Part 2
OOP Using Python

Languages like C++, Java and Python use classes and objects in their programs and are called Object Oriented Programming Languages. A Class is a module which itself contains data and methods (functions) to achieve the task. The main task is divided into several sub tasks, and these are represented as classes. Each class can perform several inter-related tasks for which several methods are written in a class. This approach is called Object Oriented Approach.

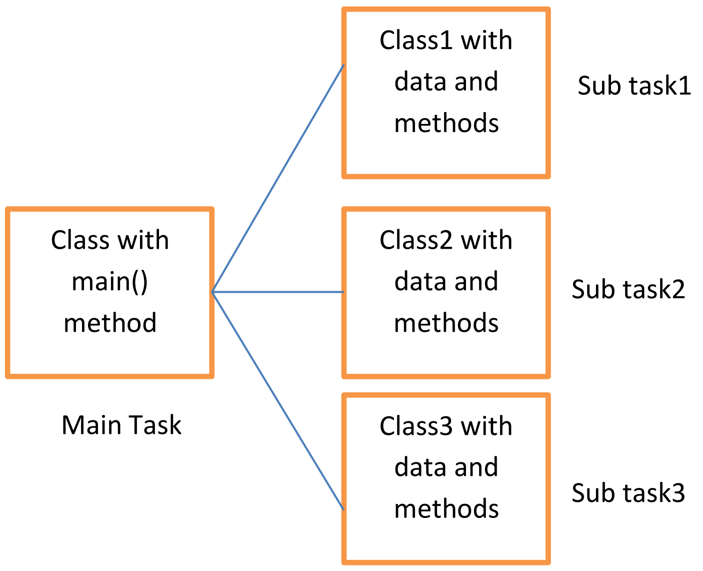

Class1 with data and methods
Sub task1
Class with main() method
Main Task
Class2 with data and methods
Sub task2
Class3 with data and methods
Sub task3

Fig: Object Oriented Approach

Even though, Python is an object oriented programming language like java, it does not force the programmers to write programs in complete object oriented way. Unlike java, Python has the blend of both the object oriented and procedure oriented features. Hence, Python programmers can write programs using procedure oriented approach (like c) or object oriented approach (like java) depending on their requirements. This is definitely an advantage for Python programmers!

Features of Object Oriented Programming System (OOPS)

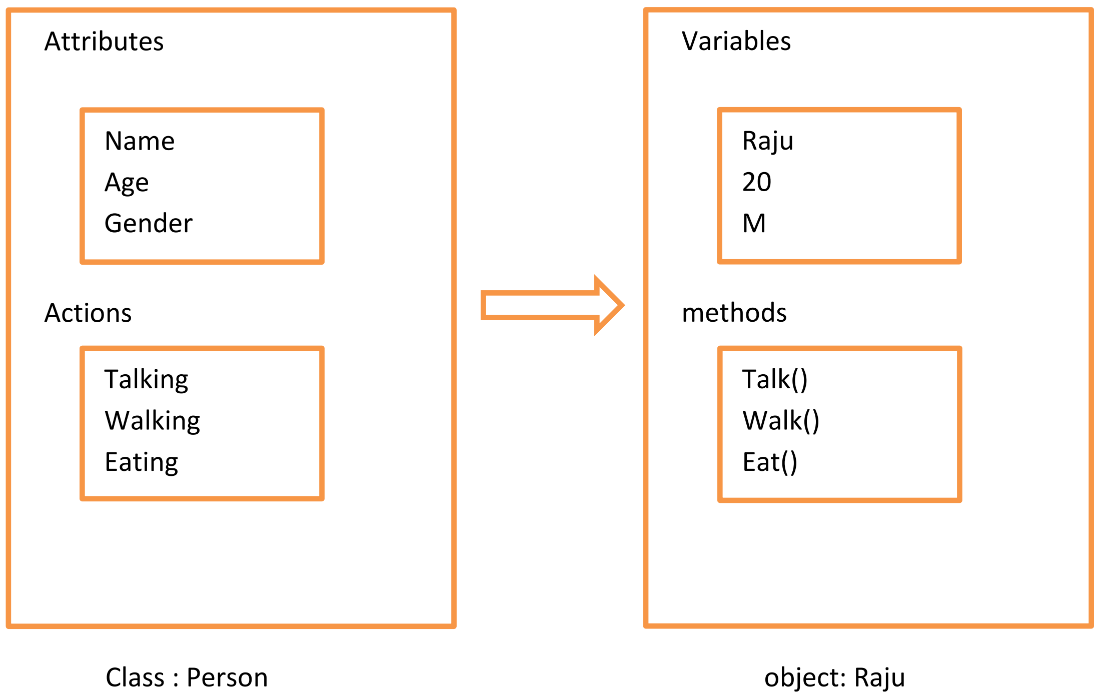
- **Classes and objects**
- **Encapsulation**
- **Abstraction**
- **Inheritance**
- **Polymorphism**

