

UNIT 1 PROGRAMMING FUNDAMENTALS

Structure

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Problem - Solving Techniques
 - 1.2.1 Steps for Problem - Solving
 - 1.2.2 Using Computer as a Problem-Solving Tool
- 1.3 Basics of Algorithms
 - 1.3.1 Definition
 - 1.3.2 Features of Algorithm
 - 1.3.3 Criteria to be followed by an Algorithm
 - 1.3.4 Top-Down Design
- 1.4 Flowcharts
 - 1.4.1 Basic Symbols used in Flowchart Design
- 1.5 Program and a Programming Language
- 1.6 Structured Programming Concepts
- 1.7 C Programming Language
 - 1.7.1 History of C Programming Language
 - 1.7.2 Salient Features of C
- 1.8 Writing a C Program
- 1.9 Compiling a C Program
 - 1.9.1 The C Compiler
 - 1.9.2 Syntax and Semantic Errors
- 1.10 Link and Run the C Program
 - 1.10.1 Run the C Program through the Menu
 - 1.10.2 Run from an Executable File
 - 1.10.3 Linker Errors
 - 1.10.4 Logical and Runtime Errors
- 1.11 Diagrammatic Representation of C Program Execution Process
- 1.12 Summary
- 1.13 Solutions / Answers
- 1.14 Further Readings

1.0 INTRODUCTION

In our daily life, we routinely encounter and solve problems. We pose problems that we need or want to solve. For this, we make use of available resources, and solve them. Some categories of resources include: the time and

efforts of yours and others; tools; information; and money. Some of the problems that you encounter and solve are quite simple. But some others may be very complex.

In this unit we introduce you to the concepts of problem-solving, especially as they pertain to computer programming.

The problem-solving is a skill and there are no universal approaches one can take to solving problems. Basically one must explore possible avenues to a solution one by one until s/he comes across a right path to a solution. In general, as one gains experience in solving problems, one develops one's own techniques and strategies, though they are often intangible. Problem-solving skills are recognized as an integral component of computer programming. It is a demand and intricate process which is equally important throughout the project life cycle especially – study, designing, development, testing and implementation stages. The computer problem solving process requires:

- Problem anticipation
- Careful planning
- Proper thought process
- Logical precision
- Problem analysis
- Persistence and attention.

At the same time it requires personal creativity, analytic ability and expression. The chances of success are amplified when the problem solving is approached in a systematic way and satisfaction is achieved once the problem is satisfactorily solved. The problems should be anticipated in advance as far as possible and properly defined to help the algorithm definition and development process.

Computer is a very powerful tool for solving problems. It is a symbol-manipulating machine that follows a set of stored instructions called a program. It performs these manipulations very quickly and has memory for storing input, lists of commands and output. A computer cannot think in the way we associate with humans. When using the computer to solve a problem, you must specify the needed initial data, the operations which need to be performed (in order of performance) and what results you want for output. If any of these instructions are missing, you will get either no results or invalid results. In either case, your problem has not yet been solved. Therefore, several steps need to be considered before writing a program. These steps may free you from hours of finding and removing errors in your program (a process called **debugging**). It should also make the act of problem solving with a computer a much simpler task.

All types of computer programs are collectively referred to as **software**. Programming languages are also part of it. Physical computer equipment such as electronic circuitry, input/output devices, storage media etc. comes under **hardware**. Software governs the functioning of hardware. Operations performed by software may be built into the hardware, while instructions

executed by the hardware may be generated in software. The decision to incorporate certain functions in the hardware and others in the software is made by the manufacturer and designer of the software and hardware. Normal considerations for this are: cost, speed, memory required, adaptability and reliability of the system. Set of instructions of the high level language used to code a problem to find its solution is referred to as **Source Program**. A translator program called **a compiler or interpreter**, translates the source program into the object program. This is the compilation or interpretation phase. All the testing of the source program as regards the correct format of instructions is performed at this stage and the errors, if any, are printed. If there is no error, the source program is transformed into the machine language program called **Object Program**. The Object Program is executed to perform calculations. This stage is the execution phase. Data, if required by the program, are supplied now and the results are obtained on the output device. The complete process is shown in fig 1.1 below:

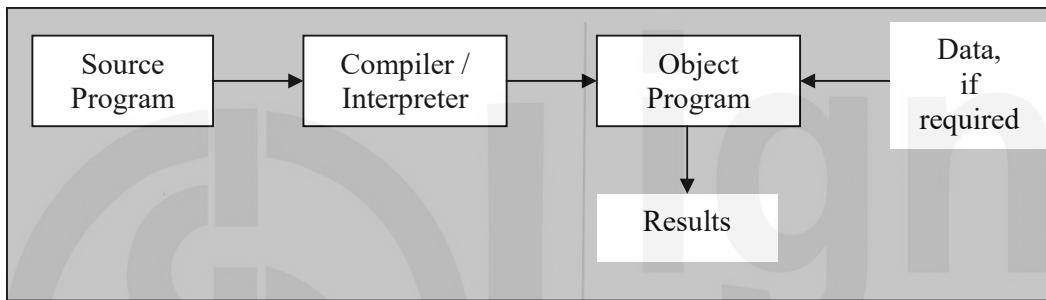


Fig 1.1: Conversion of Source Program to Object Program

1.1 OBJECTIVES

After going through this unit, you should be able to:

- apply problem solving techniques;
- define an algorithm and its features;
- design flowcharts;
- Define a program;
- Understand the history of C programming language;
- Compile a C program;
- Identify the syntax errors;
- Run a C program; and
- Understand what are run time and logical errors.

1.2 PROBLEM - SOLVING TECHNIQUES

Problem solving is a creative process which defines systematization and mechanization. There are a number of steps that can be taken to raise the level of one's performance in problem solving.

1.2.1 Steps for Problem - Solving

A problem-solving technique follows certain steps in finding the solution to a problem. Let us look into the steps one by one:

Problem definition phase

The success in solving any problem is possible only after the problem has been fully understood. That is, we cannot hope to solve a problem, which we do not understand. So, the problem understanding is the first step towards the solution of the problem. In *problem definition phase*, we must emphasize *what must be done* rather than *how is it to be done*. That is, we try to extract the precisely defined set of tasks from the problem statement. Inexperienced problem solvers too often gallop ahead with the task of problem - solving only to find that they are either solving the wrong problem or solving just one particular problem.

Getting started on a problem

There are many ways of solving a problem and there may be several solutions. So, it is difficult to recognize immediately which path could be more productive. Sometimes you do not have any idea where to begin solving a problem, even if the problem has been defined. Such block sometimes occurs because you are overly concerned with the details of the implementation even before you have completely understood or worked out a solution. The best advice is not to get concerned with the details. Those can come later when the intricacies of the problem has been understood.

The use of specific examples

To get started on a problem, we can make use of heuristics i.e., the rule of thumb. This approach will allow us to start on the problem by picking a specific problem we wish to solve and try to work out the mechanism that will allow solving this particular problem. It is usually much easier to work out the details of a solution to a specific problem because the relationship between the mechanism and the problem is more clearly defined. This approach of focusing on a particular problem can give us the foothold we need for making a start on the solution to the general problem.

Similarities among problems

One way to make a start is by considering a specific example. Another approach is to bring the experience to bear on the current problem. So, it is important to see if there are any similarities between the current problem and the past problems which we have solved. The more experience one has the more tools and techniques one can bring to bear in tackling the given problem. But sometimes, it blocks us from discovering a desirable or better solution to the problem. A skill that is important to try to develop in problem - solving is the ability to view a problem from a variety of angles. One must be able to metaphorically turn a problem upside down, inside out, sideways, backwards, forwards and so on. Once one has developed this skill it should be possible to get started on any problem.

In some cases we can assume that we already have the solution to the problem and then try to work backwards to the starting point. Even a guess at the solution to the problem may be enough to give us a foothold to start on the problem. We can systematize the investigations and avoid duplicate efforts by writing down the various steps taken and explorations made. Another practice that helps to develop the problem solving skills is, once we have solved a problem, to consciously reflect back on the way we went about discovering the solution.

