Unit-II

Arrays in Python: Arrays, Types of Arrays, Working with Arrays using numpy, Creating Arrays, Operations on Arrays, Attributes of an Array, The reshape() Method, The flatten() Method, Matrices in numpy, Matrix Addition and Multiplication. **Strings and Characters:** Creating Strings, Operations on Strings, Working with Characters, Sorting Strings, Searching Strings.

Arrays

An array is defined as a **collection of items** that are **stored at contiguous memory locations**. It is a **container** which can hold a **fixed** number of items, and these items should be of the **same type**. An array is popular in most programming languages like C/C++, JavaScript, etc.

Array is an idea of storing **multiple items** of the **same type** together and it makes **easier** to calculate the **position** of each element by simply adding an **offset** to the base value.

A combination of the arrays could **save** a lot of **time** by reducing the overall **size** of the code. It is used to **store multiple values in single variable**. If you have a list of items that are stored in their corresponding variables like this:

car1 = "Maruthi"

car2 = "Hundai"

car3 = "Kia"

If you want to loop through cars and find a specific one, you can use the array.

The array can be handled in Python by a module named **array**. It is useful when we have to manipulate only specific data values. Following are the terms to understand the concept of an array:

Element - Each item stored in an array is called an element.

Index - The location of an element in an array has a numerical index, which is used to identify the position of the element.

Array Representation

An array can be declared in various ways and different languages. The important points that should be considered are as follows:

- Index starts with 0.
- We can access each element via its index.
- The length of the array defines the capacity to store the elements.

Syntax:

array_name=array(typecode, [elements])

Type codes in arrays

Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined

Type code	C Type	Python Type	Minimum size in bytes
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	wchar_t	Unicode character	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'q'`	signed long long	int	8
`'Q'`	unsigned long long	int	8
`'f'`	float	float	4
`'d'`	double	float	8

Different styles for importing array module:

Style 1:

import array

a=array.array(‘i’,[1,2,3,4])

Style 2:

import array as ar

a=ar.array(‘u’,[‘q’,’w’,’e’,’r’,’t’,’y’])

Style 3:

from array import *

a=array(‘i’,[3,4,5,6])

Array operations

Some of the basic operations supported by an array are as follows:

- **Traverse** - It prints all the elements one by one.
- **Insertion** - It adds an element at the given index.
- **Deletion** - It deletes an element at the given index.
- **Search** - It searches an element using the given index or by the value.
- **Update** - It updates an element at the given index.

The **Array** can be created in Python by **importing** the **array module** to the python program.

In [1]:

```
import array
#creating an array

a = array.array('i',[5,6,7,8])
print("Items are:") #printing items in array
for i in a:
    print(i)
print(a)
```

```
Items are:
5
6
7
8
array('i', [5, 6, 7, 8])
```

In [5]:

```
import array as ar
#creating an array

a = ar.array('f',[5.5,6.2,7.3,8.9])
print("Items are:") #printing items in array
for i in a:
    print(i)
print(a)
```

```
Items are:
5.5
6.199999809265137
7.300000190734863
8.899999618530273
array('f', [5.5, 6.199999809265137, 7.300000190734863, 8.899999618530273])
```

In [6]:

```
from array import *
#creating an array

a = array('i',[2,3,4,5,6])
print("Items are:") #printing items in array
for i in a:
    print(i)
print(a)
```

```
Items are:
2
3
4
5
6
array('i', [2, 3, 4, 5, 6])
```

In [2]:

```
import array as arr
a = arr.array('i', [2, 4, 6, 8])
print("First element:", a[0])
print("Second element:", a[1])
print("Second last element:", a[-1])
print("Second last element:", a[-2])
```

```
First element: 2
Second element: 4
Second last element: 8
Second last element: 6
```

In [8]:

```
from array import *
array_num = array('d', [1, 3, 5, 7, 9])
print("Original array: "+str(array_num))
print("Length in bytes of one array item: "+str(array_num.itemsize))
```

```
Original array: array('d', [1.0, 3.0, 5.0, 7.0, 9.0])
Length in bytes of one array item: 8
```

In [1]:

```
from array import *
#creating an array

a = array('u',['a','r','r','a','y'])
print("Items are:") #printing items in array
for i in a:
    print(i)
print(a)
print("length of the array is ", len(a)) #printing length of the array
```

```
Items are:
a
r
r
a
y
array('u', 'array')
length of the array is  5
```

Accessing array elements

In python, to access array items refer to the index number. We will use the index operator "[]" for accessing items from the array.

Slicing

We can access a range of items in an array by using the slicing operator.

For example, if you use the index -1, you will be interacting with the last element in the array.

Syntax: arrayname[start,stop,stepsize]

In [2]:

```
import array as ar
a = ar.array('i',[2,4,6,8,10])
print("First Element of array: ", a[0])
print("Third element of array: ", a[2])
print("Last element of array: ", a[-1])
```

```
First Element of array:  2
Third element of array:  6
Last element of array:  10
```

In [3]:

```
import array as ar
a = ar.array('i', [3,5,7,9,11])
print(a[0:2])
print(a[0:4:2])
print(a[:])
```

```
array('i', [3, 5])
array('i', [3, 7])
array('i', [3, 5, 7, 9, 11])
```

In [4]:

```
from array import *
a = array('i', [11,12,13,14,15])
for n in a[1:3]:
    print(n)
```

```
12
13
```

In [24]:

```
# Python program to demonstrate
# accessing of element from list

# importing array module
import array as arr

# array with int type
a = arr.array('i', [1, 2, 3, 4, 5, 6])

# accessing element of array
print("Access element is: ", a[0])

# accessing element of array
print("Access element is: ", a[3])

# array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

# accessing element of array
print("Access element is: ", b[1])

# accessing element of array
print("Access element is: ", b[2])
```

```
Access element is:  1
Access element is:  4
Access element is:  3.2
Access element is:  3.3
```

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list

append

Adds an element at the end of the list

Syntax:

array.append(item)

In [25]:

```
import array as ar
num = ar.array('i',[20,30,40])
print(num)
num.append(50)
print(num)
```

```
array('i', [20, 30, 40])
array('i', [20, 30, 40, 50])
```

extend

Adds more elements to the end of the list

Syntax:

array.extend(list)

In [26]:

```
import array as ar
num = ar.array('i',[22,33,44,55])
print(num)
num.append(66)
print(num)
num.extend([77,88,99])
print(num)
```

```
array('i', [22, 33, 44, 55])
array('i', [22, 33, 44, 55, 66])
array('i', [22, 33, 44, 55, 66, 77, 88, 99])
```

insert

Python array insert operation enables you to insert one or more items into an array at the beginning, end, or any given index of the array. This method expects two arguments index and value.

Syntax:

arrayName.insert(index, value)

In [27]:

```
import array
balance = array.array('i', [300,200,100])
print(balance)
balance.insert(2, 150)
print(balance)
```

```
array('i', [300, 200, 100])
array('i', [300, 200, 150, 100])
```

In [28]:

```
import array as myarr
a=myarr.array('b',[2,4,6,8,10,12,14,16,18,20])
a.insert(2,56)
print(a)
```

```
array('b', [2, 4, 56, 6, 8, 10, 12, 14, 16, 18, 20])
```

In [14]:

```
import array as myarr
a=myarr.array('b',[2,4,6,8,10,8,14,16,18,20])
print("Index of 8 is ", a.index(8))
```

```
Index of 8 is  3
```

In [23]:

```
# Python program to demonstrate
# Adding Elements to a Array

# importing "array" for array creations
import array as arr

# array with int type
a = arr.array('i', [1, 2, 3])


print ("Array before insertion : ", end =" ")
for i in range (0, 3):
    print (a[i], end =" ")
print()

# inserting array using
# insert() function
a.insert(1, 4)

print ("Array after insertion : ", end =" ")
for i in (a):
    print (i, end =" ")
print()

# array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

print ("Array before insertion : ", end =" ")
for i in range (0, 3):
    print (b[i], end =" ")
print()

# adding an element using append()
b.append(4.4)

print ("Array after insertion : ", end =" ")
for i in (b):
    print (i, end =" ")
print()
```

```
Array before insertion :  1 2 3
Array after insertion :  1 4 2 3
Array before insertion :  2.5 3.2 3.3
Array after insertion :  2.5 3.2 3.3 4.4
```

remove

It is used for removing an existing element from the array and re-organizing all elements of an array.

In [29]:

```
import array as ar
num = ar.array('i',[1,2,3,5,7,10])
print(num)
num.remove(7)
for x in num:
    print(x)
```

```
array('i', [1, 2, 3, 5, 7, 10])
1
2
3
5
10
```

In [25]:

```
# Python program to demonstrate
# Removal of elements in a Array

# importing "array" for array operations
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 5])

# printing original array
print ("The new created array is : ", end ="")
for i in range (0, 5):
    print (arr[i], end =" ")

print ("\r")

# using pop() to remove element at 2nd position
print ("The popped element is : ", end ="")
print (arr.pop(2))

# printing array after popping
print ("The array after popping is : ", end ="")
for i in range (0, 4):
    print (arr[i], end =" ")

print("\r")

# using remove() to remove 1st occurrence of 1
arr.remove(1)

# printing array after removing
print ("The array after removing is : ", end ="")
for i in range (0, 3):
    print (arr[i], end =" ")
```

```
The new created array is : 1 2 3 1 5
The popped element is : 3
The array after popping is : 1 2 1 5
The array after removing is : 2 1 5
```

pop

removes elements at the specified position.

In [33]:

```
import array as ar
num = ar.array('i',[1,3,5,5,3,10])
print(num)
#num.remove(3)
num.pop(4)
for x in num:
    print(x)
```

```
array('i', [1, 3, 5, 5, 3, 10])
1
3
5
5
10
```

del

The elements can be deleted from an array using Python's **del** statement. If we want to delete any value from the array, we can do that by using the indices of a particular element.

In [34]:

```
import array as ar
num = ar.array('i',[2,4,4,6,8,10])
print(num)
del num[2]
for x in num:
    print(x)
```

```
array('i', [2, 4, 4, 6, 8, 10])
2
4
6
8
10
```

count

returns the number of elements with the specified value

In [35]:

```
import array as ar
num = ar.array('i',[2,4,4,6,8,10])
for x in num:
    print("{} occurs {} times".format(x,num.count(x)))
```

```
2 occurs 1 times
4 occurs 2 times
4 occurs 2 times
6 occurs 1 times
8 occurs 1 times
10 occurs 1 times
```

sorted

In [36]:

```
import array as ar
num = ar.array('i',[3,10,9,15,12])
print(num)
print(sorted(num))
```

```
array('i', [3, 10, 9, 15, 12])
[3, 9, 10, 12, 15]
```

Concatenation

By using concatenating operator '+' we can join two arrays

In [20]:

```
import array as ar
a = ar.array('d',[1.1,2.1,3.4,2.6,7,8])
b = ar.array('d', [3.7,8.6])
c = ar.array('d')
c = a+b
print("Array C: ", c)
```

```
Array C:  array('d', [1.1, 2.1, 3.4, 2.6, 7.0, 8.0, 3.7, 8.6])
```

In [21]:

```
#Write a Python program to find whether a given array of integers contains any
#duplicate element. Return true if any value appears at least twice in the said
#array and return false if every element is distinct.
import array as ar
a = ar.array('i', [2,4,5,4,7,9])
for e in a:
    if a.count(e) > 1:
        print("true")
        break
else:
    print("false")
```

```
true
```

In [22]:

```
#Write a Python program to find a pair with highest product from a given array
# of integers.
import array as ar
a = ar.array('i', [1, 2, 3, 4, 7, 0, 8, 4])
if len(a) <2:
    print("No pair exists")
x = a[0]
y = a[1]
for i in range(0,len(a)):
    for j in range(i+1, len(a)):
        if a[i]*a[j] > x*y:
            x = a[i]
            y = a[j]
print("original array ", a)
print("Maximum product pair is ({},{})".format(x,y))
```

```
original array  array('i', [1, 2, 3, 4, 7, 0, 8, 4])
Maximum product pair is (7,8)
```

Slicing

Slice operation is performed on array with the use of colon(:). To print elements from beginning to a range use [:Index], to print elements from end use [:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print whole List with the use of slicing operation, use [:]. Further, to print whole array in reverse order, use [::-1].

