Oluwakemi Folake Motoni

Data Structure

Data structures are a way to arrange data so that, depending on the circumstance, it can be accessed more quickly. Any programming language's fundamental building block upon which a program is based is the data structure.

List

A list is a sequenced collection of different objects such as integers, strings, and even other lists as well

- A list is represented by brackets, [ ].
- It is mutable i.e the element within a list can be modified
- An index is used to access and refer to items within a list.

There are different ways of creating a list,

You can either create an empty list and add all your items

or

You can type in all element of your list using the []

In [22]:
```
# to create an empty list
List_1 = []

List_2 = list()

print(List_1)
print(List_2)
```

```
[]
[]
```

In [1]:
```
# to create a List with elements within

List_3 = ["Kemi", 2019, 2.5, ["mango", "banana"], ("grade")]
List_3
```

Out[1]:
```
['Kemi', 2019, 2.5, ['mango', 'banana'], 'grade']
```

List can contain a string, an integer, a float, another list can be in a list and a tuple as seen above

In [24]:
```
type(List_3)
```

Out[24]:
```
list
```

In [3]:
```
#using positive and negative index to access the elements of a list

print("Element 1:\n positive index", List_3[0],
      "\n Negative Index:", List_3[-5]
     )
```

```
Element 1:
 positive index Kemi
 Negative Index: Kemi
```

In [4]:
```
# Because list is mutable, we can change the elements in a list
List_3[1] = 2022
List_3
```

Out[4]:
```
['Kemi', 2022, 2.5, ['mango', 'banana'], 'grade']
```

In [5]:
```
# We can extract a part of a list using slicing

List_3[3:6]
```

Out[5]:
```
[['mango', 'banana'], 'grade']
```

In [6]:
```
#we can clone a list
New_List = List_3[:]
New_List
```

Out[6]:
```
['Kemi', 2022, 2.5, ['mango', 'banana'], 'grade']
```

List Method

append()

the append method adds an element to the back of the last element in the list.

```
Syntax:
    append(element)
```

for exeample:

In [7]:
```
List_3.append("fruits")
List_3
```

Out[7]:
```
['Kemi', 2022, 2.5, ['mango', 'banana'], 'grade', 'fruits']
```

pop()

Removes the last element in a list, by default it removes one elements if a parameter is not passed in.

In [8]:
```
print(List_3)
print(List_3.pop())
print(List_3)
```

```
['Kemi', 2022, 2.5, ['mango', 'banana'], 'grade', 'fruits']
fruits
['Kemi', 2022, 2.5, ['mango', 'banana'], 'grade']
```

In [9]:
```
# lets pop the item on the second element
List_3.pop(2)
```

Out[9]:
```
2.5
```

Remove()

Removes an item by value, and unlike pop, remove does not return any item

for example

In [10]:
```
my_list = [1,2,3,4,5]
print(my_list.remove(2))
print(my_list)
```

```
None
[1, 3, 4, 5]
```

Index()

Use to find the index position of a value

In [11]:
```
my_list = [1,2,3,4,5]
print(my_list.index(2))
```

```
1
```

Insert()

We use the insert method to add to a particular position in a list

```
Syntax:
    insert(index,value)
```

For example:

In [12]:
```
my_list = [1,2,3,4,5]
my_list.insert(3,"New")
print(my_list)
```

```
[1, 2, 3, 'New', 4, 5]
```

clear ()

we can remove all elements in a list using the clear method

for example:

In [13]:
```
my_list = [1,2,3,4,5]
my_list.clear()
my_list
```

Out[13]:
```
[]
```

sort()

Helps to sort the elements in our list.

for example:

In [14]:
```
my_list = [90,3,5,20,100]
my_list.sort()
my_list
```

Out[14]:
```
[3, 5, 20, 90, 100]
```

extend()

We can use the method extend to add new elements to the list

In [15]:
```
my_list = [1,2,3,4]
my_list.extend([5,6,7,8])
my_list
```

Out[15]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Tuples

Tuples are very similar to lists in that they store objects of various data types.

- A tuple is represented by parentheses, ()
- It is immutable i.e the element within a tuple cannot be modified
- An index is used to access and refer to items within a tuple.

To create a tuple object, we use the tuple literal ( ).