1. Classes and objects

The entire OOPS methodology has been derived from a single root concept called 'object', An object is anything that really exists in the world and can be distinguished from others. This definition specifies that everything in this world is an object. For example, a table, a ball, a car, a dog, a person etc. will come under objects.

Every object has some behavior. The behavior of an object is represented by attributes and actions. For example, let's take a person whose name is 'Raju'. Raju us an object because he exists physically. He has attributes like name, age, gender etc. For example, 'name' is a string type variable, 'age' is an integer type variable.

Similarly, Raju can perform some actions like talking, walking, eating and sleeping. We may not write code for such actions in programming. But, we can consider calculations and processing of data as actions. These actions are performed by methods. We should understand that a function written inside a class is called a method. So an object contains variables and methods.



We can use a class as a model for creating objects. To write a class, we can write all the characteristics of objects which should follow the class. These characteristics will guide us to create objects. A class and its objects are almost the same with the difference that a class does not exist physically, while an object does.

For Example, let's say we want to construct a house. First of all, we will go to an architect who provides a plan. This plan is only an idea and exists on paper. This is called a class. However, based on this plan, if we construct the house, it is called an object since it exists physically. So we can say that we can create objects from the class. An object does not exist without a class. But a class can exist without any object.

Creating Classes and Objects in Python:

The general format of a class is given follows:

```
class classname(object):
        """docstring describing the class"""
        attributes
        def __init__(self):
        def method1():
        def method2():
```

A class is created with the keyword class and then writing the classname. After the classname, 'object' is written inside the classname. This 'object' represents the base class name from where all classes in Python are derived. Please note that writing 'object' is not compulsory since it is implied. __init__ is a special method to initialize the variables. method1() and method2() etc are methods that are intended to process variables.

```
class Student:   #class name should start with capital letter
    def __init__(self):   #this method is useful to initialized the variables.
        self.name="hiren"
        self.age=20
        self.marks=68

    def talk(self):
        print("hi i am",self.name)
        print("my age is ",self.age)
        print("my marks are",self.marks)
```

Observe that the keyword class is used to declare a class. After this, we should write the class name. So, 'Student' is our class name. Generally, a class name should start with a capital letter, hence 'S' is capital in 'Student'. In the class, we write attributes and methods. Since in python we cannot declare variables, we have written the variables inside a special method, i.e. __init__(). This method is useful to initialize the variables. Hence, the name 'init'. The method name has two underscores before and after. This indicates that this method is internally defined and we cannot call this method explicitly. Observe the parameter 'self' written after the method name in the parentheses, 'self' is a variables that refers to current class instance.

