MODULE 3

Object Oriented Programming

Object Oriented Programming is a powerful method for creating programs. So far have discussed the procedural concepts in programming. Before going into Object Oriented Programming (OOPs), we need to understand the basic terminologies used in Object Oriented Programming. While **procedure oriented** programming gives importance to **functions, Object Oriented Programming focuses on objects. A class is the base of Object Oriented Programming.**

A class contains a collection of data (variables) and methods (functions) that act on those data. Class is considered as a **blueprint** for an object. For example consider the design of a house with details of windows, doors, roofs and floors. We can build the house based on the descriptions. This can be considered as a class while the object is the house itself. **An object is said to be an instance of a class** and the **process of creating a class is called instantiation**.

The basic concepts related to OOP are as follows:

1. **Classes**
2. **Objects**
3. **Encapsulation**
4. **Data Hiding**
5. **Inheritance**
6. **Polymorphism**

Advantages of Object Oriented Programming

Object Oriented programming has following advantages:

Simplicity: The objects in case of OOP are close to the **real world objects**, so the complexity of the program is reduced making the program structure very clear and simple. For example by looking at class Student, we can simply identify with the properties and behavior of makes the class Student very simple and easy to understand.

Modifiability: It is easy to make minor changes in the data representation or the procedures in an OOP program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

Extensibility and Maintainability: It is quite easy to add new features and extend a program in case of object oriented programming. It can be simply done by introducing a few new objects and modifying some existing ones. The original base class need not be modified at all. Even objects can be maintained separately, there by locating and fixing problems easier. For example if a new attribute of the class Student needs to be added, a new derived class of the class Student may be created and no other class in the class hierarchy need to be modified.

Re-usability: Objects can be reused in different programs. The class definitions can be reused in various applications. **Inheritance makes it possible to define subclasses of data objects that share some or all of the main class characteristics.** It forces a more thorough data analysis, reduces development time and ensures more accurate coding.

Security: Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of **data hiding provides greater system security and avoids unintended data corruption.**

1. CLASS DEFINITION

The class definition in Python begins with the **keyword class**. The first statement after class definition will be the **docstring** which gives a **brief description about the class**. The following shows the syntax for defining a class.

```
class ClassName:
        'Optional class documentation string'
        class_suite
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. There are also **special attributes** in it **that begins with double underscores (__)**. For example, **__doc__** gives us the **docstring** of that class. As soon as we define a class**, a class object is created with the same name**. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

Example Program

```
class Student:
        "common base class for all students"
        studentcount = 0
        def __init__(self, rollno, name, course):
                self.rollno=rollno
                self.name = name
                self.course = course
                Student.studentcount += 1
        def displayCount (self):
                print ( "Total Students=", Student.studentcount)
        def displayStudent (self):
                print ( "Roll Number:", self.rollno)
                print ("Name: ", self.name)
                print ("Course: ", self.course)
```

In the above program, we have created a **class called Student**. The variable studentcount is a **class variable whose value is shared among all instances of this class**. This can be **accessed** from inside the class or outside the class by giving **Student.studentcount**.

The first method **init() is a special method**, which is called **class constructor initialization method** that Python calls when we **create a new instance of this class**.

Class methods have only one specific difference from ordinary functions - they **must have an extra argument in the beginning of the parameter list**. This particular argument is **self** which **is used for referring to the instance.** But we need **not give any value for this parameter** when we **call the method**. Python provides it automatically. **self is not a reserved word** in Python but just a strong naming convention and it is always convenient to use conventional names as it makes the program more readable. So while defining our class methods, we must explicitly list self as the first argument for each method, including __init__

It also implies that if we have a method which takes no arguments, then we still have to define the method to have a self argument. Self is an **instance identifier** and is required so that the statements within the methods can have automatic access to the current instance attributes.

2 CREATING OBJECTS

An object is created by **calling the class name with the arguments defined** in the method. We can access a method by using the **dot operator along with the object**. Similarly **class name can be accessed by specifying the method name with the dot operator with the class.**

Example Program

```
# First student object is created
stud1=Student (10, "Jack", "MS")
# Second student object is created.
stud2=Student (20, "Jill", "BE")
#Displays the details of First Student
stud1.displayStudent ()
# Displays the details of Second Student.
stud2.displayStudent ()
print("Total Number of Students:", Student.studentcount)
```

when the above code is executed, it produces the following result.

```
Roll Number: 10
Name: Jack
Course: MS
```

```
Roll Number: 20
Name: Jill
Course: BE
Total Number of Students: 2
```

3. BUILT-IN ATTRIBUTE METHODS

There are built-in functions **to access the attributes**. We can **access the attribute value, check whether an attribute exists or not, modify an attribute value or delete an attribute**. The following gives the built-in attribute methods.

1. **getattr(obj, name[, default]):** This method is used to access the attribute of object.
2. **hasattr(obj, name)** : This attribute is used to check if an attribute exists or not.
3. **setattr(obj, name, value)** : This method is used to set an attribute. If attribute does not exist, then it would be created
4. **delattr(obj, name)** : This method is used to delete an attribute.

Example Program

```
class Student:
   "Common base class for all students"
   def __init__ (self, rollno, name, course):
      self.rollno=rollno
      self.name = name
      self.course = course
   def displayStudent (self):
      print("Roll Number:", self.rollno)
      print("Name: ", self.name)
      print("Course: ", self.course)
stud1=Student(10, "Jack", "MS")
stud1.displayStudent ()
print ("Value of name attribute:",getattr(stud1, 'name'))
print("Is stud1 has age attribute?:", hasattr(stud1,'age'))
# New attribute inserted
print("Add new attribute age to stud1:", setattr(stud1, 'age', 21))
stud1.displayStudent()
#Attribute age deleted
print("Delete age attribute from stud1:" , delattr(stud1,'age'))
stud1.displayStudent()
```

Output

```
Roll Number: 10
Name: Jack
Course: MS
Value of name attribute: Jack
Is stud1 has age attribute?: False
Add new attribute age to stud1: None
Roll Number: 10
Name: Jack
Course: MS
Age: 21
Delete age attribute from stud1: None
Roll Number: 10
Name: Jack
Course: MS
```

4. BUILT-IN CLASS ATTRIBUTES

Python contains several built-in class attributes. These attributes are accessed using **the dot operator**. The following gives the list of built-in class attributes.

1. **__dict__**: This attribute contains the **dictionary containing the class's namespace**.
2. **__doc__:** It describes the class or it contains the **class documentation string**. If undefined, it contains None.
3. **__name__:** This attribute contains **the class name**.
4. **__module__:** This contains the **module name** in which the class is defined. This attribute is"__main__" in interactive mode.
5. **__bases__:** This contains an empty **tuple containing the base classes**, in the order of their occurrence in the base class list. The following program shows the usage of built-in class attributes.

Example Program

```
class Student:
    "Common base class for all students"
    def __init__(self, rollno, name, course):
        self.rollno=rollno
        self.name = name
        self.course = course
    def displayStudent (self):
        print ("Roll Number:", self.rollno)
        print ("Name: ", self.name)
        print("Course: ", self.course)
stud1=Student (10, "Jack", "MS")
stud1.displayStudent ()
print ("Student.__doc__: ", Student.__doc__ )
print("Student.__name__: ", Student.__name__)
print("Student.__module__: ", Student.__module__)
print("Student.__bases__: ", Student.__bases__)
print("Student.__dict__: ", Student.__dict__)
```

Output

```
Roll Number: 10
Name: Jack
Course: MS
Student.__doc__ : Common base class for all students
Student.__name__: Student
Student. __module__: __main__
Student.__bases__:( )
Student.__dict__ : { 'displayStudent': <function displayStudent at 0x7efdcc047758>, '__module__': '_main_', '__doc__': 'Common base class for all students', '__init__': <function _init_ at 0x7efdcc0476e0>}
```

5 DESTRUCTORS IN PYTHON

Python **automatically deletes an object that is no longer in use**. This **automatic destroying** of objects is known as **garbage collection**. Python periodically performs the garbage collection to free the blocks off memory that are no longer in use. But a class can implement the special method **__del__**(), called a **destructor**, that is invoked when the instance is about to be destroyed. This method might be **used to clean up any non memory resources used by an instance**. The following program shows the destructors in Python.

Example Program

```
class Student:
    "Common base class for all students"
    def __init__ (self, rollno, name, course):
        self.rollno=rollno
        self.name = name
```

```
            self.course = course
        def displayStudent (self):
            print("Roll Number:", self.rollno)
            print("Name: ", self.name)
            print ("Course: ", self.course)
        def __del__ (self):
            class_name = self.__class__.__name__
            print (class_name, "destroyed")
    stud1=Student(10, "Jack", "MS")
    stud1.displayStudent()
    del stud1
```

Output

```
    Roll Number: 10
    Name: Jack
    Course: MS
    Student destroyed
```

6 ENCAPSULATION

Encapsulation is the most basic concept of OOP. **It is the combining of data and the functions associated with that data in a single unit**. In most of the languages including Python **this unit is called a class**. In simple terms we can say that **encapsulation is implemented through classes**. In fact the data members of a class can be accessed through its member functions only. It keeps **the data safe from any external interference and misuse**. **The only way to access the data is through the functions of the class**. In the example of the class Student, the class encapsulates the data (rollno, name, course) and the associated functions into a single independent unit.

7 DATA HIDING

We can hide data in Python. For this we need to **prefix double underscore for an attribute**. Data hiding can be defined as **the mechanism of hiding the data of a class from the outside world or to be precise, from other classes**. This is done to **protect the data from any accidental or intentional access**. In most of the object oriented programming languages, encapsulation is implemented through classes. In a class, **data may be made private or public**. **Private data or function** of a class **cannot be accessed from outside** the class while public data or functions can be accessed from anywhere. **So data hiding is achieved by making the members of the class private. Access to private members** is restricted and is only available to the **member functions of the same class**. However the public part of the object is accessible outside the class. Once the attributes are prefixed with the double underscore, it will not be visible outside the class. The following program shows an example for data hiding in Python

Example Program

```
class HidingDemo:
    "Program for Hiding Data"
    __num=0                          // private variable
    def numbercount (self):
        self.__num+=1
        print("Number Count=", self.__num)
number=HidingDemo ()                  //object created
number.numbercount()
print (number.__num)                  // error….
```

Output

```
Number Count= 1
Traceback (most recent call last):
        File "main.py", line 9, in <module>
        print number. __num
AttributeError: HidingDemo instance has no attribute '__num'
```

The output shows that numbercount is displayed once. When it is attempted to call the variable number.__num from outside the function, it resulted in an error.

8 INHERITANCE

The mechanism of **deriving a new class from an old class is known as inheritance**. The **old class** is known as **base class or super class or parent class**. The **new one** is called the **subclass or derived class or child class.** Inheritance allows subclasses to inherit all the variables and methods to their parent class. The advantage of inheritance is the **reusability of code**. Fig. 1 illustrates the **single inheritance**. The attributes and methods of parent class will be available in child class after inheritance.

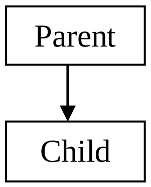


Fig 1 Single Inheritance

8.1 Deriving a Child Class

The following shows the **syntax for deriving a child class** in Python.

