4. Numpy Arrays

4.1 Introduction

Array is a collection of items of same type. Each item in the list is associated with an index. Python doest not have a built-in support for arrays. The features of arrays can be achieved using lists in python.

In python, arrays can be created using

- `array` class in the standard python library.
- `NumPy` package which is one of the fundamental scientific packages in python.

4.2 `array` Class

One dimensional arrays can be created using the `array` class. This can be done by importing the `array` module.

```
array(datatype, valuelist)
```

Listing 4.1: Syntax for creating an array

Here, `datatype` specifies the type of values in the array. It can be any data type such as integer, float, double etc. It is represented using a type code. Some of the type codes are listed below.

- `i` for signed integer
- `l` for signed long
- `f` for float
- `d` for double

```
import array as arr
#Creating an integer array
a = arr.array('i', [11, 22, 34])

#Creating a float array
b = arr.array('f', [2.5, 3.2, 3.3])
```

Listing 4.2: Example for array

4.2.1 Accessing elements of an array

Individual elements inside an array can be accessed by writing the index of the element within square brackets. A negative indexing, starting from -1 for the last element of an array can also be used. Slicing operator (:) can be used to retrieve a range of values from an array.

```
import array as arr
#Creating an integer array
a = arr.array('i', [11, 22, 34, 45])

#Accessing the element at index 2 and prints 34
print(a[2])

#Accessing the element at last index and prints 45
print(a[-1])

#Accessing the elements starting from index 1 to index 2
#Prints array('i', [22, 34])
print(a[1:3])
```

Listing 4.3: Accessing array elements

4.2.2 Adding elements to an array using `insert()` and `append()` methods

Elements can be added to an array at specified index using the `insert()` method. `append()` method is used to add an element at the end of an array.

```
insert(index, element)
append(element)
```

Listing 4.4: Syntax for `insert()` and `append`

```
import array as arr
#Creating an integer array
a = arr.array('i', [11, 22, 34, 45])

#Adds an element at index 2
a.insert(2, 99)
#Prints array('i', [11, 22, 99, 34, 45])
print(a)

#Adds an element at the end
a.append(88)
#Prints array('i', [11, 22, 99, 34, 45, 88])
print(a)
```

Listing 4.5: Adding elements to an array

Removing elements in an array using `remove()` and `pop()` methods

- `remove(element)` method can be used to remove the specified `element` from a list.
- `pop(index)` removes the item from the specified `index` and returns it. The argument `index` is optional. If not specified, the last item in the array will be popped out.

```
import array as arr
#Creating an integer array
a = arr.array('i', [11, 22, 34, 45])

#Adds an element at index 2
a.remove(34)
#Prints array('i', [11, 22, 45])
print(a)
```

```
9
10  #Pops the last element 45
11  element = a.pop()
12  #Prints 45
13  print(element)
```

Listing 4.6: Removing items in an array

4.3 NumPy Module

It is a python library which provides a multidimensional array object and a set of operations for these arrays. These operations include mathematical, logical, shape manipulation, basic linear algebra, basic statistical operations etc. `Numpy` includes an `ndarray` object which encapsulates n-dimensional arrays of elements of same type and operations on them. Some of the differences between an `ndarray` and normal python list are

- Size of an `ndarray` is fixed at the time of creation. Python lists can grow dynamically.
- All elements of an `ndarray` are of the same data type, whereas lists can have elements of different types.
- `Numpy` arrays have many mathematical and statical operations that can be executed on large amount of data more efficiently.

Using `NumPy`, an array is created using the class `ndarray`. The `array` class in python's standard library can create only one dimensional arrays whereas `ndarray` can be used to create multi dimensional arrays and it has more functionalities. Each dimension of an `ndarray` is called an **axis**. Some of the attributes of an `ndarray` object are:

- `ndarray.ndim`
  Returns the number of axes (dimensions) of the array.
- `ndarray.shape`
  Returns the dimension of the array as a tuple, containing integers indicating the size of the array in each dimension.
- `ndarray.size`
  Returns the total number elements in the array.
- `ndarray.dtype`
  Returns an object representing the type of elements in the array.

4.3.1 Creating Arrays using NumPy

There are different ways to create arrays using `NumPy` module.