In [38]:
```
tuple_1 = ("Hello", 50, 77.7)
tuple_1
```

Out[38]:
```
('Hello', 50, 77.7)
```

Out[39]:
```
tuple
```

In [40]:
```
#we can also convert a list to a tuple

List_1 = [1,2,3,4,5,6,7]

#convert to tuple

List_1 = tuple([1,2,3,4,5,6,7])
type(List_1)
```

Out[40]:
```
tuple
```

Index

The index helps to get the location each element of a tuple

In [41]:
```
print(tuple_1.index("Hello"))
```

```
0
```

Indexing and slicing

We can also access the elements of a tuple but we can not change it, because they are immutable

In [42]:
```
tuple_1[0]
```

Out[42]:
```
'Hello'
```

In [43]:
```
# We can also get the len od a tuple

len(tuple_1)
```

Out[43]:
```
3
```

Concatenate Tuples

We can concatenate or combine tuples by using the + sign:

In [44]:
```
tuple_2= tuple_1 + (1,2,3,4,5)
tuple_2
```

Out[44]:
```
('Hello', 50, 77.7, 1, 2, 3, 4, 5)
```

In [45]:
```
# Slicing, we can call ou a part of a tuple

tuple_3 = tuple_1[0:3]
tuple_3
```

Out[45]:
```
('Hello', 50, 77.7)
```

We can sort the elements of a tuple

In [1]:
```
die_roll= (4,5,6,2,3,1)

die_roll_sorted=sorted(die_roll)
die_roll_sorted
```

Out[1]:
```
[1, 2, 3, 4, 5, 6]
```

In [2]:
```
# sort descending
sorted(die_roll, reverse =True)
```

Out[2]:
```
[6, 5, 4, 3, 2, 1]
```

Nested Tuple

tuple can contain another tuple as well as other data type, this is called "nesting".

In [47]:
```
Nested_tuple = (("Hello","Python"),(1,2,3,4,5),7,8,(55.4,30.3,(4,5)))
```

In [48]:
```
#We cn access each each element of the tuple using the index

print("Element 0", Nested_tuple[0])
print("Element 1", Nested_tuple[1])
print("Element 2", Nested_tuple[2])
print("Element 3", Nested_tuple[3])
print("Element 4", Nested_tuple[4])
```

```
Element 0 ('Hello', 'Python')
Element 1 (1, 2, 3, 4, 5)
Element 2 7
Element 3 8
Element 4 (55.4, 30.3, (4, 5))
```

In [49]:
```
#We can print the element within each tuple using second index
print("Element 0", Nested_tuple[0][0])
print("Element 1", Nested_tuple[0][1])
```

```
Element 0 Hello
Element 1 Python
```

In [50]:
```
#we can also access each character of the string using the third index
print("Element 0", Nested_tuple[0][0][0])
print("Element 1", Nested_tuple[0][0][1])
print("Element 1", Nested_tuple[0][0][2])
print("Element 1", Nested_tuple[0][0][3])
print("Element 1", Nested_tuple[0][0][4])
```

```
Element 0 H
Element 1 e
Element 1 l
Element 1 l
Element 1 o
```

Count Method

It allows us to count the number of times a value appears in the tuple

In [51]:
```
tuple_4 = (1,3,1,4,1,1,"a","a","d")
print(tuple_4.count(1))
```

```
4
```

In [2]:
```
tuple_9 = (1,4,2,3,1,5)
sorted(tuple_9)
```

Out[2]:
```
[1, 1, 2, 3, 4, 5]
```

Dictionary

A Dictionary is a collection of keys and values.A dictionary can be compared to a list, but nstead of being indexed numerically like a list, dictionaries have keys.

Dictionaries use keys and are unordered, they do not support indexing or slicing.These keys are the keys that are used to access values within a dictionary.

It represented by a curly bracket.

Example of a Dictionary:

In [65]:
```
Dict_1 = {"key1":"Value1","key2":"Value2","key3":"Value3","key4":"Value4","key5":"Valu
Dict_1
```

Out[65]:
```
{'key1': 'Value1',
 'key2': 'Value2',
 'key3': 'Value3',
 'key4': 'Value4',
 'key5': 'Value5',
 6: 6,
 7.1: 7}
```

Keys can be String, Integer ,float or tuple

In [66]:
```
#To access the value of a key:

Dict_1["key1"]
```

Out[66]:
```
'Value1'
```

In [67]:
```
#We can re-assign the value of a key
Dict_1["key2"]="chemistry"
Dict_1
```

Out[67]:
```
{'key1': 'Value1',
 'key2': 'chemistry',
 'key3': 'Value3',
 'key4': 'Value4',
 'key5': 'Value5',
 6: 6,
 7.1: 7}
```

Dictionary Methods

keys()

The key method helps you to retrieve the keys in a dictionary.