1.2.2 Using Computer as a Problem - Solving Tool

The computer is a resource - a versatile tool - that can help you solve some of the problems that you encounter. A computer is a very powerful general-purpose tool. Computers can solve or help to solve many types of problems. There are also many ways in which a computer can enhance the effectiveness of the time and effort that you are willing to devote to solving a problem. Thus, it will prove to be well worth the time and effort you spend to learn how to make effective use of this tool.

In this section, we discuss the steps involved in developing a program. Program development is a multi-step process that requires you to understand the problem, develop a solution, write the program, and then test it. This critical process determines the overall quality and success of your program. If you carefully design each program using good structured development techniques, your programs will be efficient, error-free, and easy to maintain. The following are the steps in detail:

1. Develop an *Algorithm* and a *Flowchart*.
2. Write the program in a computer language (for example say C programming language).
3. Enter the program using some editor.
4. Test and debug the program.
5. Run the program, input data, and get the results.

1.3 BASICS OF ALGORITHMS

The first step in the program development is to devise and describe a precise plan of what you want the computer to do. This plan, expressed as a sequence of operations, is called an algorithm. An algorithm is just an outline or idea behind a program something resembling C or Pascal, but with some statements in English rather than within the programming language. It is expected that one could translate each pseudo-code statement to a small number of lines of actual code, easily and mechanically.

1.3.1 Definition

An **algorithm** is a finite set of steps defining the solution of a particular problem. An algorithm is expressed in pseudocode - something resembling C language or Pascal, but with some statements in English rather than within the programming language. Developing an efficient algorithm requires lot of practice and skill. It must be noted that an efficient algorithm is one which is capable of giving the solution to the problem by using minimum resources of the system such as memory and processor's time. Algorithm is a language independent, well structured and detailed. It will enable the programmer to translate into a computer program using any high-level language.

1.3.2 Features of Algorithm

Following features should be present in an algorithm:

Proper understanding of the problem

For designing an efficient algorithm, the expectations from the algorithm should be clearly defined so that the person developing the algorithm can understand the expectations from it. This is normally the outcome of the problem definition phase.

Use of procedures / functions to emphasize modularity

To assist the development, implementation and readability of the program, it is usually helpful to modularize (section) the program. Independent functions perform specific and well defined tasks. In applying modularization, it is important to watch that the process is not taken so far to a point at which the implementation becomes difficult to read because of fragmentation. The program then can be implemented as calls to the various procedures that will be needed in the final implementations.

Choice of variable names

Proper variable names and constant names can make the program more meaningful and easier to understand. This practice tends to make the program more self documenting. A clear definition of all variables and constants at the start of the procedure / algorithm can also be helpful. For example, it is better to use variable *day* for the day of the weeks, instead of the variable *a* or something else.

Documentation of the program

Brief information about the segment of the code can be included in the program to facilitate debugging and providing information. A related part of the documentation is the information that the programmer presents to the user during the execution of the program. Since, the program is often to be used by persons who are unfamiliar with the working and input requirements of the program, proper documentation must be provided. That is, the program must specify what responses are required from the user. Care should also be taken to avoid ambiguities in these specifications. Also the program should "catch"

1.3.3 Criteria to be followed by an Algorithm

The following is the criteria to be followed by an algorithm:

- **Input:** There should be zero or more values which are to be supplied.
- **Output:** At least one result is to be produced.
- **Definiteness:** Each step must be clear and unambiguous.
- **Finiteness:** If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- **Effectiveness:** Each step must be sufficiently basic that a person using only paper and pencil can in principle carry it out. In addition, not only each step is definite, it must also be feasible.

Example 1.1

Let us try to develop an algorithm to compute and display the sum of two numbers

1. Start
2. Read two numbers a and b
3. Calculate the sum of a and b and store it in sum
4. Display the value of sum
5. Stop

Example 1.2

Let us try to develop an algorithm to compute and print the average of a set of data values.

1. Start
2. Set the sum of the data values and the count to zero.
3. As long as the data values exist, add the next data value to the sum and add 1 to the count.
4. To compute the average, divide the sum by the count.
5. Display the average.
6. Stop

Example 1.3

Write an algorithm to calculate the factorial of a given number.

1. Start
2. Read the number n
3. [Initialize]
 $i \leftarrow 1, fact \leftarrow 1$
4. Repeat steps 4 through 6 until $i = n$

5. fact \leftarrow fact * i
6. i \leftarrow i + 1
7. Print fact
8. Stop

Example 1.4

Write an algorithm to check that whether the given number is prime or not.

1. Start
2. Read the number num
3. [Initialize]
i \leftarrow 2 , flag \leftarrow 1
4. Repeat steps 4 through 6 until i < num or flag = 0
5. rem \leftarrow num mod i
6. if rem = 0 then
flag \leftarrow 0
else
i \leftarrow i + 1
7. if flag = 0 then
Print Number is not prime
Else
Print Number is prime
8. Stop

1.3.4 Top Down Design

Once we have defined the problem and have an idea of how to solve it, we can then use the powerful techniques for designing algorithms. Most of the problems are complex or large problems and to solve them we have to focus on to comprehend at one time, a very limited span of logic or instructions. A technique for algorithm design that tries to accommodate this human limitation is known as **top-down design or stepwise refinement**.

Top down design provides the way of handling the logical complexity and detail encountered in computer algorithm. It allows building solutions to problems in step by step. In this way, specific and complex details of the implementation are encountered only at the stage when sufficient groundwork on the overall structure and relationships among the various parts of the problem. Before the top down design can be applied to any problem, we must at least have the outlines of a solution. Sometimes this might demand a lengthy and creative investigation into the problem while at another time the problem description may in itself provide the necessary starting point for the top-down design. Top-down design suggests taking the general statements about the solution one at a time, and then breaking them down into a more precise subtask / sub-problem. These sub-problems should more accurately describe how the final goal can be reached. The process of repeatedly breaking a task down into a subtask and then each subtask into smaller subtasks must continue

until the sub-problem can be implemented as the program statement. With each splitting, it is essential to define how sub-problems interact with each other. In this way, the overall structure of the solution to the problem can be maintained. Preservation of the overall structure is important for making the algorithm comprehensible and also for making it possible to prove the correctness of the solution.