`array()` Function

The `array()` function can be used to create arrays from regular python lists or tuples. Type of the resulting array will be determined based on the type of elements in the sequence. A sequence of sequence is transformed into a two dimensional array. A sequence of sequence of sequence is transformed into a three dimensional array and so on.

```
1  import numpy as np
2  #Creates an ndarray from the list [22,33,45]
3  a = np.array([22, 35, 45])
4  #Prints [22 35 45]
5  print(a)
6  #Prints int64
7  print(a.dtype)
8  #Creates a 2D array with dimension 2 x 2
9  a = np.array([[1,2], [3,4]])
```

```
print(a) #Prints [[1 2] [3 4]]
```

Listing 4.7: Creating an ndarray

4.3.2 `zeros()`, `ones()` and `empty()` Functions

- `zeros()` function creates an array full of zeros.
- `ones()` function creates an array full of ones.
- `empty()` function creates an array whose initial content is random and depends on the state of the memory.

```
numpy.zeros(shape, dtype, order)
numpy.ones(shape, dtype=float, order='C')
numpy.empty(shape, dtype=float, order='C')
```

Listing 4.8: `zeros()`, `ones()` and `empty()` Syntaxes

- `shape` is an integer or tuple which specifies the number of elements in each dimension.
- `dtype` is an optional argument which specifies the data type of elements. By default, the `dtype` of the created arrays will be `float64`.
- `order` is an optional argument which can take up value either `'C'` (row-major) or `'F'` (column-major). It specifies how to store the values (row major or column major). By default, the value will be `'C'`.

```
import numpy as np

print("Zeros")
a = np.zeros(5)
print(a)         #Prints [0. 0. 0. 0. 0.]
a = np.zeros((2,3))
print(a)
'''Prints [[0. 0. 0.]
           [0. 0. 0.]]'''

print("Ones")
a = np.ones(5)
print(a)         #Prints [1. 1. 1. 1. 1.]
a = np.ones((2,3))
print(a)
'''Prints [[1. 1. 1.]
           [1. 1. 1.]]'''

print("Empty")
a = np.empty(5)
print(a)         #Prints [1. 1. 1. 1. 1.]
a = np.empty((2,3))
print(a)
'''Prints [[1. 1. 1.]
           [1. 1. 1.]]'''
```

Listing 4.9: `zeros()`, `ones()` and `empty()` Functions

4.3.3 `arange()` Function

`arange()` function returns an array containing evenly spaced values within an interval.

```
numpy.arange(start, stop, step, dtype)
```

Listing 4.10: Syntax for `arange()` Function

- `start` is an integer or real number. It is an optional argument. It specifies the start of interval. The interval includes this value. The default `start` value is 0.
- `stop` is an integer or real number. It specifies the end of interval. The interval does not include this value.
- `step` is an integer or real number. It is an optional argument. It specifies spacing between values. The default `start` value is 1.
- `dtype` specifies the type of the output array. If `dtype` is not given, it will infer the data type from the other input arguments.

```
import numpy as np

'''Creates array containing elements within interval 0 (default)
and 5 (does not include). Step is 1 (default)'''
a=np.arange(5)
print(a)    #Returns [0 1 2 3 4]


'''Creates array containing elements within interval 4 (includes)
and 10 (does not include). Step is 1 (default)'''
a=np.arange(4,10)
print(a)    #Returns [4 5 6 7 8 9]

'''Creates array containing elements within interval 4 (includes)
and 7 (does not include). Step is 0.5'''
a=np.arange(4,7,0.5)
print(a)    #Returns [4. 4.5 5. 5.5 6. 6.5]
```

Listing 4.11: Example for `arange()` Function

4.3.4 `linspace()` Function

`linspace()` function returns an array containing a specified number of samples, evenly spaced within an interval.

```
numpy.linspace(start, stop, num, endpoint, rettype, dtype)
```

Listing 4.12: Syntax for `linspace()` Function

- `start` is an integer or real number. It specifies the start of interval.
- `stop` is an integer or real number. It specifies the end of interval unless `endpoint` is set to `False`. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that `stop` is excluded.
- `num` is an integer. It is an optional argument. It specifies the number of samples to be generated. Default value is `50`.
- `endpoint` is an optional boolean value. If `True`, `stop` is the last sample. Otherwise, it is not included. Default is `True`.
- `rettype` is an optional boolean value. If `True`, returns (`samples`, `step`), where `step` is the spacing between samples.
- `dtype` is an optional argument which specifies the type of the output array. If `dtype` is not given, the data type is inferred from `start` and `stop`.

```
a=np.linspace(2,6,6)
print(a)          #Prints [2. 2.8 3.6 4.4 5.2 6. ]

a=np.linspace(2,6,6,endpoint=False)
print(a) #Prints [2. 2.66666667 3.33333333 4. 4.66666667 5.33333333]

(a,step) = np.linspace(2,6,6,retstep=True)
```

```
8 print(a)          #Prints [2. 2.8 3.6 4.4 5.2 6. ]
9 print(step)         #Prints 0.8
```