In [68]:
```
Dict_1.keys()
```

Out[68]:
```
dict_keys(['key1', 'key2', 'key3', 'key4', 'key5', 6, 7.1])
```

values()

To retrieve all the values in a dictionary, we use this method

In [69]:
```
Dict_1.values()
```

Out[69]:
```
dict_values(['Value1', 'chemistry', 'Value3', 'Value4', 'Value5', 6, 7])
```

del

To delete a value associated with a key, we use the del operator.

In [70]:
```
del Dict_1["key1"]
print(Dict_1)
```

```
{'key2': 'chemistry', 'key3': 'Value3', 'key4': 'Value4', 'key5': 'Value5', 6: 6, 7.
1: 7}
```

Pop

The dictionary also has a pop method like list. However, instead of taking in an index as a parameter, the pop method takes in a key.

In [71]:
```
print(Dict_1.pop("key3"))

Dict_1
```

```
Value3
```

Out[71]:
```
{'key2': 'chemistry', 'key4': 'Value4', 'key5': 'Value5', 6: 6, 7.1: 7}
```

items()

Returns a list of the items in the dictionary, in form oa tuple containing the (key,value)

In [72]:
```
print(Dict_1.items())
```

```
dict_items([('key2', 'chemistry'), ('key4', 'Value4'), ('key5', 'Value5'), (6, 6),
(7.1, 7)])
```

In [73]:
```
# To add entry to a dictionary
Dict_1["key8"]="Value8"
print(Dict_1)
```

```
{'key2': 'chemistry', 'key4': 'Value4', 'key5': 'Value5', 6: 6, 7.1: 7, 'key8': 'Valu
e8'}
```

In [74]:
```
# we can check if an element is in the dictionary

"key4" in Dict_1
```

Out[74]:
```
True
```

for loops and in operator

We can access the keys or values using iteration

In [75]:
```
# To access the keys in the dictionary
for key in Dict_1:
    print(key)
```

```
key2
key4
key5
6
7.1
key8
```

In [76]:
```
# To access the values in the dictionary
for key in Dict_1:
    print(Dict_1[key])
```

```
chemistry
Value4
Value5
6
7
Value8
```

In [77]:
```
# we combine display both the keys and values
for key in Dict_1:
    print(key, Dict_1[key])
```

```
key2 chemistry
key4 Value4
key5 Value5
6 6
7.1 7
key8 Value8
```

Sets

Sets are also similar to lists, but they differ from tuples and lists, because:

- Their values are unique i.e does not allow duplicate values

- They are unordered and unindexed
- Represented by curlu brackets {}
- Sets are mutable

In [52]:
```
# Lets create a set
Set_1 = {"English", "Mathematics", "Chemistry","Physic","English"}
Set_1
```

Out[52]:
```
{'Chemistry', 'English', 'Mathematics', 'Physic'}
```

In [53]:
```
type(Set_1)
```

Out[53]:
```
set
```

In [1]:
```
# we can convert a list to a set

Subject = ["English", "Mathematics", "Chemistry","Physic","Commerce","Economics"]
Subject_Set = set(Subject)
Subject_Set
```

Out[1]:
```
{'Chemistry', 'Commerce', 'Economics', 'English', 'Mathematics', 'Physic'}
```

Set Operations

add()

The add method allows us to add values to a set.

Consider a grocery store with the following products

In [55]:
```
products={"Rice","Beans","Garri","Semo","Wheat","Salt"}
products
```

Out[55]:
```
{'Beans', 'Garri', 'Rice', 'Salt', 'Semo', 'Wheat'}
```

In [56]:
```
# add new stock of eggs and sugar

products.add("Eggs")
products.add("Sugar")
products
```

Out[56]:
```
{'Beans', 'Eggs', 'Garri', 'Rice', 'Salt', 'Semo', 'Sugar', 'Wheat'}
```

remove()

In [57]:
```
# we are out of stock for Rice
products.remove("Rice")
products
```

Out[57]:
```
{'Beans', 'Eggs', 'Garri', 'Salt', 'Semo', 'Sugar', 'Wheat'}
```

It clears off the entire content of the set

In [58]:
```
products.clear()
products
```

Out[58]:
```
set()
```

Set Operations

intersection

Intersection is a term used to refer to the logical **and**. When comparing two or more sets,it is a set of values that both the original sets have in common.