```
class SubclassName (ParentClass1[, ParentClass2, ...]):
        'Optional class documentation string'
        class_suite
```

The following program shows an example for single inheritance.

Example Program

```
class Student:
   "Common base class for all students"
   def getData (self, rollno, name, course):
      self.rollno=rollno
      self.name = name
      self.course = course
   def displaystudent (self):
      print("Roll Number:", self.rollno)
      print ("Name:", self.name)
      print ("Course:", self.course)

#Inheritance
class Test (Student):
   def getMarks (self, marks):
      self.marks=marks
   def displayMarks (self):
      print ("Total Marks:", self.marks)

r = int (input ("Enter Roll Number:"))
n = input ("Enter Name :")
c = input ("Enter Course Name :")
m = int (input ("Enter Marks:"))

#creating the object
print ("Result")

T=Test ()        #instance of child
T.getData (r,n,c)
T.getMarks (m)
T.displaystudent ()
T.displayMarks ()
```

Output

```
Enter Roll Number:20
Enter Name : Smith
Enter Course Name : MS
Enter Marks:200
Result
Roll Number: 20
Name: Smith
Course: MS
Total Marks: 200
```

8.2 Multilevel Inheritance

We have already discussed the mechanism of deriving a new class from one parent class. We can further **inherit the derived class to form a new child class**.

Inheritance which involves **more than one parent class but at different levels** is called **multilevel inheritance**. Figure 2.illustrates multilevel inheritance.

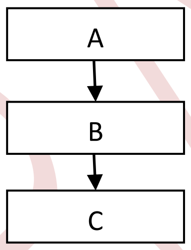


Fig. .2 Multilevel Inheritance

Example Program

```
class Student:
   "Common base class for all students"
   def getData(self, rollno, name, course):
      self.rollno=rollno
      self.name = name
      self.course = course
   def displayStudent (self):
      print("Roll Number:", self.rollno)
      print("Name:", self.name)
      print("Course:", self.course)

#Inheritance
```

```
class Test (Student):
    def getMarks (self, marks):
        self.marks=marks
    def displayMarks (self):
        print("Total Marks:", self.marks)

#Multilevel Inheritance
class Result (Test):
    def calculateGrade (self):
        if self.marks>480:
            self.grade="Distinction"
        elif self.marks>360:
            self.grade="First Class"
        elif self.marks>240:
            self.grade="Second Class"
        else: self.grade="Failed"
        print("Result:", self.grade)

#Main Program
r = int(input("Enter Roll Number:"))
n = input ("Enter Name:")
c = input("Enter Course Name:")
m = int(input("Enter Marks:"))

#creating the object
print("Results")
Res=Result()                #instance of child (Result class)
Res.getData(r, n,c)         #method of Student class
Res.getMarks (m)             #method of Test class
Res.displayStudent ()       #method of Student class
Res.displayMarks ()         #method of Test class
Res.calculateGrade ()       #method of Result class
```

Output

```
Enter Roll Number:12
Enter Name:Jones
Enter Course Name :MS
Enter Marks : 400
Results
Roll Number: 12
Name: Jones
Course: MS
Total Marks: 400
Result: First Class
```

8.3 Multiple Inheritance

It is possible to **inherit from more than one parent class**. Such type of inheritance is called **multiple inheritance**. In this case **all the attributes and methods of both the parent class will be available in the child class after inheritance**. Fig .3 illustrates multiple inheritance.

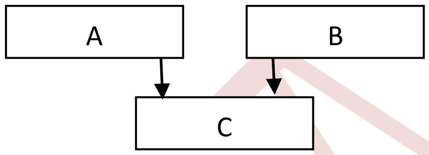


Fig.3 Multiple Inheritance

Here A and B are the parent classes and C is the child class. The attributes and methods of both classes A and B are now available in C after inheritance.

Example Program

```
class Student:
   "Common base class for all students"
   def getData (self, rollno, name, course, marks):
      self.rollno=rollno
      self.name = name
      self.course = course
      self.marks=marks
   def displayStudent (self):
      print ("Roll Number:", self.rollno)
      print ("Name:", self.name)
      print("Course:", self.course)
      print ("Total Marks:", self.marks)

class Sports:
    def getSportsMarks (self, spmarks):
        self.spmarks=spmarks
    def displaySportsMarks (self):
        print ("Sports Marks:", self.spmarks)

#Multiple Inheritance
class Result (Student, Sports):
    def calculateGrade (self):
        m=self.marks+self.spmarks
        if m>480:
            self.grade="Distinction"
        elif m>360:
            self.grade="First Class"
        elif m>240:
            self.grade="Second Class"
        else:
            self.grade=" Failed"
        print ("Grade:", self.grade)

#Main Program
r = int (input ("Enter Roll Number:"))
n = input ("Enter Name:")
c = input ( "Enter Course Name:")
m = int (input ("Enter Marks:"))
s = int (input ("Enter Sports marks:"))

#creating the object
print ("Results" )
Res=Result()                    #instance of child
Res.getData(r,n,c,m)
Res.getSportsMarks (s)
Res.displayStudent ()
Res.displaySportsMarks ()
Res.calculateGrade ()
```

Output

```
Enter Roll Number:10
Enter Name : Bob
Enter Course Name : MS
Enter Marks :190
Enter Sports marks:200
Result
Roll Number: 10
Name: Bob
Course: MS
Total Marks: 190
Sports Marks: 200
Result: First Class
```

8.4 Invoking the Base Class Constructor

In the above examples of inheritance, we have used member functions. Instead we can use constructors to pass value to the objects. In Python, the constructor of the base class can be invoked by **extending __init__().**
The class Student and class Teacher both have init__() method. The __init__ () method is defined in class Student and Teacher is extended in class School. In Python, **name of the base class** can also be used to **access the method of the base class which has been extended in derived class.**

Example Program

```
class Student ():
   def __init__(self, Id, name) :
      self. Id=Id
      self.name=name
   def showStudent (self):
      print ("Id number of the Student:", self. Id)
      print ("Name of the Student:", self.name)

class Teacher ():
   def __init__(self, tec_Id, tec_name, subject):
      self.tec_Id=tec_Id
      self.tec_name=tec_name
      self.subject=subject
   def showTeacher (self):
      print ("Id of the Teacher:", self. tec_Id)
```

```
            print ("Name of the Teacher:", self.tec_name)
            print ("Subject:", self.subject)

    class School (Student, Teacher):
        def __init__(self, ID, name, tec_Id, tec_name, subject, sch_Id):
            Student. __init__ (self, ID, name)
            Teacher. __init__(self, tec_Id, tec_name, subject)
            self.sch_Id= sch_Id
        def showSchool (self):
            print ("Id of the School:", self.sch_Id)

    #main program
    sc=School (10,"Johns", 100, "Jack" , "Math" ,80)
    sc. showStudent ()
    sc.showTeacher()
    sc.showSchool()
```

Output

```
    Id number of the Student: 10
    Name of the Student: Johns
    Id of the Teacher: 100
    Name of the Teacher: Jack
    Subject: Math
    Id of the School: 80
```

9 METHOD OVERRIDING

Polymorphism is an important characteristic of Object Oriented Programming language. In Obiect Oriented Programming, **polymorphism refers to a programming language's ability to process objects differently depending on their data type or class.** More specifically, it is the ability to **redefine methods for derived classes**. We can override methods in the parent class.

Method overriding is required when we want to define our own functionality in the child class. This is possible by defining a method in the child class that has the same name, same arguments and same return type as a method in the parent class. **When this method is called, the method defined in the child class is invoked and executed instead of the one in the parent class.** The following shows an example program for method overriding.

Example Program

```
class Parent:
    "Base class"
    def __init__(self, name):
        self.name = name
    def displayName (self):
        print ("Name: ", self.name)
class Child(Parent):
    def __init__ (self, name, address):
        Parent.__init__(self,name)
        self.name = name
        self.address= address
    def displayName (self):
        print ("Child Name: ", self.name)
        print ("Child Address:", self.address)
#Main Program
n=input("Enter Name :")
a=input ("Enter Address:")
obj=Child(n, a)                #instance of child
obj.displayName ()             #Calling child's method
```

Output

```
Enter Name :
aaa
Enter Address:
bbb
Child Name: aaa
Child Address: bbb
```

In the above program, even though there are two methods with the same name displayName() is available in child class, the method in the child class will be invoked.

The method in the child class has overridden the method in the parent class.

10 POLYMORPHISM

The word Polymorphism is formed from two words - **poly** and **morph** where poly means **many** and morph means **forms**. So **polymorphism is the ability to use an operator or function in various forms**. That is a single function or an operator behaves differently depending upon the data provided to them. **Polymorphism**

can be achieved in two ways: operator overloading and function overloading. Unlike in C++ and Java, **Python does not support function or method overloading**.

10.1 Operator Overloading

Operator overloading is one of the important features of object oriented programming. C++ and Python supports operator overloading, while Java does not support operator overloading. The mechanism of giving special meanings to an operator is known as operator overloading. For example the operator '+' is used for adding two numbers. In Python this operator can also be used for concatenating two strings. This '+' operator can be used adding member variables of two different class. The following example shows how '+' operator can be overloaded.

Example Program

```
class A:
    def __init__(self, a):
        self.a = a
      # adding two objects
    def __add__(self, o):
        return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("Python ")
ob4 = A("Programming")
print(ob1 + ob2)
print(ob3 + ob4)
```

Output

```
3
Python Programming
```

Arrays and Data Visualization

Arrays in Python

Arrays are used to store multiple values in one single variable:

Example: Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
print(cars)
```

Output:

```
["Ford", "Volvo", "BMW"]
```

An array is a special variable, which can hold more than one value at a time. If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to **an index number.**

Access the Elements of an Array

You refer to an array element by referring to the index number.

Example: Get the value of the first array item:

```
cars = ["Ford", "Volvo", "BMW"]
x = cars[0]
print(x)
```

Output:

```
Ford
```

Example: Modify the value of the first array item:

```
cars = ["Ford", "Volvo", "BMW"]
cars[0] = "Toyota"
print(cars)
```

Output:

```
["Toyota ", "Volvo", "BMW"]
```

The Length of an Array

Use the len() method to return the length of an array (the number of elements in an array).

Example: Return the number of elements in the cars array:

```
cars = ["Ford", "Volvo", "BMW"]
x = len(cars)
print(x)
```

Output:

```
3
```

Looping Array Elements

You can use the for in loop to loop through all the elements of an array.

Example: Print each item in the cars array:

```
cars = ["Ford", "Volvo", "BMW"]
for x in cars:
  print(x)
```

Output:

```
["Ford", "Volvo", "BMW"]
```

Adding Array Elements

You can use the **append()** method to add an element to an array.

Example : Add one more element to the cars array:

```
cars = ["Ford", "Volvo", "BMW"]
cars.append("Honda")
print(cars)
```

Output:

```
['Ford', 'Volvo', 'BMW', 'Honda']
```

Removing Array Elements

You can use the **pop()** method to remove an element from the array.

Example: Delete the second element of the cars array:

```
cars = ["Ford", "Volvo", "BMW"]
cars.pop(1)
print(cars)
```

Output:

```
['Ford', 'BMW']
```

You can also use the **remove()** method to remove an element from the array.

