Q1. What is Procedural Oriented Approach of Programming and drawbacks.

Early programming languages developed in 50s and 60s are recognized as procedural (or procedure oriented) languages.

A computer program describes procedure of performing certain task by writing a series of instructions in a logical order. Logic of a more complex program is broken down into smaller but independent and reusable blocks of statements called functions.

Every function is written in such a way that it can interface with other functions in the program. Data belonging to a function can be easily shared with other in the form of arguments, and called function can return its result back to calling function.

Prominent problems related to procedural approach are as follows –

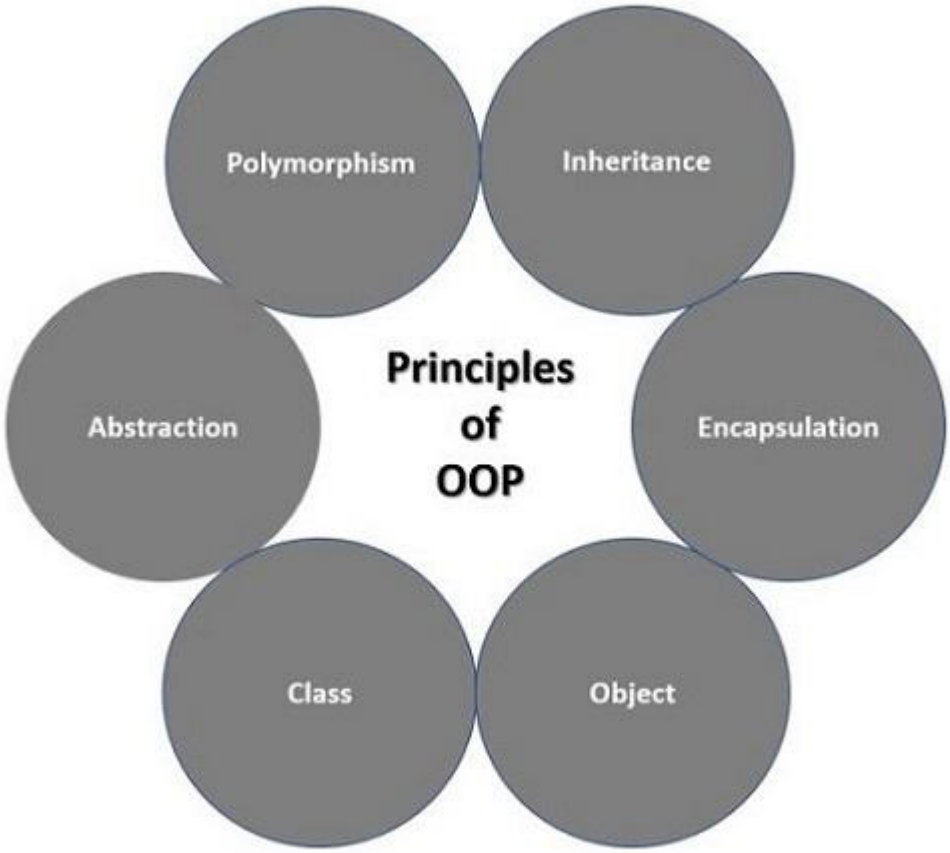
- Its top-down approach makes the program difficult to maintain.
- It uses a lot of global data items, which is undesired. Too many global data items would increase memory overhead.
- It gives more importance to process and doesn't consider data of same importance and takes it for granted, thereby it moves freely through the program.
- Movement of data across functions is unrestricted. In real-life scenario where there is unambiguous association of a function with data it is expected to process.

Q2. Explain about Object Oriented Programming .

OOPS (Object-Oriented Programming) concepts are fundamental principles for designing software using objects, focusing on real-world entities, with the core being **Class, Object, Inheritance, Polymorphism, Abstraction, and Encapsulation** (the four pillars). They help create modular, reusable, and maintainable code by binding data and behavior together, hiding complexity, and enabling code reuse through relationships like Association, Aggregation, and Composition.

Key Features of OOP in Python:

- Organizes code into classes and objects
- Supports encapsulation to group data and methods together
- Enables inheritance for reusability and hierarchy
- Allows polymorphism for flexible method implementation
- Improves modularity, scalability, and maintainability



Class & Object

A **class** is an user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

An **object** refers to an instance of a certain class. For example, an object named **obj** that belongs to a class **Circle** is an instance of that class. A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Encapsulation

Data members of class are available for processing to functions defined within the class only. Functions of class on the other hand are accessible from outside class context. So object data is hidden from environment that is external to class. Class function (also called method) encapsulates object data so that unwarranted access to it is prevented.

Data Abstraction:

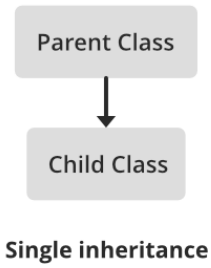
Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

Inheritance

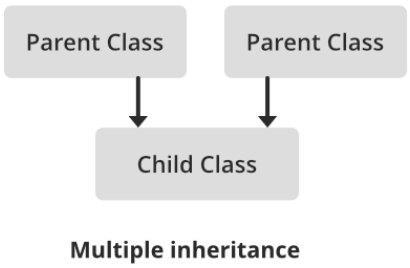
A software modelling approach of OOP enables extending capability of an existing class to build new class instead of building from scratch. In OOP terminology, existing class is called **base or parent class**, while new class is called **child or sub class**.

Child class inherits data definitions and methods from parent class. This facilitates reuse of features already available. Child class can add few more definitions or redefine a base class function.

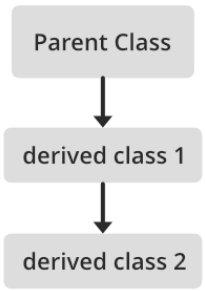
Types of Inheritance:



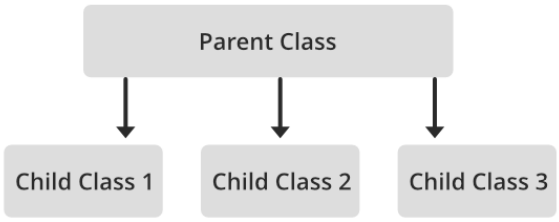
Single inheritance



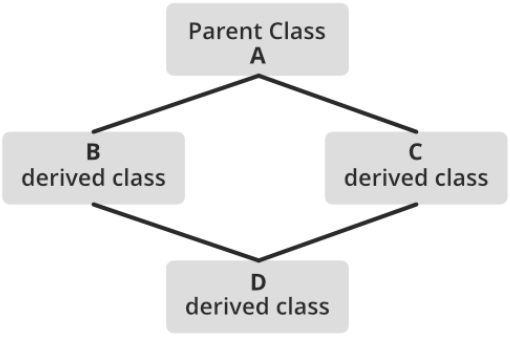
Multiple inheritance



Multilevel inheritance



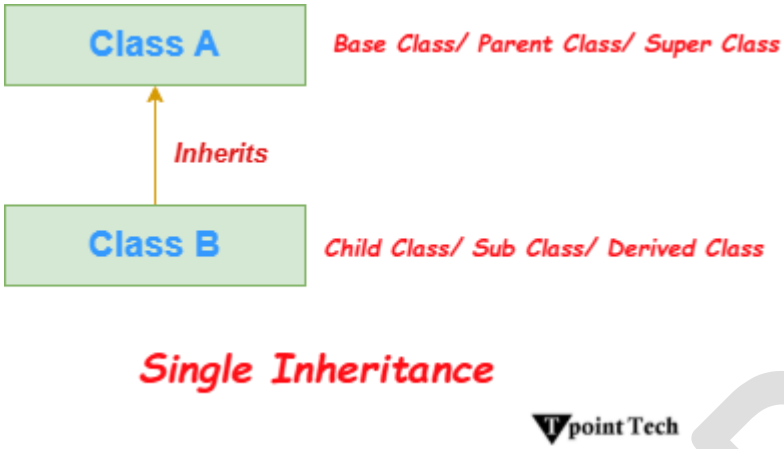
Hierarchical inheritance



Hybrid Class

Single Inheritance

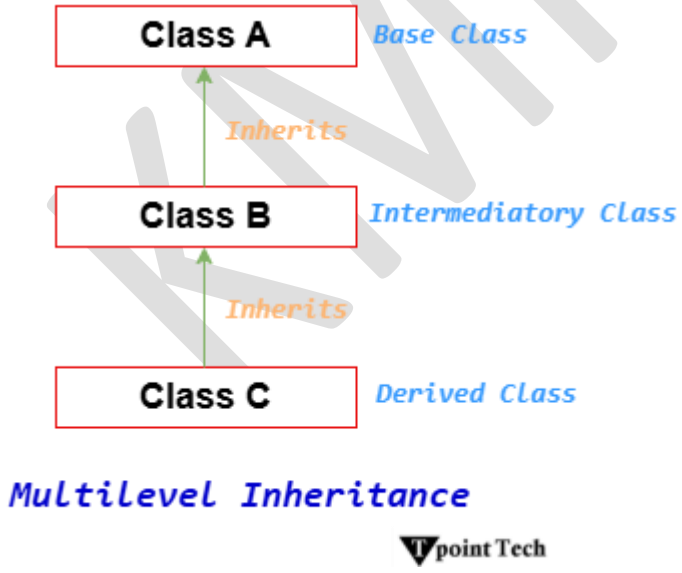
When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.



Multilevel Inheritance

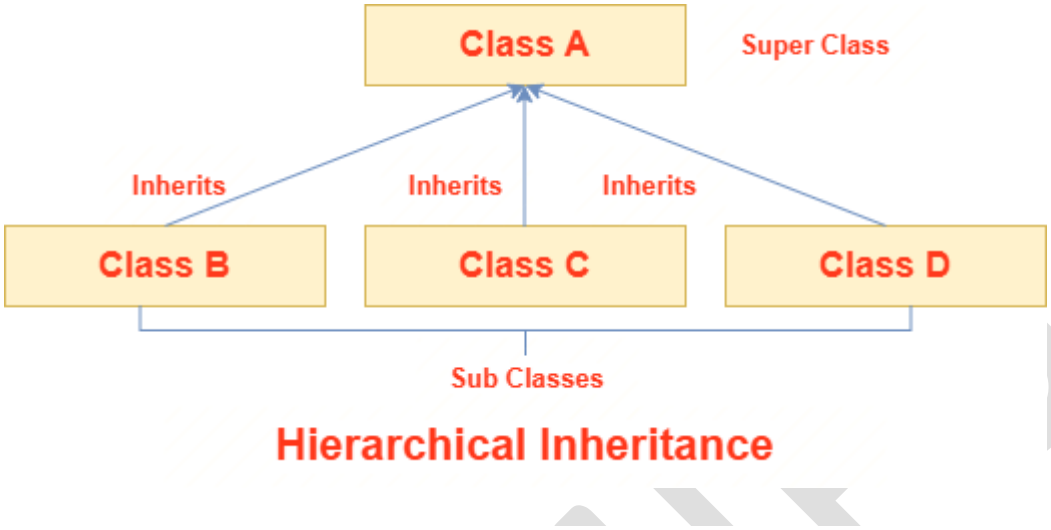
When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

To read more: Multilevel Inheritance in Java



Hierarchical Inheritance

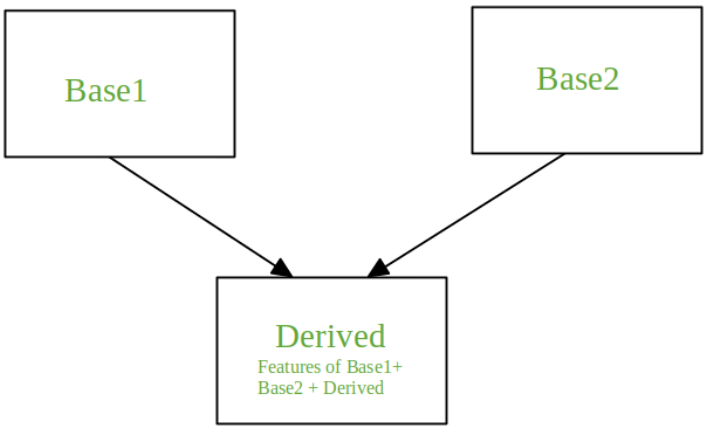
When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.



Multiple inheritance is a feature in object-oriented programming (OOP) that allows a **class to inherit attributes and methods from more than one parent class**. This allows a derived (child) class to combine the features and behaviors of multiple base (parent) classes.

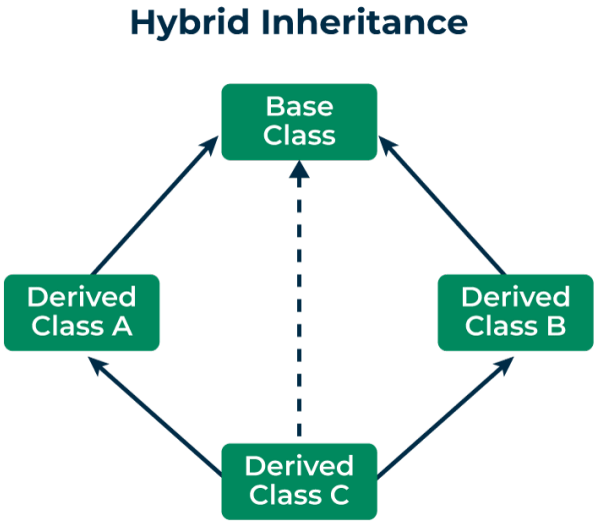
Key Concepts

- **Derived Class (Child Class):** The class that inherits from other classes.
- **Base Classes (Parent Classes):** The classes from which the derived class inherits.
- **Code Reusability:** The primary advantage, allowing developers to use existing code and functionalities in a new class without rewriting them.
- **Flexibility:** Enables modeling of complex, real-world relationships where an entity might have characteristics from multiple distinct categories (e.g., an `HourlyUnionWorker` class inheriting from both `HourlyWorker` and `UnionMember` classes).



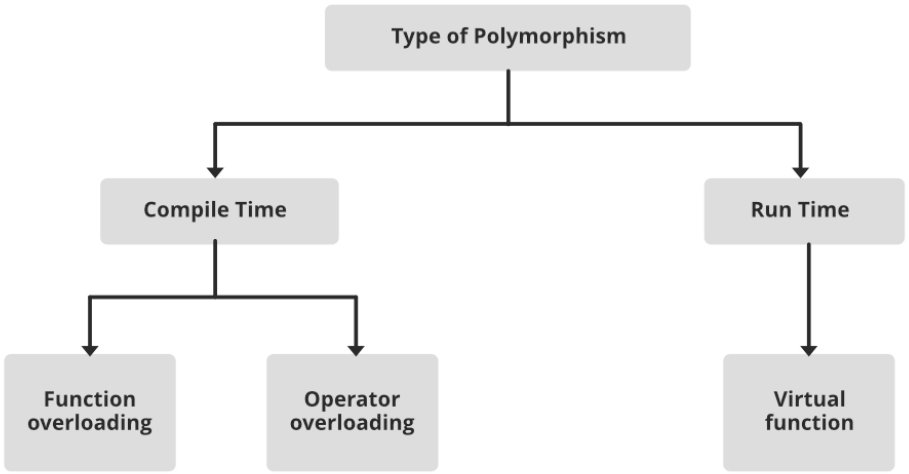
Hybrid Inhertiance:

Hybrid inheritance combines multiple inheritance types (like single, multiple, multilevel, hierarchical) in a single class hierarchy to model complex real-world relationships, allowing classes to inherit from various parents in different styles for enhanced code reuse, but it can introduce complexity, especially the "diamond problem" (ambiguity). It's a blend of inheritance styles, such as a class inheriting from two classes that themselves use multilevel inheritance.