In [26]:

```
# Python program to demonstrate
# slicing of elements in a Array

# importing array module
import array as arr

# creating a list
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

a = arr.array('i', l)
print("Initial Array: ")
for i in (a):
    print(i, end =" ")

# Print elements of a range
# using Slice operation
Sliced_array = a[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_array)

# Print elements from a
# pre-defined point to end
Sliced_array = a[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_array)

# Printing elements from
# beginning till end
Sliced_array = a[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_array)
```

```
Initial Array:
1 2 3 4 5 6 7 8 9 10
Slicing elements in a range 3-8:
array('i', [4, 5, 6, 7, 8])

Elements sliced from 5th element till the end:
array('i', [6, 7, 8, 9, 10])

Printing all elements using slice operation:
array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Searching element

In order to search an element in the array we use a python in-built index() method. This function returns the index of the first occurrence of value mentioned in arguments.

In [27]:

```
# Python code to demonstrate
# searching an element in array


# importing array module
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print ("The new created array is : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")

print ("\r")

# using index() to print index of 1st occurrence of 2
print ("The index of 1st occurrence of 2 is : ", end ="")
print (arr.index(2))

# using index() to print index of 1st occurrence of 1
print ("The index of 1st occurrence of 1 is : ", end ="")
print (arr.index(1))
```

```
The new created array is : 1 2 3 1 2 5
The index of 1st occurrence of 2 is : 1
The index of 1st occurrence of 1 is : 0
```

Updating Elements

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

In [28]:

```
# Python code to demonstrate
# how to update an element in array

# importing array module
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print ("Array before updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")

print ("\r")

# updating a element in a array
arr[2] = 6
print("Array after updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
print()

# updating a element in a array
arr[4] = 8
print("Array after updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
```

```
Array before updation : 1 2 3 1 2 5
Array after updation : 1 2 6 1 2 5
Array after updation : 1 2 6 1 8 5
```

Counting Elements

In order to count elements in an array we need to use count method.

In [29]:

```
import array

# Create an array of integers
my_array = array.array('i', [1, 2, 3, 4, 2, 5, 2])

# Count the number of occurrences of the element 2 in the array
count = my_array.count(2)

# Print the result
print("Number of occurrences of 2:", count)
```

```
Number of occurrences of 2: 3
```

Reversing Elements

In order to reverse elements of an array we need to simply use reverse method.

In [30]:

```
import array

# Create an array of integers
my_array = array.array('i', [1, 2, 3, 4, 5])

# Print the original array
print("Original array:", *my_array)

# Reverse the array in place
my_array.reverse()

# Print the reversed array
print("Reversed array:", *my_array)
```

```
Original array: 1 2 3 4 5
Reversed array: 5 4 3 2 1
```

Numpy

NumPy stands for **Numerical Python**. It is a Python library used for working with an array. It is the **core library** for **scientific computing**, which contains a powerful n-dimensional array object

In Python, we use the list for purpose of the array but it's slow to process. **NumPy array is a powerful N-dimensional array object** and its use in **linear algebra**, **Fourier transform**, and **random number** capabilities.

It provides an array object much **faster** than traditional Python lists.

Installation of Numpy: **pip install numpy**

We use python NumPy array instead of a list because of the below three reasons:

- Less Memory
- Fast
- Convenient

Types of Array:

- One Dimensional Array
- Multi-Dimensional Array

One Dimensional Array:

A one-dimensional array is a type of linear array.

In [2]:

```
# importing numpy module
import numpy as np

# creating list
list = [1, 2, 3, 4]

# creating numpy array
sample_array = np.array(list)
print("List in python : ", list)
print("Numpy Array in python :", sample_array)
```

```
List in python :  [1, 2, 3, 4]
Numpy Array in python : [1 2 3 4]
```

In [4]:

```
#data types
print(type(list))
print(type(sample_array))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

In [3]:

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

```
[1 2 3]
```

In [7]:

```
import numpy as np

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

print("li =", li, "and type(li) =", type(li))
print("numpyArr =", numpyArr, "and type(numpyArr) =", type(numpyArr))
```

```
li = [1, 2, 3, 4] and type(li) = <class 'list'>
numpyArr = [1 2 3 4] and type(numpyArr) = <class 'numpy.ndarray'>
```

In [8]:

```
#numpy array from tuple
import numpy as np

tup = (1, 2, 3, 4)
numpyArr = np.array(tup)

print("tup =", tup, "and type(tup) =", type(tup))
print("numpyArr =", numpyArr, "and type(numpyArr) =", type(numpyArr))
```

```
tup = (1, 2, 3, 4) and type(tup) = <class 'tuple'>
numpyArr = [1 2 3 4] and type(numpyArr) = <class 'numpy.ndarray'>
```

Multi-Dimensional Array:

Data in multidimensional arrays are stored in tabular form.

Note: use **[ ]** operators inside numpy.array() for multi-dimensional

Two Dimensional Array

1	2	3	4	5
6	7	8	9	10

In [5]:

```
# importing numpy module
import numpy as np

# creating list
list_1 = [1, 2, 3, 4]
list_2 = [5, 6, 7, 8]
list_3 = [9, 10, 11, 12]

# creating numpy array
sample_array = np.array([list_1, list_2, list_3])
print("Numpy multi dimensional array in python\n", sample_array)
```

```
Numpy multi dimensional array in python
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In [6]:

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

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

Numpy Operations on Arrays

ndim

it is used to find the dimension of the array.

In [11]:

```
import numpy as np
b = np.array([(2,3,5,7),(1,4,6,8)])

print("The given array is a {} dimensional array".format(b.ndim))
```

```
The given array is a 2 dimensional array
```

In [13]:

```
import numpy as np
b = np.array([(2,3,5,7),(1,4,6,8),(5,4,10,8)])

print("The given array is a {} dimensional array".format(b.ndim))
```

```
The given array is a 2 dimensional array
```

itemsize

It is used to calculate the byte size of each element.

In [14]:

```
import numpy as np
c=np.array([1,2,3])
print("Byte size of each element is ", c.itemsize)
```

```
Byte size of each element is  4
```

In [70]:

```
import numpy as np
c=np.array([1.5,2.6,3.7])
print("Byte size of each element is ", c.itemsize)
```

```
Byte size of each element is  8
```

In [10]:

```
import  numpy as np
import sys
s = range(1000) # declaring a list of 1000 elements

#printing size of each element in list
print("Size of each element of list in bytes: ", sys.getsizeof(s))

#printing size of whole list
print("Size of whole list in bytes: ", sys.getsizeof(s)*len(s))

#declaring numpy array with 1000 elements
d = np.arange(1000)

#printing size of each element in numpy array
print("Size of each element of Numpy array in bytes: ", d.itemsize)

#printing size of whole Numpy array
print("Size of whole Numpy array in bytes: ", d.size*d.itemsize)
```

```
Size of each element of list in bytes:  48
Size of whole list in bytes:  48000
Size of each element of Numpy array in bytes:  4
Size of whole Numpy array in bytes:  4000
```

dtype

It can find the data type of the elements that are stored in an array.

In [15]:

```
import numpy as np
d=np.array([1,2,3])
print("The elements of the given array are of {} type".format(d.dtype))
```

```
The elements of the given array are of int32 type
```

In [71]:

```
import numpy as np
d=np.array([1.5,2.6,3.7])
print("The elements of the given array are of {} type".format(d.dtype))
```

```
The elements of the given array are of float64 type
```

size and shape functions

size function returns number of elements of the given array and **shape** returns the shape of the array in the form of (no of rows, no of elements in each row)

In [16]:

```
import numpy as np
e= np.array([(2,3,4,5,6)])
print("The size of given array is ", e.size)
print("The shape of given array is ", e.shape)

e1 = np.array([(11,12,13),(16,17,18)])
print("The size of given array is ", e1.size)
print("The shape of array e1 is ", e1.shape)
```

```
The size of given array is  5
The shape of given array is  (1, 5)
The size of given array is  6
The shape of array e1 is  (2, 3)
```

Slicing

Slicing is basically extracting particular set of elements from an array. This slicing operation is pretty much similar to the one which is there in the list as well.

In [19]:

```
import numpy as np
a = np.array([(2,3,4,5),(8,9,10,11)])
print(a)
print("The second element in the first row is ", a[0,2])
```

```
[[ 2  3  4  5]
 [ 8  9 10 11]]
The second element in the first row is  4
```

In [21]:

```
import numpy as np
a = np.array([(2,3,4,5),(8,9,10,11)])
print(a)
print(a[0:,2])
```

```
[[ 2  3  4  5]
 [ 8  9 10 11]]
[ 4 10]
```

In [72]:

```
import numpy as np
a = np.array([(2,3,4,5),(8,9,10,11),(15,16,17,18),(21,22,23,24)])
print(a)
print(a[1:3,2])
print(a[2:3,2])
```

```
[[ 2  3  4  5]
 [ 8  9 10 11]
 [15 16 17 18]
 [21 22 23 24]]
[10 17]
[17]
```

In [23]:

```
import numpy as np
a = np.array([(2,3,4,5),(8,9,10,11),(15,16,17,18),(21,22,23,24)])
print(a)
print(a[1:3,1:3])
```

```
[[ 2  3  4  5]
 [ 8  9 10 11]
 [15 16 17 18]
 [21 22 23 24]]
[[ 9 10]
 [16 17]]
```

In [8]:

```
#print all of the values in the array that are less than 5.
import numpy as np
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(a)
print(a[a < 5])
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[1 2 3 4]
```

In [9]:

```
#numbers that are equal to or greater than 5,
#and use that condition to index an array
import numpy as np
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(a)
five_up = (a >= 5)
print(a[five_up])
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[ 5  6  7  8  9 10 11 12]
```

In [10]:

```
#elements that are divisible by 2
import numpy as np
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(a)
divisible_by_2 = a[a%2==0]
print(divisible_by_2)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[ 2  4  6  8 10 12]
```

In [11]:

```
#select elements that satisfy two conditions using the & and | operators
import numpy as np
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(a)
c = a[(a > 2) & (a < 11)]
print(c)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[ 3  4  5  6  7  8  9 10]
```

In [12]:

```
#use of the logical operators & and | in order to return boolean values
#that specify whether or not the values in an array fulfill a certain condition
import numpy as np
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(a)
five_up = (a > 5) | (a == 5)
print(five_up)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[False False False False]
 [ True  True  True  True]
 [ True  True  True  True]]
```

In [13]:

```
#use np.nonzero() to select elements or indices from an array
import numpy as np
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(a)
b = np.nonzero(a < 5)
print(b)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
(array([0, 0, 0, 0], dtype=int64), array([0, 1, 2, 3], dtype=int64))
```

In [16]:

```
#to generate a list of coordinates where the elements exist,
#you can zip the arrays, iterate over the list of coordinates, and print them
import numpy as np
a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(a)
b = np.nonzero(a < 5)
print(b)
list_of_coordinates= list(zip(b[0], b[1]))
for coord in list_of_coordinates:
    print(coord)
print(a[b])
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
(array([0, 0, 0, 0], dtype=int64), array([0, 1, 2, 3], dtype=int64))
(0, 0)
(0, 1)
(0, 2)
(0, 3)
[1 2 3 4]
```

linspace

This is another operation in python numpy which returns evenly spaced numbers over a specified interval.

Syntax:

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

In [47]:

```
import numpy as np

np.linspace(3.5, 10, 3)
```

Out[47]:

```
array([ 3.5 ,  6.75, 10.  ])
```

In [43]:

```
import numpy as np

np.linspace(3.5, 10, 3,dtype = np.int32)
```

Out[43]:

```
array([ 3,  6, 10])
```

In [64]:

```
import numpy as np
a = np.array([1,3,10])
print(a)
a = np.linspace(1,3,10)
print(a)
a = np.linspace(1,2,8)
print(a)
b = np.linspace(1,100,8)
print(b)
```

