* **Software Evolution**

Many programming approaches have been tried since the invention of the computer that includes **Modular Programming, Top-Down Programming, Bottom-up Programming** & **Structured Programming.** The main aim of each of these approaches is to handle the increasing complexity of programs that are reliable and maintainable.

If you look at the history of software development, you can classify them into the following categories (see figure 1).

* Machine Language
* Assembly Language
* Procedure-oriented Language
* Object Oriented Programming Language

0, 1

Machine Language

Assembly Language (Assembler)

Object-oriented Programming

Procedure oriented Language

(Compiler & Interpreter)

**Figure 1**

**Machine Language:**

Initially all the programs were written in the form of 0 & 1 i.e. binary language or machine language. The only advantage with machine language was to eliminate the human errors. But these programs were difficult to understand and maintain, time-consuming and error-prone.

In spite of all these difficulties, it has laid the foundation of the new era of programming.

**Assembly Language:**

They are the next level in improvement over m/c language. They are also known as symbolic language. The tedious task of remembering **opcodes** in m/c language has been replaced with mnemonic symbols. This has eased the task of programming greatly. The programs written in assembly languages must have to be translated in m/c language, since computer understands m/c codes only.

This task is achieved by programs known as **Assemblers**. It simply translated each line of assembly program into m/c language. Thus, there is one-to-one mapping for each instruction between them.

**Advantages:**

1. Improved Readability of program than that of m/c level program.
2. Easier to debug.
3. Efficient in terms of execution time & main memory usage.
4. One can fully exploit the capabilities of the computer.
5. It occupies minimum storage.

**Disadvantages:**

1. It offers little compatibility i.e. a program written for one computer may or may not work on another computer.
2. It requires writing a large number of statements to solve a given problem than that in HLL.
3. The assembly program is often difficult to write, read & maintain than program written in HLL.
4. Not useful for writing general-purpose applications.

**Procedure-Oriented Languages:**

They are called 3rd Generation Languages. They are also known as high level language (HLL) or **compiler languages**. They employ plain English. They have an extensive vocabulary of words and symbols. They are used to instruct a computer to carry out the necessary procedures, regardless of the type of machine being used.

As with a program in assembly language, the HLL program has to be translated into m/c code before it can be used. This is done by Compiler Language which compiles the source program into the object program. This involves translating the macro instructions, identifying syntax and logic error, providing a listing of object program for testing, and writing the object program onto hard disk. Once compiled and linked, this program is ready for use.

Some of the compiler languages are Basic, Pascal, C, COBOL etc.

The HLL program can be converted into m/c code with the help of **interpreter** also. Interpreter translates each line of program at a time and generates directly executable m/c code. This is in contrast with compiler. (See differences: Compiler and Interpreter.)

In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as input, process and output. These tasks are accomplished with the help of number of functions and thus the main focus is on functions. (See figure 2)

Main Program

Function - 2

Function - 1

Function - 3

Function - 4

Function - 5

Function - 1

Local Data

Function - 2

Local Data

Global Data

Global Data

**Figure 2 Figure 3**

Each function consists of group of instructions that carries out various actions like input, process and output.

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. The relationship of data and functions in procedural programming is shown in figure 3. Global data can be changed by any of these functions and thus provides opportunity for bugs in the program.

**Some characteristics exhibited by procedure-oriented programming languages are:**

* Emphasis is on doing things. (algorithms)
* Large programs are divided into smaller programs known as functions.
* Most of the functions share global data.
* Data move openly around the system from function to function.
* Functions transform data from one form to another.
* Employs top-down approach in program design.

**Object Oriented Programming Language:**

The next level of improvement is carried out with the help of Object Oriented Programming Language. Some of the flow encountered in procedure oriented language has been removed. OOPL not only allows data to move freely but protects data from accidental changes outside the system. It allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects (figure 4).

DATA

FUNCTIONS

Object A

DATA

FUNCTIONS

Object B

DATA

FUNCTIONS

Object C

**Figure 4**

**Important features of OOPL:**

* Emphasis is on data rather than on procedure.
* Programs are divided into objects.
* Data structures are designed such that they characterize the objects.
* Functions that operate on the data of an object are tied together in the data structure.
* Data is hidden and cannot be accessed by external functions.
* Objects may communicate with each other through functions.
* New data and functions can be easily added whenever necessary.
* Follows bottom-up approach in program design.