Polymorphism:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism



Features of polymorphism:

- **Multiple Behaviors**: The same method can behave differently depending on the object that calls this method.
- **Method Overriding**: A child class can redefine a method of its parent class.
- **Method Overloading**: We can define multiple methods with the same name but different parameters.
- **Runtime Decision:** At runtime, Java determines which method to call depending on the object's actual class.

Dynamic Binding:

In dynamic binding, the code to be executed in response to the function call is decided at runtime. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. Dynamic Method Binding One of the main advantages of inheritance is that some derived class D has all the members of its base class B. Once D is not hiding any of the public members of B, then an object of D can represent B in any context where a B could be used. This feature is known as subtype polymorphism.

Message Passing:

It is a form of communication used in object-oriented programming as well as parallel programming. Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

Q3. Explain about creation of Classes and Objects in Python.

In Python, a class is a user defined entity (data types) that defines the type of data an object can contain and the actions it can perform. It is used as a template for creating objects. For instance, if we want to define a class for Smartphone in a Python program, we can use the type of data like RAM, ROM, screen-size and actions like call and message.

Creating Classes in Python

The class keyword is used to create a new class in Python. The name of the class immediately follows the keyword class followed by a colon as shown below –

Syntax:

class ClassName:

attributes;

class methods

Example:

```
class Employee:
  'Common base class for all employees'
  empCount = 0
  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1
  def displayCount(self):
   print "Total Employee %d" % Employee.empCount
  def displayEmployee(self):
    print "Name : ", self.name,  ", Salary: ", self.salary
```

Object:

An object is refered to as an instance of a given Python class. Each object has its own attributes and methods, which are defined by its class.When a class is created, it only describes the structure of obejcts. The memory is allocated when an object is instantiated from a class.

Creating Objects of Classes in Python

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts.

This would create first object of Employee class

emp1 = Employee("Zara", 2000)

This would create second object of Employee class

emp2 = Employee("Manni", 5000)

Accessing Attributes of Objects in Python

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows –

emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee %d" % Employee.empCount)

Garbage Collection(Destroying Objects) in Python

Python deletes unwanted objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

```
class Account:
  def __init__(self, owner, balance):
    self.owner = owner
    self.__balance = balance  # Private attribute

  def deposit(self, amount):
    self.__balance += amount

  def withdraw(self, amount):
    if amount <= self.__balance:
      self.__balance -= amount
    else:
      print("Insufficient funds")
```

```
  def get_balance(self):
    return self.__balance

acct = Account("John Doe", 1000)
acct.deposit(500)
print(acct.get_balance())  # Output: 1500
```

Q4. Discuss about Instance attributes and Class Attributes

The properties or variables defined inside a class are called as **Attributes**. An attribute provides information about the type of data a class contains. There are two types of attributes in Python namely **instance attribute** and **class attribute**.

The **instance attribute** is defined within the constructor of a Python class and is unique to each instance of the class. And, a class attribute is declared and initialized outside the constructor of the class.

Class Attributes (Variables)

Class attributes are those variables that belong to a class and whose value is shared among all the instances of that class. A class attribute remains the same for every instance of the class.

Class attributes are defined in the class but outside any method. They cannot be initialized inside **__init__()** constructor. They can be accessed by the name of the class in addition to the object. In other words, a class attribute is available to the class as well as its object. The object name followed by dot notation (.) is used to access class attributes.

Ex:-

```
class Employee:
  name = "Bhavesh Aggarwal"
  age = "30"
# instance of the class
emp = Employee()
# accessing class attributes
print("Name of the Employee:", emp.name)
print("Age of the Employee:", emp.age)
```

They are used to define those properties of a class that should have the same value for every object of that class.Class attributes can be used to set default values for objects.

Instance Attributes

As stated earlier, an instance attribute in Python is a variable that is specific to an individual object of a class. It is defined inside the __init__() method.

The first parameter of this method is self and using this parameter the instance attributes are defined.

```
class Student:
  def __init__(self, name, grade):
    self.__name = name          self.name,self.grade-----→ instance attributes
    self.__grade = grade
    print ("Name:", self.__name, ", Grade:", self.__grade)


# Creating instances
student1 = Student("Ram", "B")
student2 = Student("Shyam", "C")
```

Instance Attributes Vs Class Attributes

SNo.	Instance Attribute	Class Attribute
1	It is defined directly inside the __init__() function.	It is defined inside the class but outside the __init__() function.
2	Instance attribute is accessed using the object name followed by dot notation.	Class attributes can be accessed by both class name and object name.
3	The value of this attribute cannot be shared among other objects.	Its value is shared among other objects of the class.
4	Changes made to the instance attribute affect only the object within which it is defined.	Changes made to the class attribute affect all the objects of the given class.

the self parameter is a **reference to the current instance of a class** and is used to access its attributes and methods. It must be the first parameter of any instance method, including the special __init__ constructor.

Q5 Explain about Constructors and Destructors in Python.

Constructor in Python

- A **constructor** is a special method used to initialize an object's attributes when it is created.
- In Python, the constructor method is **__init__()**.
- It is automatically called when a new object is created.
- Commonly used to set initial values for object properties.

Example:

```
class Car:

  def __init__(self, make, model, year):  ----→ Constructor

    self.make = make

    self.model = model

    self.year = year
```

Destructors

A destructor is a special method called when an object is destroyed. It is used to clean up resources. In Python, the __del__ method acts as a destructor.

- A **destructor** is a special method used to clean up resources before an object is destroyed.
- In Python, the destructor method is **__del__()**.
- It is called **when the object is about to be destroyed** (usually when reference count becomes zero).
- Python has **automatic garbage collection**, so destructors are less commonly used than in languages like C++.

Example:

```
class Car:

  def __init__(self, make, model, year):

    self.make = make

    self.model = model

    self.year = year

  def __del__(self):  ---→ Destructor

    print(f"Car {self.make} {self.model} is being destroyed")
```

Q6. Discuss about Inheritance in Python and how it is implemented.

Inheritance is one of the most important features of object-oriented programming languages like Python. It is used to inherit the properties and behaviours of one class to another. The class that inherits another class is called a child class and the class that gets inherited is called a base class or parent class.

Creating a Parent Class

The class whose attributes and methods are inherited is called as parent class. It is defined just like other classes i.e. using the class keyword.

Syntax

The syntax for creating a parent class is shown below –

```
class ParentClassName:
  {class body}
```

Creating a Child Class

Classes that inherit from base classes are declared similarly to their parent class, however, we need to provide the name of parent classes within the parentheses.

Syntax

Following is the syntax of child class –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
  {sub class body}
```

Example:

```
# parent class
class Parent:
  def parentMethod(self):
    print ("Calling parent method")
# child class
class Child(Parent):
  def childMethod(self):
    print ("Calling child method")
```

```
# instance of child

c = Child()

# calling method of child class

c.childMethod()

# calling method of parent class

c.parentMethod()
```

Python - Multiple Inheritance

Multiple inheritance in Python allows you to construct a class based on more than one parent classes. The Child class thus inherits the attributes and method from all parents. The child can override methods inherited from any parent.

Syntax

```
class parent1:

  #statements

class parent2:

  #statements

class child(parent1, parent2):

  #statements
```

Example

Python's standard library has a built-in divmod() function that returns a two-item tuple. First number is the division of two arguments, the second is the mod value of the two operands.