```
[ 1  3 10]
[1.         1.22222222 1.44444444 1.66666667 1.88888889 2.11111111
 2.33333333 2.55555556 2.77777778 3.        ]
[1.         1.14285714 1.28571429 1.42857143 1.57142857 1.71428571
 1.85714286 2.        ]
[  1.          15.14285714  29.28571429  43.42857143  57.57142857
  71.71428571  85.85714286 100.        ]
```

max/min/sum

To find the **minimum**, **maximum** as well the **sum** of the numpy array. **max** function returns the biggest element in the given array. **min** function returns the smallest element in the given array. **sum** function returns the sum of the elements in the array.

In [27]:

```
import numpy as np
a = np.array([21,9,27])
print("Biggest element in the array is :", a.max())
print("Smallest element in the array is :", a.min())
print("Sum of the elements in the array is :", a.sum())
```

```
Biggest element in the array is : 27
Smallest element in the array is : 9
Sum of the elements in the array is : 57
```

In [30]:

```
import numpy as np
b = np.array([(2,3,4,5),(8,9,10,11)])
print(b)
print("Biggest element in the array is :", b.max())
print("Smallest element in the array is :", b.min())
print("Sum of the elements in the array is :", b.sum())
```

```
[[ 2  3  4  5]
 [ 8  9 10 11]]
Biggest element in the array is : 11
Smallest element in the array is : 2
Sum of the elements in the array is : 52
```

numpy.fromiter():

The fromiter() function create a **new one-dimensional array** from an **iterable object**.

Syntax:

numpy.fromiter(iterable, dtype, count=-1)

In [2]:

```
#Import numpy module
import numpy as np

# iterable
iterable = (a*a for a in range(8))

arr = np.fromiter(iterable, float)

print("fromiter() array :",arr)
```

```
iterable
fromiter() array : [ 0.  1.  4.  9. 16. 25. 36. 49.]
```

In [40]:

```
import numpy as np

var = "CMR Technical Campus"

arr = np.fromiter(var, dtype = 'U2')

print("fromiter() array :", arr)
```

```
fromiter() array : ['C' 'M' 'R' ' ' 'T' 'e' 'c' 'h' 'n' 'i' 'c' 'a' 'l' '
' 'C' 'a' 'm' 'p'
 'u' 's']
```

numpy.arange()

This is an inbuilt NumPy function that returns evenly spaced values within a given interval.

Syntax:

numpy.arange([start,]stop, [step,]dtype=None)

In [41]:

```
import numpy as np

np.arange(1, 20 , 2, dtype = np.float32)
```

Out[41]:

```
array([ 1.,  3.,  5.,  7.,  9., 11., 13., 15., 17., 19.], dtype=float32)
```

numpy.empty()

This function create a new array of given shape and type, without initializing value.

Syntax:

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

In [65]:

```
import numpy as np

np.empty([4, 3], dtype = np.int32, order = 'f')
```

Out[65]:

```
array([[4128860, 6619251, 4259912],
       [6029375, 7536754, 8257614],
       [3801155, 5111900,      49],
       [5570652, 5505109,       0]])
```

numpy.ones()

This function is used to get a new array of given shape and type, filled with ones(1).

Syntax:

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

In [2]:

```
import numpy as np

np.ones([4, 3],
        dtype = np.int32,
        order = 'f')
```

Out[2]:

```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

numpy.zeros()

This function is used to get a new array of given shape and type, filled with zeros(0).

Syntax:

numpy.zeros(shape, dtype=None)

In [3]:

```
import numpy as np
np.zeros([4, 3],
         dtype = np.int32,
         order = 'f')
```

Out[3]:

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Sort

sort the numbers in ascending order

In [4]:

```
import numpy as np
arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
np.sort(arr)
```

Out[4]:

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

concatenate

In [6]:

```
import numpy as np
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
np.concatenate((a, b))
```

Out[6]:

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

In [7]:

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6]])
np.concatenate((x, y), axis=0)
```

Out[7]:

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

In [66]:

```
import random
# Create 3 lists, each with 8 random numbers
list_1 = [random.random() for i in range(8)]
list_2 = [random.random() for i in range(8)]
list_3 = [random.random() for i in range(8)]
```

In [71]:

```
list_1
```

Out[71]:

```
[0.9056522981574985,
 0.7879920437594492,
 0.11253752927353988,
 0.03715637383606296,
 0.8724022875586012,
 0.43893521058806506,
 0.24089633140995192,
 0.4178225352814675]
```

In [72]:

```
list_2
```

Out[72]:

```
[0.3061021933579968,
 0.509067426516029,
 0.3308741815964783,
 0.294912528871764,
 0.03510072181913748,
 0.8831077947158158,
 0.6753705067546735,
 0.26403427734582074]
```

In [69]:

```
list_3
```

Out[69]:

```
[0.8747571345017265,
 0.866225357933028,
 0.09664184684600097,
 0.678577533310394,
 0.08440405018452735,
 0.09299641245962276,
 0.7991669708121977,
 0.04991780715163063]
```

In [73]:

```
A = np.concatenate([list_1,list_2,list_3])
A
```

Out[73]:

```
array([0.9056523 , 0.78799204, 0.11253753, 0.03715637, 0.87240229,
       0.43893521, 0.24089633, 0.41782254, 0.30610219, 0.50906743,
       0.33087418, 0.29491253, 0.03510072, 0.88310779, 0.67537051,
       0.26403428, 0.87475713, 0.86622536, 0.09664185, 0.67857753,
       0.08440405, 0.09299641, 0.79916697, 0.04991781])
```

In [18]:

```
A.shape
```

Out[18]:

```
(24,)
```

In [24]:

```
A = A.reshape(1,-1)
A.shape
```

Out[24]:

```
(1, 24)
```

In [22]:

```
A = A.reshape(-1,1)
A.shape
```

Out[22]:

```
(24, 1)
```

Basic Array Operations

addition, subtraction, multiplication, division

In [39]:

```
import numpy as np
data = np.array([1, 2])
ones = np.ones(2, dtype=int)
data + ones
```

Out[39]:

```
array([2, 3])
```

In [40]:

```
data - ones
```

Out[40]:

```
array([0, 1])
```

In [41]:

```
data * data
```

Out[41]:

```
array([1, 4])
```

In [42]:

```
data / data
```

Out[42]:

```
array([1., 1.])
```

Broadcasting

There are times when you might want to carry out an operation between an array and a single number (also called an operation between a vector and a scalar) or between arrays of two different sizes.

In [43]:

```
import numpy as np
data = np.array([1.0, 2.0])
data * 1.6
```

Out[43]:

```
array([1.6, 3.2])
```

Attributes of an Array

NumPy arrays have a set of attributes that you can access. These attributes include the array's size, shape, number of dimensions, and data type.

ndarray.shape

This array attribute returns a tuple consisting of array dimensions. It can also be used to resize the array.

In [48]:

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

```
(2, 3)
```

In [75]:

```
# this resizes the ndarray
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
print(a)
a.shape = (3,2)
print("after reshape")
print(a)
```

```
[[1 2 3]
 [4 5 6]]
after reshape
[[1 2]
 [3 4]
 [5 6]]
```

In [52]:

```
#NumPy also provides a reshape function to resize an array.
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
b = a.reshape(3,2)
print(b)
```

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

ndarray.ndim

This array attribute returns the number of array dimensions.

In [53]:

```
# an array of evenly spaced numbers
import numpy as np
a = np.arange(24)
print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

In [76]:

```
# this is one dimensional array
import numpy as np
a = np.arange(24)
a.ndim
```

Out[76]:

```
1
```

In [78]:

```
# now reshape it
b = a.reshape(3,4,2)
print(b)
# b is having three dimensions
```

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

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

 [[16 17]
  [18 19]
  [20 21]
  [22 23]]]
```

numpy.itemsize

This array attribute returns the length of each element of array in bytes.

In [67]:

```
# dtype of array is int8 (1 byte)
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.int8)
print(x.itemsize)
```

```
1
```

In [68]:

```
# dtype of array is now float32 (4 bytes)
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.float32)
print(x.itemsize)
```

```
4
```

numpy.flags

The ndarray object has the following attributes. Its current values are returned by this function

Sr.No.	Attribute & Description
1	**C_CONTIGUOUS (C)** The data is in a single, C-style contiguous segment
2	**F_CONTIGUOUS (F)** The data is in a single, Fortran-style contiguous segment
3	**OWNDATA (O)** The array owns the memory it uses or borrows it from another object
4	**WRITEABLE (W)** The data area can be written to. Setting this to False locks the data, making it read-only
5	**ALIGNED (A)** The data and all elements are aligned appropriately for the hardware
6	**UPDATEIFCOPY (U)** This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array

In [69]:

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

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

reshape method

Reshape is when you change the number of rows and columns which gives a new view to an object.

The **numpy.reshape()** function is available in NumPy package. As the name suggests, reshape means 'changes in shape'. The numpy.reshape() function helps us to get a new shape to an array without changing its data.

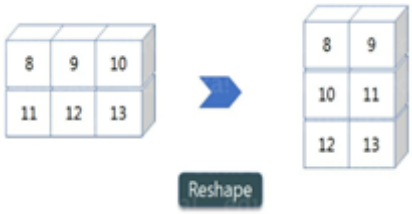
Syntax:

numpy.reshape(arr, new_shape)

Parameters

arr: This is the source array which we want to reshape.

new_shape: The shape in which we want to convert our original array.



In [17]:

```
import numpy as np
f = np.array([(8,9,10),(11,12,13)])
print("Array before reshape:\n", f)
f = f.reshape(3,2)
print("Array after reshape:\n", f)
f1 = np.reshape(f, (2,3))
print("Array after second reshape:\n", f1)
```

```
Array before reshape:
 [[ 8  9 10]
 [11 12 13]]
Array after reshape:
 [[ 8  9]
 [10 11]
 [12 13]]
Array after second reshape:
 [[ 8  9 10]
 [11 12 13]]
```

Flatten method

In Python, for some cases, we need a one-dimensional array rather than a 2-D or multi-dimensional array.

For this purpose, the numpy module provides a function called **numpy.ndarray.flatten()**, which returns a copy of the array in one dimensional rather than in 2-D or a multi-dimensional array.

Syntax:

ndarray.flatten(order='C')

Parameters:

- order: **{'C', 'F', 'A', 'K'}**(optional)
- If we set the order parameter to **'C'**, it means that the array gets flattened in **row-major** order.
- If **'F'** is set, the array gets flattened in **column-major** order.
- If **'A'** is set, the array get flattened in **row-major order**.
- The last order is **'K'**, which flatten the array in **same order** in which the elements occurred in the memory.
- By **default**, this parameter is set to **'C'**.

In [35]:

```
import numpy as np
a = np.array([[1,4,7],[2,5,8],[3,6,9]])
print(a)
b = a.flatten()
print(b)
```

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

In [36]:

```
import numpy as np
a = np.array([[1,4,7],[2,5,8],[3,6,9]])
print(a)
b = a.flatten("C") #using order c
print(b)
```

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

In [34]:

```
import numpy as np
a = np.array([[1,4,7],[2,5,8],[3,6,9]])
print(a)
b = a.flatten("F") #using order F
print(b)
```

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

In [37]:

```
import numpy as np
a = np.array([[1,4,7],[2,5,8],[3,6,9]])
print(a)
b = a.flatten("A") #using order A
print(b)
```

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

In [107]:

```
import numpy as np
a = np.array([[1,4,7],[2,5,8],[3,6,9]])
print(a)
b = a.flatten("K") #using order K
print(b)
```

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

In [92]:

```
import numpy as np
a = np.arange(1, 10)
a.shape = (3, 3)
print("a = ")
print(a)
print("\nAfter flattening")
print("------------------")
print(a.flatten())
```

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

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

Matrices in numpy

Random integers in a specific range

The first parameter determines the upper bound of the range. The lower bound is 0 by default but we can also specify it. The size parameter is used to specify the size, as expected.

In [114]:

```
import numpy as np
np.random.randint(5,size=10)
```

Out[114]:

```
array([3, 3, 1, 1, 1, 4, 3, 3, 1, 4])
```

In [117]:

```
import numpy as np
np.random.randint(2,100,size=(4,5))
```

Out[117]:

```
array([[52, 89, 80,  4, 41],
       [52, 39, 85, 93, 55],
       [88, 37, 28, 93, 34],
       [42, 81, 28, 93, 39]])
```

Random floats between 0 and 1

A 1-dimensional array of floats between 0 and 1. It is useful to create random noise data.

In [118]:

```
import numpy as np
np.random.random(100)
```

Out[118]:

```
array([0.02901457, 0.71474263, 0.54761842, 0.86068982, 0.19512298,
       0.72172834, 0.56559598, 0.37947743, 0.91035372, 0.78708975,
       0.10958591, 0.02261699, 0.19318949, 0.00573727, 0.59769839,
       0.92955357, 0.90281079, 0.11171462, 0.50075825, 0.54451272,
       0.1493147 , 0.99126197, 0.35080799, 0.70339292, 0.43182974,
       0.62198211, 0.65879963, 0.61002776, 0.20273748, 0.92614717,
       0.01174967, 0.93561684, 0.06682915, 0.28053718, 0.52407988,
       0.74399641, 0.78174856, 0.94305984, 0.81941048, 0.61430647,
       0.90103436, 0.65734534, 0.65180952, 0.81889688, 0.34392473,
       0.70820734, 0.19086601, 0.76756342, 0.35772151, 0.20666907,
       0.18931905, 0.81128279, 0.74213985, 0.59089356, 0.6474277 ,
       0.90773513, 0.31157361, 0.91331591, 0.50295779, 0.40978179,
       0.59758017, 0.51426161, 0.00754165, 0.45039679, 0.81173481,
       0.61809398, 0.93379407, 0.37438654, 0.76827659, 0.61604502,
       0.57437943, 0.2494756 , 0.04502579, 0.86915462, 0.61896668,
       0.11657972, 0.65131394, 0.26272712, 0.9028757 , 0.37514775,
       0.64188754, 0.52537279, 0.70565906, 0.80759737, 0.87523512,
       0.33126004, 0.23002976, 0.33773986, 0.93660877, 0.42711366,
       0.24779807, 0.56307303, 0.11938331, 0.10717519, 0.79461838,
       0.52616954, 0.02709545, 0.47529388, 0.54324112, 0.59937121])
```

Sample from a standard normal distribution

Np.random.randn() is used to create a sample from a standard normal distribution (i.e. zero mean and unit

In [12]:

```
import numpy as np
print(np.random.randn(100).mean())
print(np.random.randn(100).std())
```

```
0.04468769766113926
1.0703002193281377
```

Create a matrix in a Numpy

There is another way to create a matrix in python. It is using the numpy matrix() methods. It is the lists of the list.

In [94]:

```
import numpy as np
list1 = [2,5,1]
list2 = [1,3,5]
list3 = [7,5,8]

matrix2 = np.matrix([list1,list2,list3])
matrix2
```

Out[94]:

```
matrix([[2, 5, 1],
        [1, 3, 5],
        [7, 5, 8]])
```

In [95]:

```
matrix2.shape
```

Out[95]:

```
(3, 3)
```

Matrix with ones and zeros

A matrix can be considered as a 2-dimensional array. We can create a matrix with zeros or ones with np.zeros and np.ones, respectively.

In [15]:

```
import numpy as np
np.ones((4,3))
```

Out[15]:

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

In [16]:

```
import numpy as np
np.zeros((4,3))
```

Out[16]:

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Identity matrix

An identity matrix is a square matrix **(nxn)** that has **ones on the diagonal and zeros on every other position**. **Np.eye** or **np.identity** can be used to create one.

In [17]:

```
import numpy as np
np.eye(3)
```

Out[17]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [18]:

```
import numpy as np
np.identity(4)
```

Out[18]:

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [19]:

```
import numpy as np
np.eye(4,3)
```

Out[19]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])
```

Array with only one value

We can create an array that has the same value at every position using **np.full**.

In [119]:

```
import numpy as np
np.full((3,4),7)
```

Out[119]:

```
array([[7, 7, 7, 7],
       [7, 7, 7, 7],
       [7, 7, 7, 7]])
```

In [21]:

```
import numpy as np
np.full((3,5),5,dtype='float')
```

Out[21]:

```
array([[5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.]])
```

Manipulating arrays

Ravel

Ravel function flattens the array (i.e. convert to a 1-dimensional array).

In [23]:

```
import numpy as np
a=np.random.randint(5,size=(3,4))
a
```

Out[23]:

```
array([[0, 0, 2, 2],
       [3, 4, 3, 4],
       [3, 0, 1, 4]])
```

In [24]:

```
a.ravel(order='F')
```

Out[24]:

```
array([0, 3, 3, 0, 4, 0, 2, 3, 1, 2, 4, 4])
```

Transpose

Transposing a matrix is to switch rows and columns.

In [80]:

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

```
[[ 1  2  3]
 [ 1  1 -3]]
```

In [26]:

```
import numpy as np
a = np.array([[1, 2, 3]])
print("a = ")
print(a)
print("\na.T = ")
print(a.T)
```

```
a =
[[1 2 3]]

a.T =
[[1]
 [2]
 [3]]
```

In [25]:

```
import numpy as np

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

print("\nWith np.transpose(a) function")
print(np.transpose(a))

print("\nWith ndarray.transpose() method")
print(a.transpose())

print("\nWith ndarray.T short form")
print(a.T)
```

```
a =
[[1 2]
 [3 4]
 [5 6]]

With np.transpose(a) function
[[1 3 5]
 [2 4 6]]

With ndarray.transpose() method
[[1 3 5]
 [2 4 6]]

With ndarray.T short form
[[1 3 5]
 [2 4 6]]
```

Trace

The trace is the sum of diagonal elements in a square matrix. There are two methods to calculate the trace. We can simply use the **trace()** method of an ndarray object or get the diagonal elements first and then get the sum.

In [86]:

```
import numpy as np
a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
print("\nTrace:", a.trace())
print("Trace:", sum(a.diagonal()))
```

```
a =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Trace: 7
Trace: 7
```

Vsplit

Splits an array into multiple sub-arrays vertically.

In [29]:

```
import numpy as np
a=np.random.randint(5,size=(4,3))
a
```

Out[29]:

```
array([[4, 4, 2],
       [2, 3, 2],
       [2, 1, 0],
       [1, 0, 4]])
```

In [30]:

```
np.vsplit(a,2)
```

Out[30]:

```
[array([[4, 4, 2],
        [2, 3, 2]]),
 array([[2, 1, 0],
        [1, 0, 4]])]
```

In [128]:

```
import numpy as np
a=np.random.randint(5,size=(8,4))
a
```

Out[128]:

```
array([[0, 4, 3, 2],
       [3, 1, 3, 2],
       [2, 0, 1, 3],
       [2, 3, 1, 0],
       [4, 4, 0, 2],
       [1, 0, 4, 0],
       [1, 1, 1, 4],
       [0, 2, 3, 3]])
```

In [129]:

```
np.vsplit(a,4)[0]
```

Out[129]:

```
array([[0, 4, 3, 2],
       [3, 1, 3, 2]])
```

In [131]:

```
np.vsplit(a,4)[3]
```

Out[131]:

```
array([[1, 1, 1, 4],
       [0, 2, 3, 3]])
```

Hsplit

It is similar to vsplit but works horizontally.

In [37]:

```
import numpy as np
a=np.array([1,2,2,2,5,8,9,7])
a
```

Out[37]:

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

In [40]:

```
np.hsplit(a,4)
```

Out[40]:

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

In [41]:

```
import numpy as np
a=np.random.randint(5,size=(6,3))
a
```

Out[41]:

```
array([[4, 0, 2],
       [0, 2, 0],
       [4, 2, 0],
       [1, 0, 4],
       [2, 3, 0],
       [3, 1, 3]])
```

In [42]:

```
np.hsplit(a,3)[0].shape
```

Out[42]:

```
(6, 1)
```

Access matrix elements, rows and columns

Access matrix elements

Similar like lists, we can access matrix elements using index. Let's start with a one-dimensional NumPy array.

In [81]:

```
import numpy as np
A = np.array([2, 4, 6, 8, 10])

print("A[0] =", A[0])     # First element
print("A[2] =", A[2])     # Third element
print("A[-1] =", A[-1])   # Last element
```

```
A[0] = 2
A[2] = 6
A[-1] = 10
```

Now, let's see how we can access elements of a two-dimensional array (which is basically a matrix).

In [82]:

```
import numpy as np

A = np.array([[1, 4, 5, 12],
    [-5, 8, 9, 0],
    [-6, 7, 11, 19]])

#  First element of first row
print("A[0][0] =", A[0][0])

# Third element of second row
print("A[1][2] =", A[1][2])

# Last element of last row
print("A[-1][-1] =", A[-1][-1])
```

```
A[0][0] = 1
A[1][2] = 9
A[-1][-1] = 19
```

Access rows of a Matrix

In [83]:

```
import numpy as np

A = np.array([[1, 4, 5, 12],
    [-5, 8, 9, 0],
    [-6, 7, 11, 19]])

print("A[0] =", A[0]) # First Row
print("A[2] =", A[2]) # Third Row
print("A[-1] =", A[-1]) # Last Row (3rd row in this case)
```

```
A[0] = [ 1  4  5 12]
A[2] = [-6  7 11 19]
A[-1] = [-6  7 11 19]
```

Access columns of a Matrix

In [84]:

```
import numpy as np

A = np.array([[1, 4, 5, 12],
    [-5, 8, 9, 0],
    [-6, 7, 11, 19]])

print("A[:,0] =",A[:,0]) # First Column
print("A[:,3] =", A[:,3]) # Fourth Column
print("A[:,-1] =", A[:,-1]) # Last Column (4th column in this case)
```

```
A[:,0] = [ 1 -5 -6]
A[:,3] = [12  0 19]
A[:,-1] = [12  0 19]
```

Combining arrays

We may need to combine arrays in some cases. NumPy provides functions and methods to combine array in many different ways.

Concatenate

It is similar to the concat function of pandas.

In [43]:

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

Out[43]:

```
array([1, 2, 3, 4, 3, 3, 3])
```

We can convert these arrays to column vectors using the reshape function and then concatenate vertically.

In [47]:

```
import numpy as np
a=np.array([1,2,3,4])
b=np.array([5,5,5,5])
np.concatenate((a.reshape(-1,1),b.reshape(-1,1)), axis=1)
```

Out[47]:

```
array([[1, 5],
       [2, 5],
       [3, 5],
       [4, 5]])
```

Vstack

It is used to stack arrays vertically (rows on top of each other).

In [48]:

```
import numpy as np
a=np.array([1,2,3,4])
b=np.array([3,3,3,3])
c=np.array([0,1,5,6])
np.vstack((a,b,c))
```

Out[48]:

```
array([[1, 2, 3, 4],
       [3, 3, 3, 3],
       [0, 1, 5, 6]])
```

It also works with higher dimensional arrays.

In [49]:

```
import numpy as np
a=np.random.randint(5,size=(2,2,2))
b=np.random.randint(5,size=(2,2,2))
c=np.random.randint(5,size=(2,2,2))
np.vstack((a,b,c))
```

Out[49]:

```
array([[[3, 4],
        [4, 1]],

       [[1, 2],
        [0, 2]],

       [[1, 4],
        [3, 3]],

       [[4, 3],
        [1, 1]],

       [[2, 3],
        [3, 1]],

       [[4, 1],
        [3, 1]]])
```

Hstack

Similar to **vstack** but works horizontally (column-wise).

In [50]:

```
import numpy as np
a=np.array([1,2,3,4]).reshape(-1,1)
b=np.array([3,3,3,3]).reshape(-1,1)
c=np.array([0,1,5,6]).reshape(-1,1)
np.hstack((a,b,c))
```

Out[50]:

```
array([[1, 3, 0],
       [2, 3, 1],
       [3, 3, 5],
       [4, 3, 6]])
```

Simultaneous Linear Equations

In a matrix, you can solve the linear equations using the matrix.