To create an instance, the following syntax is used:

Instancename=Classname()

So, to create an instance (or object) to the student class, we can write as:

s1.Student()

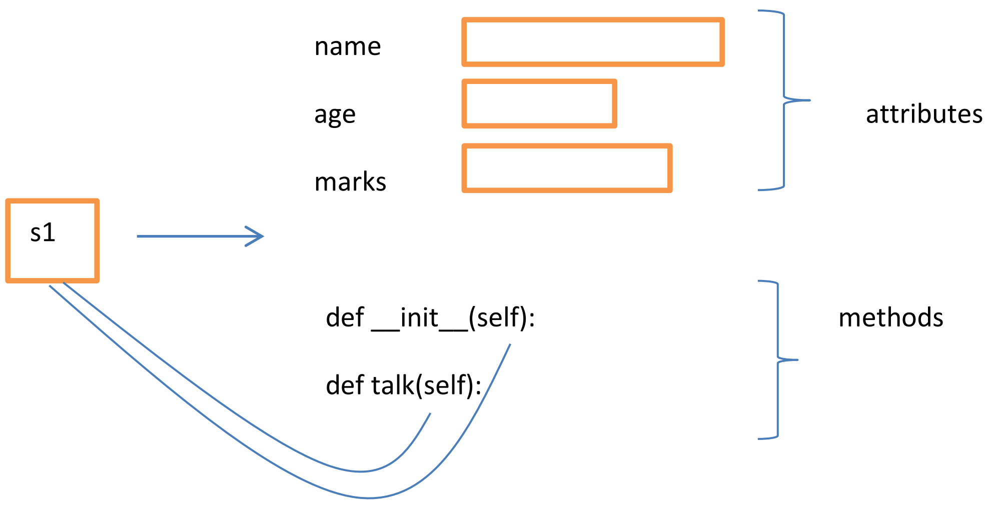


Fig: Student class instance in memory

prog: A Python program to define Student class and create an object to it. Also, we will call the method and display the student's details.

```
class Student:
   #this i a special method called constructor
   def __init__(self):
      self.name="raju"
      self.age=20
      self.marks=88

   #this is an instance method
   def talk(self):
      print("Hi, i am ",self.name)
      print("My age is ",self.age)
      print("My marks are ",self.marks)

#create an instance to Student class
s1=Student()

#call the method using the instance
s1.talk()
```

‘self’ is a default variable that contains the memory of the instance of the current class. So, we can use ‘self’ to refer to all the instance variables and instance methods.
Here, ‘s1’ contains the memory address of the instance. This memory address is internally and by default passed to ‘self’ variable. Since, ‘self’ knows the memory address of the instance, it can refer to all the members of the instance. We use ‘self’ in two ways.

def __init__(self): #the ‘self’ variable is used as first parameter in the constructor as

def talk(self): #’self’ can be used as first parameter in the instance methods as

<u>Constructor:</u>

A constructor is a special method that is used to initialize the instance variables of a class.

```
class Student:
   #this is constructor
   def __init__(self,n=".",m=0):
      self.name=n
      self.marks=m

   #this is an instance method
   def display(self):
      print("Hi",self.name)
      print("Your marks",self.marks)

#constuctor is called without any arguments
s=Student()
s.display()
print("--------------")

#constructor is called with 2 arguments
s1=Student("xyz",85)
s1.display()
print("-------------")
```

<u>Types of Variables:</u>
The variables which are written inside a class are of 2 types:
- Instance variables
- Class variables or static variables

Instance variables are the variables whose separate copy is created in every instance (or object). For example, if ‘x’ is an instance variable and if we create 3 instances, there will be 3 copies of ‘x’ in these 3 instances. When we modify the copy ‘x’ in any instance, it will not modify the other two copies.

