V_transpose)
```

Output:

```
Left Singular Matrix: [[-0.70097269 -0.06420281 -0.7102924 ]
[-0.6748668 -0.26235919 0.68972636]
[-0.23063411 0.9628321 0.14057828]]

Diagonal Matrix: [8.42757145    4.89599358    0.07270729]

Right Singular Matrix: [[-0.84363943 -0.48976369 -0.2200092]
[-0.13684207 -0.20009952 0.97017237]
[ 0.51917893 -0.84858218 -0.10179157]]
```

Eigenvectors and Eigenvalues using NumPy

- Eigenvectors and Eigenvalues are the tools required to understand linear mapping and transformation.
- Eigenvalues are solutions to the equation $Ax = \lambda x$.

  Here, A is the square matrix, x is the eigenvector, and λ is eigenvalues.
- The `numpy.linalg` subpackage provides two functions, `eig()` and `eigvals()`.
- The `eig()` function returns a tuple of eigenvalues and eigenvectors, and `eigvals()` returns the eigenvalues.

- Eigenvectors and eigenvalues are the core fundamentals of linear algebra. Eigenvectors and eigenvalues are used in SVD, spectral clustering, and PCA.

- Create the matrix using the NumPy mat() function:

```
# Import numpy
import numpy as np
# Create matrix using NumPy
mat=np.mat([[2,4],[5,7]])
print("Matrix:\n",mat)
```

This results in the following output:

```
Matrix: [[2 4]
[5 7]]
```

Compute eigenvectors and eigenvalues using the eig() function, like this:

```
# Calculate the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(mat)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```

This results in the following output:

```
Eigenvalues: [-0.62347538 9.62347538]
Eigenvectors: [[-0.83619408 -0.46462222]
[ 0.54843365 -0.885509 ]]
```

Compute eigenvalues using the `eigvals()` function, like this:

```
# Compute eigenvalues
eigenvalues= np.linalg.eigvals(mat)
print("Eigenvalues:", eigenvalues)
```

This results in the following output:

```
Eigenvalues: [-0.62347538 9.62347538]
```

Generating random numbers

- Random numbers offer a variety of applications such as Monte Carlo simulation, cryptography, initializing passwords, and stochastic processes.
- It is not easy to generate real random numbers, so in reality, most applications use pseudo-random numbers.
- Pseudo numbers are adequate for most purposes except for some rare cases.
- Random numbers can be generated from discrete and continuous data.
- The numpy.random() function will generate a random number matrix for the given input size of the matrix.

```
# Import numpy
import numpy as np
# Create an array with random values
random_mat=np.random.random((3,3))
print("Random Matrix: \n",random_mat)
```

output:

```
Random Matrix: [[0.90613234 0.83146869 0.90874706]
[0.59459996 0.46961249 0.61380679]
[0.89453322 0.93890312 0.56903598]]
```