In [96]:

```
left_hand_side = np.matrix([[1,1,-1], # x + y - z = 4
                        [1,-2,3],      # x - 2y + 3z = -6
                        [2,3,1]])      #  2x + 3y + z = 7
left_hand_side
```

Out[96]:

```
matrix([[ 1,  1, -1],
        [ 1, -2,  3],
        [ 2,  3,  1]])
```

In [97]:

```
right_hand_side = np.matrix([[4],
                             [-6],
                             [7]])
right_hand_side
```

Out[97]:

```
matrix([[ 4],
        [-6],
        [ 7]])
```

In [98]:

```
left_hand_side_inverse = left_hand_side.I
left_hand_side_inverse
```

Out[98]:

```
matrix([[ 0.84615385,  0.30769231, -0.07692308],
        [-0.38461538, -0.23076923,  0.30769231],
        [-0.53846154,  0.07692308,  0.23076923]])
```

In [99]:

```
solution = left_hand_side_inverse*right_hand_side
solution
```

Out[99]:

```
matrix([[ 1.],
        [ 2.],
        [-1.]])
```

In [101]:

```
#You can verify the solution is correct or not by the following
left_hand_side*solution - right_hand_side
```

Out[101]:

```
matrix([[-8.8817842e-16],
        [ 8.8817842e-16],
        [ 0.0000000e+00]])
```

Linear algebra with NumPy arrays (numpy.linalg)

Linear algebra is fundamental in the field of data science. NumPy being the most widely used scientific computing library provides numerous linear algebra operations.

Rank

The **rank** of a matrix is the dimensions of the vector space spanned (generated) by its columns or rows. In other words, it can be defined as the maximum number of linearly independent column vectors or row vectors.

The rank of a matrix can be found using the **matrix_rank()** function.

In [87]:

```
import numpy as np
a = np.arange(1, 10)
a.shape = (3, 3)
print("a = ")
print(a)
rank = np.linalg.matrix_rank(a)
print("\nRank:", rank)
```

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

Rank: 2
```

Det

Returns the determinant of a matrix.

The determinant of a square matrix can be calculated det() function. If the determinant is 0, that matrix is not invertible. It is known as a singular matrix in algebra terms.

In [51]:

```
import numpy as np
a=np.random.randint(5,size=(3,3))
a
```

Out[51]:

```
array([[3, 0, 3],
       [1, 4, 1],
       [2, 2, 4]])
```

In [52]:

```
np.linalg.det(a)
```

Out[52]:

```
24.000000000000004
```

Note: A matrix must be square (i.e. the number of rows is equal to the number of columns) to calculate the determinant. For a higher-dimensional array, the last two dimensions must be square.

In [88]:

```
import numpy as np
a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
```

```
a =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Determinant: 5.0
```

Inv

Calculates the inverse of a matrix.

The true inverse of a square matrix can be found using the inv() function. If the determinant of a square matrix is not 0, it has a true inverse.

In [53]:

```
import numpy as np
a=np.random.randint(5,size=(3,3))
a
```

Out[53]:

```
array([[2, 3, 4],
       [1, 3, 0],
       [4, 2, 3]])
```

In [54]:

```
np.linalg.inv(a)
```

Out[54]:

```
array([[-0.29032258,  0.03225806,  0.38709677],
       [ 0.09677419,  0.32258065, -0.12903226],
       [ 0.32258065, -0.25806452, -0.09677419]])
```

Note: The **inverse** of a matrix is the matrix that gives the identity matrix when multiplied with the original matrix. Not every matrix has an inverse. If matrix A has an inverse, then it is called **invertible** or **non-singular**.

In [89]:

```
import numpy as np
a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
inv = np.linalg.inv(a)
print("\nInverse of a = ")
print(inv)
```

```
a =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Determinant: 5.0

Inverse of a =
[[ 0.8 -0.4 -0.2]
 [-0.2  0.6 -0.2]
 [-0.2 -0.4  0.8]]
```

If you try to compute the true inverse of a singular matrix (a square matrix whose determinant is 0), you will get an error.

In [90]:

```
import numpy as np
a = np.array([[2, 8],
              [1, 4]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
inv = np.linalg.inv(a)
print("\nInverse of a = ")
print(inv)
```

```
a =
[[2 8]
 [1 4]]

Determinant: 0.0

---------------------------------------------------------------------------
-
LinAlgError                               Traceback (most recent call las
t)
Cell In[90], line 8
      6 det = np.linalg.det(a)
      7 print("\nDeterminant:", np.round(det))
----> 8 inv = np.linalg.inv(a)
      9 print("\nInverse of a = ")
     10 print(inv)

File <__array_function__ internals>:180, in inv(*args, **kwargs)

File ~\anaconda3\lib\site-packages\numpy\linalg\linalg.py:552, in inv(a)
    550 signature = 'D->D' if isComplexType(t) else 'd->d'
    551 extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 552 ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
    553 return wrap(ainv.astype(result_t, copy=False))

File ~\anaconda3\lib\site-packages\numpy\linalg\linalg.py:89, in _raise_li
nalgerror_singular(err, flag)
     88 def _raise_linalgerror_singular(err, flag):
---> 89     raise LinAlgError("Singular matrix")

LinAlgError: Singular matrix
```

Pseudo inverse

The pseudo (not genuine) inverse can be calculated even for a singular matrix (a square matrix whose determinant is 0) using the pinv() function.

In [91]:

```
import numpy as np
a = np.array([[2, 8],
              [1, 4]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
pinv = np.linalg.pinv(a)
print("\nPseudo Inverse of a = ")
print(pinv)
```

```
a =
[[2 8]
 [1 4]]

Determinant: 0.0

Pseudo Inverse of a =
[[0.02352941 0.01176471]
 [0.09411765 0.04705882]]
```

Eig

Computes the eigenvalues and right eigenvectors for a square matrix.

In [55]:

```
import numpy as np
a=np.random.randint(5,size=(3,3))
a
```

Out[55]:

```
array([[1, 1, 1],
       [3, 0, 3],
       [0, 0, 3]])
```

In [56]:

```
np.linalg.eig(a)
```

Out[56]:

```
(array([ 2.30277564, -1.30277564,  3.        ]),
 array([[ 0.60889368, -0.3983218 ,  0.53452248],
        [ 0.79325185,  0.91724574,  0.80178373],
        [ 0.        ,  0.        ,  0.26726124]]))
```

In [93]:

```
import numpy as np
a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
w, v = np.linalg.eig(a)
print("\nEigenvalues:")
print(w)
print("\nEigenvectors:")
print(v)
```

```
a =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Eigenvalues:
[5. 1. 1.]

Eigenvectors:
[[ 0.57735027  0.90453403  0.36601928]
 [ 0.57735027 -0.30151134 -0.55609927]
 [ 0.57735027 -0.30151134  0.74617926]]
```

Dot

Calculates the dot product of two vectors which is the sum of the products of elements with regards to their position. The first element of the first vector is multiplied by the first element of the second vector and so on.

In [57]:

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

Out[57]:

```
array([1, 2, 3, 4])
```

In [58]:

```
b=np.array([2,2,2,2])
b
```

Out[58]:

```
array([2, 2, 2, 2])
```

In [59]:

```
np.dot(a,b)
```

Out[59]:

```
20
```

In [79]:

```
import numpy as np
a=np.array([[3,6,7],[5,-3,0]])
b=np.array([[1,1],[2,1],[3,-3]])
c=a.dot(b)
print(c)
```

```
[[ 36 -12]
 [ -1   2]]
```

In [85]:

```
import numpy as np

# Matrices as ndarray objects
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6, 7], [8, 9, 10]])
print("a", type(a))
print(a)
print("\nb", type(b))
print(b)

# Matrices as matrix objects
c = np.matrix([[1, 2], [3, 4]])
d = np.matrix([[5, 6, 7], [8, 9, 10]])
print("\nc", type(c))
print(c)
print("\nd", type(d))
print(d)
print("\ndot product of two ndarray objects")
print(np.dot(a, b))
print("\ndot product of two matrix objects")
print(np.dot(c, d))
```

```
a <class 'numpy.ndarray'>
[[1 2]
 [3 4]]

b <class 'numpy.ndarray'>
[[ 5  6  7]
 [ 8  9 10]]

c <class 'numpy.matrix'>
[[1 2]
 [3 4]]

d <class 'numpy.matrix'>
[[ 5  6  7]
 [ 8  9 10]]

dot product of two ndarray objects
[[21 24 27]
 [47 54 61]]

dot product of two matrix objects
[[21 24 27]
 [47 54 61]]
```

Matmul

It performs matrix multiplication.

In [61]:

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

Out[61]:

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

In [62]:

```
b=([[3,4],[3,1]])
b
```

Out[62]:

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

In [63]:

```
np.matmul(a,b)
```

Out[63]:

```
array([[ 9,  6],
       [21, 16]])
```

More Examples

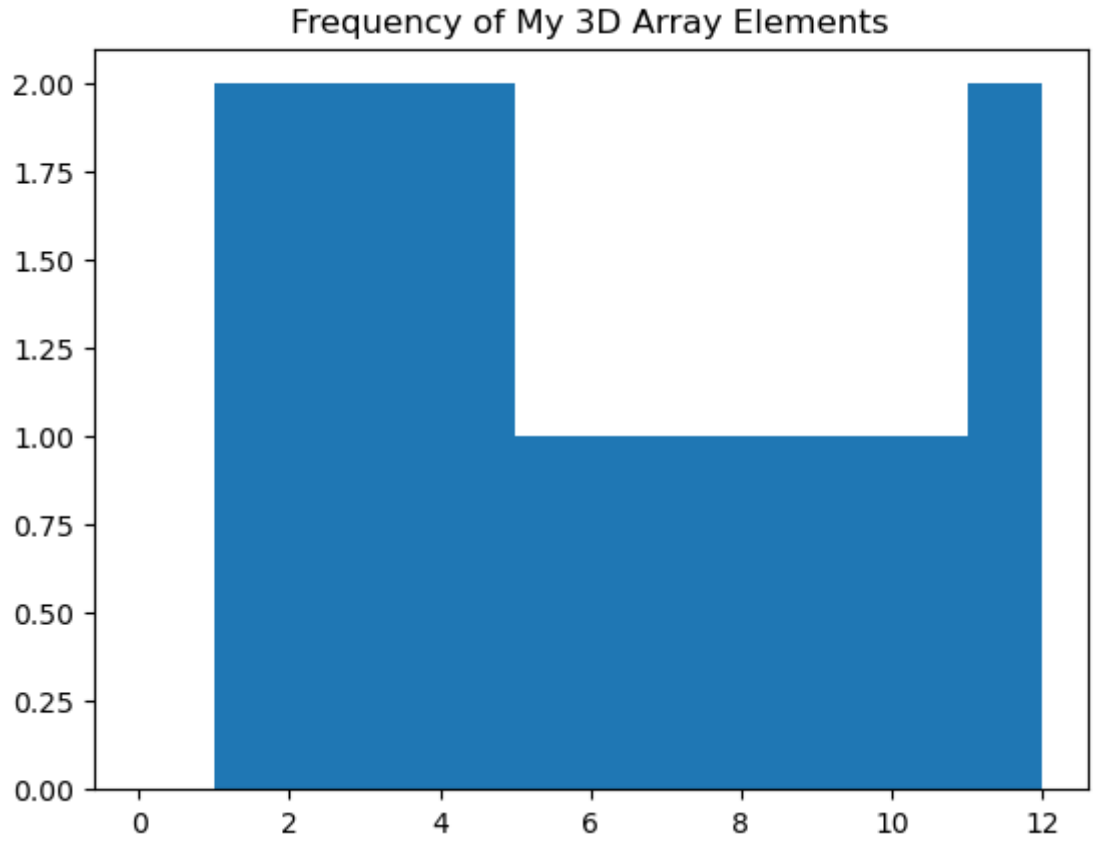
Visualize NumPy Arrays

With np.histogram()

The np.histogram() function doesn't draw the histogram but it does compute the occurrences of the array that fall within each bin; This will determine the area that each bar of your histogram takes up.

In [105]:

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

# Initialize your array
my_3d_array = np.array([[[1,2,3,4], [5,6,7,8]], [[1,2,3,4], [9,10,11,12]]], dtype=np.int

# Construct the histogram with a flattened 3d array and a range of bins
plt.hist(my_3d_array.ravel(), bins=range(0,13))

# Add a title to the plot
plt.title('Frequency of My 3D Array Elements')

# Show the plot
plt.show()
```