In [59]:
```
set_1 = {"Rice","Beans","Garri","Semo"}
set_2 = {"Palm oil","Beans","Wheat","Salt"}
common_products_1 = set_1.intersection(set_2)
common_products_2 = set_1 & set_2
print(common_products_1)
print(common_products_2)
```

```
{'Beans'}
{'Beans'}
```

In [60]:
```
# You can find all the elements that are only contained in set_1 using the difference

print(set_1.difference(set_2))
print(set_1 -set_2)
```

```
{'Semo', 'Rice', 'Garri'}
{'Semo', 'Rice', 'Garri'}
```

union

When comparing two or more sets, it is the set containing the combined values of all the individual sets without duplicates(sets does not allow duplicate).

In [61]:
```
print(set_1.union(set_2))
print(set_1|set_2)
```

```
{'Salt', 'Garri', 'Wheat', 'Semo', 'Beans', 'Rice', 'Palm oil'}
{'Salt', 'Garri', 'Wheat', 'Semo', 'Beans', 'Rice', 'Palm oil'}
```

In [62]:
```
#Class Activity
#set_1=(set_1 & set_2)|(set_1 - Set_2), prove
```

issubset

A subset is a set of elements or values that appear in another set of equal or larger size.

issuperset

A superset is defined as a set of elements or values of which some or all appear in another set

isdisjoint

When comparing two sets and there are no elements in common, we say that the sets are known as disjoint sets.

In [63]:
```
set_1={1,2,3,4}
set_2={1,2,3,4,5,6,7,8}
set_3={9,10}

print(set_1.issubset(set_2))
print(set_2.issuperset(set_1))
print(set_1.isdisjoint(set_3))
```

```
True
True
True
```

In [64]:
```
set_1 = {1,2,3,4}
set_2 = {2,4}

intersection_set = set()
union_set = set()
#find the intersection

for elem in set_1:
    if elem in set_2:
        intersection_set.add(elem)
# find if is disjoint

if len(intersection_set) == 0:
    print("sets are disjoint")

# find the union
for elem in set_1 or set_2:
    union_set.add(elem)

print(union_set)
print(intersection_set)
```

```
{1, 2, 3, 4}
{2, 4}
```

In [ ]:
```
set_1 = {1,2,3,4}
set_2 = {2,4}

intersection_set = set()
union_set = set()
#find the intersection


# find if is disjoint



# find the union


print(union_set)
```

Numpy

Numpy is the short pronunciation for Numerical Python

Travis Oliphant designed NumPy in 2005. It is an open source project.

NumPy is a Python library used for working with arrays.

It also provides functions for working with linear algebra, fourier transforms, and matrices.

Importance of Numpy

We have lists in Python that act as arrays, however they are slow to process.

NumPy intends to deliver a 50-fold quicker array object than ordinary Python lists.

The array object in NumPy is named ndarray, and it comes with a slew of helper functions to make working with it a breeze.

Arrays are very frequently used in data science, where speed and resources are very important

Installing Numpy

install using conda install numpy for anaconda Navigator

pip install numpy for python IDE

In [1]:
```
# import numpy

import numpy as np
```

Let's create a numpy array

In [3]:
```
x = np.array([0,1,2,3,4,5])
x
```

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

In [8]:
```
np.append(x,3)
```

Out[8]:
```
array([0, 1, 2, 3, 4, 5, 3])
```

In [9]:
```
x
```

Out[9]:
```
array([0, 1, 2, 3, 4, 5])
```

We access each element of the array

```
0
```

We can also check the numpy version

In [4]:
```
print(np.__version__)
```

```
1.21.5
```

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

i - integer

b - boolean

u - unsigned integer

f - float

c - complex float

m - timedelta

M - datetime

O - object

S - string

U - unicode string

V - fixed chunk of memory for other type (void)

Type

We can check the type of the array and also check the data type of the aaray elements, note, a numpy array contains data of the same type

In [5]:
```
# check the type of the array
type(x)
```

Out[5]:
```
numpy.ndarray
```

In [6]:
```
# check the data type stored in your array

x.dtype
```

Out[6]:
```
dtype('int32')
```

Specify data type

While creating your arrange you can specify the data type

In [7]:
```
p= np.array([1,2,3,4,5], dtype="F")
p
```

Out[7]:
```
array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j], dtype=complex64)
```