Example: Delete the element that has the value "Volvo":

```
cars = ["Ford", "Volvo", "BMW"]
cars.remove("Volvo")
print(cars)
```

Output:

```
['Ford', 'BMW']
```

Array Methods

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

Method	Description	Example	Output
append()	Adds an element at the end of the list		

clear()	Removes all the elements from the list	`fruits = ["apple", "banana", "cherry"]`
		`fruits.clear()`
		`print(fruits)`
			[ ]

copy()	Returns a copy of the list	`fruits = ["apple", "banana", "cherry"]`
		`x = fruits.copy()`
		`print(x)`
			['apple', 'banana', 'cherry']

count()	Returns the number of elements with the specified value	`fruits = ["cherry", "apple", "banana", "cherry"]`
		`x = fruits.count("cherry")`
		`print(x)`
			2

extend()	Add the elements of a list (or any iterable), to the end of the current list	`fruits = ['apple', 'banana', 'cherry']`
		`cars = ['Ford', 'BMW', 'Volvo']`
		`fruits.extend(cars)`
		`print(fruits)`
			['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']

index()	Returns the index of the first element with the specified value	fruits = ['apple', 'banana', 'cherry']
		x = fruits.index("cherry")
		print(x)
			2

insert()	Adds an element at the specified position	fruits = ['apple', 'banana', 'cherry']
		fruits.insert(1, "orange")
		print(fruits)
			['apple', 'orange', 'banana', 'cherry']

pop()	Removes the element at the specified position		
remove()	Removes the first item with the specified value		

reverse()	Reverses the order of the list	fruits = ['apple', 'banana', 'cherry']
		fruits.reverse()
		print(fruits)
			['cherry', 'banana', 'apple']

sort()	Sorts the list	cars = ['Ford', 'BMW', 'Volvo']
		cars.sort()
		print(cars)
			['BMW', 'Ford', 'Volvo']

Numpy Module

NumPy is a **Python library** that is the core library for **scientific computing in Python**. It contains a collection of tools and techniques that can be used to solve on a computer mathematical models of problems in Science and Engineering. One of these tools is a **high-performance multidimensional array object** that is a powerful data structure for **efficient computation of arrays and matrices**. To work with these arrays, there's a vast amount of high-level mathematical functions operate on these matrices and arrays.

NumPy's main object is the homogeneous multidimensional array. **It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers.** In NumPy dimensions are called axes.
For example, the coordinates of a point in 3D space [1, 2, 3] has one axis. That axis has 3 elements in it, so we say it has a length of 3.
NumPy's array class is called **ndarray**. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality.

Attributes of an ndarray object:
ndarray.ndim
The number of axes (dimensions) of the array.
ndarray.shape
The dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.
ndarray.size
The total number of elements of the array. This is equal to the product of the elements of shape.
ndarray.dtype

An object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

ndarray.itemsize

The size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

ndarray.data

The buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

Example

```
import numpy as np
a = np.array([2, 3, 4, 5, 6, 7, 8])
print(a.dtype)
print(a.ndim)
print(a.size)
print(a.shape)
print(a.itemsize)
print(a.data)
```

Output

```
int64
1
7
(7,)
8
<memory at 0x7f600c5cf400>
```

Array Creation

We can create an array from a regular Python list or tuple using the **array** function.

```
import numpy as np
a = np.array([1,2, 5, 6, 7])
print(a.dtype)
b=np.array([1.1, 2.1, 3.2])
print(b.dtype)
```

Output

```
int64
float64
```

A frequent error consists in calling array with multiple arguments, rather than providing a single sequence as an argument.

```
a = np.array(1, 2, 3, 4)        # WRONG
a = np.array([1, 2, 3, 4])      # RIGHT
```

array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
import numpy as np
b = np.array([(1.5, 2, 3), (4, 5, 6)])
print(b)
```

Output

```
[[1.5 2. 3. ]
  [4. 5. 6. ]]
```

The type of the array can also be explicitly specified at creation time:

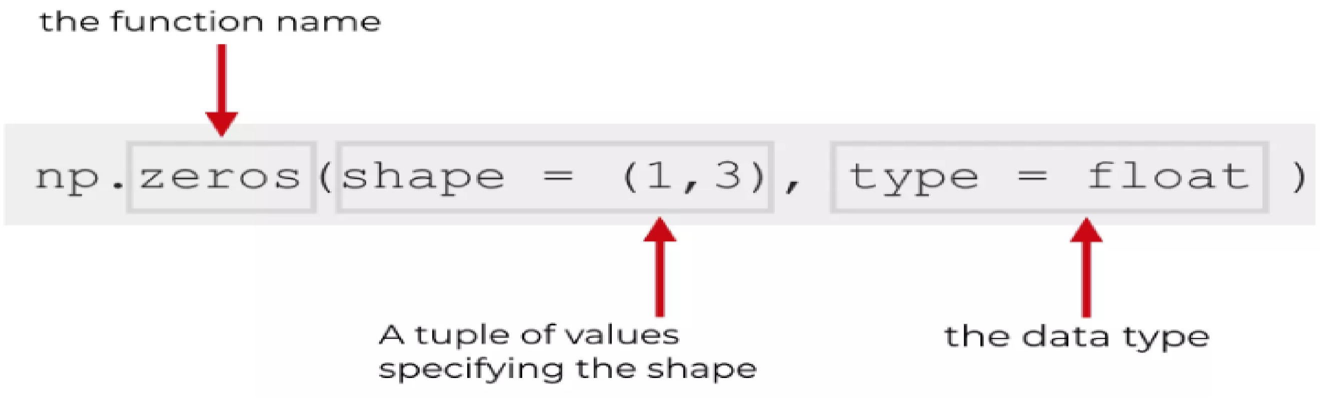
```
import numpy as np
c = np.array([[1, 2], [3, 4]], dtype=complex)
print(c)
```

Output

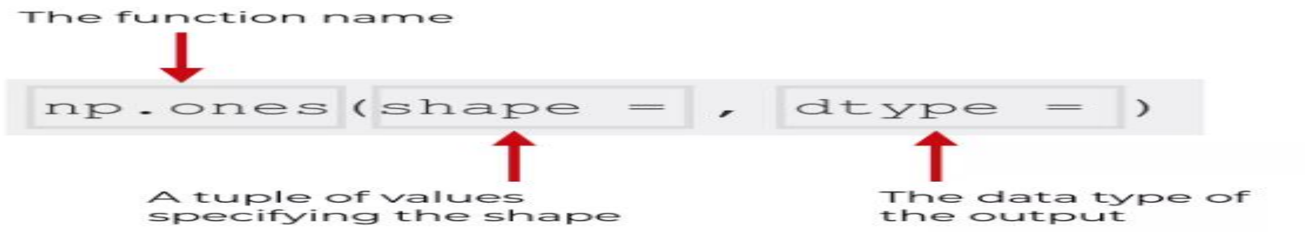
```
[[1.+0.j 2.+0.j]
  [3.+0.j 4.+0.j]]
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function **zeros** creates an array full of zeros, the function **ones** creates an array full of ones, and the function **empty** creates an array whose initial content is random and depends on the state of the memory. By default, the **dtype of the created array is float64**, but it can be specified via the key word argument dtype.



the function name
np.zeros(shape = (1,3), type = float)
A tuple of values specifying the shape
the data type



The function name
np.ones(shape = , dtype =)
A tuple of values specifying the shape
The data type of the output

Syntax of empty():

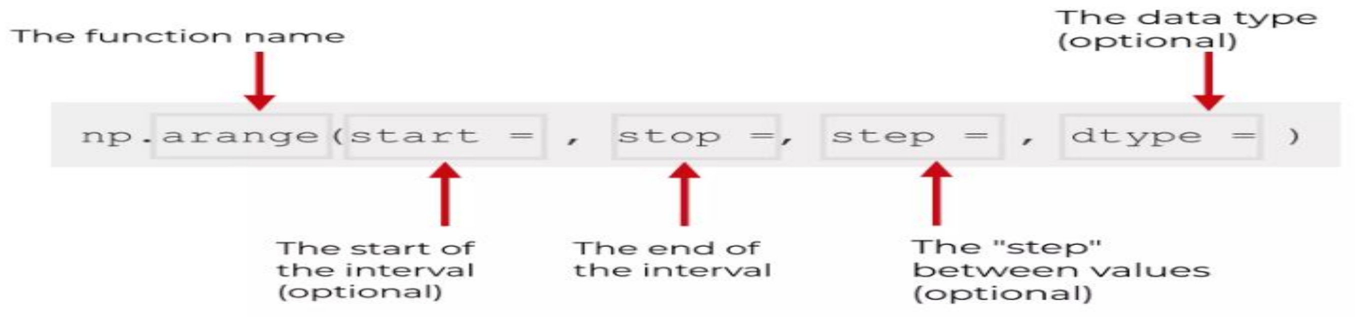
np.empty(shape=, dtype=)

```
import numpy as np
a=np.zeros((3, 4))
print(a)
print(a.dtype)
print()
b=np.ones((3, 4),dtype=np.int16)
print(b)
print(b.dtype)
print()
c=np.empty((3, 4),dtype=np.int16)
print(c)
```

Output:

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
float64

[[1 1 1 1]
[1 1 1 1]
[1 1 1 1]]
int16

 [[-13672     2828      32666      0]              # may vary
  [-13672     2828      32666      0]
   [ 0          0          0        0]]
```

To create sequences of numbers, NumPy provides the **arange** function which is analogous to the Python built-in range, but returns an array.

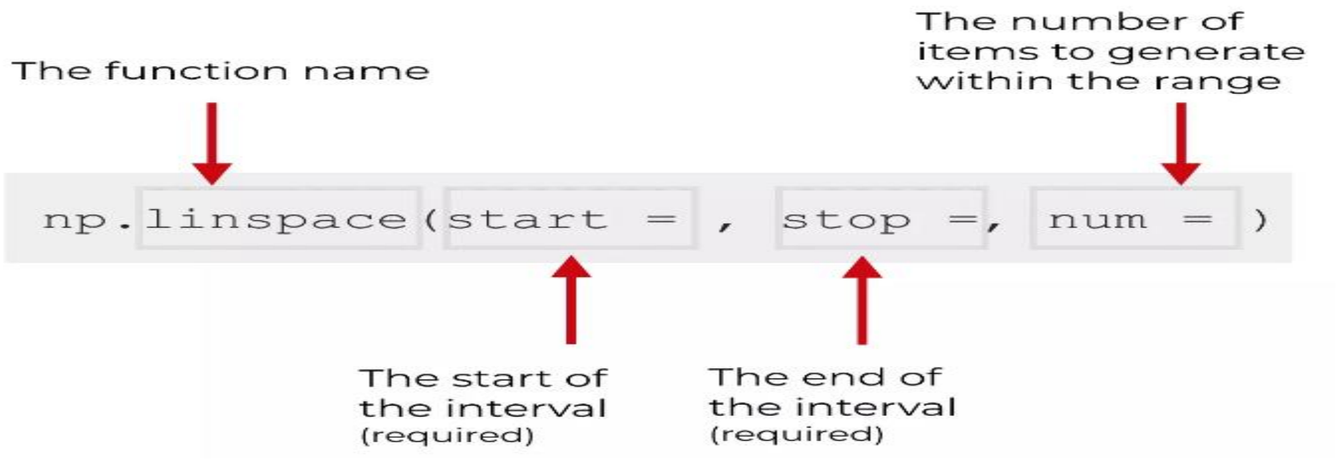

The function name
The data type (optional)
np.arange(start = , stop =, step = , dtype =)
The start of the interval (optional)
The end of the interval
The "step" between values (optional)

```
import numpy as np
a=np.arange(10, 30, 5)
print(a)
b=np.arange(0, 2, 0.3)                    # it accepts float arguments
print(b)
```

Output

```
[10 15 20 25]
[0. 0.3 0.6 0.9 1.2 1.5 1.8]
```

When **arange** is used with **floating point arguments,** it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function **linspace** that receives as an argument the **number of elements that we want**, instead of the step:


The function name
The number of items to generate within the range
np.linspace(start = , stop =, num =)
The start of the interval (required)
The end of the interval (required)

```
import numpy as np
print(np.linspace(0, 2, 9) )          # 9 numbers from 0 to 2
```

Output

```
[0. 0.25 0.5 0.75 1. 1.25 1.5 1.75 2. ]
```

Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists. One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

Example