**Basic Concepts of OOP:**

Some of the important concepts used extensively in OOP are:

* Objects
* Classes
* Data Abstraction and Encapsulation
* Inheritance
* Polymorphism
* Dynamic Binding
* Message Passing

**Objects:**

Objects are the basic run-time entities in an object-oriented system. You take any real-world object. Example includes: A person, a place, a bird, a table, a bank account, vectors, time and lists, etc. Problem is analyzed in terms of objects and the nature of communication between them.

When a program is executed, the objects interact by sending messages to one another. Each object contains data and functions to manipulate the data. Objects can interact without having to know details of each other's data or function.

Object: Circle

Data

Radius

Area

Functions

Set\_radius( )

Calc\_area( )

Circle

Set\_radius()

Calc\_area()

**Figure 5**

Two popular notations used to represent the object are mentioned in figure 5.

**Class:**

**A class is a collection of objects of similar type.** Each object is associated with the data of type class with which they are created. In another words, objects are variables of the type class. E.g. objects sparrow, pigeon, peacock are members of the class bird. Class is a user-defined data type but behaves like the built-in data types of a programming language.

**Examples of Classes and Objects:**

Bird is a class. Sparrow, Hen, Peacock are objects of Bird class.

Player is a class. Sachin, Sehvag, P V Sindhu are objects of Player class.

Flower is a class. Rose, Lily, Asopalav are objects of Flower class.

Instrument is a class. Sitar, Flute, Violin are objects of Instrument class.

Thus, class is generic in nature, whereas an object is specific in nature.

Multiple objects can be created from a class. The process of creating object from a class is known as Instantiation.

**Data Abstraction and Encapsulation:**

**The wrapping up of data and functions into a single unit (called class) is known as encapsulation.** Here, the data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. Thus we can hide the data.

**Abstraction refers to the act of representing essential features without including the background details or explanations.** You are driving a car. You know the essential features like gears, brakes, steering etc. but you don’t know about how engine is working. You only start the switches but you don’t know how it is working internally. This is known as abstraction.

Since the classes use the concept of data abstraction, they are also known as **Abstract Data Types (ADT)**.

**Inheritance:**

**Inheritance is the process by which objects of one class acquire the properties of objects of another class (**See figure 6)**.** The class which is defined first, whose properties and behavior are used by other classes is known as Base Class. The class which inherits the properties of Base Class is known as Derived Class. You can add new features into derived class without modifying the base class.

Animal

Mammal

Reptile…

Canine

Feline…

Domesticus

Lupus…

Retriever

Poodle…

Labrador

Golden

**Figure 6**

**Polymorphism:**

Polymorphism (from the Greek, meaning "many forms") means an operation may show different behaviours in different situations. Polymorphism is achieved by function overloading or operator overloading.

**Function overloading**: **Using a single function name to perform different types of tasks**. E.g. from class shape, we derive three different types of objects called square, rectangle and circle. Class shape has function called calc\_area. calc\_area function will take into account only length for square object, length and breadth for rectangle object and radius for circle object to calculate area.

**Operator overloading:** Here, **same operator is used to work differently in different situations**. E.g. 70 + 80 generates 150 as output while "Pandit Deendayal " + " Energy University" will generate "Pandit Deendayal Energy University " as output.

Polymorphism is extensively used in implementing inheritance.

**Dynamic Binding:**

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known at the time of compiling a program but at the time of the call at run-time. It is associated with polymorphism and inheritance.

**Message Passing:**

An object-oriented program consists of a set of objects that communicate with each other by sending and receiving information (see figure 4). Message passing involves specifying the name of the object, the name of the function (procedure/method) and the information to be sent (argument).

E.g. A.set\_radius(5);

Here A is the object. We are asking to call the set\_radius function for object A. 5 is the message passed to the object A via function set\_radius.

Each object has its own life cycle. It can be created and destroyed. Communication is feasible only if it is alive. Message passing techniques make the interface descriptions with external systems much simpler.

**Benefits of OOP:**

OOP offers several benefits to both the program designer and the user. It helps solve many problems associated with the development and quality of software products. The benefits are:

* Reduction in Program Development Time (Greater Programmer Productivity),
* Lesser Maintenance Cost,
* Better Quality of Software,
* Reuse of existing classes (Inheritance),
* Removal of duplicate code (Inheritance),
* Secure programs as data is hidden,
* Multiple instances of an object co-exist without any interference,
* Easier to partition the work in a project based on object,
* Interface with external system is much simpler (Message Passing),
* Software complexity (from small to large) can be easily managed,

**Classes and objects in Programming:**

* In Python every type is a class. so **int**, **float**, **complex**, **bool**, **str**, **list**, **tuple**, **set**, **dict** are all classes.
* A class has a name, whereas objects are nameless. Since objects do not have names, they are referred using their addresses in the memory.
* When we use simple statement **num = 20**, a nameless object of type **int** is created in the memory and its address is stored in **num.** Thus **num** refers to or points to the nameless object containing value 20.
* However, instead of saying that **num** refers to a nameless **int** object, often for sake of convenience, it is said that **num** is an **int** object.
* Different objects of a particular type may contain different data, but same methods.
  + s1 = “Darshit”
  + s2 = “Ragi”
  + Here, s1 and s2, both are object of type str but they have same methods like **upper()**, **capitalize()**, **lower()**, etc.