In [8]:
```
p[0]
```

Out[8]:
```
(1+0j)
```

you can convert from int32 to int64

In [9]:
```
y = x.astype(np.int64)
```

In [10]:
```
y.dtype
```

Out[10]:
```
dtype('int64')
```

In [11]:
```
y.astype(np.int32)
```

Out[11]:
```
array([0, 1, 2, 3, 4, 5])
```

In [12]:
```
y
```

Out[12]:
```
array([0, 1, 2, 3, 4, 5], dtype=int64)
```

In [13]:
```
z = np.array([3.1,33.4,5.6,77.5])
```

In [14]:
```
type(z)
```

Out[14]:
```
numpy.ndarray
```

In [15]:
```
z.dtype
```

Out[15]:
```
dtype('float64')
```

We can convert from other data type to array

In [16]:
```
list_1 = [1,2,3,4,5,6]
a= np.array(list_1)
```

In [17]:
```
a
```

Out[17]:
```
array([1, 2, 3, 4, 5, 6])
```

In [18]:
```
type(a)
```

Out[18]:
```
numpy.ndarray
```

In [19]:
```
a.dtype
```

Out[19]:
```
dtype('int32')
```

In [20]:
```
tuple_1 = (1,2,3,4,5)
```

In [21]:
```
b = np.array(tuple_1)
```

In [22]:
```
b
```

Out[22]:
```
array([1, 2, 3, 4, 5])
```

In [23]:
```
b.dtype
```

Out[23]:
```
dtype('int32')
```

You can re-assign the value of an array

In [24]:
```
a[1]=50
a
```

Out[24]:
```
array([ 1, 50,  3,  4,  5,  6])
```

Slicing

Slicing can also pe perform in numpy, it is like coping a part of a variable

In [25]:
```
c = a[0:3]
c
```

Out[25]:
```
array([ 1, 50,  3])
```

You can also re-assign as follow

In [26]:
```
a[2:5] = 350,230,678
a
```

Out[26]:
```
array([  1,  50, 350, 230, 678,   6])
```

In [27]:
```
a[0]
```

Out[27]:
```
1
```

We can also slice by step

In [28]:
```
print(a[0:5:2])
```

```
[  1 350 678]
```

print out the odd numbers in the given array

d =([1,2,3,4,5,6,7,8,9,10])

We can assign value using the list

In [32]:
```
list_index = [0,1,2,3,4]
list_index
```

Out[32]:
```
[0, 1, 2, 3, 4]
```

In [33]:
```
# let's call out the values on the list_index

f = a[list_index]
f
```

Out[33]:
```
array([  1,  50, 350, 230, 678])
```

We re_assign based on the index

In [36]:
```
a[list_index] = 800,200,300,9,4
a
```

Out[36]:
```
array([800, 200, 300,   9,   4,   6])
```

Checking the size

Size is the number of elements in an array

In [ ]:
```
a.size
```

checking the array dimension or rank

In [ ]:
```
a.ndim # it is a 1 dimensional array (1d array)
```

checking the shape of the array

In [ ]:
```
a.shape
```

Find the size of the array below

g = np.array([33,23,55,66,12,55,89])

In [ ]:
```
y = np.array([3,4,5,6,7])
```

mean

In [ ]:
```
y_mean = y.mean()
y_mean
```

standard deviation

In [ ]:
```
y_std = y.std()
y_std
```

max and min

In [ ]:
```
y_max = y.max()
print(y_max)

y_min = y.min()
print(y_min)
```

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

In [ ]:
```
z=np.median(y)
z
```

In [ ]:

In [ ]:

Calculate the mean, median, min and max value of the data below

Operation

Addition

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

In [ ]:
```
x = np.array([34,55,66,77,88])
y = np.array([23,44,55,66,98])
```

In [ ]:
```
z = np.add(x, y)
z
```

Subtraction

In [ ]:
```
p = np.subtract(x ,y)
```

Multiplication

In [ ]:
```
r = np.multiply(x,y)
r
```

In [ ]:
```
## we can also multiply a vector by a scalar

q = np.multiply(x,2)
q
```