```
class division:   -→Parent class-1
  def __init__(self, a,b):
    self.n=a
    self.d=b
  def divide(self):
    return self.n/self.d
class modulus:  --→-→ Parent class-2
  def __init__(self, a,b):
    self.n=a
    self.d=b
  def mod_divide(self):
    return self.n%self.d
```

```
class div_mod(division,modulus):    --→Multiple inheritance
  def __init__(self, a,b):
    self.n=a
    self.d=b
  def div_and_mod(self):
    divval=division.divide(self)
    modval=modulus.mod_divide(self)
    return (divval, modval)
x=div_mod(10,3)
print ("division:",x.divide())
 print ("mod_division:",x.mod_divide())
print ("divmod:",x.div_and_mod())
```

Python - Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class. There exists multiple layers of inheritance. We can imagine it as a grandparent-parent-child relationship.

Example

In the following example, we are illustrating the working of multilevel inheritance.

```
# parent class
class Universe:
  def universeMethod(self):
    print ("I am in the Universe")
# child class
class Earth(Universe):
  def earthMethod(self):
    print ("I am on Earth")
# another child class
class India(Earth):
  def indianMethod(self):
    print ("I am in India")
# creating instance
person = India()
# method calls
person.universeMethod()
person.earthMethod()
person.indianMethod()
```

Python - Hierarchical Inheritance

This type of inheritance contains multiple derived classes that are inherited from a single base class. This is similar to the hierarchy within an organization.

Example

The following example illustrates hierarchical inheritance. Here, we have defined two child classes of **Manager class**.

```
# parent class
class Manager:
  def managerMethod(self):
    print ("I am the Manager")
# child class
class Employee1(Manager):
  def employee1Method(self):
    print ("I am Employee one")
# second child class
class Employee2(Manager):
  def employee2Method(self):
    print ("I am Employee two")
# creating instances
emp1 = Employee1()
emp2 = Employee2()
# method calls
emp1.managerMethod()
emp1.employee1Method()
emp2.managerMethod()
emp2.employee2Method()
```

Q7. Write about super() function in Python.

In Python, **super()** function allows you to access methods and attributes of the parent class from within a child class.

Example

In the following example, we create a parent class and access its constructor from a subclass using the super() function.

```
# parent class
class ParentDemo:
  def __init__(self, msg):
    self.message = msg
  def showMessage(self):
    print(self.message)
# child class
class ChildDemo(ParentDemo):
  def __init__(self, msg):
    # use of super function
    super().__init__(msg)
# creating instance
obj = ChildDemo("Welcome to Tutorialspoint!!")
obj.showMessage()
```

Q8. Explain about Access Modifiers in Python.

The Python access modifiers are used to restrict access to class members (i.e., variables and methods) from outside the class. There are three types of access modifiers namely public, protected, and private.

Public members – A class member is said to be public if it can be accessed from anywhere in the program.

Protected members – They are accessible from within the class as well as by classes derived from that class.

Private members – They can be accessed from within the class only.

Usually, methods are defined as public and instance variable are private. This arrangement of private instance variables and public methods ensures implementation of principle of encapsulation.

Access Modifiers in Python

Unlike C++ and Java, Python does not use the Public, Protected and Private keywords to specify the type of access modifiers. By default, all the variables and methods in a Python class are public.Python doesn't enforce restrictions on accessing any instance variable or method. However, Python prescribes a convention of prefixing name of variable/method with single or double underscore to emulate behavior of protected and private access modifiers.

- To indicate that an instance variable is **private**, prefix it with double underscore (such as "__age").
- To imply that a certain instance variable is **protected**, prefix it with single underscore (such as "_salary").

```
class Employee:
  def __init__(self, name, age, salary):
    self.name = name # public variable
    self.__age = age # private variable
    self._salary = salary # protected variable
  def displayEmployee(self):
    print ("Name : ", self.name, ", age: ", self.__age, ", salary: ", self._salary)

e1=Employee("Bhavana", 24, 10000)

print (e1.name)
print (e1._salary)
print (e1.__age)
```

Q9. Discuss about Polymorphism in Python and how it is implemented.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It is the ability to redefine methods for derived classes. The term **polymorphism** refers to a function or method taking different forms in different contexts. Since Python is a dynamically typed language, polymorphism in Python is very easily implemented.

If a method in a parent class is overridden with different business logic in its different child classes, the base class method is a polymorphic method.

Ways of implementing Polymorphism in Python

There are four ways to implement polymorphism in Python –

- Duck Typing
- Operator Overloading
- Method Overriding
- Method Overloading

i.Duck Typing in Python

Duck typing is a concept where the type or class of an object is less important than the methods it defines. Using this concept, you can call any method on an object without checking its type, as long as the method exists.

Ex:-

```
class Duck:
  def sound(self):
    return "Quack, quack!"
class AnotherBird:
  def sound(self):
    return "I'm similar to a duck!"
def makeSound(duck):
  print(duck.sound())
```

```
# creating instances
duck = Duck()
anotherBird = AnotherBird()
# calling methods
makeSound(duck)
makeSound(anotherBird)
```

ii.Method Overriding in Python

In method overriding, a method defined inside a subclass has the same name as a method in its superclass but implements a different functionality.

```
from abc import ABC, abstractmethod
class shape(ABC):
  @abstractmethod
  def draw(self):
    "Abstract method"
    return
class circle(shape):
  def draw(self):
    super().draw()
    print ("Draw a circle")
    return
class rectangle(shape):
  def draw(self):
    super().draw()
    print ("Draw a rectangle")
    return
```

iii)Method Overloading in Python

When a class contains two or more methods with the same name but different number of parameters then this scenario can be termed as method overloading.

Python does not allow overloading of methods by default, however, we can use the techniques like variable-length argument lists, multiple dispatch and default parameters to achieve this.

Example:

```
def add(*nums):
  return sum(nums)
# Call the function with different number of parameters
result1 = add(10, 25)
result2 = add(10, 25, 35)
print(result1)
print(result2)
```

iv) Overloading Operators in Python

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you. we define the **__add__** method in your class to perform vector addition and then the plus operator would behave as per expectation –

```
class Vector:
  def __init__(self, a, b):
    self.a = a
    self.b = b
  def __str__(self):
    return 'Vector (%d, %d)' % (self.a, self.b)
  def __add__(self,other):
    return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
```

Q10. Explain about implementation of Polymorphism in Python using Dynamic Binding

In object-oriented programming, the concept of dynamic binding is closely related to polymorphism. In Python, dynamic binding is the process of resolving a method or attribute at runtime, instead of at compile time.

According to the polymorphism feature, different objects respond differently to the same method call based on their implementations. This behavior is achieved through method overriding, where a subclass provides its implementation of a method defined in its superclass.

The Python interpreter determines which is the appropriate method or attribute to invoke based on the object's type or class hierarchy at runtime. This means that the specific method or attribute to be called is determined dynamically, based on the actual type of the object.

Example:

```
class shape:
  def draw(self):
    print ("draw method")
    return

class circle(shape):
  def draw(self):
    print ("Draw a circle")
    return

class rectangle(shape):
  def draw(self):
    print ("Draw a rectangle")
    return

shapes = [circle(), rectangle()]
for shp in shapes:
  shp.draw()
```

Q11. Explain about Python Abstract Classes in Python.

Abstraction is one of the important principles of object-oriented programming. It refers to a programming approach by which only the relevant data about an object is exposed, hiding all the other details. This approach helps in reducing the complexity and increasing the efficiency of application development.

Types of Python Abstraction

There are two types of abstraction. One is data abstraction, wherein the original data entity is hidden via a data structure that can internally work through the hidden data entities. Another type is called process abstraction. It refers to hiding the underlying implementation details of a process.

Python Abstract Class

In object-oriented programming terminology, a class is said to be an abstract class if it cannot be instantiated, that is you can have an object of an abstract class. You can however use it as a base or parent class for constructing other classes.

Create an Abstract Class

To create an abstract class in Python, it must inherit the ABC class that is defined in the ABC module. This module is available in Python's standard library. Moreover, the class must have at least one abstract method. Again, an abstract method is the one which cannot be called but can be overridden. You need to decorate it with @abstractmethod decorator.

Example:-