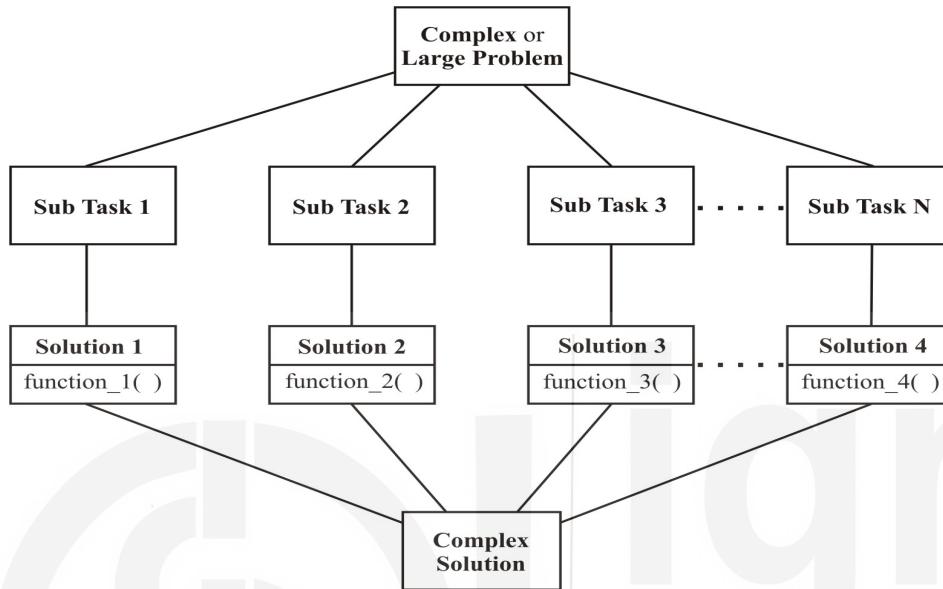


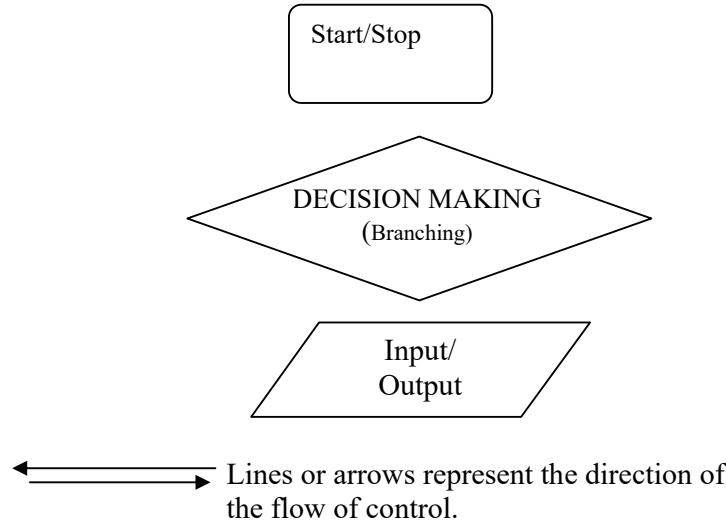
Figure 1.2: Schematic breakdown of a problem into subtasks as employed in top down design

Let us see how to represent the algorithm in a graphical form using a flowchart in the following section.

1.4 FLOWCHARTS

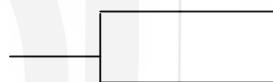
The next step after the algorithm development is the flowcharting. Flowcharts are used in programming to diagram the path in which information is processed through a computer to obtain the desired results. Flowchart is a graphical representation of an algorithm. It makes use of symbols which are connected among them to indicate the flow of information and processing. It will show the general outline of how to solve a problem or perform a task. It is prepared for better understanding of the algorithm.

1.4.1 Basic Symbols used in flowchart design



Connector (connect one part of the flowchart to another)

Process, Instruction



Comments, Explanations, Definitions

Additional Symbols Related to more advanced programming



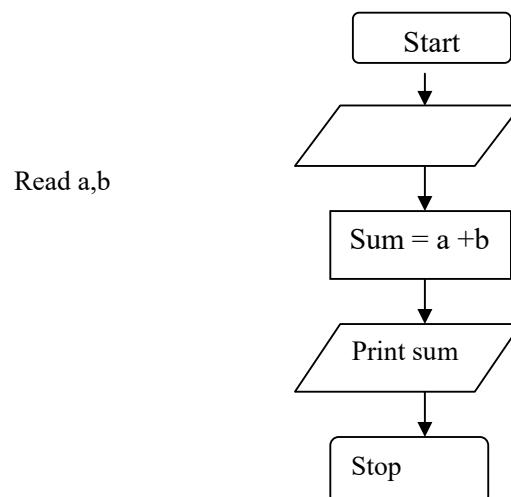
Preparation (may be used with “do Loops”)



Refers to separate flowchart

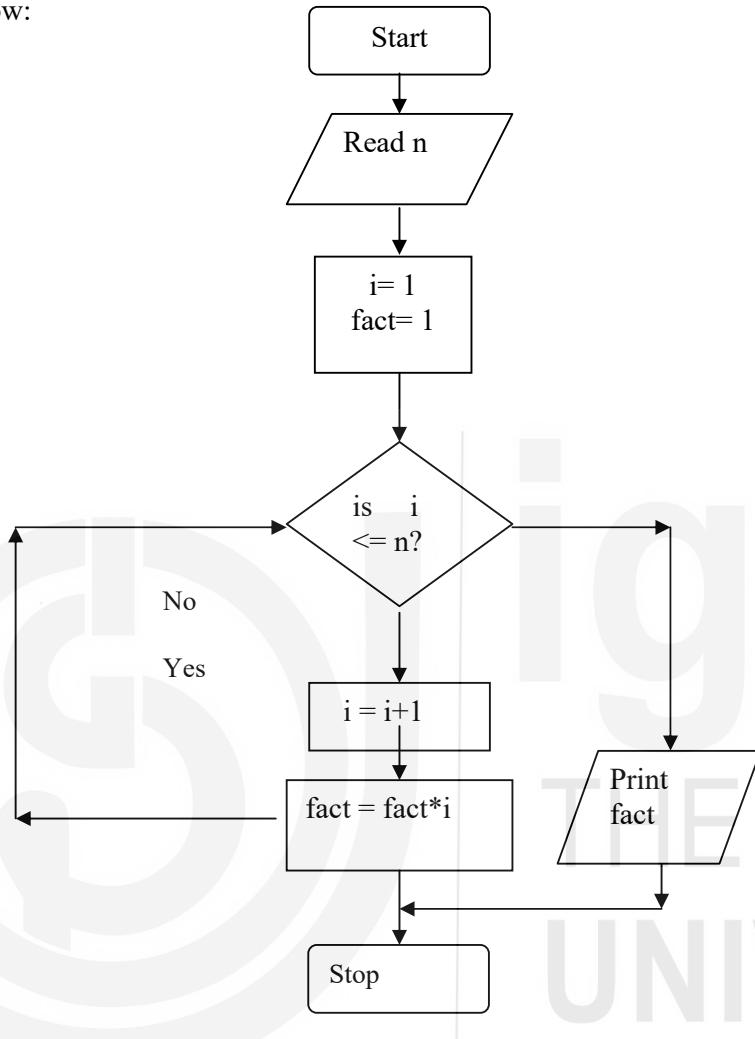
Example 1.5

The flowchart for the Example 1.1 is shown below:

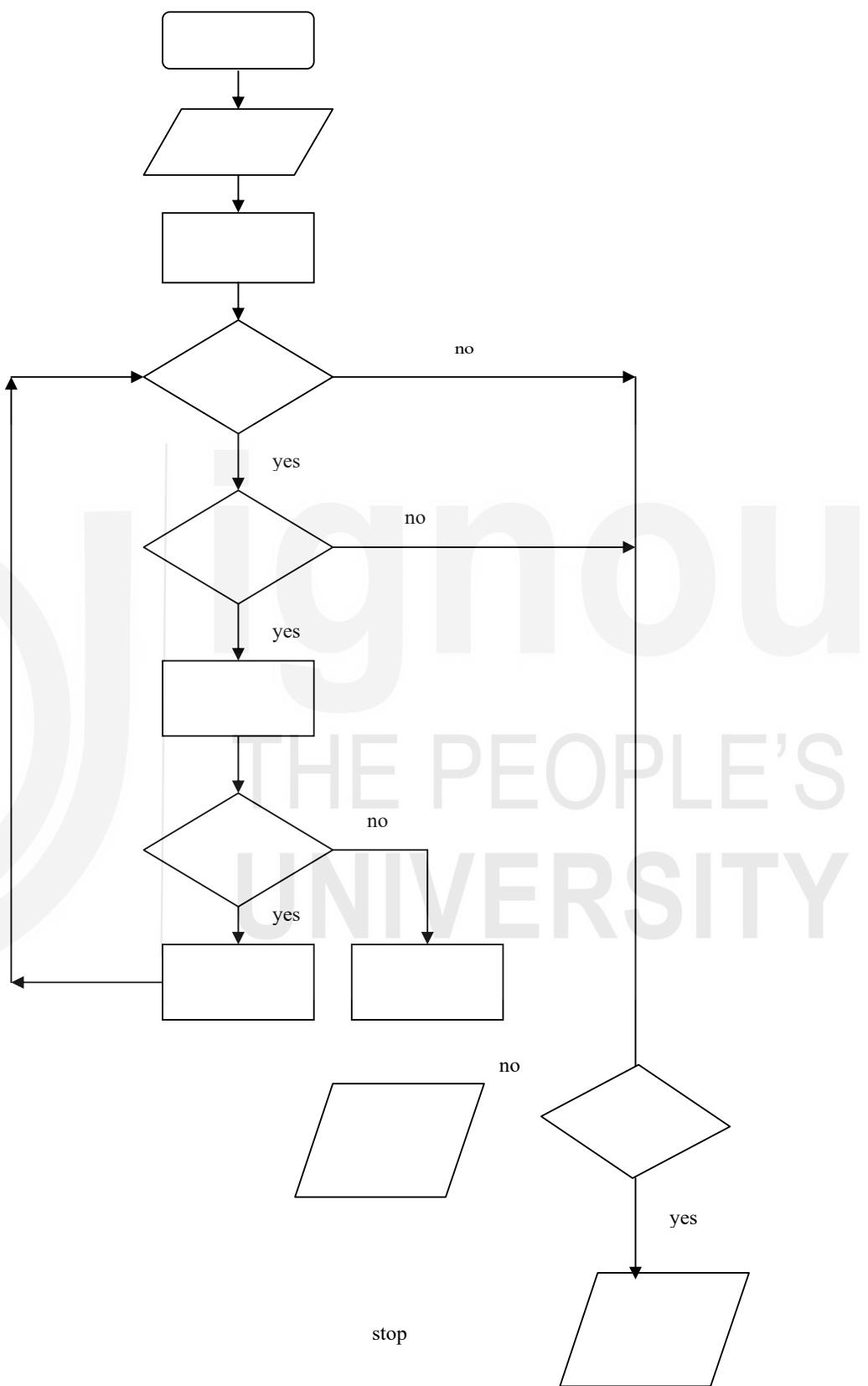


Example 1.6

The flowchart for the Example 1.3 (to find factorial of a given number) is shown below:

**Example 1.7:**

The flowchart for Example 1.4 is shown below:



Check Your Progress 1

Programming
Fundamentals

1. Differentiate between flowchart and algorithm.

.....
.....
.....
.....
.....

2. Compute and print the sum of a set of data values.

.....
.....
.....
.....
.....

3. Write the following steps are suggested to facilitate the problem solving process using computer.

.....
.....
.....
.....
.....

4. Draw an algorithm and flowchart to calculate the roots of quadratic equation

$$Ax^2 + Bx + C = 0.$$

.....
.....
.....
.....
.....

1.5 PROGRAM AND PROGRAMMING LANGUAGE

A language is a mode of communication between two people. It is necessary for those two people to understand the language in order to communicate. But even if the two people do not understand the same language, a translator can help to convert one language to the other, understood by the second person. Similar to a translator is the mode of communication between a user and a computer is a computer language. One form of the computer language is understood by the user, while in the other form it is understood by the

computer. A translator (or compiler) is needed to convert from user's form to computer's form. Like other languages, a computer language also follows a particular grammar known as the syntax.

In this unit we will introduce you the basics of programming language C.

We have seen in the earlier section's that a computer has to be fed with a detailed set of instructions and data for solving a problem. Such a procedure which we call an *algorithm* is a series of steps arranged in a logical sequence. Also we have seen that a *flowchart* is a pictorial representation of a sequence of instructions given to the computer. It also serves as a document explaining the procedure used to solve a problem. In practice it is necessary to express an algorithm using a *programming language*. A procedure expressed in a programming language is known as a *computer program*.

Computer programming languages are developed with the primary objective of facilitating a large number of people to use computers without the need for them to know in detail the internal structure of the computer. Languages are designed to be *machine-independent*. Most of the programming languages ideally designed, to execute a program on any computer regardless of who manufactured it or what model it is.

Programming languages can be divided into two categories:

- i) **Low Level Languages or Machine Oriented Languages:** The language whose design is governed by the circuitry and the structure of the machine is known as the **Machine language**. This language is difficult to learn and use. It is specific to a given computer and is different for different computers i.e. these languages are **machine-dependent**. These languages have been designed to give a better machine efficiency, i.e. faster program execution. Such languages are also known as Low Level Languages. Another type of Low-Level Language is the Assembly Language. We will code the assembly language program in the form of mnemonics. Every machine provides a different set of mnemonics to be used for that machine only depending upon the processor that the machine is using.
- ii) **High Level Languages or Problem Oriented Languages:** These languages are particularly oriented towards describing the procedures for solving the problem in a concise, precise and unambiguous manner. Every high level language follows a precise set of rules. They are developed to allow application programs to be run on a variety of computers. These languages are *machine-independent*. Languages falling in this category are FORTRAN, BASIC, PASCAL etc. They are easy to learn and programs may be written in these languages with much less effort. However, the computer cannot understand them and they need to be translated into machine language with the help of other programs known as Compilers or Translators.

Prior to writing C programs, it would be interesting to find out what really is C language, how it came into existence and where does it stand with respect to other computer languages. We will briefly outline these issues in the following section.

1.6.1 History of C Programming Language

C is a programming language developed at AT&T's Bell Laboratory of USA in 1972. It was designed and written by Dennis Ritchie. As compared to other programming languages such as Pascal, C allows a precise control of input and output.

Now let us see its historical development. The late 1960s were a turbulent era for computer systems research at Bell Telephone Laboratories. By 1960, many programming languages came into existence, almost each for a specific purpose. For example COBOL was being used for Commercial or Business Applications, FORTRAN for Scientific Applications and so on. So, people started thinking why could not there be a one general purpose language.

Therefore, an International Committee was set up to develop such a language, which came out with the invention of ALGOL60. But this language never became popular because it was too abstract and too general. To improve this, a new language called Combined Programming Language (CPL) was developed at Cambridge University. But this language was very complex in the sense that it had too many features and it was very difficult to learn. Martin Richards at Cambridge University reduced the features of CPL and developed a new language called Basic Combined Programming Language (BCPL). But unfortunately it turned out to be much less powerful and too specific. Ken Thompson at AT & T's Bell Labs, developed a language called B at the same time as a further simplification of CPL. But like BCPL this was also too specific. Ritchie inherited the features of B and BCPL and added some features on his own and developed a language called C. C proved to be quite compact and coherent. Ritchie first implemented C on a DEC PDP-11 that used the UNIX Operating System.

For many years the *de facto* standard for C was the version supplied with the UNIX version 5 operating system. The growing popularity of microcomputers led to the creation of large number of C implementations. At the source code level most of these implementations were highly compatible. However, since no standard existed there were discrepancies. To overcome this situation, ANSI established a committee in 1983 that defined an ANSI standard for the C language.

1.6.2 Salient features of C

C is a general purpose, structured programming language. Among the two types of programming languages discussed earlier, C lies in between these two categories. That's why it is often called a ***middle level language***. It means that it combines the elements of high level languages with the functionality of

assembly language. It provides relatively good programming efficiency (as compared to machine oriented language) and relatively good machine efficiency as compared to high level languages). As a middle level language, C allows the manipulation of bits, bytes and addresses – the basic elements with which the computer executes the inbuilt and memory management functions. C code is very portable, that it allows the same C program to be run on machines with different hardware configurations. The flexibility of C allows it to be used for systems programming as well as for application programming.

C is commonly called a structured language because of structural similarities to ALGOL and Pascal. The distinguishing feature of a structured language is compartmentalization of code and data. Structured language is one that divides the entire program into modules using top-down approach where each module executes one job or task. It is easy for debugging, testing, and maintenance if a language is a structured one. C supports several control structures such as **while**, **do-while and for** and various data structures such as **structs**, **files**, **arrays** etc. as would be seen in the later units. The basic unit of a C program is a **function** - C's standalone subroutine. The structural component of C makes the programming and maintenance easier.

Check Your Progress 2

1. “A Program written in Low Level Language is faster.” Why?

.....
.....
.....
.....

2. What is the difference between high level language and low level language?

.....
.....
.....
.....

3. Why is C referred to as middle level language?

.....
.....
.....
.....