Listing 4.13: Example for `linspace()` Function

- Line No. 1: Creates an array containing 6 evenly spaced elements within interval 2 and 6
- Line No. 4: Creates an array containing 6 evenly spaced elements within interval 2 and 6. Since `endpoint` is set to `False`, first it creates 7 evenly spaced elements within interval 2 and 6 and then excludes the last one (6).
- Line No. 7: Creates an array containing 6 evenly spaced elements within interval 2 and 6. Also returns the value for `step`.

4.3.5 `random()` Function

`random()` function generates random samples within the half-open interval [0.0, 1.0). It returns an array of specified shape and fills it with random floats.

```
1 numpy.random.random(size)
```

Listing 4.14: Syntax for `random()` Function

where `size` is an integer or tuple of integers which specifies the number of elements in each dimension of the array.

```
1 import numpy as np
2
3 a = np.random.random(5)
4 print(a)
5
6 a = np.random.random((2,3))
7 print(a)
```

Listing 4.15: Example for `random()` Function

- Line No. 3: Creates an array containing 5 random number in the half open interval [0.0,1.0).
- Line No. 6: Creates an array containing 6 random number in the half open interval [0.0,1.0) and arranges them as a 2 dimensional array (2×3).

4.3.6 Indexing, Slicing and Iterating

Indexing

- Indexing a `NumPy` array means accessing the elements inside the array. Array elements can be accessed using the `[]` operator. Index starts at 0.
- Multi dimensional array elements can be accessed using comma separated indices (one for each dimension) inside `[]`.
- A negative indexing, starting from -1 for the last element of an array can also be used.

```
1  import numpy as np
2  a = np.array((11,22,33,44,55))
3  print(a[1])                    #Prints 22
4  a = np.array((11,22,33,44,55))
5  print(a[-1])                   #Prints 55
6  a = np.array((11,22,33,44,55))
7  print(a[-2])                   #Prints 44
8  #2D Array
9  b = np.array(((1,2,3),(4,5,6)))
10 print(b[1,1])                  #Prints 5
11 b = np.array(((1,2,3),(4,5,6)))
12 print(b[1,-1])                 #Prints 6
```

Listing 4.16: Indexing Example

Slicing

- Part of an array can be retrieved using a slice instead of index inside `[]`.
- A slice can be specified as `[start:end:step]`.
  - `start` is optional. Default is 0.
  - `end` is optional. Default is the last index of the array.
  - `step` is 1 by default.
- For multi dimensional arrays, slice can be specified in each dimension.

```
import numpy as np
#Creates arrray [10 11 12 13 14 15 16 17 18 19]
a=np.arange(10,20)
print(a[1:7:2])   #Prints [11 13 15]
print(a[:7:2])    #Prints [10 12 14 16]
print(a[:7])      #Prints [10 11 12 13 14 15 16]
print(a[::2])     #Prints [10 12 14 16 18]
print(a[-4:-1])   #Prints [16 17 18]

b=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(b[1,1:3])     #Prints [6,7]
print(b[0:2,2])     #Prints [3,7]
print(b[1:3,1:3])   #Prints [[6,7] [10,11]]
```

Listing 4.17: Slicing Example

Iterating

- `NumPy` arrays can be iterated using `for` loop.
- Nested loops are required for iterating over multi dimensional array.

```
import numpy as np
#Creates arrray [10 11 12 13 14 15 16 17 18 19]
a = np.arange(10,20)
#Prints individual elements of the array
for e in a:
  print(e)
#Creating a 2D Array
b = np.array([[1,2,3],[4,5,6],[7,8,9]])
#Prints each row
for r in b:
  print(e)
#Prints individual elements
for r in b:
  #For each element in the row
  for e in r:
    print(e)
```