**User-defined Classes:**

* We can create our own classes and create objects from them apart from classes like **int**, **float**, **tuple**, etc.
* Rules for defining and using a user-defined class and a standard class are same.

class number:

def set\_num(self,n):

self.num = n

def get\_num(self):

* class number contains different methods like set\_num,

get\_num, print\_num, isnegative, isdivisibleby, abs\_val.

* a = number() 🡨 creates one nameless object of class number.
* The address of the nameless object is stored in **a**.
* Generally, each object will have its own data but methods are

shared amongst objects.

* The Syntax to call an object’s method is object.method()
* Whenever we call an instance method using an object, address

of the object gets passed to the method implicitly and collected

by variable **self**.

* Self is like **this** pointer of C++ or **this** reference of Java. Any

variable can be used in place of self.

**Access Convention**

* We can access class methods as well as variables from outside

the class.

* + c = number()
  + c.num = 25
  + print (c.num)

return self.num

def \_\_init\_\_(self):

self.num = 0

def print\_num(self):

print (self.num)

def isnegative(self):

if self.num < 0:

return "-ve"

elif self.num > 0:

return "+ve"

else:

return "Zero"

def isdivisibleby(self,n):

if self.num % 2 == 0:

return "Yes, divisible"

else:

return "No, not divisible"

def abs\_val(self):

if self.num < 0:

return -(self.num)

else:

return self.num

def \_\_del\_\_(self):

print ("deleting Object:",str(self))

x = int(input("Enter a number:"))

a = number()

a.set\_num(x)

b = a.get\_num()

print ("You entered value :",b)

a.print\_num()

print(a.isnegative())

print(a.isdivisibleby(10))

print("Absolute Value: ", a.abs\_val())

a = None

# Accessing object variables from outside the class.

# Ideally one should not do that but Python allows it.

c = number()

c.num = 25

print (c.num)

d = number()

print (d.num)

**Object Initialization:**

* **Two ways:**
  + using methods like set\_num().
  + using special method \_\_init\_\_()
* **\_\_init\_\_():**
  + **\_\_init\_\_()** is always called when an object is created. So address of the object is passed to **\_\_init\_\_()**.
  + **\_\_init\_\_()** parameters can take default values.
  + **\_\_init\_\_()** doesn’t return any value.
  + If we don’t define **\_\_init\_\_()** , then Python inserts a default **\_\_init\_\_()** method in user-defined class.
  + **\_\_init\_\_()** is called only once during entire lifetime of an object.
  + A class can have **\_\_init\_\_()** as well as other user-defined functions like set\_num() in above example.
    - **\_\_init\_\_()** 🡪 to initialize object.
    - **set\_num() 🡪** to modify an already initialized object.
  + **\_\_init\_\_()** method is similar to constructor function of C++/Java.

**Object Deletion:**

* **\_\_del\_\_():**
  + **\_\_del\_\_()** method gets called automatically when an object goes out of scope. Cleanup activity, if any, should be done in **\_\_del\_\_().**
  + **\_\_del\_\_()** is similar to destructor function of C++.

**Class Variables and Methods:**

* If we wish to share a variable amongst all objects of a class, we must declare the variable as a **class variable** or **class attribute.**
* A variable without prepending it with **self** within a class, is a **class variable.**
* It doesn’t become part of objects of a class.
* Use syntax **classname.varname** to use it.
* **Class methods** don’t receive a **self** argument.
* Use syntax **classname.methodname** to access it.
* Class variables can be used to count how many objects have been created from a class.
* Class variables and methods are like static members in C++/java.