As we have already seen, to solve a problem there are three main things to be considered. Firstly, what should be the output? Secondly, what should be the inputs that will be required to produce this output and thirdly, the steps of instructions which use these inputs to produce the required output. As stated earlier, every programming language follows a set of rules; therefore, a program written in C also follows predefined rules known as syntax. C is a case sensitive language. All C programs consist of one or more functions. One function that must be present in every C program is **main()**. This is the first function called up when the program execution begins. Basically, **main()** outlines what a program does. Although **main** is not given in the keyword list, it cannot be used for naming a variable. The structure of a C program is illustrated in Figure 1.3 where functions *func1()* through *funcn()* represent user defined functions.

Preprocessor directives

```
Global data declarations
main () /* main function*/
{
    Declaration part;
    Program statements;
}

/*User defined functions*/
func1()
{
    .....
}

func2 ()
{
    .....
}

.
funcn ()
{
    .....
}
```

Figure 1.3: Structure of a C Program.

A Simple C Program

From the above sections, you have become familiar with, a programming language and structure of a C program. It's now time to write a simple C program. This program will illustrate how to print out the message "This is a C program".

Example 1.8: Write a program to print a message on the screen.

```
/*Program to print a message*/
```

```
#include<stdio.h>           /* header file*/
```

```
main()          /* main function*/
{
    printf("This is a C program\n"); /* output statement*/
}
```

Though the program is very simple, a few points must be noted.

Every C program contains a function called **main()**. This is the starting point of the program. This is the point from where the execution begins. It will usually call other functions to help perform its job, some that we write and others from the standard libraries provided.

#include <stdio.h> is a reference to a special file called stdio.h which contains information that must be included in the program when it is compiled. The inclusion of this required information will be handled automatically by the compiler. You will find it at the beginning of almost every C program. Basically, all the statements starting with # in a C program are called preprocessor directives. These will be considered in the later units. Just remember, that this statement allows you to use some predefined functions such as, *printf()*, in this case.

main() declares the start of the function, while the two curly brackets { } shows the start and finish of the function. Curly brackets in C are used to group statements together as a function, or in the body of a loop. Such a grouping is known as a compound statement or a block. Every statement within a function ends with a terminator semicolon (;).

printf("This is a C program\n"); prints the words on the screen. The text to be printed is enclosed in double quotes. The \n at the end of the text tells the program to print a newline as part of the output. That means now if we give a second printf statement, it will be printed in the next line.

Comments may appear anywhere within a program, as long as they are placed within the delimiters /* and */. Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

While useful for teaching, such a simple program has few practical uses. Let us consider something rather more practical. Let us look into the example given below, the complete program development life cycle.

Example 1.9

Develop an algorithm, flowchart and program to add two numbers.

Algorithm

1. Start
2. Input the two numbers **a** and **b**
3. Calculate the sum as **a+b**
4. Store the result in **sum**
5. Display the result
6. Stop.

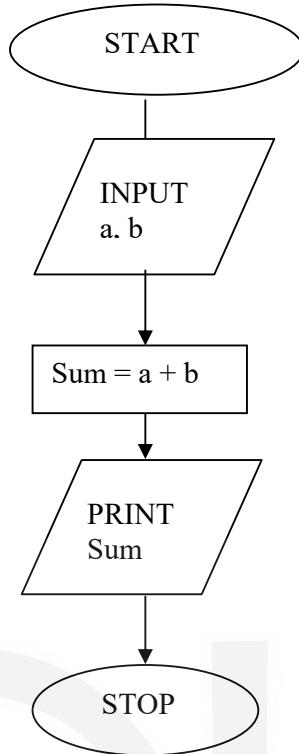


Figure 1.4: Flow chart to add two numbers

Program

```

#include<stdio.h>

main()
{
int a,b,sum;           /* variables declaration*/

printf("\n Enter the values for a and b: \n");
scanf("%d, %d", &a, &b);

sum=a+b;

printf("\nThe sum is %d",sum);  /*output statement*/
}
  
```

OUTPUT

Enter the values of a and b:

2 3

The sum is 5

In the above program considers two variables *a* and *b*. These variables are declared as integers (**int**), it is the data type to indicate integer values. Next statement is the **printf** statement meant for prompting the user to input the values of *a* and *b*. **scanf** is the function to intake the values into the program provided by the user. Next comes the processing / computing part which computes the **sum**. Again the **printf** statement is a bit different from the first program; it includes a format specifier (%d). The format specifier indicates the

kind of value to be printed. We will study about other data types and format specifiers in detail in the following units. In the `printf` statement above, `sum` is not printed in double quotes because we want its value to be printed. The number of format specifiers and the variable should match in the `printf` statement.

At this stage, don't go much in detail. However, in the following units you will be learning all these details.

1.8 WRITING A C PROGRAM

A C program can be executed on platforms such as DOS, UNIX etc. DOS stores C program with a file extension `.c`. Program text can be entered using any text editor such as EDIT or any other. To edit a file called `testprog.c` using edit editor, gives:

C:> edittestprog.c

If you are using **Turbo C**, then Turbo C provides its own editor which can be used for writing the program. Just give the full pathname of the executable file of Turbo C and you will get the editor in front of you. For example:

C:> turboc\bin\tc

Here, `tc.exe` is stored in bin subdirectory of `turboc` directory. After you get the menu just type the program and store it in a file using the menu provided. The file automatically gets the extension of `.c`.

UNIX also stores C program in a file with extension is `.c`. This identifies it as a C program. The easiest way to enter your text is using a text editor like `vi`, `emacs` or `xedit`. To edit a file called `testprog.c` using `vi`,

\$ vi testprog.c

The editor is also used to make subsequent changes to the program

1.9 COMPILING A C PROGRAM

After you have written the program the next step is to save the program in a file with extension `.c`. This program is in high-level language. But this language is not understood by the computer. So, the next step is to convert the high-level language program (source code) to machine language (object code). This task is performed by a software or program known as a compiler. Every language has its own compiler that converts the source code to object code. The compiler will compile the program successfully if the program is syntactically correct; else the object code will not be produced. This is explained pictorially in Figure 1.5.

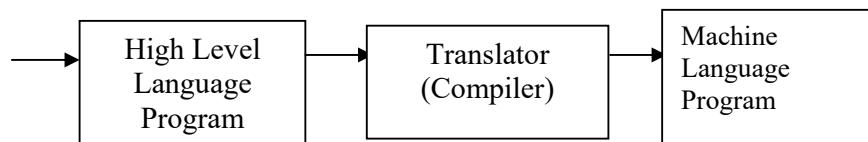


Figure 1.5: Process of Translation

1.9.1 The C Compiler

Programming
Fundamentals

If you are working on UNIX platform, then if the name of the program file is testprog.c, to compile it, the simplest method is to type

```
cc testprog.c
```

This will compile testprog.c, and, if successful, will produce a executable file called **a.out**. If you want to give the executable file any other, you can type

```
cc testprog.c -o testprog
```

This will compile **testprog.c**, creating an executable file testprog.

If you are working with TurboC on DOS platform then the option for compilation is provided on the menu. If the program is syntactically correct then this will produce a file named as **testprog.obj**. If not, then the syntax errors will be displayed on the screen and the object file will not be produced. The errors need to be removed before compiling the program again. This process of removing the errors from the program is called as the **debugging**.

1.9.2 Syntax and Semantic Errors

Every language has an associated grammar, and the program written in that language has to follow the rules of that grammar. For example in English a sentence such a “Shyam, is playing, with a ball”. This sentence is syntactically incorrect because commas should not come the way they are in the sentence.

Likewise, C also follows certain syntax rules. When a C program is compiled, the compiler will check that the program is syntactically correct. If there are any syntax errors in the program, those will be displayed on the screen with the corresponding line numbers. Let us consider the following program.