Prog: A Python program to understand instance variables.

```
#instance vars example
class Sample:
   #this is a constuctor
   def __init__(self):
      self.x=10

   #this is an instance method
   def modify(self):
      self.x+=1

#create 2 instances
s1=Sample()
s2=Sample()
print("x in s1=",s1.x)
print("x in s2=",s2.x)

#modify x in s1
s1.modify()
print("x in s1=",s1.x)
print("x in s2=",s2.x)
```

output

```
x in s1= 10
x in s2= 10
x in s1= 11
x in s2= 10
```

Prog: A Python program to understand class variables or static variables

```
#class vars or static vars example
class Sample:
   #this is a class var
   x=10

   #this is a class method
   @classmethod     #this is a decorator
   def modify(cls):     #cls must be the first parameter
       cls.x+=1            #cls.x refers to class variable x

#create 2 instances
s1=Sample()
s2=Sample()
print("x in s1=",s1.x)
print("x in s2=",s2.x)

#modify x in s1
s1.modify()
print("x in s1=",s1.x)
print("x in s2=",s2.x)
```

output:

```
x in s1= 10
x in s2= 10
x in s1= 11
x in s2= 11
```

Observe above program the class variable ‘x’ is defined in the class and initialized with value 10. A method by the name ‘modify’ is used to modify the value of ‘x’. This method is called ‘class method’ since it is acting on the class variable. To mark this method as class method, we should use build-in decorator statement @classmethod.

2. Inheritance

When a programmer develops a class, he will use its features by creating an instance to it. When another programmer wants to create another class which is similar to the class already created, then he need not create the class from the search. He can simply use the features of the existing class in creating his own class.

Prog: A Python program to create Teacher class and store it into teacher.py

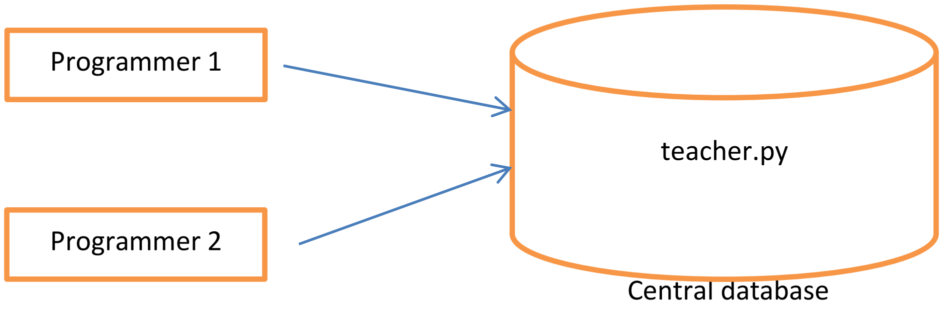
#this is Teacher class. Save this code in teacher.py

```
class Teacher:
   def setid(self,id):
       self.id=id

   def getid(self):
       return self.id

   def setname(self,name):
       self.name=name

   def getname(self):
       return self.name

   def setaddress(self,address):
       self.address=address

   def getaddress(self):
       return self.address

   def setsalary(self,salary):
       self.salary=salary

   def getsalary(self):
       return self.salary
```

This file can simply import this class into his program and use it as shown here.

Prog: A Python program to use the Teacher class.

```
#save this file inh.py
#using Teacher class
from teacher import Teacher
#create instance
t=Teacher()

#store data into the instance
t.setid(101)
t.setname("Amit")
t.setaddress("Liliya Road, Amreli")
t.setsalary(35000.00)

#retrieve data from instance and display
print("id = ",t.getid())
print("Name = ",t.getname())
print("Address = ",t.getaddress())
print("Salary = ",t.getsalary())
```



Prog: A Python program to create Student class by deriving it from the Teacher class.

```
from teacher import Teacher

class Student(Teacher):

    def setmarks(self,marks):

        self.marks=marks


    def getmarks(self):

        return self.marks


s=Student()

s.setid(100)

s.setname("rakesh")

s.setaddress("Chakkargadh road, amreli")

s.setmarks(85)
```

```
print("Id = ",s.getid())

print("Name = ",s.getname())

print("Address = ",s.getaddress())

print("marks = ",s.getmarks())
```