**vars() and dir() Functions:**

* There are two useful built-in functions **vars()** and **dir().**
* **vars()** returns a dictionary of attributes and their values, whereas **dir()** returns a list of attributes.
* Both the built-in functions can be used with a class as well as an object.
* When used with class, vars() returns a dictionary of the class’s attributes and their values and dir() that of only attributes.
* When used with object, vars() returns a dictionary of the object’s attributes and their values. In addition, it also returns the object’s class’s attributes and recursively the attributes of its class’s base classes.
* When used with object, dir() returns a list of the object’s attributes, object’s class’s attributes and recursively the attributes of its class’s base classes.

import math

a = 184

nm = "PDEU"

class fruit:

count = 0

def \_\_init\_\_(self,name = '', size = 0, color = ''):

self.name = name

self.size = size

self.color = color

fruit.count += 1

def display():

print(fruit.count)

f1 = fruit('Banana',4,'Yellow')

print("print(vars(fruit))")

print(vars(fruit))

print()

print("print(vars(f1))")

print(vars(f1))

print()

print("print (dir(fruit))")

print (dir(fruit))

print()

print("print (dir(f1))")

print (dir(f1))

print()

print("print (vars())")

print (vars())

print()

print("print (vars(math))")

print (vars(math))

print()

print("print (dir())")

print (dir())

print()

print("print (dir(math))")

print (dir(math))

* Unlike C++ and Java, Python does not have keywords private, protected or public to mark the attributes.

**Operator Overloading:**

* + Since **Complex** is a user-defined class, Python doesn’t know how to add objects of this class. We can teach it how to do it, by overloading the + operator as shown below:

class Complex:

def \_\_init\_\_(self, r = 0.0, i = 0.0):

self.real = r;

self.imag = i

def \_\_add\_\_(self,other):

z = Complex()

z.real = self.real + other.real

z.imag = self.imag + other.imag

return z

def \_\_sub\_\_(self,other):

z = Complex()

z.real = self.real - other.real

z.imag = self.imag - other.imag

return z

def display(self):

print(self.real, self.imag)

c1 = Complex(1.1,0.2)

c2 = Complex(2.2,0.4)

c3 = c1 + c2

c3.display()

c4 = c1 - c2

c4.display()

* + To overload the **+ operator** we need to define **\_\_add\_\_( )** function within the **Complex** class.
  + Likewise, to overload the **- operator** we need to define **\_\_sub\_\_( )** function within the **Complex** class.
  + In the expression c3 = c1 + c2, c1 becomes available in **self,** whereas **c2** is collected in **other.**

**Which Operators to Overload?:**

**Arithmetic Operators**

+ \_\_add\_\_(self, other)

- \_\_sub\_\_(self, other)

\* \_\_mul\_\_(self, other)

/ \_\_truediv\_\_(self, other)

% \_\_mod\_\_(self, other)

\*\* \_\_pow\_\_(self, other)

// \_\_floordiv\_\_(self, other)

**Comparision Operators**

< \_\_lt\_\_(self, other)

> \_\_gt\_\_(self, other)

<= \_\_le\_\_(self, other)

>= \_\_ge\_\_(self, other)

== \_\_eq\_\_(self, other)

!= \_\_ne\_\_(self, other)

**Compound Assignment Operators**

= \_\_isub\_\_(self, other)

+= \_\_iadd\_\_(self, other)

\*= \_\_imul\_\_(self, other)

/= \_\_idiv\_\_(self, other)

//= \_\_idiv\_\_(self, other)

%= \_\_imod\_\_(self, other)

\*\*= \_\_ipow\_\_(self, other)

**Function Overloading**

* Unlike many other languages like C++, Java, etc., Python doesn’t support function overloading. It means function names in a program, or method names within a class should be unique.
* If we define two functions or methods by same name, we won’t get an error message, but the latest version would prevail.
* **Write following programs considering classes and objects in mind:**

1. Write a program to create a class that represents Complex numbers containing real and imaginary parts and then use it to perform complex number addition, subtraction, multiplication and division.
2. Write a program that implements a Matrix class and performs addition, multiplication and transpose operations on 3x3 matrices.
3. Write a program to create a class that can calculate the surface area and volume of a solid. The class should also have a provision to accept the data relevant to the solid.
4. Write a program to create a class that can calculate the perimeter/circumference and area of a regular shape. The class should also have a provision to accept the data relevant to the shape.
5. Write a program that creates and uses a Time class to perform various time arithmetic operations.
6. Write a program to create a class **Date** that has a list containing day, month and year attributes. Define an overloaded == operator to compare two **Date** objects.
7. Create a class **Weather** that has a list containing weather parameters. Define an overloaded **in** operator that checks whether an item is present in the list. (Hint: define the function **\_\_contains\_\_( )**in a class.)
8. Implement a **String** class containing the following functions:
   1. Overloaded += operator function to perform string concatenation
   2. Method **toLower()** to convert upper case letters to lower case.
   3. Method **toUpper()** to convert lower case letters to upper case.