Example 1.10: Write a program to print a message on the screen.

```
/* Program to print a message on the screen*/
#include <stdio.h>
main( )
{
    printf("Hello, how are you\n")
```

Let the name of the program be **test.c**. If we compile the above program as it is we will get the following errors:

Error test.c 1:No file name ending

Error test.c 5: Statement missing ;

Error test.c 6: Compound statement missing }

Edit the program again, correct the errors mentioned and the corrected version appears as follows:

```
#include <stdio.h>
main( )
{
```

```
    printf ("Hello, how are you\n");
}
```

Apart from syntax errors, another type of errors that are shown while compilation are semantic errors. These errors are displayed as warnings. These errors are shown if a particular statement has no meaning. The program does compile with these errors, but it is always advised to correct them also, since they may create problems while execution. The example of such an error is that say you have declared a variable but have not used it, and then you get a warning “code has no effect”. These variables are unnecessarily occupying the memory.

Check Your Progress 3

1. What is the basic unit of a C program?

.....
.....
.....
.....
.....

2. “The program is syntactically correct”. What does it mean?

.....
.....
.....
.....
.....

3. Indicate the syntax errors in the following program code:

```
include <stdio.h>
main()
[
    printf("hello\n");
]
```

.....
.....
.....
.....
.....

1.10 LINK AND RUN THE C PROGRAM

After compilation, the next step is linking the program. Compilation produces a file with an extension **.obj**. Now this **.obj** file cannot be executed since it

contains calls to functions defined in the standard library (header files) of C language. These functions have to be linked with the code you wrote. C comes with a standard library that provides functions that perform most commonly needed tasks. When you call a function that is not the part of the program you wrote, C remembers its name. Later the linker combines the code you wrote with the object code already found in the standard library. This process is called *linking*. In other words, Linker is a program that links separately compiled functions together into one program. It combines the functions in the standard C library with the code that you wrote. The output of the linker in an executable program i.e., a file with an extension **.exe**.

1.10.1 Run the C Program Through the Menu

When we are working with TurboC in DOS environment, the menu in the GUI that pops up when we execute the executable file of TurboC contains several options for executing the program:

- i) Link , after compiling
- ii) Make, compiles as well as links
- iii) Run

All these options create an executable file and when these options are used we also get the output on user screen. To see the output we have to shift to user screen window.

1.10.2 Run From an Executable File

An **.exe** file produced by can be directly executed.

UNIX also includes a very useful program called **make**. **Make** allows very complicated programs to be compiled quickly, by reference to a configuration file (usually called **makefile**). If your C program is a single file, you can usually use make by simply typing –

make testprog

This will compile **testprog.c** as well as link your program with the standard library so that you can use the standard library functions such as **printf** and put the executable code in **testprog**.

In case of DOS environment , the options provided above produce an executable file and this file can be directly executed from the DOS prompt just by typing its name without the extension. That is if the name of the program is **test.c**, after compiling and linking the new file produced is **test.exe** only if compilation and linking is successful.

This can be executed as:

c>test

1.10.3 Linker Errors

If a program contains syntax errors then the program does not compile, but it may happen that the program compiles successfully but we are unable to get the executable file, this happens when there are certain linker errors in the

program. For example, the object code of certain standard library function is not present in the standard C library; the definition for this function is present in the header file that is why we do not get a compiler error. Such kinds of errors are called linker errors. The executable file would be created successfully only if these linker errors are corrected.

1.10.4 Logical and Runtime Errors

After the program is compiled and linked successfully we execute the program. Now there are three possibilities:

- 1) The program executes and we get correct results,
- 2) The program executes and we get wrong results, and
- 3) The program does not execute completely and aborts in between.

The first case simply means that the program is correct. In the second case, we get wrong results; it means that there is some logical mistake in our program. This kind of error is known as **logical error**. This error is the most difficult to correct. This error is corrected by debugging. Debugging is the process of removing the errors from the program. This means manually checking the program step by step and verifying the results at each step. Debugging can be made easier by a tracer provided in Turbo C environment. Suppose we have to find the average of three numbers and we write the following code:

Example 1.11: Write a C program to compute the average of three numbers

```
/* Program to compute average of three numbers */
#include<stdio.h>
main( )
{
    int a,b,c,sum,avg;
    a=10;
    b=5;
    c=20;
    sum = a+b+c;
    avg = sum / 3;
    printf("The average is %d\n", avg);
}
```

OUTPUT

The average is 8.

The exact value of average is 8.33 and the output we got is 8. So we are not getting the actual result, but a rounded off result. This is due to the logical error. We have declared variable **avg** as an integer but the average calculated is a real number, therefore only the integer part is stored in **avg**. Such kinds of errors which are not detected by the compiler or the linker are known as **logical errors**.

The third kind of error is only detected during execution. Such errors are known as **run time errors**. These errors do not produce the result at all, the program execution stops in between and the run time error message is flashed on the screen. Let us look at the following example:

Example 1.12: Write a program to divide a sum of two numbers by their difference

Programming
Fundamentals

```
/* Program to divide a sum of two numbers by their difference*/
```

```
#include <stdio.h>
```

```
main( )  
{  
  
    int a,b;  
    float c;  
  
    a=10;  
    b=10;  
  
    c = (a+b) / (a-b);  
    printf("The value of the result is %f\n",c);  
}
```

The above program will compile and link successfully, it will execute till the first *printf* statement and we will get the message in this statement, as soon as the next statement is executed we get a runtime error of “Divide by zero” and the program halts. Such kinds of errors are **runtime errors**.

1.11 DIAGRAMMATIC REPRESENTATION OF C PROGRAM EXECUTION PROCESS

The following figure 1.6 shows the diagrammatic representation of the program execution process.

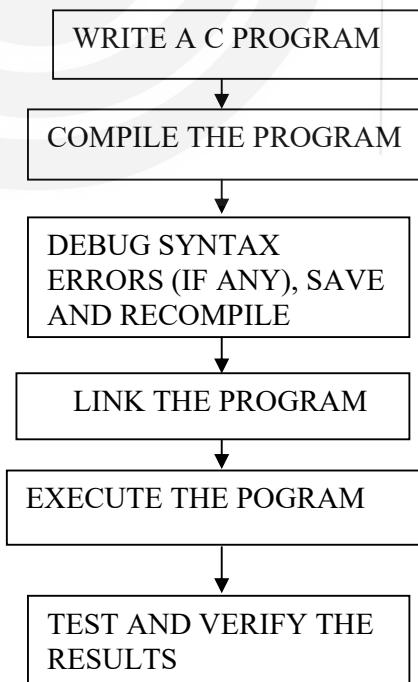


Figure 1.6: Program Execution Process

Check Your Progress 4

1. What is the extension of an executable file?

.....
.....
.....
.....
.....

2. What is the need for linking a compiled file?

.....
.....
.....
.....
.....

3. How do you correct the logical errors in the program?

.....
.....
.....
.....
.....

1.12 SUMMARY

To solve a problem different problem - solving tools are available that help in finding the solution to problem in an efficient and systematic way. Steps should be followed to solve the problem that includes writing the algorithm and drawing the flowchart for the solution to the stated problem. Top down design provides the way of handling the logical complexity and detail encountered in computer algorithm. It allows building solutions to problems in a stepwise fashion. In this way, specific and complex details of the implementation are encountered only at the stage when sufficient groundwork on the overall structure and relationships among the various parts of the problem. We present C language - a standardized, industrial-strength programming language known for its power and portability as an implementation vehicle for these problem solving techniques using computer.

In this unit, you have learnt about a program and a programming language. You can now differentiate between high level and low level languages. You can now define what is C, features of C. You have studied the emergence of C. You have seen how C is different, being a middle level Language, than other High Level languages. The advantage of high level language over low level language is discussed.

You have seen how you can convert an algorithm and flowchart into a C program. We have discussed the process of writing and storing a C program in a file in case of UNIX as well as DOS environment.

You have learnt about compiling and running a C program in UNIX as well as on DOS environment. We have also discussed about the different types of errors that are encountered during the whole process, i.e. syntax errors, semantic errors, logical errors, linker errors and runtime errors. You have also learnt how to remove these errors. You can now write simple C programs involving simple arithmetic operators and the *printf()* statement. With these basics, now we are ready to learn the C language in detail in the following units.