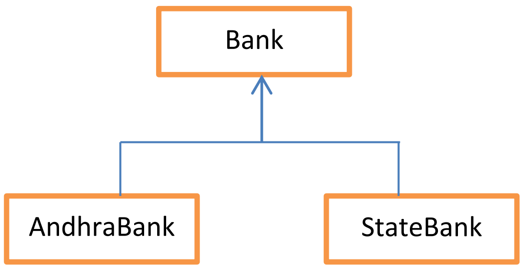
Output:

```
Id =  100
Name =  rakesh
Address =  Chakkargadh road, amreli
marks =  85
```

Types of Inheritance:

- **Single Inheritance**
- **Multiple Inheritance**

Single Inheritance



Prog: A Python program showing single inheritance in which two sub classes are derived from a single base class.

```
#single inheritance
class Bank(object):
   cash=500000
   @classmethod
   def available_cash(cls):
      print(cls.cash)

class AndhraBank(Bank):
   pass
```

```
class StateBank(Bank):
    cash=200000
    @classmethod
    def available_cash(cls):
        print(cls.cash+Bank.cash)

a=AndhraBank()
a.available_cash()

s=StateBank()
s.available_cash()
```

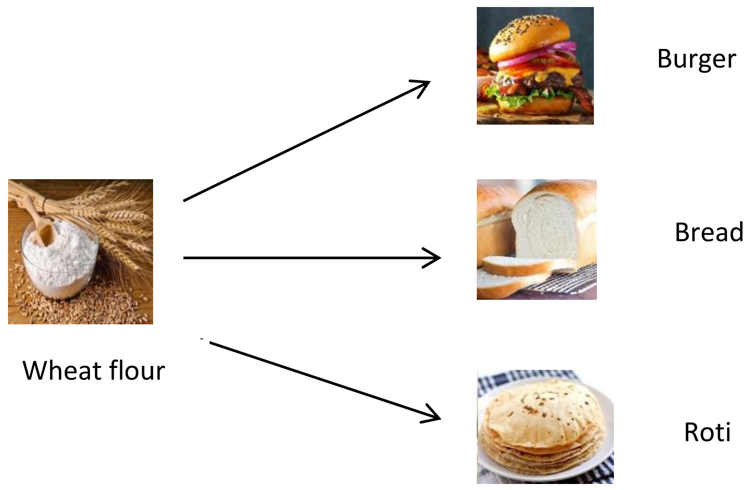
Multiple Inheritances

Prog: A Python program to implement multiple inheritance using two base classes.

```
#multiple inheritance
class Father:
    def height(self):
        print("Height is 6.0 Foot")

class Mother:
    def color(self):
        print("Color is Brown")

class Child(Father,Mother):
    pass

c=Child()
print("Child's inherited qualities: ")
c.height()
c.color()
```

Polymorphism

Polymorphism is a word that came from two greek words, poly means many and morphos means forms. If something exhibits various forms, it is called polymorphism. For Example, Assume that we have wheat flour. Using this wheat flour we can make burgers, rotis, or bread. It means same wheat flour same wheat flour is taking different edible forms and hence we can say wheat flour is exhibiting polymorphism.



Encapsulations in Python: (Information Hiding)

An objects variables should not always be directly accessible.

To prevent accidental change, an objects variables can sometimes only be changed with an objects methods. Those type of variables are private variables.

The methods can ensure the correct values are set. If an incorrect value is set, the method can return an error.

Private methods

We create a class Car which has two methods: drive() and updateSoftware(). When a car object is created, it will call the private methods __updateSoftware().