```
import numpy as np
a = np.arange(6)                # 1d array
print("One Dimensional Array:\n ", a)
b = np.arange(12)
print("Array:\n ",b)
c = np.arange(12).reshape(4, 3)     # 2d array
print("Two Dimensional Array: \n",c)
d = np.arange(24).reshape(2, 3, 4)  # 3d array
print("Three Dimensional Array:\n ",d)
```

Output

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

reshape()

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.
Syntax:

array-name.reshape(shape)

where: shape: It is the tuple of integer values, according to which the elements are reshaped.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
print(arr)
newarr = arr.reshape(4, 3)
print(newarr)
```

Output

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

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
import numpy as np
print(np.arange(10000))
print()
print(np.arange(10000).reshape(100, 100))
```

Output

```
 [   0    1    2 ... 9997 9998 9999]

 [[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using **set_printoptions**.

```
np.set_printoptions(threshold=sys.maxsize)  # sys module should be imported
```

Example:

```
import numpy as np
import sys
np.set_printoptions(threshold=sys.maxsize)
print(np.arange(500).reshape(5, 100))
```

Output

```
[[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99]
[100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145
146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168
169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199]
[200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222
223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245
246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268
269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291
292 293 294 295 296 297 298 299]
[300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322
323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345
346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368
369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391
392 393 394 395 396 397 398 399]
[400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422
423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445
446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468
469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491
492 493 494 495 496 497 498 499]]
```

Basic Operations

Arithmetic operators on arrays apply **element wise**. A new array is created and filled with the result.
Unlike in many matrix languages, the **product operator * operates** element wise in NumPy arrays. The **matrix product** can be performed using **the @ operator** (in python >=3.5) or **the dot function** or method:

Example

```
import numpy as np
a = np.array([20, 30, 40, 50])
b=np.arange(4)
print(a)
print(b)
print("Sum : " , a+b)
print("Subtraction : ", a-b)
print("Product :", a*b)
print("Product :", a@b)
print("Product :", a.dot(b))
print("Square of array b : ", b**2)
print("Sine of a : ", 10 * np.sin(a))
print(a>30)
```

Output

```
[20 30 40 50]
```

```
[0 1 2 3]
Sum : [20 31 42 53]
Subtraction : [20 29 38 47]
Product : [ 0 30 80 150]
Product : 260
Product : 260
Square of array b : [0 1 4 9]
Sine of a : [ 9.12945251 -9.88031624 7.4511316 -2.62374854]
[False False True True]
```

Random in NumPy

Random number does NOT mean a different number every time. Random means something that cannot be predicted logically. NumPy offers the **random** module to work with random numbers. The random module's **rand()** method returns a random float between 0 and 1.

Example

```
from numpy import random
x = random.randint(100)
y = random.rand()
print(x)
print(y)
```

Output

```
52
0.7985497296519793
```

Generate Random Array

In NumPy we work with arrays, and you can use the two methods from the above examples to make **random arrays.**

Integers

The randint() method takes a size parameter where you can specify the shape of an array.

Example: Generate a 1-D array containing 5 random integers from 0 to 100:

```
from numpy import random
x=random.randint(100, size=(5))
print(x)
```

Output

```
[62 73 85 29 80]
```

Example: Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100:

```
from numpy import random
x = random.randint(100, size=(3, 5))
print(x)
```

Output

```
[[80 54 19 74 65]
 [26 60 69 34 25]
 [50 16 53 84 90]]
```

Floats

The rand() method also allows you to specify the shape of the array.

Example: Generate a 1-D array containing 5 random floats:

```
from numpy import random
x = random.rand(5)
print(x)
```

Output

```
[0.3956229    0.5312976   0.2147550     0.6804279     0.7865929]
```

Example: Generate a 2-D array with 3 rows, each row containing 5 random numbers:

```
from numpy import random
x = random.rand(3, 5)
print(x)
```

Output

```
[[0.03379952    0.78263517      0.9834899      0.47851523      0.02948659]

 [0.36284007    0.10740884      0.58485016    0.20708396      0.00969559]

 [0.88232193    0.86068608      0.75548749    0.61233486       0.06325663]]
```

Generate Random Number From Array

The **choice()** method allows you to generate a random value based on an array of values. The choice() method takes an array as a parameter and randomly returns one of the values.

Example: Return one of the values in an array:

```
from numpy import random
x = random.choice([3, 5, 7, 9])
print(x)
```

Output

```
5
```

The choice() method also allows you to return an array of values. Add a size parameter to specify the shape of the array.

Example: Generate a 2-D array that consists of the values in the array parameter (3, 5, 7, and 9):

```
from numpy import random
x = random.choice([3, 5, 7, 9], size=(3, 5))
print(x)
```

Output

```
[[5   9    7    5    9]
 [3    7    7    9   7]
 [3     7    9   9   5]]
```

Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

Example

```
import numpy as np
a=np.array([(1,2,3),(4,5,6)])
print(a)
b = np.array([(1.1,2.1,3.1),(4.1,5.1,6.1)])
print(b)
a *= 3
print(a)
b += a
```

```
print(b)
a += b                    # b is not automatically converted to integer type
print(a)
```

Output

```
[[1 2 3]
 [4 5 6]]
[[1.1 2.1 3.1]
 [4.1 5.1 6.1]]
[[ 3  6  9]
 [12 15 18]]
[[ 4.1  8.1 12.1]
 [16.1 20.1 24.1]]
Traceback (most recent call last):
  File "D:\Python\1 Files\MYPGM\num.py", line 10, in <module>
    a += b           # b is not automatically converted to integer type
numpy.core._exceptions.UFuncTypeError: Cannot cast ufunc 'add' output from dtype('float64') to
dtype('int32') with casting rule 'same_kind'
```

When operating with **arrays of different types**, the type of the resulting array corresponds to the more general or precise one (a behavior known as **upcasting**).

Many **unary operations**, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

Example

```
from numpy import random
a=random.randint(100, size=(3,5))
print(a)
print("Sum of elements :", a.sum())
print("Smallest element: " ,a.min())
print("Largest element : ", a.max())
```

Output

```
[[30 81 13 27 66]
 [11 84 63 11  9]
 [ 3 21 14  3 52]]
Sum of elements : 488
Smallest element:  3
Largest element :  84
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

Example

```
from numpy import random
b=random.randint(100, size=(3,4))
print(b)
print(" Sum of each column: ", b.sum(axis=0))                # sum of each column
print("Smallest of each row : ", b.min(axis=1))              # min of each row
print("Cumulative sum in each row: ", b.cumsum(axis=1) )     # cumulative sum along each row
```

Output

```
[[91 40 95 58]
 [76 31 14 48]
 [58 45 86 37]]
 Sum of each column:  [225 116 195 143]
Smallest of each row :  [40 14 37]
Cumulative sum in each row:
 [[ 91 131 226 284]
 [ 76 107 121 169]
 [ 58 103 189 226]]
```

[Cumulative sum means partially adding the elements in array.
E.g. The partial sum of [1, 2, 3, 4] would be [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10].]

Numpy ufunc | Universal functions

Universal functions in Numpy are simple mathematical functions. It is just a term that we gave to mathematical functions in the Numpy library. Numpy provides various universal functions that cover a wide variety of operations.

These functions include standard trigonometric functions, functions for arithmetic operations, handling complex numbers, statistical functions, etc.

Universal functions have various characteristics which are as follows-

- These functions operates on ndarray (N-dimensional array) i.e Numpy's array class.
- It performs fast element-wise array operations.
- Numpy, universal functions are objects those belongs to numpy.ufunc class.
- Python functions can also be created as a universal function using from pyfunc library function.
- Some ufuncs are called automatically when the corresponding arithmetic operator is used on arrays. For example when addition of two array is performed element-wise using '+' operator then np.add() is called internally.

Some of the basic universal functions in Numpy are-

Arithmetic functions

We can use arithmetic operators + - * / directly between NumPy arrays.

Addition

The add() function sums the content of two arrays, and return the results in a new array.

Subtraction

The subtract() function subtracts the values from one array with the values from another array, and return the results in a new array.

Multiplication

The multiply() function multiplies the values from one array with the values from another array, and return the results in a new array.

Division

The divide() function divides the values from one array with the values from another array, and return the results in a new array.

Power

The power() function rises the values from the first array to the power of the values of the second array, and return the results in a new array.

Remainder

Both the mod() and the remainder() functions return the remainder of the values in the first array corresponding to the values in the second array, and return the results in a new array.

You get the same result when using the remainder() function:

Quotient and Mod

The **divmod()** function return both the quotient and the the mod. The return value is two arrays, the first array contains the quotient and second array contains the mod.

Absolute Values

Both the absolute() and the abs() functions do the same absolute operation element-wise but we should use absolute() to avoid confusion with python's inbuilt math.abs()

Example:

```
import numpy as np
arr1 = np.array([10, 22, 32, 13, 14, 15])
arr2 = np.array([2, 3, 5, 2, 4, 3])
print(arr1)
print(arr2)
newarr = np.add(arr1, arr2)
print("Sum:\n",newarr)
newarr = np.subtract(arr1, arr2)
print("Subtraction:\n",newarr)
newarr = np.multiply(arr1, arr2)
print("Mul:\n",newarr)
newarr = np.divide(arr1, arr2)
print("Div;\n",newarr)
newarr = np.power(arr1, arr2)
print("Pow:\n",newarr)
newarr = np.mod(arr1, arr2)
```

```
print("Mod:\n",newarr)
newarr = np.remainder(arr1, arr2)
print("Reminder:\n",newarr)
newarr = np.divmod(arr1, arr2)
print("Divmod:\n",newarr)
newarr = np.absolute(arr1)
print("Absolute:\n",newarr)
```

Output

```
[10 22 32 13 14 15]
[2 3 5 2 4 3]
Sum:
 [12 25 37 15 18 18]
Subtraction:
 [ 8 19 27 11 10 12]
Mul:
 [ 20  66 160  26  56  45]
Div;
 [5.         7.33333333 6.4        6.5        3.5        5.        ]
Pow:
 [      100     10648 33554432       169     38416      3375]
Mod:
 [0 1 2 1 2 0]
Reminder:
 [0 1 2 1 2 0]
Divmod:
 (array([5, 7, 6, 6, 3, 5], dtype=int32), array([0, 1, 2, 1, 2, 0], dtype=int32))
Absolute:
 [10 22 32 13 14 15]
```

Trigonometric functions:

These functions work on **radians**, so angles need to be converted to radians by **multiplying by pi/180**. Only then we can call trigonometric functions. They take **an array as input** arguments. It includes functions like-

Function	**Description**
sin, cos, tan	Compute sine, cosine and tangent of angles

arcsin, arccos, arctan	Calculate inverse sine, cosine and tangent
Hypot	Calculate hypotenuse of given right triangle
sinh, cosh, tanh	Compute hyperbolic sine, cosine and tangent
arcsinh, arccosh, arctanh	Compute inverse hyperbolic sine, cosine and tangent
deg2rad	Convert degree into radians
rad2deg	Convert radians into degree

# Python code to demonstrate trigonometric function

```
import numpy as np
 # create an array of angles
angles = np.array([45, 60, 90, 180])
 # conversion of degree into radians  using deg2rad function
radians = np.deg2rad(angles)
 # sine of angles
print("Sine of angles in the array:")
sine_value = np.sin(radians)
print(np.sin(radians))
 # inverse sine of sine values
print("Inverse Sine of sine values:")
print(np.rad2deg(np.arcsin(sine_value)))
# hyperbolic sine of angles
print("Sine hyperbolic of angles in the array:")
sineh_value = np.sinh(radians)
print(np.sinh(radians))
# inverse sine hyperbolic
print("Inverse Sine hyperbolic:")
print(np.sin(sineh_value))
# hypot function demonstration
base = 4
height = 3
print("hypotenuse of right triangle is:")
print(np.hypot(base, height)
```

Output:

```
Sine of angles in the array:
[7.07106781e-01 8.66025404e-01 1.00000000e+00 1.22464680e-16]
```

```
Inverse Sine of sine values:
[4.5000000e+01 6.0000000e+01 9.0000000e+01 7.0167093e-15]
Sine hyperbolic of angles in the array:
[ 0.86867096  1.24936705  2.3012989  11.54873936]
Inverse Sine hyperbolic:
[ 0.76347126  0.94878485  0.74483916 -0.85086591]
hypotenuse of right triangle is:
5.0
```

Statistical functions:

These functions are used to calculate mean, median, variance, minimum of array elements.

It includes functions like-

Function	**Description**
amin, amax	returns minimum or maximum of an array or along an axis
ptp	returns range of values (maximum-minimum) of an array or along an axis
percentile(a, p, axis)	calculate pth percentile of array or along specified axis
median	compute median of data along specified axis
mean	compute mean of data along specified axis
std	compute standard deviation of data along specified axis
var	compute variance of data along specified axis
average	compute average of data along specified axis

# Python code demonstrate statistical function

```
import numpy as np
# construct a weight array
weight = np.array([50.7, 52.5, 50, 58, 55.63, 73.25])
# minimum and maximum
print("Minimum and maximum weight of the students: ")
print(np.amin(weight), np.amax(weight))
# range of weight i.e. max weight-min weight
print("Range of the weight of the students: ")
print(np.ptp(weight))
# percentile
print("Weight below which 70 % student fall: ")
print(np.percentile(weight, 70))
```

```
# mean
print("Mean weight of the students: ")
print(np.mean(weight))
# median
print("Median weight of the students: ")
print(np.median(weight))
# standard deviation
print("Standard deviation of weight of the students: ")
print(np.std(weight))
# variance
print("Variance of weight of the students: ")
print(np.var(weight))
# average
print("Average weight of the students: ")
print(np.average(weight))
```

Output:

```
Minimum and maximum weight of the students:
50.0 73.25
Range of the weight of the students:
23.25
Weight below which 70 % student fall:
56.815
Mean weight of the students:
56.68
Median weight of the students:
54.065
Standard deviation of weight of the students:
7.908487002370723
Variance of weight of the students:
62.54416666666666
Average weight of the students:
56.68
```

Bit-twiddling functions:

These functions accept integer values as input arguments and perform bitwise operations on binary representations of those integers. It include functions like-

Function	**Description**
bitwise_and	performs bitwise and operation on two array elements
bitwies_or	performs bitwise or operation on two array elements
bitwise_xor	performs bitwise xor operation on two array elements
Invert	performs bitwise inversion of an array elements
left_shift	shift the bits of elements to left
right_shift	shift the bits of elements to right

```
# Python code to demonstrate bitwise-function
        import numpy as np
        # construct an array of even and odd numbers
        even = np.array([0, 2, 4, 6, 8, 16, 32])
        odd = np.array([1, 3, 5, 7, 9, 17, 33])
        # bitwise_and
        print("bitwise_and of two arrays: ")
        print(np.bitwise_and(even, odd))
        # bitwise_or
        print("bitwise_or of two arrays: ")
        print(np.bitwise_or(even, odd))
        # bitwise_xor
        print("bitwise_xor of two arrays: ")
        print(np.bitwise_xor(even, odd))
        # invert or not
        print("inversion of even no. array: ")
        print(np.invert(even))
        # left_shift
        print("left_shift of even no. array: ")
        print(np.left_shift(even, 1))
        # right_shift
        print("right_shift of even no. array: ")
        print(np.right_shift(even, 1))
```

Output:

```
bitwise_and of two arrays:
[ 0  2  4  6  8 16 32]
bitwise_or of two arrays:
[ 1  3  5  7  9 17 33]
bitwise_xor of two arrays:
[1 1 1 1 1 1 1]
inversion of even no. array:
[ -1  -3  -5  -7  -9 -17 -33]
left_shift of even no. array:
[ 0  4  8 12 16 32 64]
right_shift of even no. array:
[ 0  1  2  3  4  8 16]
```

Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
import numpy as np
a = np.arange(10)
print(a)
print("Second position element ; ", a[2])
print("Elements from 2 to 5 position : ", a[2:5])
a[0:6:2] = 1000      # from start to position 6, exclusive, set every 2nd element to 1000
print(a)
print("Reverse of a ; ", a[::-1])      # reversed a
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print("Last element from 2nd dim: ", arr[1, -1])
print("Last element from 1 dim: ", arr[0, -1])
print("Second Last element from 1 dim: ", arr[0, -2])
```

Output

```
[0 1 2 3 4 5 6 7 8 9]
Second position element ;  2
Elements from 2 to 5 position :  [2 3 4]
[1000    1 1000    3 1000    5    6    7    8    9]
Reverse of a ;  [   9    8    7    6    5 1000    3 1000    1 1000]
Last element from 2nd dim:  10
Last element from 1 dim:  5
Second Last element from 1 dim:  4
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
import numpy as np
```

```
b=np.array([[1,2,3],[4,5,6]])
print(b)
print("Last row: ",b[-1] )                    # the last row. Equivalent to b[-1, :]
print("Last row: ",b[-1,:])
```

Output

```
[[1 2 3]
 [4 5 6]]
Last row:  [4 5 6]
Last row:  [4 5 6]
```

Iterating over multidimensional arrays is done with respect to the first axis:

```
import numpy as np
b=np.array([[1,2,3],[4,5,6]])
print("Elements are: ")
for row in b:
    print(row)
```

Output

```
Elements are:
[1 2 3]
[4 5 6]
```

However, if one wants to perform an operation on each element in the array, one can use the flat attribute which is an iterator over all the elements of the array:

```
import numpy as np
b=np.array([[1,2,3],[4,5,6]])
print(b)
print("Elements +10 are: ")
for row in b.flat:
    print(row+10)
```

Output

```
[[1 2 3]
 [4 5 6]]
Elements +10 are:
11
12
13
14
15
16
```

Example: **flat**

```
import numpy as np
b=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(b)
print(b.flat[0:10])
```

Output

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

Splitting one array into several smaller ones

Using **hsplit**, you can split an array along its horizontal axis (column-wise). hsplit is equivalent to split with axis=1, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur. **vsplit()** function split an array into multiple sub-arrays vertically (row-wise). vsplit is equivalent to split with axis=0 (default).

Example

```
import numpy as np
b= np.arange(16).reshape(4, 4)
print(b)
#split array b into 3 smaller arrays
print(np.hsplit(b,2))
```

Output

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]]),
 array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])]
```

Example

```
import numpy as np
x = np.arange(16).reshape(4, 4)
print(x)
print(np.vsplit(x, 2))
```

Output

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])]
```

Shape Manipulation

Changing the shape of an array

An array has a shape given by the number of elements along each axis:

```
import numpy as np
```

```
a = np.array([[1,2,3],[4,5,6]])
print("Elements are : \n", a)
print("Shape:", a.shape)
print("Flat: ",a.ravel())
print("Shape changed :\n", a.reshape(3,2))
print("Transpose : \n", a.T)
```

Output

```
Elements are :
 [[1 2 3]
 [4 5 6]]
Shape: (2, 3)
Flat:  [1 2 3 4 5 6]
Shape changed :
 [[1 2]
 [3 4]
 [5 6]]
Transpose :
 [[1 4]
 [2 5]
 [3 6]]
```

ravel() which is used to change a 2-dimensional array or a multi-dimensional array into a contiguous flattened array.

We can also use the ndarray.**resize** method modifies the shape of the array.

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print("Elements are : \n", a)
print("Shape:", a.shape)
print(a.reshape(6,1))
a.resize(2,3)
print(a)
```

Output

```
Elements are :
 [[1 2 3]
 [4 5 6]]
Shape: (2, 3)
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
[[1 2 3]
 [4 5 6]]
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are **automatically calculated**:

```
import numpy as np
a = np.array([[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]])
print("Elements are : \n", a)
print("Shape:", a.shape)
print(a.reshape(2,6))
print(a.reshape(3,-1))
```

Output

```
Elements are :
 [[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]
  [10 11 12]]]
Shape: (1, 4, 3)
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Copies and Views

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. There are three cases:

No Copy at All

Normal assignments do not make the copy of an array object. Instead, it uses the exact same id of the original array to access it. Further, any changes in either get reflected in the other.

```
import numpy as np
arr = np.array([2, 4, 6, 8, 10])
print(arr)
nc = arr              # assigning arr to nc
# both arr and nc have same id
print("id of arr", id(arr))
print("id of nc", id(nc))
# updating nc
nc[0]= 12
# printing the values
print("original array- ", arr)
print("assigned array- ", nc)
```

Output

```
[ 2  4  6  8 10]
id of arr 140606645307872
id of nc 140606645307872
original array-  [12  4  6  8 10]
assigned array-  [12  4  6  8 10]
```

View or Shallow Copy

This is also known as **Shallow Copy**. The view is just a view of the original array and view does not own the data. When we make changes to the view it affects the original array, and when changes are made to the original array it affects the view. Both have different id.

```
import numpy as np
arr = np.array([2, 4, 6, 8, 10])
# creating view
v = arr.view()
# both arr and v have different id
print("id of arr", id(arr))
print("id of v", id(v))
# changing original array will effect view
arr[0] = 12
# printing array and view
print("original array- ", arr)
print("view- ", v)
```

Output

```
id of arr 139932570884496
id of v 139932570884576
original array-  [12  4  6  8 10]
view-  [12  4  6  8 10]
```

Copy

Copy: This is also known as **Deep Copy**. The copy is completely a new array and copy owns the data. When we make changes to the copy it does not affect the original array, and when changes are made to the original array it does not affect the copy. Both have different id.

```
import numpy as np
arr = np.array([2, 4, 6, 8, 10])
# creating copy of array
c = arr.copy()
# both arr and c have different id
print("id of arr", id(arr))
print("id of c", id(c))
# changing original array this will not effect copy
arr[0] = 12
# printing array and copy
print("original array- ", arr)
print("copy- ", c)
```

Output

```
id of arr 140212025436640
id of c 140212025436720
original array-  [12  4  6  8 10]
copy-  [ 2  4  6  8 10]
```

Data visualization

A graph or chart is used to present numerical data in visual form. A graph is one of the easiest ways to compare numbers. They should be used to make facts clearer and more understandable. Results of mathematical computations are often presented in graphical format. In this section, we will explore the Python modules used for generating two and three dimensional graphs of varioustypes.

The Matplotlib Module

Matplotlib is a python package that produces quality figures. It also provides many functions for matrix manipulation. You can generate plots, histograms, bar charts etc, with just a few lines of code and have full control of line styles, font properties, axes properties, etc.

The **data points** to the plotting functions are supplied as Python **lists or Numpy arrays**.

If you import matplotlib, the plotting functions from the submodules pyplot will be available as local functions.

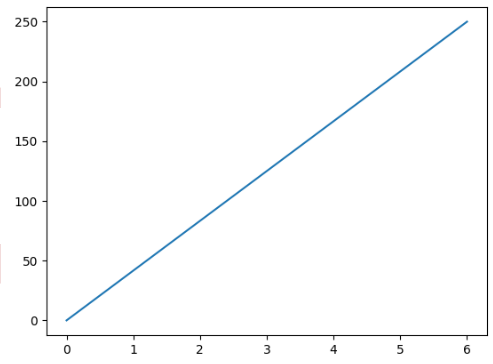
Pyplot

Most of the Matplotlib utilities lies under the **pyplot** submodule, and are usually imported under the plt alias:

import matplotlib.pyplot as plt

Now the Pyplot package can be referred to as plt.

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([0, 6])
ypoints = np.array([0, 250])
plt.plot(xpoints, ypoints)
plt.show()
```



By default, the **color is blue** and the **line style is continuous**. This can be changed by an optional argument after the coordinate data, which is the format string that indicates the color and line type of the plot. The default format string is **'b-'** (blue, continuous line).