```
from abc import ABC, abstractmethod

class democlass(ABC):

  @abstractmethod

  def method1(self):

    print ("abstract method")

    return

  def method2(self):

    print ("concrete method")
```

```
class concreteclass(democlass):
  def method1(self):
    super().method1()
    return

obj = concreteclass()
obj.method1()
obj.method2()
```

Q12. Explain about Interfaces and its implementation in Python.

In software engineering, an interface is a software architectural pattern. It is similar to a class but its methods just have prototype signature definition without any executable code or implementation body. The required functionality must be implemented by the methods of any class that inherits the interface.The method defined without any executable code is known as abstract method.

Interfaces in Python

Python doesn't have it or any similar keyword like interface . It uses abstract base classes (in short ABC module) and **@abstractmethod decorator** to create interfaces.

An abstract class and interface appear similar in Python. The only difference in two is that the abstract class may have some non-abstract methods, while all methods in interface must be abstract, and the implementing class must override all the abstract methods.

Rules for implementing Python Interfaces

We need to consider the following points while creating and implementing interfaces in Python –

- Methods defined inside an interface must be abstract.
- Creating object of an interface is not allowed.

- A class implementing an interface needs to define all the methods of that interface.
- In case, a class is not implementing all the methods defined inside the interface, the class must be declared abstract.

interfaces in Python are implemented using abstract base class (ABC). To use this class, you need to import it from the **abc** module.

Example

In this example, we are creating a formal interface with two abstract methods.

```
from abc import ABC, abstractmethod
# creating interface
class demoInterface(ABC):
  @abstractmethod
  def method1(self):
    print ("Abstract method1")
    return

  @abstractmethod
  def method2(self):
    print ("Abstract method1")
    return
```

Let us provide a class that implements both the abstract methods.

```
# class implementing the above interface
class concreteclass(demoInterface):
  def method1(self):
    print ("This is method1")
    return
  def method2(self):
    print ("This is method2")
    return
# creating instance
obj = concreteclass()
# method call
obj.method1()
obj.method2()
```

Q13. Explain about Exception Handling in Python.

In Python, syntax errors are among the most common errors encountered by programmers, especially those who are new to the language. This tutorial will help you understand what syntax errors are, how to identify them, and how to fix them.

Syntax Error

A syntax error in Python (or any programming language) is an error that occurs when the code does not follow the syntax rules of the language. Syntax errors are detected by the interpreter or compiler at the time of parsing the code, and they prevent the code from being executed.

These errors occur because the written code does not conform to the grammatical rules of Python, making it impossible for the interpreter to understand and execute the commands.

Common Causes of Syntax Errors

Following are the common causes of syntax errors –

- Missing colons (:) after control flow statements (e.g., if, for, while) – Colons are used to define the beginning of an indented block, such as in functions, loops, and conditionals.
- Incorrect indentation – Python uses indentation to define the structure of code blocks. Incorrect indentation can lead to syntax errors
- Misspelled keywords or incorrect use of keywords.
- Unmatched parentheses, brackets, or braces – Python requires that all opening parentheses (, square brackets [, and curly braces { have corresponding closing characters),], and }.

Exception Handling in Python

Exception handling in Python refers to managing runtime errors that may occur during the execution of a program. In Python, exceptions are raised when errors or unexpected situations arise during program execution, such as division by zero, trying to access a file that does not exist, or attempting to perform an operation on incompatible data types.

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them , they are Exception Handling and Assertions .

Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The assert Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The **syntax** for assert is –

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a trace back.

Example:-

```
def KelvinToFahrenheit(Temperature):
  assert (Temperature >= 0),"Colder than absolute zero!"
  return ((Temperature-273)*1.8)+32
print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
```

Exception

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python

script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an Exception in Python

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a **try**: block. After the **try**: block, include an **except**: statement, followed by a block of code which handles the problem as elegantly as possible.

- The **try**: block contains statements which are susceptible for exception
- If exception occurs, the program jumps to the **except**: block.
- If no exception in the **try**: block, the **except**: block is skipped.

Syntax:-

try:

You do your operations here

.....

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

.....

else:

If there is no exception then execute this block.

- A single **try** statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic **except** clause, which handles any exception.

- After the except clause(s), you can include an **else** clause. The code in the **else** block executes if the code in the try: block does not raise an exception.
- The **else** block is a good place for code that does not need the try: block's protection

Example:-

```
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
```

The except Clause with Multiple Exceptions

You can also use the same except statement to handle multiple exceptions as follows

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

The try-finally Clause

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this –

Syntax:-

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use else clause as well along with a finally clause.

Example

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print ("Error: can\'t find file or read data")
```

Standard Exceptions

Here is a list of Standard Exceptions available in Python –

Sr.No.	**Exception Name & Description**
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError**

	Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.

Q14. Explain about Data Structures and its types.

A **data structure** is a specialized format for organizing, storing, and managing data in a computer's memory to allow for efficient access and modification. The choice of a data structure significantly impacts the efficiency of a program's algorithms.

Data structures are generally classified into two primary categories: linear and non-linear, with further categorization into primitive and non-primitive types.

Primary Classification of Data Structures

1. Linear Data Structures

In a linear data structure, data elements are arranged sequentially or linearly, where each element is connected to its previous and next adjacent elements. Data traversal is straightforward, typically involving iterating from one element to the next.

Examples of linear data structures include:

- **Arrays:** Elements of the same data type are stored in contiguous memory locations, allowing for quick access using an index.
- **Linked Lists:** Elements (nodes) are linked by pointers and can be dynamic. Types include singly, doubly, and circular linked lists.
- **Stacks:** Uses the Last-In, First-Out (LIFO) principle for adding and removing elements from the top.

- **Queues:** Uses the First-In, First-Out (FIFO) principle, adding elements to the rear and removing them from the front.

2. Non-Linear Data Structures

These structures don't follow a simple sequence, allowing for more complex relationships between data elements. Traversal is typically more involved.

Examples of non-linear data structures include:

- **Trees:** Hierarchical structures with a root node and child nodes. Examples include binary trees and binary search trees (BSTs).
- **Graphs:** Composed of nodes (vertices) connected by links (edges), representing networks.
- **Hash Tables:** Use a hash function to map keys to an array index, enabling fast data retrieval.

An array is a type of linear data structure that is defined as a collection of elements with same or different data types. They exist in both single dimension and multiple dimensions. These data structures come into picture when there is a necessity to store multiple elements of similar nature together at one place

Need for Arrays

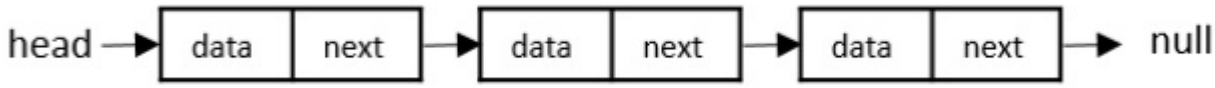
Arrays are used as solutions to many problems from the small sorting problems to more complex problems like travelling salesperson problem. There are many data structures other than arrays that provide efficient time and space complexity for these problems, so what makes using arrays better? The answer lies in the random access lookup time.