This function cannot be called on the object directly, only from within the class

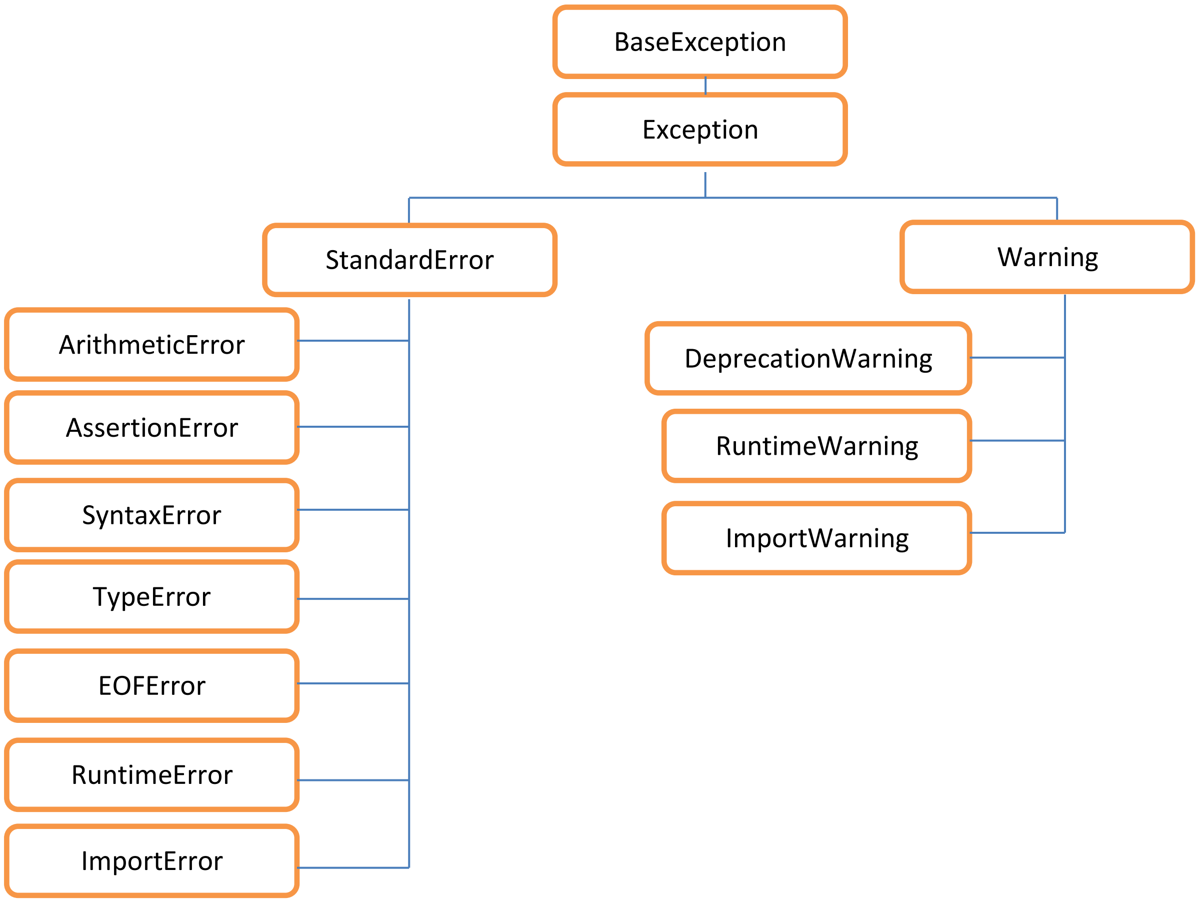
```
class Car:
  def __init__(self):
    self.__updateSoftware()

  def drive(self):
    print("Driving")

  def __updateSoftware(self):
    print("updating software")
```

```
        redcar = Car()
        redcar.drive()
        #redcar.__updateSoftware()  not accesible from object.
```

Exceptions

An exception is a runtime error which can be handled by the programmer. That means if the programmer can guess an error in the program and he can do something to eliminate the harm caused by the error, then it is called an ‘exception’. If the programmer cannot do anything in case of an error, then it is called an ‘error’ and not an exception.