Plot

Plotting x and y points

The plot() function is used to draw points (markers) in a diagram. By default, the plot() function draws a line from point to point. The function takes parameters for specifying points in the diagram.

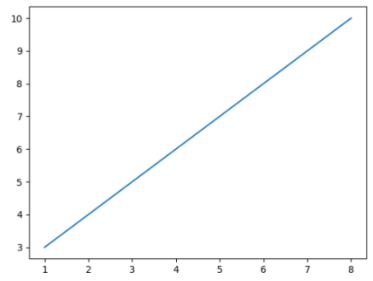
Parameter 1 is an array containing the points on the **x-axis**.

Parameter 2 is an array containing the points on the **y-axis**.

If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

Draw a line in a diagram from position (1, 3) to position (8, 10):

```
import matplotlib.pyplot as plt
import numpy as np
xpoints=np.array([1, 8])
ypoints=np.array([3, 10])
plt.plot(xpoints,ypoints)
plt.show()
```

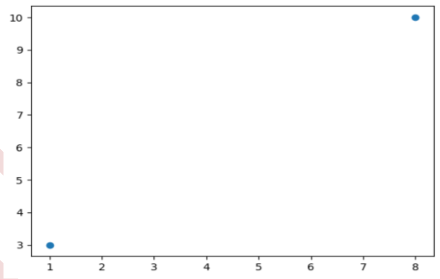


Plotting Without Line

To plot only the markers, you can use shortcut string notation parameter 'o', which means 'rings'.

Draw two points in the diagram, one at position (1, 3) and one in position (8, 10):

```
import matplotlib.pyplot as plt
import numpy as np
xpoints=np.array([1, 8])
ypoints=np.array([3, 10])
plt.plot(xpoints,ypoints, 'o')
plt.show()
```

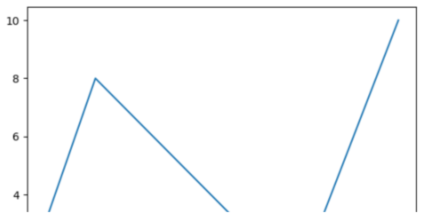


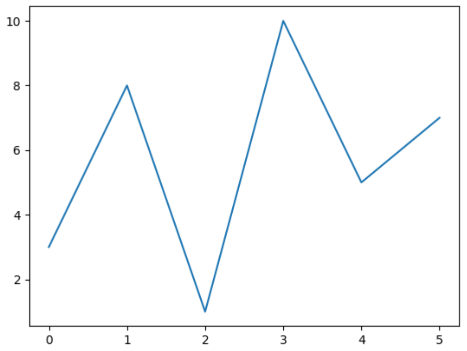
Multiple Points

You can plot as many points as you like, just make sure you have the same number of points in both axis.

Draw a line in a diagram from position (1, 3) to (2, 8) then to (6, 1) and finally to position (8, 10):

```
import matplotlib.pyplot as plt
import numpy as np
xpoints=np.array([1, 2, 6, 8])
ypoints=np.array([3, 8, 1, 10])
plt.plot(xpoints,ypoints)
plt.show()
```





Default X-Points

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points).

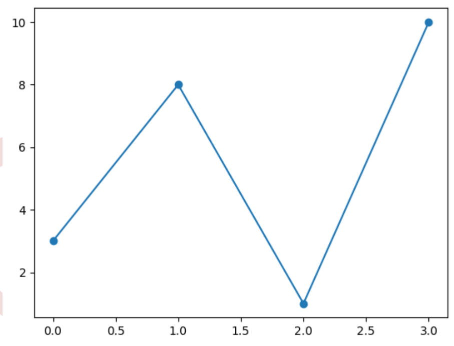
So, if we take the same example as above, and leave out the x-points, the diagram will look like this:

Plotting without x-points:

```
import matplotlib.pyplot as plt
import numpy as np
ypoints=np.array([3, 8, 1, 10, 5, 7])
plt.plot(ypoints)
plt.show()
```

Markers

You can use the keyword argument `marker` to **emphasize each point** with a specified marker:



Mark each point with a circle:

```
import matplotlib.pyplot as plt
import numpy as np
ypoints=np.array([3, 8, 1, 10])
plt.plot(ypoints,marker= 'o')
plt.show()
```

Marker Reference

You can choose any of these markers:

Marker	Description	Marker	Description
'o'	Circle	'P'	Plus (filled)
'*'	Star	's'	Square
'.'	Point	'D'	Diamond
','	Pixel	'd'	Diamond (thin)
'x'	X	'p'	Pentagon
'X'	X (filled)	'H'	Hexagon
'+'	Plus		

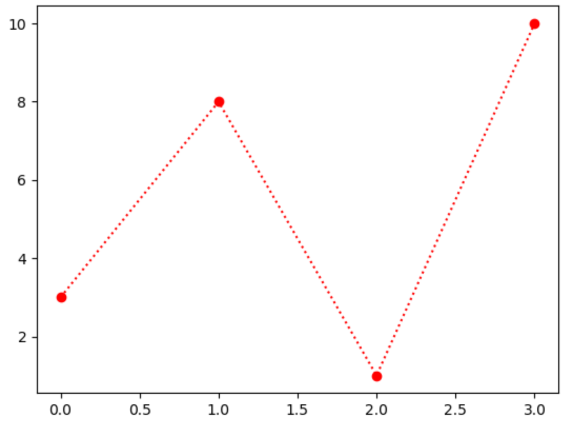
Format Strings `fmt`

You can use also use the *shortcut string notation* parameter to specify the marker. This parameter is also called fmt, and is written with this syntax:

marker|line|color

Mark each point with a circle:

```
import matplotlib.pyplot as plt
import numpy as np
ypoints=np.array([3, 8, 1, 10])
plt.plot(ypoints, 'o:r')
plt.show()
```



The line value can be one of the following:

Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

The short color value can be one of the following:

Color Reference

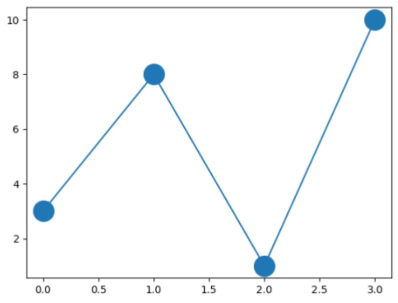
Color Syntax	Description	Color Syntax	Description
'r'	Red	'm'	Magenta
'g'	Green	'y'	Yellow
'b'	Blue	'k'	Black
'c'	Cyan	'w'	White

Marker Size

You can use the keyword argument `markersize` or the shorter version, `ms` to set the size of the markers:

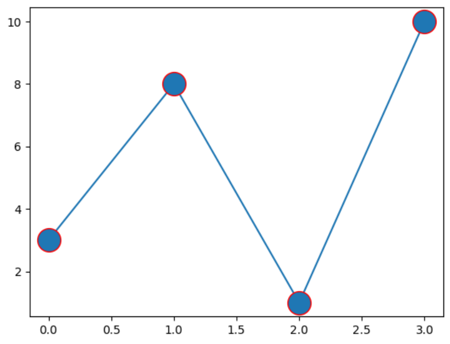
Set the size of the markers to 20:

```
import matplotlib.pyplot as plt
import numpy as np
ypoints=np.array([3, 8, 1, 10])
plt.plot(ypoints,marker= 'o',ms= 20)
plt.show()
```



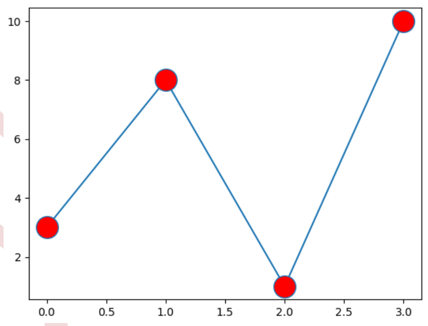
Marker Color

You can use the keyword argument `markeredgecolor` or the shorter `mec` to set the color of the *edge* of the markers: Set the EDGE color to red:



```
import matplotlib.pyplot as plt
import numpy as np
ypoints=np.array([3, 8, 1, 10])
plt.plot(ypoints,marker= 'o',ms= 20,mec= 'r')
plt.show()
```

You can use the keyword argument `markerfacecolor` or the shorter `mfc` to set the color inside the edge of the markers: Set the FACE color to red:

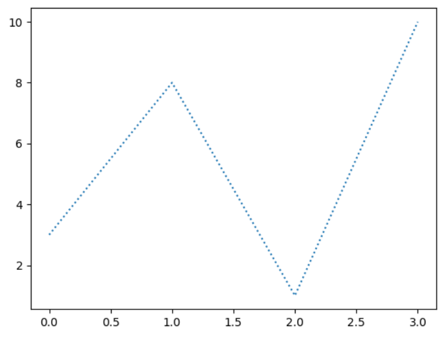


```
import matplotlib.pyplot as plt
import numpy as np
ypoints=np.array([3, 8, 1, 10])
plt.plot(ypoints,marker= 'o',ms= 20,mfc= 'r')
plt.show()
```

Linestyle

You can use the keyword argument `linestyle,` or shorter `ls,` to change the style of the plotted line:

Use a dotted line:



```
import matplotlib.pyplot as plt
import numpy as np
ypoints=np.array([3, 8, 1, 10])
plt.plot(ypoints,linestyle= 'dotted')
plt.show()
```

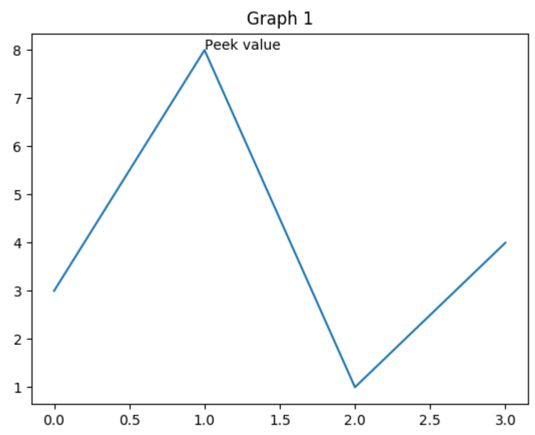
Shorter Syntax

The line style can be written in a shorter syntax:

`linestyle` can be written as `ls`

`dotted` can be written as `:`

`dashed` can be written as `–`



Working with text

The **text()** command can be used to add text in an arbitrary location.