1.13 SOLUTIONS / ANSWERS

Check Your Progress 1

1. The process to devise and describe a precise plan (in the form of sequence of operations) of what you want the computer to do, is called an **algorithm**. An algorithm may be symbolized in a flowchart or pseudocode.
2.
 1. Start
 2. Set the sum of the data values and the count of the data values to zero.
 3. As long as the data values exist, add the next data value to the sum and add 1 to the count.
 4. Display the average.
 5. Stop
3. The following steps are suggested to facilitate the problem solving process:
 - a) Define the problem
 - b) Formulate a mathematical model
 - c) Develop an algorithm
 - d) Design the flowchart
 - e) Code the same using some computer language
 - f) Test the program

Check Your Progress 2

1. A program written in Low Level Language is faster to execute since it needs no conversion while a high level language program need to be converted into low level language.
2. Low level languages express algorithms on the form of numeric or mnemonic codes while High Level Languages express algorithms in the using concise, precise and unambiguous notation. Low level languages are machine dependent while High level languages are machine independent. Low level languages are difficult to program and to learn, while High

level languages are easy to program and learn. Examples of High level languages are FORTRAN, Pascal and examples of Low level languages are machine language and assembly language.

3. C is referred to as middle level language as with C we are able to manipulate bits, bytes and addresses i.e. interact with the hardware directly. We are also able to carry out memory management functions.

Check Your Progress 3

1. The basic unit of a C program is a C function.
2. It means that program contains no grammatical or syntax errors.
3. Syntax errors:
 - a) # not present with include
 - b) {brackets should be present instead of [brackets.

Check Your Progress 4

1. The extension of an executable file is .exe.
2. The C program contains many C pre-defined functions present in the C library. These functions need to be linked with the C program for execution; else the C program may give a linker error indicating that the function is not present.
3. Logical errors can be corrected through debugging or self checking.

1.14 FURTHER READINGS

1. How to solve it by Computer, 5th Edition, *R G Dromey*, PHI, 1992.
2. Introduction to Computer Algorithms, Second Edition, *Thomas H. Cormen*, MIT press, 2001.
3. *Fundamental Algorithms*, Third Edition, *Donald E Knuth*, Addison-Wesley, 1997.
4. C Programming Language, *Kernighan & Richie*, PHI Publication.
5. Programming with C, Second Edition, *Byron Gottfried*, Tata Mc Graw Hill, 2003.
6. The C Complete Reference, Fourth Editon, *Herbert Schildt*, Tata Mc Graw Hill, 2002.
7. Programming with ANSI and Turbo C, *Ashok N. Kamthane*, Pearson Education Asia, 2002.
8. Computer Science A Structured Programming Approach Using C, Second Edition, *Behrouza A. Forouzan, Richard F. Gilberg*, Brooks/Cole, Thomson Learning, 2001.

UNIT 2 DATA TYPES, OPERATORS AND EXPRESSIONS

Structure

- 2.0 Introduction
- 2.1 Objectives
- 2.2 C Language Character Set
- 2.3 Identifiers and Keywords
 - 2.3.1 Rules for Forming Identifiers
 - 2.3.2 Keywords
- 2.4 Data Types and Storage
- 2.5 Data Type Qualifiers
- 2.6 Variables
- 2.7 Declaring Variables
- 2.8 Initializing Variables
- 2.9 Constants
 - 2.9.1 Integer Constants
 - 2.9.2 Floating Point Constants
 - 2.9.3 Character Constants
 - 2.9.4 String Constants
- 2.10 Symbolic Constants and Others
- 2.11 Expressions and Operators – An Introduction
- 2.12 Assignment Statements
- 2.13 Arithmetic Operators
- 2.14 Relational Operators
- 2.15 Logical Operators
- 2.16 Comma and Conditional Operators
- 2.17 Type Cast Operator
- 2.18 Size of Operator
- 2.19 C Shorthand
- 2.20 Priority of Operators
- 2.21 Summary
- 2.22 Solutions / Answers
- 2.23 Further Readings

2.0 INTRODUCTION

As every natural language has a basic character set, computer languages also have a character set, rules to define words. Words are used to form statements. These in turn are used to write the programs.

Computer programs usually work with different types of data and need a way to store the values being used. These values can be numbers or characters. C

language has two ways of storing number values—**variables and constants**—with many options for each. Constants and variables are the fundamental elements of each program. Simply speaking, a program is nothing else than defining them and manipulating them.

A variable is a data storage location that has a value that can change during program execution. In contrast, a constant has a fixed value that can't change.

This unit is concerned with the basic elements used to construct simple C program statements. These elements include the C character set, identifiers and keywords, data types, constants, variables and arrays, declaration and naming conventions of variables.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- define identifiers, data types and keywords in C;
- know name the identifiers as per the conventions;
- describe memory requirements for different types of variables;
- define constants, symbolic constants and their use in programs.write and evaluate arithmetic expressions;
- express and evaluate relational expressions;
- write and evaluate logical expressions;
- write and solve compute complex expressions (containing arithmetic, relational and logical operators), and
- use simple conditions using conditional operators.

2.2 C LANGUAGE CHARACTER SET

When you write a program, you express C source files as text lines containing characters from the character set. When a program executes in the target environment, it uses characters from the character set. These character sets are related, but need not have the same encoding or all the same members.

Every character set contains a distinct code value for each character in the **basic C character set**. A character set can also contain additional characters with other code values. The C language character set has alphabets, numbers, and special characters as shown below:

1. Alphabets including both lowercase and uppercase alphabets - A-Z and a-z.
2. Numbers 0-9
3. Special characters include:

;	:	{	,	'	"	
}	>	<	/	\	~	
[]	!	\$?	*	+
=	()	-	%	#	^
					@	&

2.3 IDENTIFIERS AND KEYWORDS

Identifiers are the names given to various program elements such as constants, variables, function names and arrays etc. Every element in the program has its own distinct name but one cannot select any name unless it conforms to valid name in C language. Let us study first the rules to define names or identifiers.

2.3.1 Rules for Forming Identifiers

Identifiers are defined according to the following rules:

1. It consists of letters and digits.
2. First character must be an alphabet or underscore.
3. Both upper and lower cases are allowed. Same text of different case is not equivalent, for example: **TEXT** is not same as **text**.
4. Except the special character underscore (_), no other special symbols can be used.

For example, some valid identifiers are shown below:

Y
X123
_XI
temp
tax_rate

For example, some invalid identifiers are shown below:

123	First character to be alphabet
"X."	. not allowed
order-no	Hyphen not allowed
error flag	Blank space not allowed

2.3.2 Keywords

Keywords are reserved words which have standard, predefined meaning in C. They cannot be used as program-defined identifiers.

The list of keywords in C language are as follows:

char	while	do	typedef	auto
int	if	else	switch	case
printf	double	struct	break	static
long	enum	register	extern	return
union	const	float	short	unsigned
continue	for	signed	void	default
goto	sizeof	volatile		

Note: Generally all keywords are in lower case although uppercase of same names can be used as identifiers.

2.4 DATA TYPES AND STORAGE

To store data inside the computer we need to first identify the type of data elements we need in our program. There are several different types of data, which may be represented differently within the computer memory. The data type specifies two things:

1. Permissible range of values that it can store.
2. Memory requirement to store a data type.

C Language provides four basic data types viz. int, char, float and double. Using these, we can store data in simple ways as single elements or we can group them together and use different ways (to be discussed later) to store them as per requirement. The four basic data types are described in the following table 2.1:

Table 2.1: Basic Data Types

DATA TYPE	TYPE OF DATA	MEMORY	RANGE
int	Integer	2 Bytes	- 32,768 to 32,767
char	character	1 Byte	- 128 to 128
float	Floating point number	4 bytes	3.4e - 38 to 3.4e +38
double	Floating point number with higher precision	8 bytes	1.7e - 308 to 1.7e + 308

Memory requirements or size of data associated with a data type indicates the range of numbers that can be stored in the data item of that type.