Arrays provide **O(1)** random access lookup time. That means, accessing the 1^{st} index of the array and the 1000^{th} index of the array will both take the same time. This is due to the fact that array comes with a pointer and an offset value. The pointer points to the right location of the memory and the offset value shows how far to look in the said memory.

Q15. Explain about Linked List Data structure.

A linked list is a linear data structure which can store a collection of "nodes" connected together via links i.e. pointers. Linked lists nodes are not stored at a contiguous location, rather they are linked using pointers to the different memory locations. A node consists of the data value and a pointer to the address of the next node within the linked list.

A linked list is a dynamic linear data structure whose memory size can be allocated or de-allocated at run time based on the operation insertion or deletion, this helps in using system memory efficiently. Linked lists can be used to implment various data structures like a stack, queue, graph, hash maps, etc.



A linked list starts with a head node which points to the first node. Every node consists of data which holds the actual data (value) associated with the node and a next pointer which holds the memory address of the next node in the linked list. The last node is called the tail node in the list which points to null indicating the end of the list.

Linked Lists vs Arrays

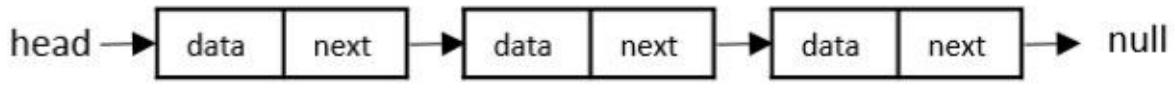
In case of arrays, the size is given at the time of creation and so arrays are of fixed lenghth where as Linked lists are dynamic in size and any number of nodes can be added in the linked lists dynamically. An array can accommodate similar types of data types where as linked lists can store various nodes of different data types.

Types of Linked List

Following are the various types of linked list.

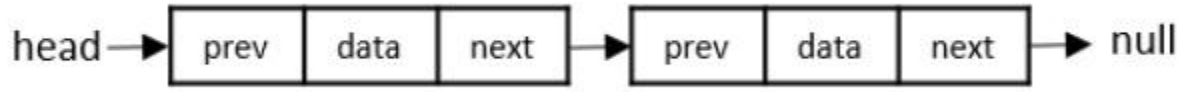
Singly Linked Lists

Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.



Doubly Linked Lists

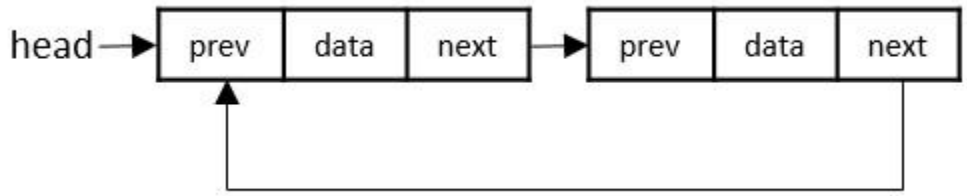
Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.



Circular Linked Lists

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



Operations in Linked List

Insertion at Beginning

In this operation, we are adding an element at the beginning of the list.

1. START
2. Create a node to store the data
3. Check if the list is empty
4. If the list is empty, add the data to the node and
   assign the head pointer to it.
5. If the list is not empty, add the data to a node and link to the
   current head. Assign the head to the newly added node.
6. ENDAlgorithm

Ex:-

```
class Node:
  def __init__(self, data=None):
    self.data = data
    self.next = None
class SLL:
  def __init__(self):
    self.head = None

# Print the linked list
  def listprint(self):
    printval = self.head
    print("Linked List: ")
    while printval is not None:
      print (printval.data)
      printval = printval.next
  def AddAtBeginning(self,newdata):
    NewNode = Node(newdata)

    # Update the new nodes next val to existing node
    NewNode.next = self.head
    self.head = NewNode
l1 = SLL()
l1.head = Node("731")
e2 = Node("672")
e3 = Node("63")

l1.head.next = e2
e2.next = e3

l1.AddAtBeginning("122")
l1.listprint()
```

Insertion at Ending

In this operation, we are adding an element at the ending of the list.

Algorithm

1. START

2. Create a new node and assign the data
3. Find the last node
4. Point the last node to new node
5. END

Python Program

```
class Node:
  def __init__(self, data=None):
    self.data = data
    self.next = None
class LL:
  def __init__(self):
    self.head = None
  def listprint(self):
    val = self.head
    print("Linked List:")
    while val is not None:
      print(val.data)
      val = val.next

l1 = LL()
l1.head = Node("23")
l2 = Node("12")
l3 = Node("7")
l4 = Node("14")
l5 = Node("61")
# Linking the first Node to second node
l1.head.next = l2
# Linking the second Node to third node
l2.next = l3
l3.next = l4
l4.next = l5
l1.listprint()
```

Deletion in linked lists is also performed in three different ways. They are as follows

Deletion at Beginning

In this deletion operation of the linked, we are deleting an element from the beginning of the list. For this, we point the head to the second node.

Algorithm:

1. START
2. Assign the head pointer to the next node in the list
3. END

```
from typing import Optional
class Node:
    def __init__(self, data: int, next: Optional['Node'] = None):
        self.data = data
        self.next = next
class LinkedList:
    def __init__(self):
        self.head = None
     #display the list
    def print_list(self):
        p = self.head
        print("\n[", end="")
        while p:
            print(f" {p.data} ", end="")
            p = p.next
        print("]")
     #Insertion at the beginning
    def insert_at_begin(self, data: int):
        lk = Node(data)
         #point it to old first node
        lk.next = self.head
        #point firt to new first node
        self.head = lk
    def delete_at_begin(self):
        self.head = self.head.next
if __name__ == "__main__":
    linked_list = LinkedList()
    linked_list.insert_at_begin(12)
    linked_list.insert_at_begin(22)
```

```
linked_list.insert_at_begin(30)
linked_list.insert_at_begin(44)
linked_list.insert_at_begin(50)
#print list
print("Linked List: ", end="")
linked_list.print_list()
linked_list.delete_at_begin()
print("Linked List after deletion: ", end="")
linked_list.print_list()
```

Linked List - Search Operation

Searching for an element in the list using a key element. This operation is done in the same way as array search; comparing every element in the list with the key element given.

Algorithm

1 START
2 If the list is not empty, iteratively check if the list
contains the key
3 If the key element is not present in the list, unsuccessful
search
4 END
Example c

```
lass Node:
  def __init__(self, data=None):
    self.data = data
    self.next = None
class SLL:
  def __init__(self):
    self.head = None

# Print the linked list
  def listprint(self):
    printval = self.head
    print("Linked List: ")
    while printval is not None:
      print (printval.data)
      printval = printval.next
```

```
l1 = SLL()
l1.head = Node("731")
e2 = Node("672")
e3 = Node("63")

l1.head.next = e2
e2.next = e3

l1.listprint()
```