All exception are represented as classes in Python. The exceptions which are already available in python are called ‘built-in’ exceptions. The base class for all built-in exceptions is ‘BaseException’ class. From BaseException class, the sub class ‘Exception’ is derived. From Exception class, the sub classes ‘StandardError’ and ‘Warning’ are derived.



Exception Handling:

Syntax:

```
try:
    statements
except Exception1:
   handler1
except Exception2:
   handler2
else:
   statements
finally:
   statements
```

Types of Exceptions:

Exception class name	Description
Exception	Represents any type of exception.
ArithmeticError	Represents the base class for arithmetic errors like OverflowError,ZeroDivisionErrror, FloatingPointError
AssertionError	Raised when an assert statement gives error
IOError	Raised when an input or output operation failed.
SyntaxError	Raised when the compiler encounters a syntax error.
ZeroDivisionError	Raised when the denominator is zero in a division.

```
try:
   x=int(input("Enter number1:"))
   y=int(input("Enter number2:"))
   print(x/y)
except ZeroDivisionError:
   print("Can not divide zero")
```

Prog:A Python program to handle the ZeroDivisionError example

```
#an exception handling example
try:
   f=open("myfile","w")
   a,b=[int(x) for x in input("Enter two numbers: ").split()]
   c=a/b
   f.write("Writing %d into myfile"%c)
```

```
except ZeroDivisionError:
    print("Division by zero happened")
    print("please do not enter 0 in input")

finally:
    f.close()
    print("File closed")
```

Prog: A Python program to handle syntax error given by eval() function

```
#example for syntax error
try:
    date=eval(input("Enter date: "))
except SyntaxError:
    print("Invalid date entered")
else:
    print("You entered: ",date)
```

```
Enter date: 2016,10,3
You entered:  (2016, 10, 3)

Enter date: 2016,10b,3
Invalid date entered
```

The assert statement

The assert statement is useful to ensure that a given condition is True. If it is not true, it raises AssertionError. Syntax is

```
assert condition, message
```

prog: A Python program using the assert statement and catching AssertionError.

```
#handling AssertionError
try:
    x=int(input("Enter a number between 5 and 10: "))
    assert x>=5 and x<=10
    print("The number entered:",x)
except AssertionError:
    print("The condition is not fulfilled")
```

Search Algorithm in Python

Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for. This is known as Linear search. It is inefficient and rarely used, but creating a program for it gives an idea about how we can implement some advanced search algorithms.

- **Linear Search**

  In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data structure.

Prog: **A python program for linear search.**

```
def linear_search(values, search_for):
   search_at = 0
   search_res = False

# Match the value with each data element
   while search_at < len(values) and search_res is False:
      if values[search_at] == search_for:
         search_res = True
      else:
         search_at = search_at + 1

   return search_res

list1=[64, 34, 25, 12, 22, 11, 90]
print(linear_search(list1, 12))
```

Binary Search

Binary search follows a divide and conquer methodology. It is faster than linear search but requires that the array be sorted before the algorithm is executed.

Assuming that we're searching for a value val in a sorted array, the algorithm compares val to the value of the middle element of the array, which we'll call mid.

- If mid is the element we are looking for (best case), we return its index.
- If not, we identify which side of mid val is more likely to be on based on whether val is smaller or greater than mid, and discard the other side of the array.

- We then recursively or iteratively follow the same steps, choosing a new value for mid, comparing it with val and discarding half of the possible matches in each iteration of the algorithm.

  The binary search algorithm can be written either recursively or iteratively. Recursion is generally slower in Python because it requires the allocation of new stack frames.