Division

In [ ]:
```
t = np.array([50,100,150])
t
```

In [ ]:
```
k = np.array([5,10,15])
k
```

In [ ]:
```
n = np.divide(t,k)
n
```

Dot Product

In [ ]:
```
x=np.array([5,5])
y = np.array([2,2])
```

In [ ]:
```
y
```

In [ ]:
```
np.dot(x,y)
```

In [ ]:
```
print(x[0])
print(x[1])
```

Adding constant to Numpy Array

In [ ]:
```
p = np.array([1,2,3,4,5,6])
p
```

In [ ]:
```
p + 5
```

Mathematical Function

In [ ]:
```
# pi

np.pi
```

In [ ]:
```
# create the numpy array in radians
```

In [ ]:
```
x
```

We can apply the sin to the array x

In [ ]:
```
d = np.sin(x)
d
```

Linespace

Linespace returns an evenly spaces numbers over a specified interval.

Syntax: numpy.linspace(start, stop, num = int value)

start : start of interval range

stop : end of interval range

num : Number of samples to generate.

In [ ]:
```
# numpy array within -5,5 with 9 elements

np.linspace(-5,5, num = 5)
```

We can use the function linspace to generate 100 evenly spaced samples from the interval 0 to 2π:

In [ ]:
```
r = np.linspace(0, 2*np.pi, num = 100)
r
```

In [ ]:
```
# let's apply the sine function

p = np.sin(r)
```

In [ ]:
```
p
```

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [ ]:
```
plt.plot(r,p)
```

Iterating

When we iterate over a 1-D array, each element is examined one by one.

We obtain **the array format** if we run the numpy array

In [ ]:
```
array_1 = np.array([4,5,6,7])
print(array_1)
```

However, if you want the result to be in the form of a list, you can use the for loop:

In [ ]:
```
for x in array_1:
    print(x)
```

Class Activity

h = [1,2,3] g = [4,5,6] Do the following:

1. Create a numpy array
2. perform the following operations: Addition Subtraction multiplication dot operation

Dimension of Array

so far we have being working on a 1-D array, there are other dimension or ranks of array.

Scalar

Refers to a single element

In [ ]:
```
scalar = np.array(65)

print(scalar)

scalar.ndim
```

1-D Array

Also known as a uni-dimensional array

In [ ]:
```
one_d_array =np.array([1,2,3,4,5,6])
print(one_d_array)
```

2-D Array

In [ ]:
```
# Create a list

b = [[11,12,13],[21,22,23],[31,32,33]]
b
```

In [ ]:
```
# Convert to Numpy

B = np.array(b)
B
```

The attribute ndim can be used to get the number of axes or dimensions, often known as the rank.

In [ ]:
```
# Numpy dimension or rank

B.ndim
```

The size or number of each dimension is returned by attribute shape as a tuple.

In [ ]:
```
# shape of numpy
B.shape
```

The attribute size specifies the total number of elements in the array.

In [ ]:
```
# Size
B.size
```

Access to Numpy Elements

Rectangular brackets can be used to access the array's various elements.

In [ ]:
```
# Access the element on the second row and third column

B[1, 2]
```

In [ ]:
```
# Access the element on the second row and third column

B[1][2]
```

In [ ]:
```
# Access the element on the first row and first column

B[0][0]
```

Slicing

In [ ]:
```
# Access the element on the first row and first and second columns

B[0][0:2]
```

In [ ]:
```
# Access the element on the second row and first to third columns
B[1][0:3]
```

In [ ]:
```
# Access the element on the first and second rows and third column

B[0:2, 2]
```

Arithmetic Operations

Addition

In [ ]:
```
# Create a numpy array X and Y
Y = np.array([[2,1],[1,2]])
```

In [ ]:
```
# Add the two arrays

p = X + Y
p
```

Multiplication

A numpy array multiplied by a scaler is the same as a matrix multiplied by a scaler. When we multiply the matrix Y by the scaler 2, every element in the matrix is multiplied by two, as illustrated below.

In [ ]:
```
Z = 2 *Y
Z
```

Multiplication of 2 array

An element-wise product, also known as a Hadamard product, is the result of multiplying two arrays. For example,matrices X and Y. Multiplying elements in the same location will produce a new matrix of the same size as matrix Y or X.

In [ ]:
```
print(X)
```

In [ ]:
```
print(Y)
```

In [ ]:
```
# Multiplying X by Y
Z = X * Y
Z
```

Numpy dot

Matrix multiplication can also be done with the numpy arrays A and B, as seen below.