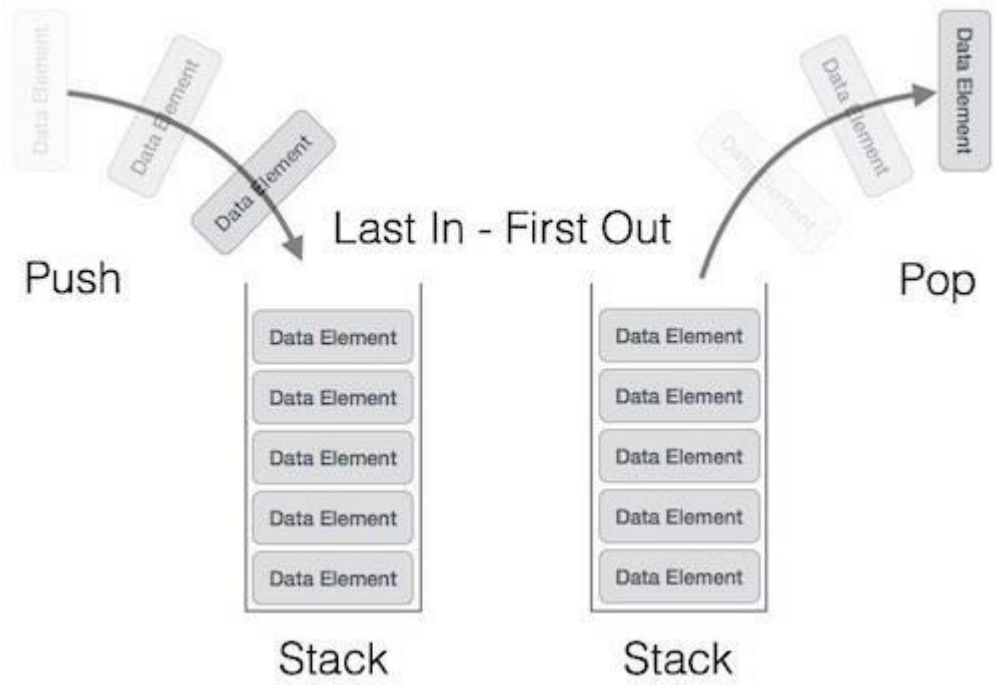
Q16. Explain about Stack Operations

A stack is a linear data structure where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted. A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages. It is named stack because it has the similar operations as the real-world stacks, for example – a pack of cards or a pile of plates, etc.

Stack Representation

A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations on Stacks

Stack operations are usually performed for initialization, usage and, de-initialization of the stack ADT.

The most fundamental operations in the stack ADT include: push(), pop(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the stack.Stack uses pointers that always point to the topmost element within the stack, hence called as the top pointer.

Stack Insertion: push()

The push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.

Algorithm

1. Checks if the stack is full.
2. If the stack is full, produces an error and exit.
3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.

Stack Deletion: pop()

The pop() is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way.

Algorithm

1. Checks if the stack is empty.
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.
5. Returns success.

Retrieving topmost Element from Stack: peek()

The peek() is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.

Algorithm

1. START
2. return the element at the top of the stack
3. END

Verifying whether the Stack is full: isFull()

The isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the size of the stack is equal to the top position of the stack,
   the stack is full. Return 1.
3. Otherwise, return 0.
4. END

Python Implementation

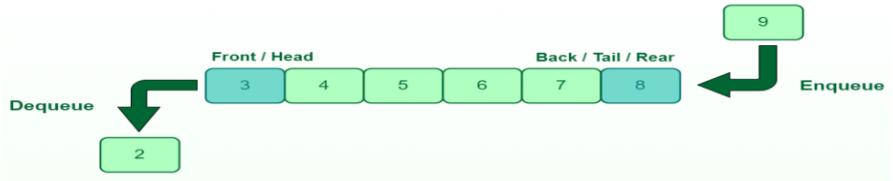
```
MAXSIZE = 8
stack = [0] * MAXSIZE
top = -1;

def isempty():
   if(top == -1):
      return 1
   else:
      return 0

def isfull():
   if(top == MAXSIZE):
      return 1
   else:
      return 0

def peek():
   return stack[top]

def pop():
   global data, top
   if(isempty() != 1):
      data = stack[top];
      top = top - 1;
      return data
   else:
      print("Could not retrieve data, Stack is empty.")
   return data
```

```
def push(data):
    global top
    if(isfull() != 1):
        top = top + 1
        stack[top] = data
    else:
        print("Could not insert data, Stack is full.")
    return data
#-----------------------------MAIN PROGRAM-------------
push(44)
push(10)
push(62)
push(123)
push(15)
print("Element at top of the stack: ", peek())
print("Elements: ")
while(isempty() != 1):
    data = pop();
    print(data, end = " ")
print("\nStack full: ",bool({True: 1, False: 0} [isfull() == 1]))
print("Stack empty: ",bool({True: 1, False: 0} [isempty() == 1]))
```

Q17. Explain about QUEUE operations

Queue is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed. A queue is an Abstract Data Type (ADT) similar to stack, the thing that makes queue different from stack is that a queue is open at both its ends. The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.



Queue Data Structure

Representation of Queues

Similar to the stack ADT, a queue ADT can also be implemented using arrays, linked lists, or pointers. As a small example in this tutorial, we implement queues using a one-dimensional array.

Basic Operations in Queue

Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.The most fundamental operations in the queue ADT include: **enqueue(), dequeue(), peek(), isFull(), isEmpty().** These are all built-in operations to carry out data manipulation and to check the status of the queue.

Queue uses two pointers – front and rear. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

Queue Insertion Operation: Enqueue()

The enqueue() is a data manipulation operation that is used to insert elements into the stack. The following algorithm describes the enqueue() operation in a simpler way.

Algorithm

1. START
2. Check if the queue is full.
3. If the queue is full, produce overflow error and exit.
4. If the queue is not full, increment rear pointer to point the next empty space.
5. Add data element to the queue location, where the rear is pointing.
6. return success.
7. END

Queue Deletion Operation: dequeue()

The dequeue() is a data manipulation operation that is used to remove elements from the stack. The following algorithm describes the dequeue() operation in a simpler way.

Algorithm

1. START
2. Check if the queue is empty.
3. If the queue is empty, produce underflow error and exit.
4. If the queue is not empty, access the data where front is pointing.
5. Increment front pointer to point to the next available

data element.
6. Return success.
7. END

Queue - The peek() Operation

The peek() is an operation which is used to retrieve the frontmost element in the queue, without deleting it. This operation is used to check the status of the queue with the help of the pointer.

Algorithm

1. START
2. Return the element at the front of the queue
3. END

Queue - The isFull() Operation

The isFull() operation verifies whether the stack is full.

Algorithm

1. START
2. If the count of queue elements equals the queue size,
   return true
3. Otherwise, return false
4. END

Queue - The isEmpty() operation

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the count of queue elements equals zero, return true
3. Otherwise, return false
4. END

Queue Implementation in Python

```
MAX = 6
intArray = [0] * MAX
front = 0
rear = -1
itemCount = 0
def peek():
   return intArray[front]
def isEmpty():
   return itemCount == 0

def isFull():
   return itemCount == MAX
```

```
def size():
    return itemCount

def insert(data):
    global rear, itemCount
    if not isFull():
        if rear == MAX-1:
            rear = -1
        rear += 1
        intArray[rear] = data
        itemCount += 1

def removeData():
    global front, itemCount
    data = intArray[front]
    if front == MAX-1:
        front = 0
    else:
        front += 1
    itemCount -= 1
    return data
#---------------------------------MAIN----------------
def main():
    insert(3)
    insert(5)
    insert(9)
    insert(1)
    insert(12)
    insert(15)
    print("Queue size: ", size())
    print("Queue: ")
    for i in range(MAX):
        print(intArray[i], end = " ")
    if isFull():
        print("\nQueue is full!")
    num = removeData()
    print("Element removed: ", num)
    print("Queue size after deletion: ", size())
    print("Element at front: ", peek())

main()
```

Q19. Explain about DQueue Operations.