  Since a good search algorithm should be as fast and accurate as possible, let's consider the iterative implementation of binary search:

Prog: A Python program for Binary Search.

```
def BinarySearch(lys, val):
    first = 0
    last = len(lys)-1
    index = -1
    while (first <= last) and (index == -1):
        mid = (first+last)//2
        if lys[mid] == val:
            index = mid
        else:
            if val<lys[mid]:
                last = mid -1
            else:
                first = mid +1
    return val
print(BinarySearch([10,20,30], 20))
```

Bubble Sort

Bubble sort is the one usually taught in introductory CS classes since it clearly demonstrates how sort works while being simple and easy to understand. Bubble sort steps through the list and compares adjacent pairs of elements. The elements are swapped if they are in the wrong order. The pass through the unsorted portion of the list is repeated until the list is sorted. Because Bubble sort repeatedly passes through the unsorted part of the list, it has a worst case complexity of $O(n^2)$.

Prog: Bubble Sort

```
def bubble_sort(arr):
    def swap(i, j):
        arr[i], arr[j] = arr[j], arr[i]

    n = len(arr)
    swapped = True
```

```
   x = -1
   while swapped:
      swapped = False
      x = x + 1
      for i in range(1, n-x):
         if arr[i - 1] > arr[i]:
            swap(i - 1, i)
            swapped = True

   print(arr)
bubble_sort([1,2,4,3,5])
```

Selection Sort

Selection sort is also quite simple but frequently outperforms bubble sort. If you are choosing between the two, it’s best to just default right to selection sort. With Selection sort, we divide our input list / array into two parts: the sublist of items already sorted and the sublist of items remaining to be sorted that make up the rest of the list. We first find the smallest element in the *unsorted sublist* and place it at the end of the *sorted sublist.* Thus, we are continuously grabbing the smallest unsorted element and placing it in sorted order in the *sorted sublist.* This process continues iteratively until the list is fully sorted.

```
def selection_sort(arr):
   for i in range(len(arr)):
      minimum = i

      for j in range(i + 1, len(arr)):
         # Select the smallest value
         if arr[j] < arr[minimum]:
            minimum = j

      # Place it at the front of the
      # sorted end of the array
      arr[minimum], arr[i] = arr[i], arr[minimum]

   print(arr)
selection_sort([20,10,30,40])
```

Hashtables:

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words Hash table stores key-value pairs but the key is generated through a hashing function.

So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.

- The keys of the dictionary are hashable i.e. the are generated by hashing function which generates unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.

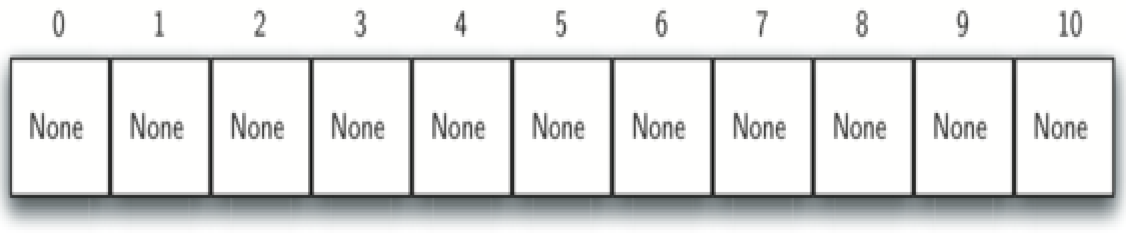
Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

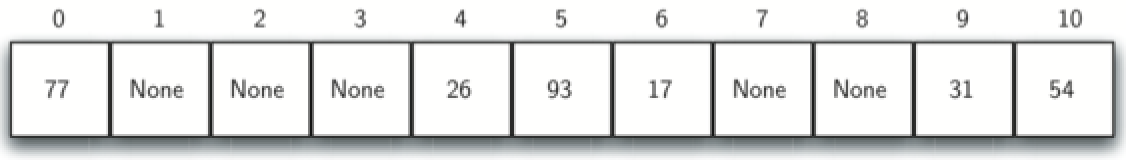
```
# Declare a dictionary

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}



# Accessing the dictionary with its key

print "dict['Name']: ", dict['Name']

print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Zara
dict['Age']:  7
```



Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9


0
1
2
3
4
5
6
7
8
9
10
77
None
None
None
26
93
17
None
None
31
54