In [ ]:
```
A = np.array([[0,1,1],[1,0,1]])
A
```

In [ ]:
```
B = np.array([[1,1],[1,1],[-1,1]])
B
```

To multiply the arrays together, we use the numpy function dot.

In [ ]:
```
C = np.dot(A,B)
C
```

In [ ]:
```
np.sin(C)
```

The numpy attribute T can be used to calculate the transposed matrix

In [ ]:
```
# Create a matrix D

D = np.array([[1,1],[2,2],[3,3]])
D
```

In [ ]:
```
# Get the transposed of C

D.T
```

Class Activity

X = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

Do the following:

1. Convert to a numpy array
2. size
3. Access the element on the second row and first and second columns.
4. Perform matrix multiplication with the numpy arrays X and Y

Given Y as [[0, 1], [1, 0], [1, 1], [-1, 0]]

Copy and View

The primary distinction between a copy and a view of an array is that the copy creates a new array, whereas the view simply displays the original array.

Any modifications made to the copy will have no effect on the original array, and any changes made to the original array will have no effect on the copy.

The view does not own the data, therefore any changes to the view will have an impact on the original array, and any changes to the original array will have an impact on the view.

In [ ]:
```
# Copy

g = np.array([1,2,3,4,5])
h = g.copy()
g[1]=6

print(g)
print(h)
```

In [ ]:
```
# View
```

```
g[1]=6

print(g)
print(k)
```

In [ ]:
```
k[3] = 15

print(g)
print(k)
```

Check the data owner

As previously stated, copies own the data, whereas views do not, but how can we verify this?

The base attribute in NumPy arrays returns None if the array owns the data.

The base attribute otherwise refers to the original object.

In [ ]:
```
array_1 = np.array([1,2,3,4,5])
x= array_1.copy()
y=array_1.view()

print(x.base)
print(y.base)
```

Subsetting using boolean

In [ ]:
```
y=np.array([1,2,3,4,5])
z = y>2
```

In [ ]:
```
print(z)
```

In [ ]:
```
print(y[z])
```

Generating Data in Numpy

Random Numbers

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

In [ ]:
```
height = np.round(np.random.normal(1.91, 0.19,6000),2)
weight = np.round(np.random.normal(65.55,15,6000),2)
np_team = np.column_stack((height,weight))
```

In [ ]:
```
np_team.shape
```

Mean

In [ ]:
```
height_mean = np.round(np.mean(np_team[:,0]),2)
height_mean
```

In [ ]:
```
height_median = np.round(np.median(np_team[:,0]),2)
height_median
```

In [ ]:
```
np_team_corr = np.corrcoef(np_team[:,0],np_team[:,1])
np_team_corr
```

In [ ]:
```
np.std(np_team[:,0])
```

Advantages of Data Structure

1- Data structure helps in efficient storage of data in the storage device.

2- Data structure usage provides convenience while retrieving the data from storage device.

3- Data structure provides effective and efficient processing of small as well as large amount of data.

4- Usage of proper data structure, can help programmer save lots of time or processing time while operations such as storage, retrieval or processing of data.

5- Manipulation of large amount of data can be carried out easily with the use of good data structure approach.

6- Most of the well organized data structures like Array, stack, queues, graph, tree, linked list has well built and pre-planned approach for operations like storage, addition, retrieval, manipulation, deletion, etc. While using them, programmer can be completely rely on these data structures.

Disadvantages of Data Structure

1- An application using data structure requires highly qualified professional resource to manage the operations related to data structure. For example, consider the array example that we explained above. If we need to get the elements of the above array in ascending or descending order then we must know sorting technique algorithms like insertion sort, bubble sort, etc. Or a good coder will also be able to design their own sorting algorithm. Similarly there may be other complex operations to be performed which may require dedicated professional.

2- Bigger the application or data structure involved in creating and maintaining application more is the requirement of man power. This can increase maintaining data structure costs. For example, we have several data structure available like array, queue, stack, linked list, tree, graph, etc. Bigger the application is more the amount of such data structures will be involved. Thus you may need several professional to create and maintain the application.

3- Designing your own data structure may involve complex algorithm and may require lot of time and testing to conclude they are full-proof and ready to use for organization purpose. This again will come with increase cost. And after development is complete it may be found that the new data structure designed is not that effective as was expected.