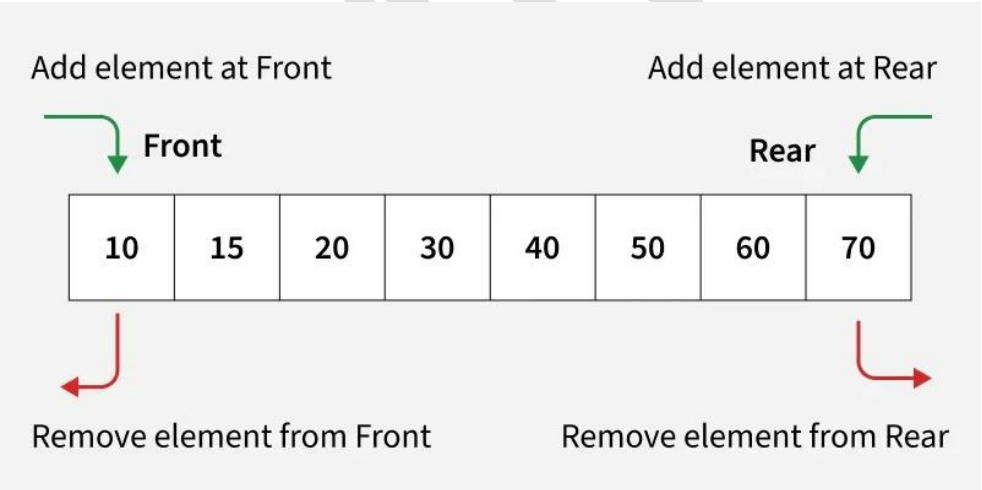
Deque is a hybrid data structure that combines the features of a stack and a queue. It allows us to insert and delete elements from both ends of the queue. The name Deque is an abbreviation of Double-Ended Queue.

Imagine an event where you have two gates to enter and exit a place. People are entering from the front gate and some are entering from the side gate. Now, when people are leaving, they are leaving from the front gate and some sneak from the side gate. Now, we need to manage flow of people from both ends. This is where Deque comes into play.

Operations on Deque

Following are the major operations on Deque –

push_front(x): Insert element x at the front of the deque.
push_back(x): Insert element x at the back of the deque.
pop_front(): Remove the element from the front of the deque.
pop_back(): Remove the element from the back of the deque.
peek_front(): Get the element from the front of the deque.
peek_back(): Get the element from the back of the deque.
size(): Get the number of elements in the deque.
isEmpty(): Check if the deque is empty.



Implementation of Deque

Let's understand how we can implement deque using array. For this, we need to maintain two pointers, front and rear, to keep track of the front and back of the deque. We also need to define the size of the deque.

The push_front(x) Operation on Deque
When we insert an element at the front of the deque, we need to shift all the elements to the right by one position. We will increment the front pointer by one and insert the element at the front of the deque.

Algorithm for push_front(x)
Following are the steps to insert an element at the front of the deque –
1. Check if the deque is full. If it is full, return an error message.
2. Increment the front pointer by one.
3. Insert the element at the front position.
4. Increment the size of the deque.

push_back(x) Operation
This operation is used for inserting an element to the back of the deque. When we insert an element at the back of the deque, we need to increment the rear pointer by one and insert the element at the back of the deque.

Algorithm for push_back(x)
Following are the steps to insert an element at the back of the deque –

1. Check if the deque is full.
2. Increment the rear pointer by one.
3. Insert the element at the rear position.
4. Increment the size of the deque.

The pop_front() and pop_back() Operations on Deque
These operation is done when we need to remove elements from front or back. When we remove an element from the front of the deque, we need to increment the front pointer by one.

Similarly, when we remove an element from the back of the deque, we need to decrement the rear pointer by one.

Algorithm for pop_front() and pop_back()
Following are the steps to remove an element from the front or back of the deque –

1. Check if the deque is empty.
2. Remove the element from the front or back of the deque.
3. Increment or decrement the front or rear pointer.
4. Decrement the size of the deque.

The peek_front() and peek_back() Operations on Deque

When we want to get the element from the front or back of the deque, we can use the peek_front() and peek_back() operations.

Algorithm for peek_front() and peek_back()

Following are the steps to get the element from the front or back of the deque –

1. Check if the deque is empty.
2. If not empty, return the element from the front or back of the deque.

Applications of Deque

Some of the applications of deque are as follows –

- **Deque** is used for **undo** operation in text editors.
- It is also used in implementation of the **sliding window algorithm**.
- **Deque** is used in implementing the data structures like **double-ended priority queue** and **double-ended stack**.

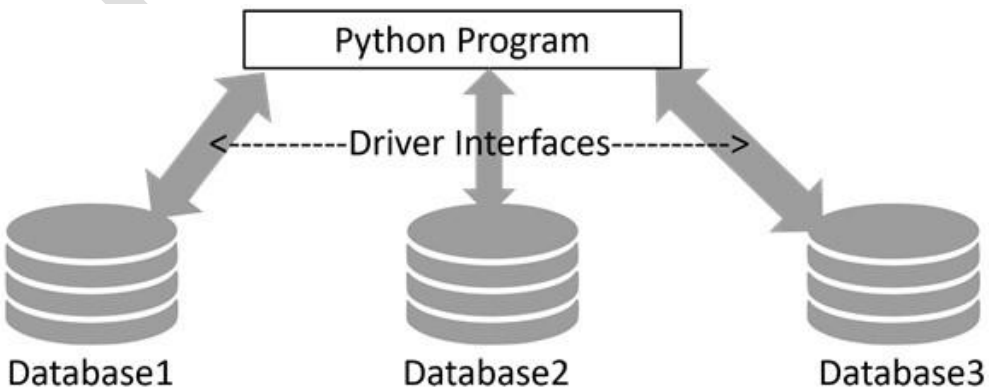
In summary, we use **deque** when we need to perform insertion and deletion operations at both ends of the data structure.

Q20. Explain Database Connection in Python.

Python being a high-level language provides support for various databases. We can connect and run queries for a particular database using Python and without writing raw queries in the terminal or shell of that particular database, we just need to have that database installed in our system.

DB-API (Database API)

To address this issue of compatibility, Python Enhancement Proposal (PEP) 249 introduced a standardized interface known as DB-API. This interface provides a consistent framework for database drivers, ensuring uniform behavior across different database systems. It simplifies the process of transitioning between various databases by establishing a common set of rules and methods.



The process generally follows these four key steps:

The Connection Process

1. Import the Driver: You must first import the appropriate Python module for your specific database. While Python includes the sqlite3 module in its standard library, other databases like MySQL or PostgreSQL require installing a third-party library (e.g., mysql-connector-python or psycopg2) using pip.
2. Establish a Connection: Use the driver's connect() method to establish a connection to the database server. This function typically requires credentials such as the host, user, password, and database name. This method returns a connection object which manages the session and transactions.
3. Create a Cursor: From the connection object, you create a cursor object using the cursor() method. The cursor acts as an interface for executing SQL commands and fetching results from the database.
4. Execute Queries and Close: You can now execute SQL statements (CRUD operations) using the cursor's execute() method. After operations are complete, you should commit any changes using connection.commit() and then close both the cursor and the connection to free up resources using the close() methods.

Python MySQL Connector is a Python driver that helps to integrate Python and MySQL. This Python MySQL library allows the conversion between Python and MySQL data types. MySQL Connector API is implemented using pure Python and does not require any third-party library.

Implementation in Python

```
import mysql.connector

try:
    # 1. Establish the connection
    connection = mysql.connector.connect(
        host="localhost",
        user="yourusername",
        password="yourpassword",
        database="yourdatabase"
    )

    if connection.is_connected():
        # 2. Create a cursor object
        cursor = connection.cursor()

        # 3. Execute a query
```

```
        cursor.execute("SELECT * FROM your_table")

        # Fetch all results and iterate
        records = cursor.fetchall()
        for row in records:
            print(row)

except mysql.connector.Error as e:
    print(f"Error connecting to MySQL database: {e}")

finally:
    # 4. Close the connection
    if connection.is_connected():
        cursor.close()
        connection.close()
        print("MySQL connection is closed")
```