Listing 4.18: Iteration Example

- `NumPy` package contains an iterator object `nditer`, which allows to perform advanced iterations on `NumPy` arrays.
- `nditer` allows to visit each element of an array regardless of number of dimensions of the array.

```
import numpy as np
#Creating a 2D Array
b = np.array([[1,2,3],[4,5,6],[7,8,9]])
#Prints each element
for e in np.nditer(b):
  print(e, end=' ')   #Prints 1 2 3 4 5 6 7 8 9
```

Listing 4.19: Iteration using `nditer`

- The `order` argument of `nditer` function can be used to specify the order in which the elements are to be iterated. The default value for `order` argument is `'K'` which the order of elements in memory. This can be overridden with `order='C'` for C order (row major) and `order='F'` for Fortran order (column major).

```
1  import numpy as np
2
3  #Creating a 2D Array
4  b = np.array([[1,2,3],[4,5,6],[7,8,9]])
5
6  #Prints each element in the order in memory
7  for e in np.nditer(b,order='K'):
8    print(e, end=' ')  #Prints 1 2 3 4 5 6 7 8 9
9  print()
10
11 #Prints each element in row major order
12 for e in np.nditer(b,order='C'):
13   print(e, end=' ')  #Prints 1 2 3 4 5 6 7 8 9
14 print()
15
16 #Prints each element in column major order
17 for e in np.nditer(b,order='F'):
18   print(e, end=' ')  #Prints 1 4 7 2 5 8 3 6 9
19
```

Listing 4.20: Changing order of iteration using `order` argument

4.3.7 Copying Arrays

`copy` method can be used to create a copy of an `ndarray`.

```
1  numpy.copy(a, order)
```

- `a` is the array whose elements are to be copied.
- `order` is an optional argument which specifies the order in which the elements are to be copied. For example, `'C'` for row major, `'F'` for column major.

```
1  import numpy as np
2
3  a = np.arange(10)
4  print(a)  #Prints [0 1 2 3 4 5 6 7 8 9]
5
6  b = a
7  c = np.copy(a)
8
9  a[1]=11
10
11 print(a)  #Prints [0 11 2 3 4 5 6 7 8 9]
12 print(b)  #Prints [0 11 2 3 4 5 6 7 8 9]
13 print(c)  #Prints [0 1 2 3 4 5 6 7 8 9]
```

Listing 4.21: Copying an array using `copy`

- In line no. 6, `a` is assigned to `b`. Hence both will be representing the same object. Changing an element in `a` will be reflected in `b`.
- In line no. 7, `copy` function, returns a new array containing all the elements of `a` and this new array is assigned to `c`. Hence changes made to `a` will not be reflected in `c`.

4.3.8 Splitting Arrays

The `split` function splits an array into multiple sub-arrays and returns a list of these sub-arrays.

```
numpy.split(a, indices_or_sections, axis)
```

- a is the array to be divided in to sub arrays.
- If indices_or_sections is an integer, N, the array will be divided into N equal arrays along axis. If such a split is not possible, an error is raised.
- If indices_or_sections is a 1-D array of sorted integers, the entries indicate where along axis the array is split.
- axis is optional integer which specifies the axis along which the array is to be split. Default value for axis is 0.

```
1  import numpy as np
2
3  a = np.arange(8)
4  splits = np.split(a,4)
5  print(splits)
6  #Prints [array([0, 1]), array([2, 3]), array([4, 5]), array([6, 7])]
7  splits = np.split(a,[2,4])
8  print(splits)
9  #Prints [array([0, 1]), array([2, 3]), array([4, 5, 6, 7])]
10 b = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
11 splits = np.split(b,2)
12 print(splits)
13 #Prints [array([[1, 2, 3],
14 #               [4, 5, 6]]), array([[ 7, 8, 9],
15 #                                   [10, 11, 12]])]
```

Listing 4.22: Splitting an array using split() Function

- Line No. 4: Splits the array in to 4 equal sized sub arrays
- Line No. 7: Splits the array into 3 sub arrays. First sub array consists of elements from index 0 to 1. Second sub array consists of elements from index 2 to 4 (not including 4). Third sub array consists of remaining elements.
- Line No. 11: Splits the 2 dimensional array into 2 sub arrays. Each sub array is a two dimensional array.