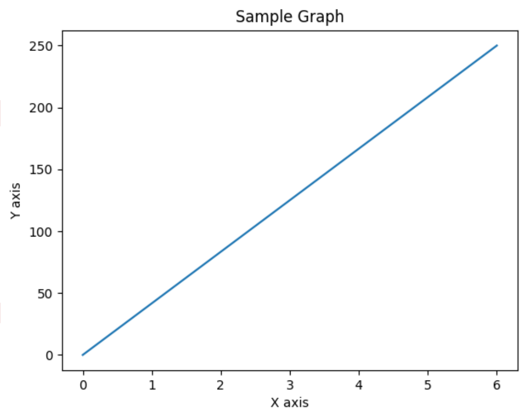
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 4])
plt.text(1,8,'Peek value')
plt.plot(x,y)
plt.title('Graph 1')
plt.show()
```

Labels and Title

With Pyplot, you can use the `xlabel()` and `ylabel()` functions to set a label for the x- and y-axis.

With Pyplot, you can use the `title()` function to set a title for the plot.

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([0, 6])
ypoints = np.array([0, 250])
plt.plot(xpoints, ypoints)
plt.title('Sample Graph')
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.show()
```



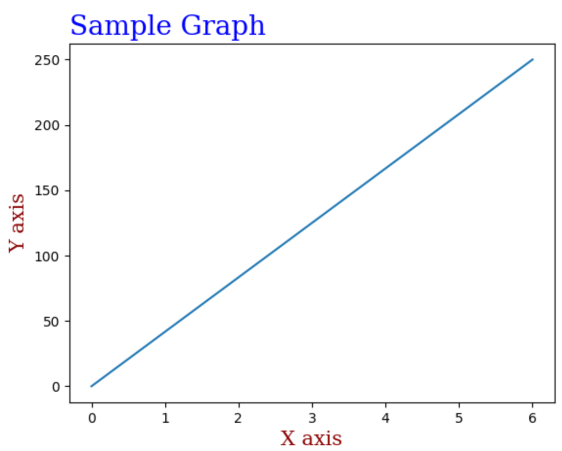
Set Font Properties for Title and Labels

You can use the `fontdict` parameter in `xlabel()`, `ylabel()`, and `title()` to set font properties for the title and labels.

You can use the `loc` parameter in `title()` to position the title.

Legal values are: **'left', 'right', and 'center'**. Default value is 'center'.

```
import matplotlib.pyplot as plt
import numpy as np
font1 = {'family':'serif','color':'blue','size':20}
```



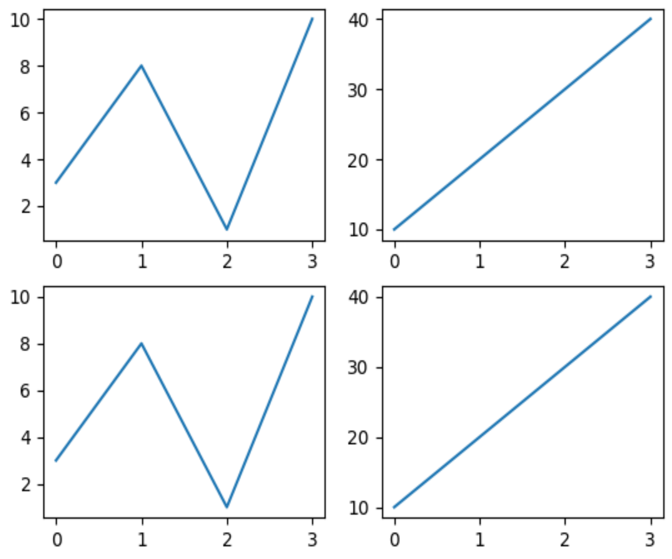
```
font2 = {'family':'serif','color':'darkred','size':15}
xpoints = np.array([0, 6])
ypoints = np.array([0, 250])
plt.plot(xpoints, ypoints)
plt.title('Sample Graph',fontdict = font1, loc = 'left')
plt.xlabel('X axis', fontdict = font2)
plt.ylabel('Y axis', fontdict = font2)
plt.show()
```

Subplots

With the `subplot()` function you can draw multiple plots in one figure. The **subplot()** function takes **three arguments** that describes the layout of the figure. The layout is organized in rows and columns, which are represented by the *first* and *second* argument. The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)          #the figure has 1 row, 2 columns, and this plot is the first plot.
plt.subplot(1, 2, 2)          #the figure has 1 row, 2 columns, and this plot is the second plot.
```

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 2, 1)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 2, 2)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 2, 3)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 2, 4)
plt.plot(x,y)
```

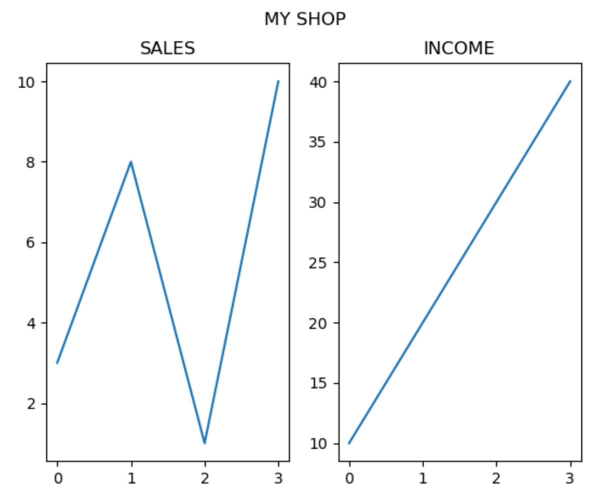


plt.show()

You can add a title to each plot with the `title()` function. You can add a title to the entire figure with the `suptitle()` function.

Add a title and suptitle for the figure:

```
import matplotlib.pyplot as plt
import numpy as np
#plot1:
x= np.array([0, 1, 2, 3])
y=np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")
#plot2:
x=np.array([0, 1, 2, 3])
y=np.array([10, 20, 30, 40])
plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")
plt.suptitle("MYSHOP")
plt.show()
```

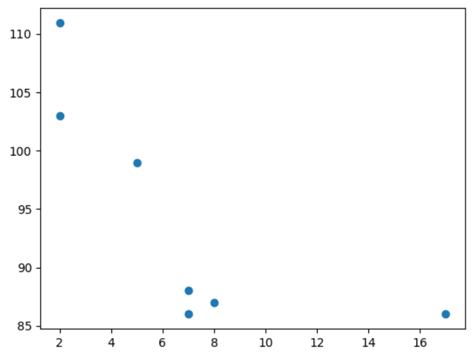


Scatter

With Pyplot, you can use the `scatter()` function to draw a scatter plot. The `scatter()` function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

A simple scatter plot:

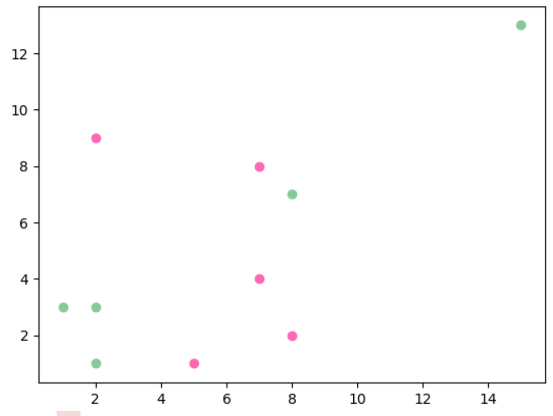
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([5,7,8,7,2,17,2])
y = np.array([99,86,87,88,111,86,103])
plt.scatter(x, y)
plt.show()
```



Colors

You can set your own color for each scatter plot with the `color` or the `c` argument:

Set your own color of the markers:

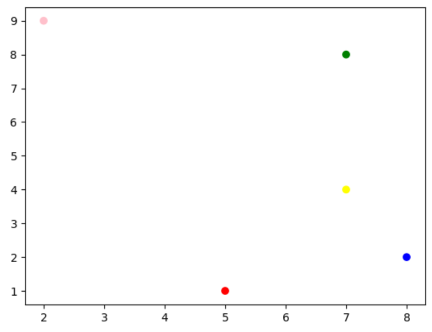


```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2])
y = np.array([1,8,2,4,9])
plt.scatter(x, y, color = 'hotpink')

x = np.array([2,2,8,1,15])
y = np.array([3,1,7,3,13])
plt.scatter(x, y, color = '#88c999')
plt.show()
```

Color Each Dot

You can even set a specific color for each dot by using an array of colors as value for the `c` argument:

Set your own color of the markers:

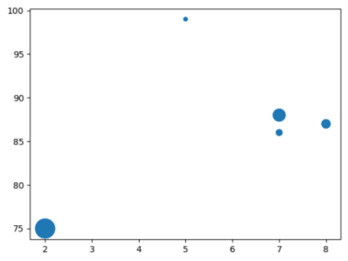


```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([5,7,8,7,2])
y = np.array([1,8,2,4,9])
colors=np.array(["red","green","blue","yellow","pink"])
plt.scatter(x, y, c=colors)
plt.show()
```

Size

You can change the **size** of the dots with the `s` argument.

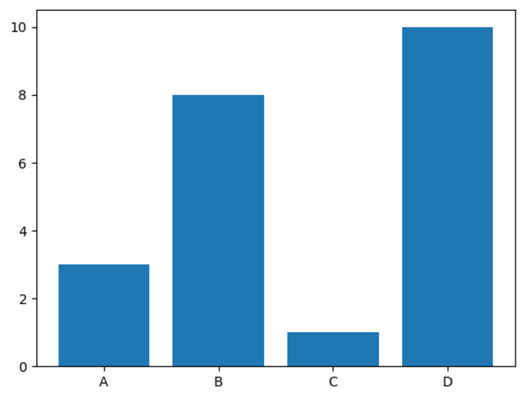
Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:



```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([5,7,8,7,2])
y = np.array([99,86,87,88,75])
sizes = np.array([20,50,100,200,500])
plt.scatter(x, y, s=sizes)
plt.show()
```

Bars

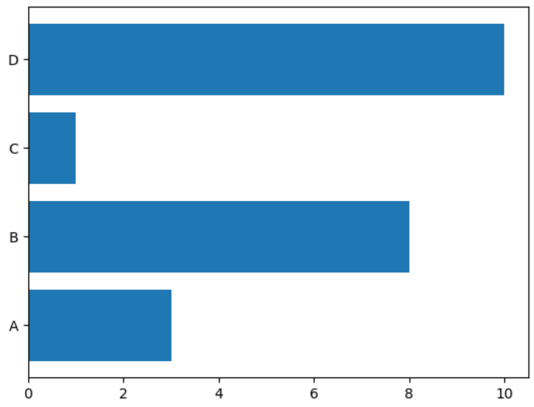
With Pyplot, you can use the `bar()` function to draw bar graphs:



```
import matplotlib.pyplot as plt
import numpy as np
x=np.array(["A", "B", "C", "D"])
y=np.array([3, 8, 1, 10])
plt.bar(x,y)
plt.show()
```

Horizontal Bars

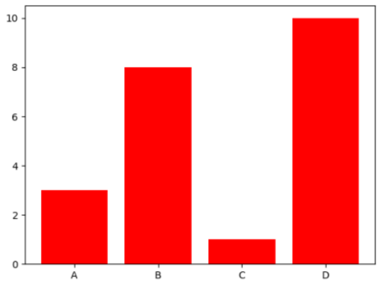
If you want the bars to be displayed horizontally instead of vertically, use the `barh()` function:



```
import matplotlib.pyplot as plt
import numpy as np
x=np.array(["A", "B", "C", "D"])
y=np.array([3, 8, 1, 10])
plt.barh(x,y)
plt.show()
```

Bar Color

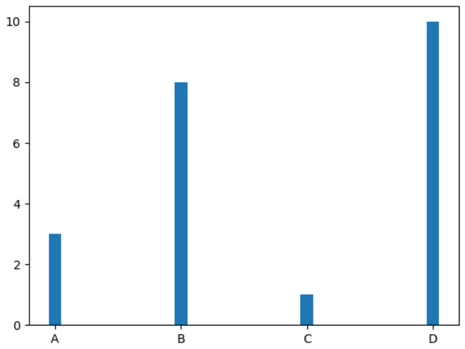
The `bar()` and `barh()` takes the keyword argument `color` to set the color of the bars:



```
import matplotlib.pyplot as plt
import numpy as np
x=np.array(["A", "B", "C", "D"])
y=np.array([3, 8, 1, 10])
plt.bar(x,y,color= "red")
plt.show()
```

Bar Width

The `bar()` takes the keyword argument `width` to set the width of the bars:



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

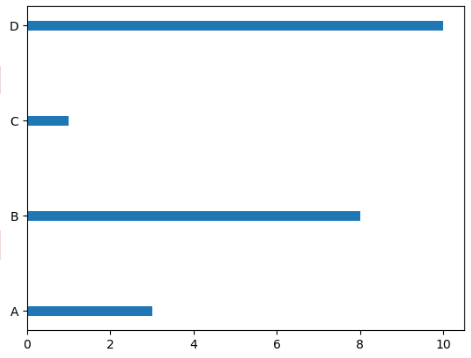
```
x=np.array(["A", "B", "C", "D"])
y=np.array([3, 8, 1, 10])

plt.bar(x,y,width= 0.1)
plt.show()
```

The default width value is 0.8

Bar Height

The `barh()` takes the keyword argument `height` to set the height of the bars:

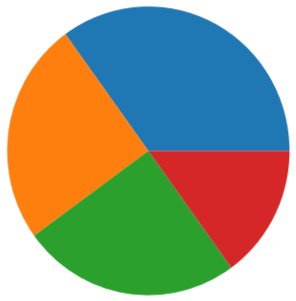
```
import matplotlib.pyplot as plt
import numpy as np
x=np.array(["A", "B", "C", "D"])
y=np.array([3, 8, 1, 10])
plt.barh(x,y,height= 0.1)
plt.show()
```



Pie Charts

With Pyplot, you can use the `pie()` function to draw pie charts:

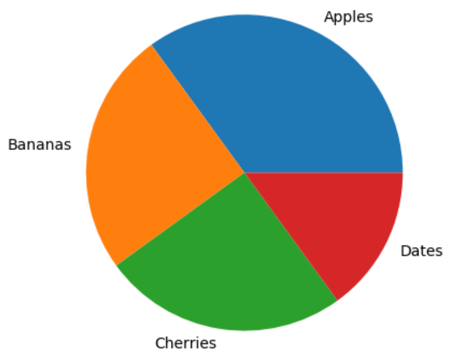
```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
plt.pie(y)
plt.show()
```



Labels

Add labels to the pie chart with the `label` parameter. The `label` parameter must be an array with one label for each wedge:

```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels = mylabels)
plt.show()
```



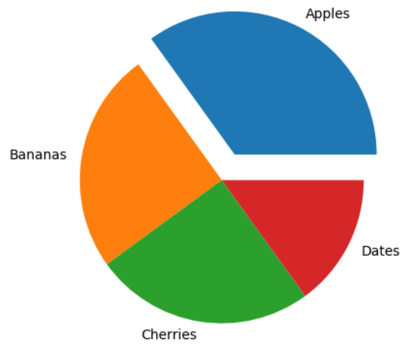
Explode

Maybe you want one of the wedges to stand out? The `explode` parameter allows you to do that.

The `explode` parameter, if specified, and not `None`, must be an array with one value for each wedge.

Each value represents how far from the center each wedge is displayed:

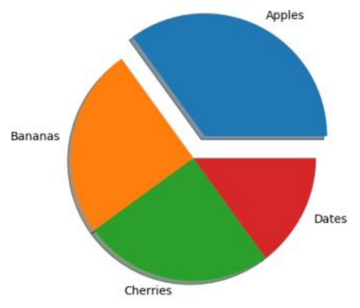
Pull the "Apples" wedge 0.2 from the center of the pie:



```
import matplotlib.pyplot as plt
import numpy as np
y
= np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels = mylabels, explode = myexplode)
plt.show()
```

Shadow

Add a shadow to the pie chart by setting the `shadows` parameter to `True`:

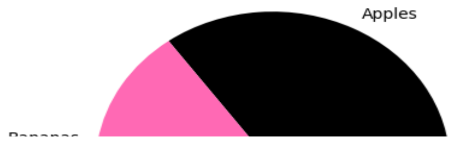


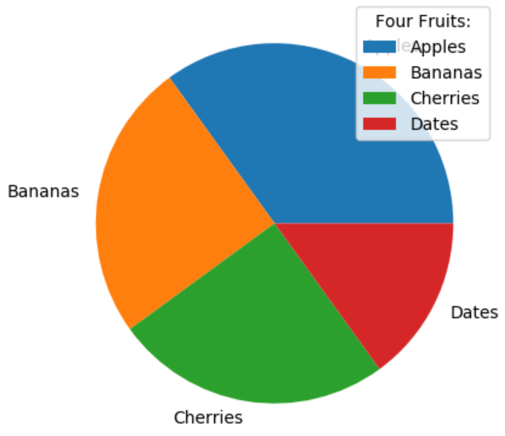
```
import matplotlib.pyplot as plt
import numpy as np
y =
np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels = mylabels, explode = myexplode, shadow = True)
plt.show()
```

Colors

You can set the color of each wedge with the `colors` parameter.

The `colors` parameter, if specified, must be an array with one value for each wedge:





```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
mycolors = ["black", "hotpink", "b", "#4CAF50"]
```

```
plt.pie(y, labels = mylabels, colors = mycolors)
plt.show()
```

Legend

To add a list of explanation for each wedge, use the `legend()` function. To add a header to the legend, add the `title` parameter to the `legend` function.

```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend(title = "Four Fruits:")
plt.show()
```

Histogram

A histogram is a graph showing *frequency* distributions.

It is a graph showing the number of observations within each given interval.

Example: Say you ask for the height of 250 people, you might end up with a histogram like this:

You can read from the histogram that there are approximately:

2 people from 140 to 145cm

5 people from 145 to 150cm

15 people from 151 to 156cm

31 people from 157 to 162cm

46 people from 163 to 168cm

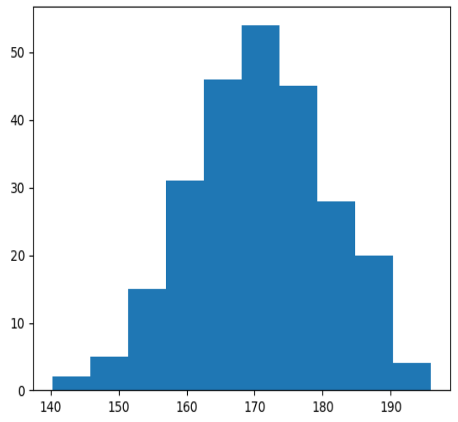
53 people from 168 to 173cm

45 people from 173 to 178cm

28 people from 179 to 184cm

21 people from 185 to 190cm

4 people from 190 to 195cm

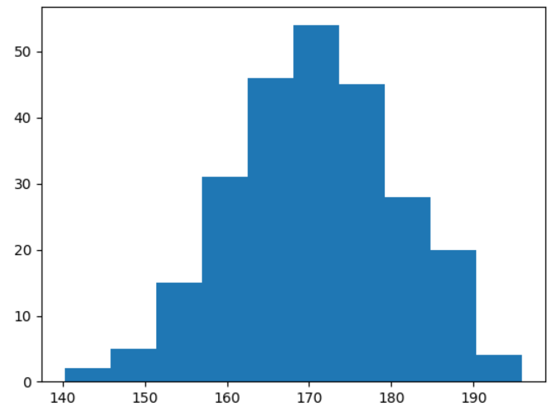


In Matplotlib, we use the **hist()** function to create histograms.

The hist() function will use an array of numbers to create a histogram, the array is sent into the function as an argument.

For simplicity we use NumPy to randomly generate an array with 250 values, where the values will concentrate around 170, and the standard deviation is 10.

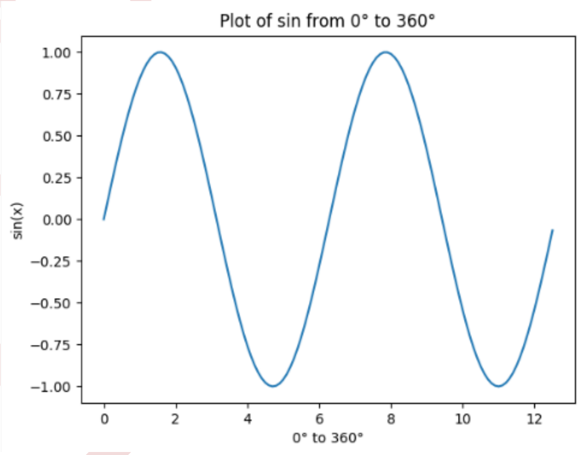
```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.normal(170, 10, 250)
plt.hist(x)
plt.show()
```



Plotting Simple Mathematical Functions

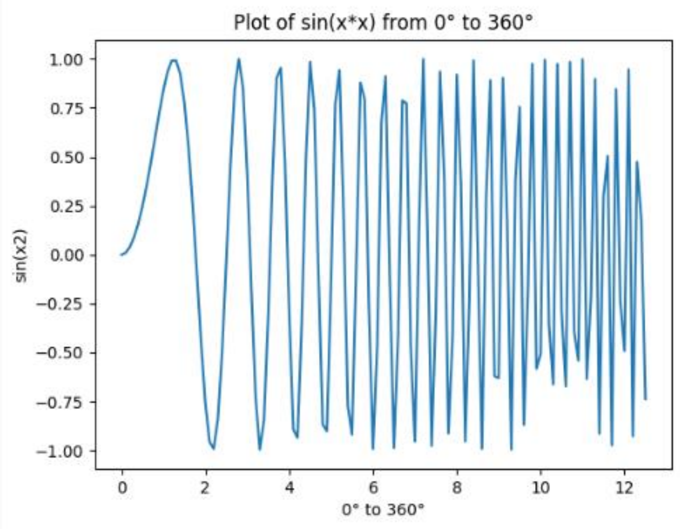
Sin(x)

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

x = np.arange(0,4*np.pi,0.1)

sin = np.sin(x)

plt.plot(x, sin)

plt.ylabel("sin(x)")

plt.xlabel("0° to 360°")

plt.title("Plot of sin from 0° to 360°")

plt.show()
```



sin(x^2)

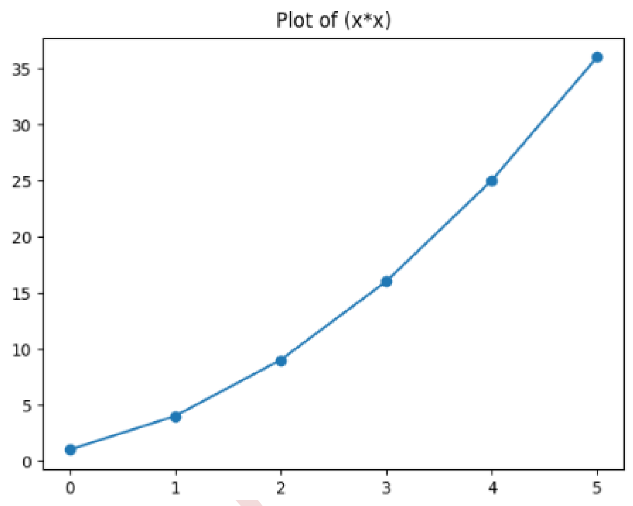
```
import matplotlib.pyplot as plt

import numpy as np

x = np.arange(0,4*np.pi,0.1)

sin = np.sin(x*x)

plt.plot(x,sin)

plt.ylabel("sin(x2)")
```



```
plt.xlabel("0° to 360°")

plt.title("Plot of sin(x*x) from 0° to 360°")

plt.show()
```

x^2

```
import matplotlib.pyplot as plt

import numpy as np

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

plt.plot(x*x,'o-')

plt.title("Plot of (x*x)")

plt.show()
```


Plot of (x*x)