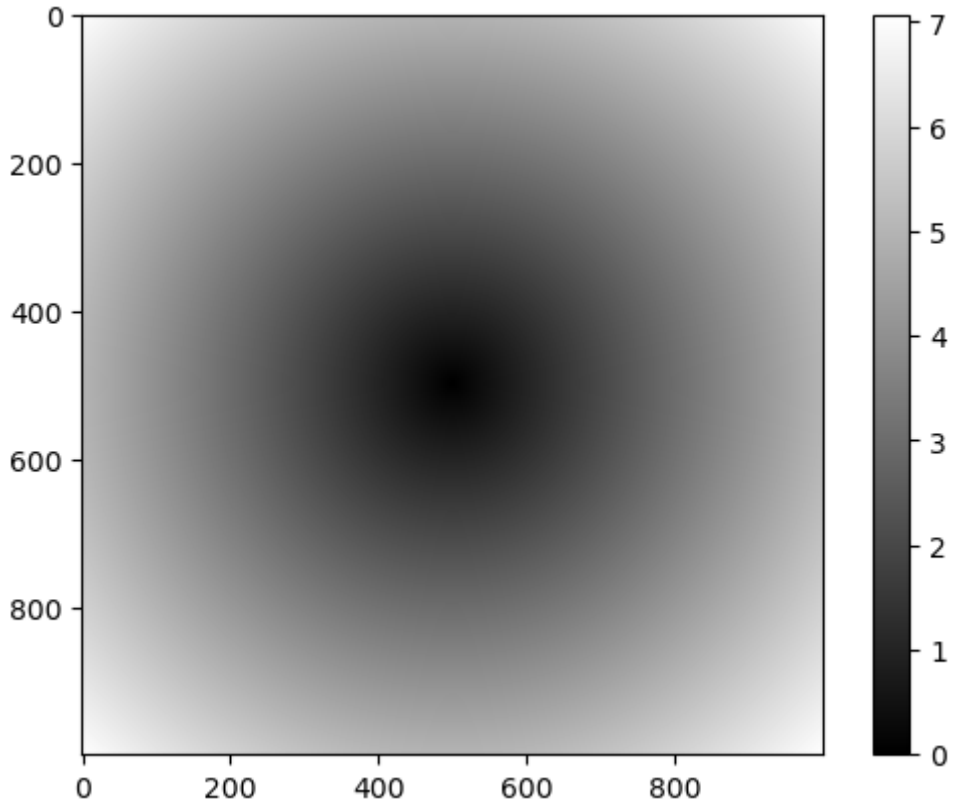
Using np.meshgrid()

Another way to (indirectly) visualize your array is by using np.meshgrid(). The problem that you face with arrays is that you need 2-D arrays of x and y coordinate values. With the above function, you can create a rectangular grid out of an array of x values and an array of y values: the np.meshgrid() function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of (x, y) in the two arrays. Then, you can use these matrices to make all sorts of plots.

np.meshgrid() is particularly useful if you want to evaluate functions on a grid.

In [104]:

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

# Create an array
points = np.arange(-5, 5, 0.01)

# Make a meshgrid
xs, ys = np.meshgrid(points, points)
z = np.sqrt(xs ** 2 + ys ** 2)

# Display the image on the axes
plt.imshow(z, cmap=plt.cm.gray)

# Draw a color bar
plt.colorbar()

# Show the plot
plt.show()
```



Importing and exporting a CSV

In [132]:

```
import pandas as pd
# If all of your columns are the same type:
x = pd.read_csv('music.csv', header=0).values
print(x)
```

```
[['Billie Holiday' 'Jazz' '13,00,000' '2,70,00,000']
 ['Jimi Hendrix' 'Rock' '27,00,000' '7,00,00,000']
 ['Miles Davis' 'Jazz' '15,00,000' '4,80,00,000']
 ['SIA' 'Pop' '20,00,000' '7,40,00,000']]
```

In [107]:

```
# You can also simply select the columns you need:
x = pd.read_csv('music.csv', usecols=['Artist', 'Plays']).values
print(x)
```

```
[['Billie Holiday' '2,70,00,000']
 ['Jimi Hendrix' '7,00,00,000']
 ['Miles Davis' '4,80,00,000']
 ['SIA' '7,40,00,000']]
```

Creating a Pandas dataframe from the values in your array and then write the data frame to a CSV file with Pandas.

In [108]:

```
a = np.array([[-2.58289208,  0.43014843, -1.24082018, 1.59572603],
              [ 0.99027828, 1.17150989,  0.94125714, -0.14692469],
              [ 0.76989341,  0.81299683, -0.95068423, 0.11769564],
              [ 0.20484034,  0.34784527,  1.96979195, 0.51992837]])
```

In [109]:

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

```
          0         1         2         3
0 -2.582892  0.430148 -1.240820  1.595726
1  0.990278  1.171510  0.941257 -0.146925
2  0.769893  0.812997 -0.950684  0.117696
3  0.204840  0.347845  1.969792  0.519928
```

In [111]:

```
#save dataframe
df.to_csv('pd.csv')
```

In [113]:

```
#read CSV
data = pd.read_csv('pd.csv')
data
```

Out[113]:

	Unnamed: 0	0	1	2	3
0	0	-2.582892	0.430148	-1.240820	1.595726
1	1	0.990278	1.171510	0.941257	-0.146925
2	2	0.769893	0.812997	-0.950684	0.117696
3	3	0.204840	0.347845	1.969792	0.519928

In [114]:

```
# We can also save array with the NumPy savetxt method.
np.savetxt('np.csv', a, fmt='%.2f', delimiter=',', header='1,  2,  3,  4')
```

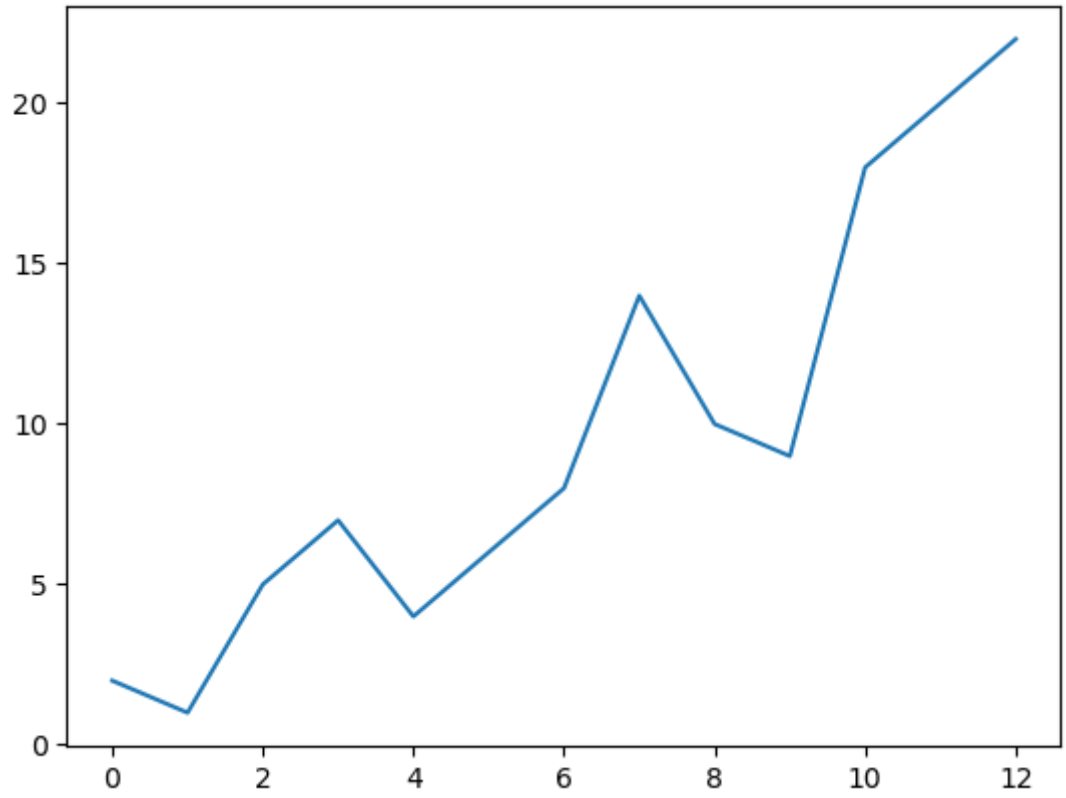
Plotting arrays with Matplotlib

In [134]:

```
import numpy as np
import matplotlib.pyplot as plt
a = np.array([2, 1, 5, 7, 4, 6, 8, 14, 10, 9, 18, 20, 22])
plt.plot(a)
```

Out[134]:

```
[<matplotlib.lines.Line2D at 0x2609bb783d0>]
```

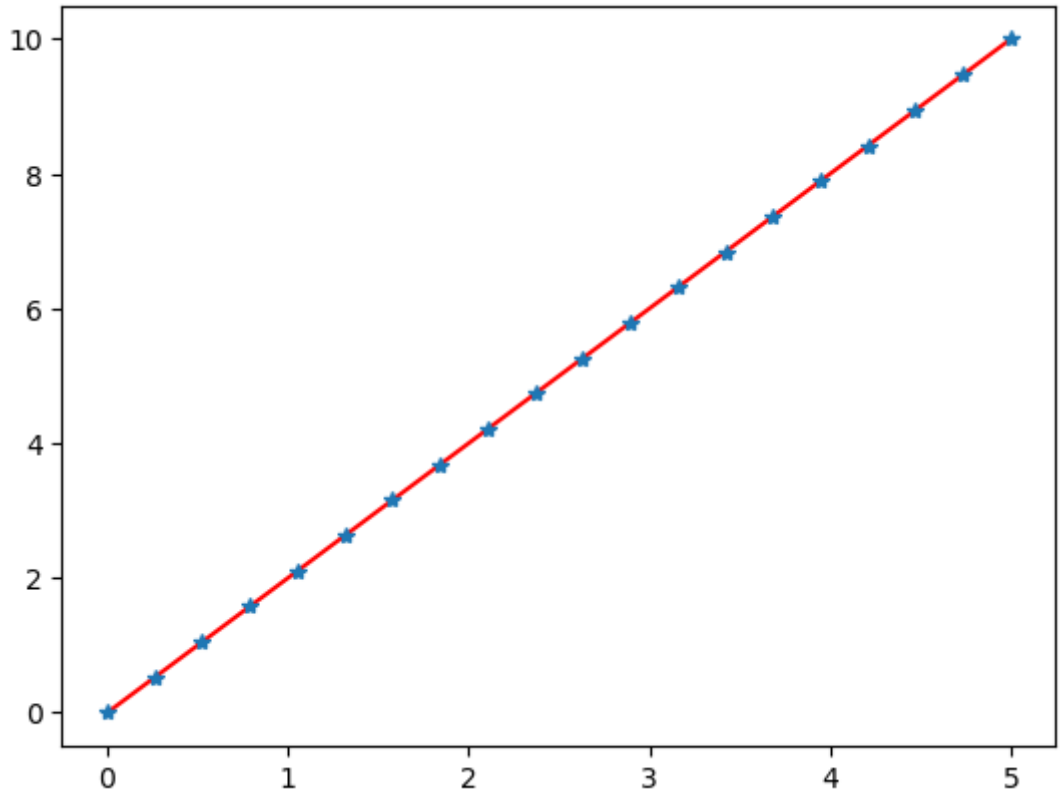


In [136]:

```
#plot a 1D array
x = np.linspace(0, 5, 20)
y = np.linspace(0, 10, 20)
plt.plot(x, y, 'red') # line
plt.plot(x, y, '*')      # dots
```

Out[136]:

```
[<matplotlib.lines.Line2D at 0x2609bc3a7f0>]
```

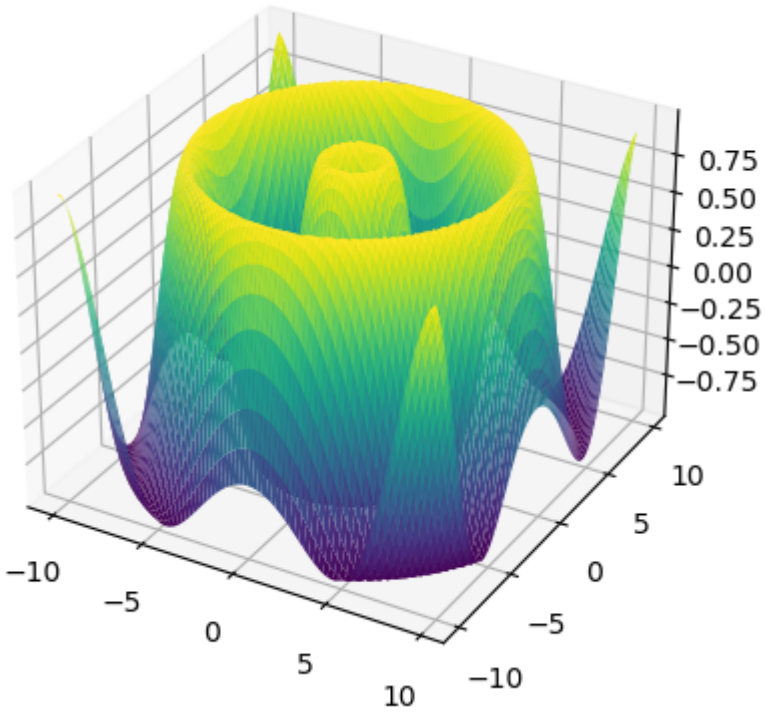


In [137]:

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
X = np.arange(-10, 10, 0.25)
Y = np.arange(-10, 10, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis')
```

Out[137]:

```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x2609bcbaf10>
```



Strings and Characters

There are quite a few data structures available. The builtins data structures are: lists, tuples, dictionaries, strings, sets and frozensets.

Lists, strings and tuples are ordered sequences of objects. Unlike strings that contain only characters, list and tuples can contain any type of objects.

Lists and tuples are like arrays. Tuples like strings are immutables.

Lists are mutables so they can be extended or reduced at will. Sets are mutable unordered sequence of unique elements whereas frozensets are immutable sets.

Strings

Strings can be defined as sequential collections of characters. This means that the individual characters that make up a string are in a particular order from left to right.

A string that contains no characters, often referred to as the empty string, is still considered to be a string. It is simply a sequence of zero characters and is represented by ‘’ or “” (two single or two double quotes with nothing in between).

In short, strings are immutable sequence of characters. There are a lot of methods to ease manipulation and creation of strings.

https://youtu.be/T435lvYXE_w (https://youtu.be/T435lvYXE_w)

Creating String

Single and double quotes are special characters. There are used to defined strings. There are actually 3 ways to define a string using either single, double or triple quotes:

text = 'CMR Technical Campus'

text = "CMR Technical Campus"

text = '''CMR Technical Campus'''

In fact the latest is generally written using triple double quotes:

text = """CMR Technical Campus"""

In [1]:

```
text = 'CMR Technical Campus'
print(text)
```

```
CMR Technical Campus
```

In [2]:

```
text = "CMR Technical Campus"
print(text)
```

```
CMR Technical Campus
```

In [3]:

```
text = '''CMR Technical Campus'''
print(text)
```

```
CMR Technical Campus
```

In [4]:

```
text = """CMR Technical Campus"""
text
```

Out[4]:

```
'CMR Technical Campus'
```

Strings in double quotes work exactly the same as in single quotes but allow to insert single quote character inside them.

The interest of the triple quotes (‘’’ or “””) is that you can specify multi-line strings. Moreover, single quotes and double quotes can be used freely within the triple quotes:

text = """ CMR Technical Campus " and ' CSE Department """

In [5]:

```
text = """ a string with special character " and ' inside """
print(text)
```

```
 a string with special character " and ' inside
```

In [6]:

```
text = """Gandhi's birthday, 2 October, is commemorated in India as Gandhi Jayanti, a na
and worldwide as the International Day of Nonviolence. """
print(text)
print(len(text))
```

```
Gandhi's birthday, 2 October, is commemorated in India as Gandhi Jayanti,
a national holiday,
and worldwide as the International Day of Nonviolence.
150
```

The ” and ‘ characters are part of the Python language; they are special characters. To insert them in a string, you have to escape them (i.e., with a \ chracter in front of them to indicate the special nature of the character).

In [7]:

```
text = " a string with escaped special character \", \' inside "
print(text)
```

```
 a string with escaped special character ", ' inside
```

Operations on Strings

String Length

To get the length of a string, use the len() function.

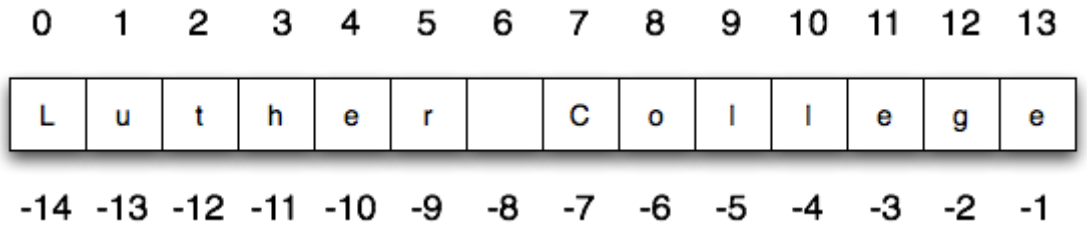
In [8]:

```
a = "Hello, CMR TC!"
print(len(a))
```

Index Operator: Working with the Characters of a String

The indexing operator (Python uses square brackets to enclose the index) selects a single character from a string. The characters are accessed by their position or index value.

For example, in the string shown below, the 14 characters are indexed left to right from postion 0 to position 13.



It is also the case that the positions are named from right to left using negative numbers where -1 is the rightmost index and so on.

Note that the character at index 6 (or -8) is the blank character.

In [9]:

```
college = "CMR Technical Campus"
m = college[2]
print(m)

l = college[-2]
print(l)
```

```
R
u
```

In [10]:

```
s = "CMR Technical Campus"
print(s[2] + s[-4])
```

```
Rm
```

Strings are immutable

You can access to any character using slicing, However, you cannot change any character

In [11]:

```
s = "CMR Technical Campus"
print(s[:])
print(s[-4:-1])
print(s[:4])
s
```

```
CMR Technical Campus
mpu
CMR
```

Out[11]:

```
'CMR Technical Campus'
```

In [12]:

```
s[2]='K'
print(s)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[12], line 1
----> 1 s[2]='K'
      2 print(s)

TypeError: 'str' object does not support item assignment
```

One final thing that makes strings different from some other Python collection types is that you are not allowed to modify the individual characters in the collection. It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.

For example, in the following code, we would like to change the first letter of greeting.

In [13]:

```
greeting = "Hello, CMR TC"
greeting[0] = 'J'
print(greeting)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[13], line 2
      1 greeting = "Hello, CMR TC"
----> 2 greeting[0] = 'J'
      3 print(greeting)

TypeError: 'str' object does not support item assignment
```

Strings are immutable, which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original.

In [14]:

```
greeting = "Hello, CMR TC"
newGreeting = 'J' + greeting[1:]
print(newGreeting)
print(greeting)
```

```
Jello, CMR TC
Hello, CMR TC
```

Check String

To check if a certain phrase or character is present in a string, we can use the keywords 'in' or 'not in'.

In [15]:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x)
```

```
True
```

In [16]:

```
txt = "The rain in Spain stays mainly in the plain"
x = "Ain" not in txt
print(x)
```

```
True
```

String Operators

In [17]:

```
str = "CMR"
str1 = " Technical Campus"
print(str*3)
print(str+str1)
print(str[2])
print(str[1:2])
print('c' in str)
print('ac' not in str1)
```

```
CMRCMRCMR
CMR Technical Campus
R
M
False
True
```

Compare Two Strings

We use the == operator to compare two strings. If two strings are equal, the operator returns **True**. Otherwise, it returns **False**.

In [72]:

```
str1 = "Hello, world!"
str2 = "I love Python."
str3 = "Hello, world!"

# compare str1 and str2
print(str1 == str2)

# compare str1 and str3
print(str1 == str3)
```

```
False
True
```

Strings and for loops

Since a string is simply a sequence of characters, the for loop iterates over each character automatically. (As always, try to predict what the output will be from this code before your run it.)

In [18]:

```
for achar in "CMR Technical Campus":
    print(achar)
```

```
C
M
R

T
e
c
h
n
i
c
a
l

C
a
m
p
u
s
```

In [19]:

```
#How many times is the word CMR printed by the following statements?
s = "python data structures"
print(len(s))
for ch in s:
   print("CMR")
```

```
22
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
CMR
```

In [20]:

```
#How many times is the word CMR printed by the following statements?
s = "python data structures"
for ch in s[3:8]:
   print(ch)
```

```
h
o
n

d
```

String Methods (Working with Characters)

There are a wide variety of methods for string objects. Here are some of the most common string methods. A method is like a function, but it runs "on" an object.

s.lower(), s.upper()

returns the lowercase or uppercase version of the string

In [22]:

```
ss = "Hello, CMR Technical Campus"
print(ss.upper())

tt = ss.lower()
print(tt)
print(ss)
```

```
HELLO, CMR TECHNICAL CAMPUS
hello, cmr technical campus
Hello, CMR Technical Campus
```

s.strip()

returns a string with whitespace removed from the start and end

In [23]:

```
ss = "    Hello, World    "

print(ss)
els = ss.count("l")

print(els)

print(ss.strip())

print("***"+ss+"***")

print("***"+ss.strip()+"***")

news = ss.replace("o", "***")

print(news)
print(ss)
```

```
    Hello, World
3
Hello, World
***    Hello, World    ***
***Hello, World***
    Hell***, W***rld
    Hello, World
```

In [24]:

```
s = "CMR Technical Campus"
print(s.count("a") + s.count("m"))
```

```
3
```

In [25]:

```
s = "CMR Technical Campus"
print(s[1]*s.index("n"))
print(s.index("n")*s[1])
```

```
MMMMMMMM
MMMMMMMM
```

s.isalpha()/s.isdigit()/s.isspace()

tests if all the string chars are in the various character classes

In [26]:

```
s = "CMR"
a="123"
b=" "
x=s.isalpha()
print(x)
print(a.isalpha())
```

```
True
False
```

In [27]:

```
s = "Cmr1234"

d = 0
l = 0
u = 0
for i in s:
    if i.isdigit():
        d = d + 1
    elif i.islower():
        l = l + 1
    else:
        u = u+1
print("No of digits in given string: ", d)
print("No of lower case letters in given string: ", l)
print("No of upper case letters in given string: ", u)
```

```
No of digits in given string:  4
No of lower case letters in given string:  2
No of upper case letters in given string:  1
```

In [28]:

```
a="a123"
y=a.isdigit()
print(y)
```

```
False
```

In [29]:

```
b="aaa"
z=b.isspace()
print(z)
```

```
False
```

s.startswith(s1), s.endswith(s1)

tests if the string starts or ends with the given other string

In [30]:

```
s = "CMR Technical Campus"
s.startswith('C')
```

Out[30]:

```
True
```

In [31]:

```
s = "CMR Technical Campus"
s1="CmR"
s.startswith(s1)
```

Out[31]:

```
False
```

In [32]:

```
s = "CMR Technical Campus"
s.endswith('Campus')
```

Out[32]:

```
True
```

In [33]:

```
s = "CMR Technical Campus"
s1="Campus"
s.endswith(s1)
```

Out[33]:

```
True
```

Searching Strings (s.find(s1))

searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found

In [34]:

```
s = "CMR Technical Campus"
s.find('kal')
```

Out[34]:

```
-1
```

In [35]:

```
s = "CMR Technical Campus"
s1="Tech"
s.find(s1)
```

Out[35]:

```
4
```

s.replace('old', 'new')

returns a string where all occurrences of 'old' have been replaced by 'new'

In [43]:

```
s = "CMR Technical Campus"
print(s)
s.replace('Technical','Professional')
```

```
CMR Technical Campus
```

Out[43]:

```
'CMR Professional Campus'
```

String Reversing

With Accessing Characters from a string, we can also reverse them. We can Reverse a string by writing [::-1] and the string will be reversed.

In [21]:

```
#Program to reverse a string
abc = "CMR Technical Campus"
print(abc[::-1])
```

```
supmaC lacinhceT RMC
```

s.split('delim')

returns a list of substrings separated by the given delimiter.

The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.

Two of the most useful methods on strings involve lists of strings. The split method breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary.

In [44]:

```
s = "CMR Technical Campus"
print(type(s))
k=s.split()
print(type(k))
print(k)
```

```
<class 'str'>
<class 'list'>
['CMR', 'Technical', 'Campus']
```

In [138]:

```
s = "CMR Tech,nical, Campus"
k=s.split(',')
print(k)
```

```
['CMR Tech', 'nical', ' Campus']
```

In [46]:

```
song = "The rain in Spain..."
wds = song.split()
print(wds)
```

```
['The', 'rain', 'in', 'Spain...']
```

In [47]:

```
song = "The rain in Spain..."
wds = song.split('ai')
print(wds)
```

```
['The r', 'n in Sp', 'n...']
```

In [139]:

```
#palindrome
str = input("Enter any string")
str1 = str[::-1]
if str == str1:
    print("Given string is a Palindrome")
else:
    print("Given string is not a Palindrome")
```

```
Enter any string125
Given string is not a Palindrome
```

s.join(list)

opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

The inverse of the split method is join. You choose a desired separator string, (often called the glue) and join the list with the glue between each of the elements.

In [50]:

```
lst=['CMR', 'Technical', 'Campus']
s = ' '.join(lst)
print(s)
```

```
CMR Technical Campus
```

In [51]:

```
lst=['CMR', 'Technical', 'Campus']
s=','.join(lst)
print(s)
```

```
CMR,Technical,Campus
```

In [52]:

```
wds = ["red", "blue", "green"]
glue = ';'
s = glue.join(wds)
print(s)
print(wds)

print("***".join(wds))
print("".join(wds))
```

```
red;blue;green
['red', 'blue', 'green']
red***blue***green
redbluegreen
```

In [59]:

```
#We can also reverse a string by using built-in join and reversed function.
# Program to reverse a string

abc = "cmr technical campus"

# Reverse the string using reversed and join function
abc = "".join(reversed(abc))

print(abc)
```

```
supmac lacinhcet rmc
```

String Slicing

To access a range of characters in the String, the method of slicing is used. Slicing in a String is done by using a Slicing operator (colon).

In [60]:

```
String1 = "CMR Technical Campus"
print("Initial String: ")
print(String1)

# Printing 3rd to 12th character
print("\nSlicing characters from 3-12: ")
print(String1[3:12])

# Printing characters between 3rd and 2nd last character
print("\nSlicing characters between " + "3rd and 2nd last character: ")
print(String1[3:-2])
```

```
Initial String:
CMR Technical Campus

Slicing characters from 3-12:
 Technica

Slicing characters between 3rd and 2nd last character:
 Technical Camp
```

Deleting/Updating from a String

In Python, the Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. Although deletion of the entire String is possible with the use of a built-in del keyword.

This is because Strings are immutable, hence elements of a String cannot be changed once it has been assigned. Only new strings can be reassigned to the same name.

In [61]:

```
#Updation of a character
String1 = "Hello, I'm Bhaskr"
print("Initial String: ")
print(String1)

# Updating a character of the String
## As python strings are immutable, they don't support item updation directly
### there are following two ways
#1
list1 = list(String1)
list1[2] = 'p'
String2 = ''.join(list1)
print("\nUpdating character at 2nd Index: ")
print(String2)

#2
String3 = String1[0:2] + 'p' + String1[3:]
print(String3)
```

```
Initial String:
Hello, I'm Bhaskr

Updating character at 2nd Index:
Heplo, I'm Bhaskr
Heplo, I'm Bhaskr
```

In [62]:

```
# Python Program to Update entire String

String1 = "Hello, I'm Bhaskar"
print("Initial String: ")
print(String1)

# Updating a String
String1 = "Welcome to the CMR Technical Campus"
print("\nUpdated String: ")
print(String1)
```

```
Initial String:
Hello, I'm Bhaskar

Updated String:
Welcome to the CMR Technical Campus
```

In [63]:

```
# Python Program to Delete characters from a String

String1 = "Hello, I'm Bhaskar"
print("Initial String: ")
print(String1)

# Deleting a character of the String
String2 = String1[0:2] + String1[3:]
print("\nDeleting character at 2nd Index: ")
print(String2)
```

```
Initial String:
Hello, I'm Bhaskar

Deleting character at 2nd Index:
Helo, I'm Bhaskar
```

In [64]:

```
# Python Program to Delete entire String

String1 = "Hello, I'm Bhaskar"
print("Initial String: ")
print(String1)

# Deleting a String with the use of del
del String1
print("\nDeleting entire String: ")
print(String1)
```

```
Initial String:
Hello, I'm Bhaskar

Deleting entire String:

---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[64], line 10
      8 del String1
      9 print("\nDeleting entire String: ")
---> 10 print(String1)

NameError: name 'String1' is not defined
```

Escape Sequencing

While printing Strings with single and double quotes in it causes SyntaxError because String already contains Single and Double Quotes and hence cannot be printed with the use of either of these. Hence, to print such a String either Triple Quotes are used or Escape sequences are used to print such Strings.

In [65]:

```
# Python Program for Escape Sequencing of String

# Initial String
String1 = '''I'm "Bhaskar"'''
print("Initial String with use of Triple Quotes: ")
print(String1)

# Escaping Single Quote
String1 = 'I\'m  "Bhaskar"'
print("\nEscaping Single Quote: ")
print(String1)

# Escaping Double Quotes
String1 = "I'm \"Bhaskar\""
print("\nEscaping Double Quotes: ")
print(String1)

# Printing Paths with the use of Escape Sequences
String1 = "C:\\Python\\II CSE A\\"
print("\nEscaping Backslashes: ")
print(String1)

# Printing Paths with the use of Tab
String1 = "Hi\tBhaskar"
print("\nTab: ")
print(String1)

# Printing Paths with the use of New Line
String1 = "CMR Technical Campus\nBhaskar"
print("\nNew Line: ")
print(String1)
```

```
Initial String with use of Triple Quotes:
I'm "Bhaskar"

Escaping Single Quote:
I'm  "Bhaskar"

Escaping Double Quotes:
I'm "Bhaskar"

Escaping Backslashes:
C:\Python\II CSE A\

Tab:
Hi	Bhaskar

New Line:
CMR Technical Campus
Bhaskar
```

Formatting of Strings

Strings in Python can be formatted with the use of format() method which is a very versatile and powerful tool for formatting Strings. Format method in String contains curly braces {} as placeholders which can hold

In [68]:

```
# Python Program for Formatting of Strings

# Default order
String1 = "{} {} {}".format('CMR', 'Technical', 'Campus')
print("Print String in default order: ")
print(String1)

# Positional Formatting
String1 = "{1} {0} {2}".format('CMR', 'Technical', 'Campus')
print("\nPrint String in Positional order: ")
print(String1)

# Keyword Formatting
String1 = "{l} {f} {g}".format(g='CMR', f='Technical', l='Campus')
print("\nPrint String in order of Keywords: ")
print(String1)
```

```
Print String in default order:
CMR Technical Campus

Print String in Positional order:
Technical CMR Campus

Print String in order of Keywords:
Campus Technical CMR
```

In [69]:

```
# Formatting of Integers
String1 = "{0:b}".format(16)
print("\nBinary representation of 16 is ")
print(String1)

# Formatting of Floats
String1 = "{0:e}".format(165.6458)
print("\nExponent representation of 165.6458 is ")
print(String1)

# Rounding off Integers
String1 = "{0:.2f}".format(1/6)
print("\none-sixth is : ")
print(String1)
```

```
Binary representation of 16 is
10000

Exponent representation of 165.6458 is
1.656458e+02

one-sixth is :
0.17
```

In [70]:

```
# String alignment
String1 = "|{:<10}|{:^10}|{:>10}|".format('CMR', 'Technical', 'Campus')
print("\nLeft, center and right alignment with Formatting: ")
print(String1)

# To demonstrate aligning of spaces
String1 = "\n{0:^16} was founded in {1:<4}!".format("CMR Technical Campus",2009)
print(String1)
```

```
Left, center and right alignment with Formatting:
|CMR       |Technical |    Campus|

CMR Technical Campus was founded in 2009!
```

In [71]:

```
# Python Program for Old Style Formatting of Integers

Integer1 = 12.3456789
print("Formatting in 3.2f format: ")
print('The value of Integer1 is %3.2f' % Integer1)
print("\nFormatting in 3.4f format: ")
print('The value of Integer1 is %3.4f' % Integer1)
```

```
Formatting in 3.2f format:
The value of Integer1 is 12.35

Formatting in 3.4f format:
The value of Integer1 is 12.3457
```

The Accumulator Pattern with Strings

We can also accumulate strings rather than accumulating numbers, as you've seen before.

The following program isn't particularly useful for data processing, but we will see more useful things later that accumulate strings.

In [57]:

```
s = input("Enter some text")
ac = ""
for c in s:
    ac = ac + c + "-" + c + "-"

print(ac)
```

```
Enter some text: bhaskar
b-b-h-h-a-a-s-s-k-k-a-a-r-r-
```

In [58]:

```
#What is printed by the following statements:
s = "ball"
r = ""
for item in s:
    r = r + item.upper()
print(r)
```

```
BALL
```

Sorting Strings

Sort a Python String with Sorted

Python comes with a function, **sorted()**, built-in. This function takes an iterable item and sorts the elements by a given key. The default value for this key is None, which compares the elements directly.

The function returns a list of all the sorted elements.

In [85]:

```
# Sort a Python String with sorted()
word = 'datagy'

sorted_word = sorted(word)
print(sorted_word)
```

```
['a', 'a', 'd', 'g', 't', 'y']
```

Note: We can see that when we apply the sorted function to our string, a list is returned. The list items are sorted alphabetically.

In [86]:

```
#How can we turn this list back into a string? We can use the str.join() method, as show
word = 'datagy'

sorted_word = sorted(word)
sorted_word = ''.join(sorted_word)
print(sorted_word)
```

```
aadgty
```

In [87]:

```
#combine this into a single line by writing
word = 'datagy'

sorted_word = ''.join(sorted(word))
print(sorted_word)
```

```
aadgty
```

In [88]:

```
#Now, what happens when we include some capitals in our string? Let's capitalize the fir
#see what happens
word = 'Datagy'

sorted_word = ''.join(sorted(word))
print(sorted_word)
```

```
Daagty
```

Sort a Python String with Sorted without Case Sensitivity

The key= argument allows us to pass in a function that allows Python to create a comparison key for each element in the iterable.

In [99]:

```
word = 'Datagy'

sorted_word = ''.join(sorted(word.lower()))
print(sorted_word)
```

```
aadgty
```

Sort a Python String with Unique Characters Only

We can use the uniqueness to our advantage here – we can turn our string into a set and then sort the values. Finally, we can convert it back to a string using the .join string method.

In [101]:

```
word = 'Datagy'

sorted_word = ''.join(sorted(set(word)))
print(sorted_word)
```

```
Dagty
```

In [105]:

```
word = 'Datagy'

sorted_word = ''.join(sorted(set(word.lower())))
print(sorted_word)
```

```
adgty
```

Sort a Python String and Remove White Space and Punctuation

Python strings will often contain characters that are not alphabetical, we can find a way to sort our string ignoring these values

In [106]:

```
word = 'da ta ?gy!'

sorted_word = ''.join(filter(lambda x:x.isalpha(), sorted(word,key=lambda x:x.lower())))

print(sorted_word)
```

```
aadgty
```

In [ ]:

```

```