4.3.9 Shape Manipulation

reshape() Function

The reshape() functions is used to change the shape of an array without changing its data and size. It returns the new array.

```
numpy.reshape(a, newshape, order)
```

- a is the array to be reshaped.
- newshape is an integer or tuple of integers.The new shape should be compatible with the original shape. If shape is an integer, then the result will be a 1-Dimensional array of that length.
- order is an optional argument which specifies the order in which elements of the array a should be read while reshaping. For example, order can be 'C' for row major and 'F' for column major.

```
import numpy as np
#Creates array [0 1 2 3 4 5]
a = np.arange(6)
print(a)        #Prints [0 1 2 3 4 5]
new_a = np.reshape(a,(2,3))
print(new_a)
#2D Array
b = np.array([[1,2,3],[4,5,6]])
```

```
9  new_b = np.reshape(b,6)
10 print(new_b)  #Prints [1 2 3 4 5 6]
```

Listing 4.23: Reshaping an array using `reshape()` Function

- Line No. 5: Reshapes the 1D array to a to 2D array with 2 rows and 3 columns.
- Line No. 9: Reshapes the 2D array to a 1D array with 6 elements.

4.3.10 Transposing an Array

The `transpose()` functions is used to permute the axes of an array. It returns the modified array.

```
1 numpy.transpose(a, axes)
```

- `a` is the array to be transposed.
- `axes` is an tuple or list integers. If specified, it must be a tuple or list which contains a permutation of `[0,1,..,N-1]` where `N` is the number of axes of `a`. The i^{th} axis of the returned array will correspond to the axis numbered `axes[i]` of the input. If not specified, it reverses the order of the axes.

```
1 import numpy as np
2
3 #Creates a 2D array with 2 rows and 3 columns
4 a = np.arange(6).reshape((2,3))
5 print(a)
6
7 #Transpose of a will have 3 rows and 2 columns
8 b = np.transpose(a)
9 print(b)
```

Listing 4.24: Reshaping an array using `reshape()` Function

- Line No. 5 prints the following

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

- Line No. 9 prints the following

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

4.3.11 Resizing an Array

The `resize()` function is used to return a new array with the specified shape.

```
1 numpy.resize(a, new_shape)
```

- `a` is the array to be resized.
- `shape` is an integer or tuple of integers. If the new array is larger than the original array, then the new array is filled with repeated copies of `a`.

```
1 import numpy as np
2
3 #Creates a 2D array with 2 rows and 3 columns
4 a = np.arange(6).reshape((2,3))
5 print(a)
6
7 #Resize the array a with 3 rows and 4 columns
8 b = np.resize(a,(3,4))
9 print(b)
```

Listing 4.25: Resizing an array using `resize()` Function

- Line No. 9 prints the following

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

Using `resize` method of ndarray object

The `resize()` method of an ndarray object resizes an array in to specified shape. If the new size is larger than the original array, then the new array is filled with zeros.

```
ndarray.resize(new_shape)
```

```
import numpy as np

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

#Resize the array a with 3 rows and 4 columns
a.resize(3,4)
print(a)
```

Listing 4.26: Resizing an array using `ndarray.resize()` Method

- Line No. 8 prints the following

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

4.3.12 Arithmetic Operations on Arrays

Arithmetic operations on arrays can be performed using some mathematical functions or operators. For performing arithmetic operations, the arrays must of compatible size.

- `add()` function can be used to add two arrays element wise. The `+` operator can be used as a shorthand for `np.add` on ndarrays.
- `subtract()` function can be used to subtract one array from the other element wise. The `-` operator can be used as a shorthand for `np.subtract` on ndarrays.
- `multiply()` function can be used to add multiply arrays element wise. The `*` operator can be used as a shorthand for `np.multiply` on ndarrays.
- `divide()` function can be used to add divide arrays element wise. The `/` operator can be used as a shorthand for `np.divide` on ndarrays.

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
b = np.array([[11,12,33],[44,55,66]])
#Element wise addition using add function
print(np.add(a,b))
#Element wise addition using + operator
print(a+b)
#Element wise subtraction using subtract function
print(np.subtract(a,b))
#Element wise subtraction using - operator
print(a-b)
#Element wise multiplcation using multiply function
print(np.multiply(a,b))
#Element wise multiplcation using * operator
print(a*b)
#Element wise division using divide function
print(np.divide(b,a))
```

```
#Element wise division using / operator
print(b/a)
```

Matrix Multiplication using `dot` Function and `@` Operator

`dot()` function or `@` operator can be used to perform matrix multiplication.

```
import numpy as np

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

#Matrix multiplication using dot function
c = np.dot(a,b)
print(c)

#Matrix multiplication using @ operator
d = a @ b
print(d)
```