2.5 DATA TYPE QUALIFIERS

Short, long, signed, unsigned are called the data type qualifiers and can be used with any data type. A *short int* requires less space than *int* and *long int* may require more space than *int*. If *int* and *short int* takes 2 bytes, then *long int* takes 4 bytes.

Unsigned bits use all bits for magnitude; therefore, this type of number can be larger. For example ***signed int*** ranges from -32768 to +32767 and ***unsigned int*** ranges from 0 to 65,535. Similarly, ***char*** data type of data is used to store a character. It requires 1 byte. ***Signed char*** values range from -128 to 127 and ***unsigned char*** value range from 0 to 255. These can be summarized as follows:

Data type	Size (bytes)	Range
Short int or int	2	-32768 to 32,767
Long int	4	-2147483648 to 2147483647
Signed int	2	-32768 to 32767
Unsigned int	2	0 to 65535
Signed char	1	-128 to 127
Unsigned char	1	0 to 255

Variable is an identifier whose value changes from time to time during execution. It is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. Note that a value must be assigned to the variables at some point of time in the program which is termed as assignment statement. The variable can then be accessed later in the program. If the variable is accessed before it is assigned a value, it may give garbage value. The data type of a variable doesn't change whereas the value assigned to can change. All variables have three essential attributes:

- the name
- the value
- the memory, where the value is stored.

2.7 DECLARING VARIABLES

Before any data can be stored in the memory, we must assign a name to these locations of memory. For this we make declarations. Declaration associates a group of identifiers with a specific data type. All of them need to be declared before they appear in program statements, else accessing the variables results in junk values or a diagnostic error. The syntax for declaring variables is as follows:

data-type variable-name(s);

For example,

```
int a;  
short int a, b;  
int c, d;  
long c, f;  
float r1, r2;
```

2.8 INITIALISING VARIABLES

Variable initialization means assigning a value to the variable. Initial values can be assigned to them in two ways:

a) **Within a Type Declaration**

The value is assigned at the declaration time.

For example,

```
int      a = 10;  
float   b = 0.4 e -5;  
char    c = 'a';
```

b) Using Assignment Statement

The values are assigned just after the declarations are made.

For example,

```
int a;  
float b;  
char c;
```

```
a = 10;  
b = 0.4 e -5;  
c = 'a';
```

Check Your Progress 1

- 1) Identify keywords and valid identifiers among the following:

hello	function	day-of-the-week
student_1	max_value	“what”
1_student	int	union

.....
.....
.....
.....
.....

- 2) Declare variables roll no, total_marks and percentage with appropriate datatypes.
-
.....
.....
.....
.....

- 3) How many byte(s) are assigned to store for the following?

a) Unsigned character b) Unsigned integer c) Double

.....
.....
.....
.....
.....

2.9 CONSTANTS

A constant is an identifier whose value cannot be changed throughout the execution of a program whereas the variable value keeps on changing. In C there are four basic types of **constants**. They are:

1. Integer constants

2. Floating point constants
3. Character constants
4. String constants

Integer and Floating Point constants are numeric constants and represent numbers.

Rules to form Integer and Floating Point Constants

- No comma or blankspace is allowed in a constant.
- It can be preceded by – (minus) sign if desired.
- The value should lie within a minimum and maximum permissible range decided by the word size of the computer.

2.9.1 Integer Constants

Further, these constant can be classified according to the base of the numbers as:

1. Decimal integer constants

These consist of digits 0 through 9 and first *digit should not be 0*.

For example,

1 443 32767

are valid decimal integer constants.

2. Invalid Decimal integer Constants

12 ,45 , not allowed

1 010 Blankspace not allowed

10 – 10 – not allowed

0900 The first digit should not be a zero

3. Octal integer constants

These consist of digits 0 through 7. The first digit must be zero in order to identify the constant as an octal number.

Valid octal integer constants are:

0 01 0743 0777

Invalid octal integer constants are:

743 does not begin with 0

0438 illegal character 8

0777.77 illegal char

4. Hexadecimal integer constants

To specify a hexadecimal integer constant, start the hexadecimal sequence with a 0 followed by the character X (or x). Follow the X or x with one or more hexadecimal characters (the digits 0 to 9 and the upper or lowercase letters A to F). The value of a hexadecimal constant is computed in base 16 (the letters A to F have the values 10 to 15, respectively).

Valid Hexadecimal integer constants are:

0X0	0X1	0XF77	0xABCD
-----	-----	-------	--------

Invalid Hexadecimal integer constants are:

0BEF	x is not included
0x.4bff	illegal char (.)
0XGBC	illegal char G

Unsigned integer constants: Exceed the ordinary integer by magnitude of 2, they are not negative. A character U or u is postfix to the number to make it unsigned.

Long Integer constants: These are used to exceed the magnitude of ordinary integers and are appended by L.

For example,

50000U	decimal unsigned
1234567889L	decimal long
0123456L	octal long
0777777U	octal unsigned

2.9.2 Floating Point Constants

A floating-point constant consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- e or E and a signed integer exponent (optional)
- Type suffix: f or F or l or L (optional)

Either decimal integer or decimal fraction (but not both) can be omitted.

Either decimal point or letter e (or E) with a signed integer exponent (but not both) can be omitted. These rules allow conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with an unary operator minus (-) prefixed. If there is a need for a floating-point constant that exceeds these limits, user should add l or L suffix, making the constant a long double type.

Here are some examples:

0.	// is equal to 0.0
-1.23	// is equal to -1.23
23.45e6	// is equal to 23.45 X 10 ⁶
2e-5	// is equal to 2.0 X 10 ⁻⁵
3e+10	// is equal to 3.0 X 10 ¹⁰
.09E34	// is equal to 0.09 X 10 ³⁴

2.9.3 Character Constants

This constant is a single character enclosed in apostrophes (' ').

For example, some of the character constants are shown below:

‘A’, ‘x’, ‘3’, ‘\$’

‘\0’ is a null character having value zero.

Character constants have integer values associated depending on the character set adopted for the computer. ASCII character set is in use which uses 7-bit code with $2^7 = 128$ different characters. The digits 0-9 have ASCII values of 48-57, upper case alphabets from ‘A’ to ‘Z’ have ASCII values from 65 to 90 and lower case alphabets ‘a’ to ‘z’ have ASCII values from 97 to 122.

Escape Sequence

Many programming languages support a concept called Escape Sequence. When a character is preceded by a backslash (\), it is called an escape sequence and it has a special meaning to the compiler. For example, \n in the following statement is a valid character and it is called a new line character in C language.

2.9.4 String Constants

It consists of sequence of characters enclosed within double quotes. For example,

“red” “Blue Sea” “41213*(I+3)”

2.10 SYMBOLIC CONSTANTS AND OTHERS

Symbolic Constant is a name that substitutes for a sequence of characters or a numeric constant, a character constant or a string constant. When program is compiled each occurrence of a symbolic constant is replaced by its corresponding character sequence. The syntax is as follows:

#define *name* *text*

where *name* implies symbolic name in caps.
text implies value or the text.

Examples:

```
#define printf print
#define MAX 100
#define TRUE 1
#define FALSE 0
#define SIZE 10
#define PI 3.141592
```

The # character is used for preprocessor commands. A *preprocessor* is a system program, which comes into action prior to Compiler, and it replaces the replacement text by the actual text. This will allow correct use of the statement printf.

Advantages of using Symbolic Constants are:

- They can be used to assign names to values.
- Replacement of value has to be done at one place and wherever the name appears in the text it gets the value by execution of the preprocessor. This saves time. if the symbolic constant appears 20 times in the program; it needs to be changed at one place only.

Enumerated Data Type

An enumerated type is used to specify the possible values of an object from a predefined list. Elements of the list are called *enumeration constants*. The main use of enumerated types is to explicitly show the symbolic names, and therefore the intended purpose, of objects whose values can be represented with integer values. Objects of enumerated type are interpreted as objects of type `signed int`, and are compatible with objects of other integral types.

The compiler automatically assigns integer values to each of the enumeration constants, beginning with 0. The following example declares an enumerated object `background_color` with a list of enumeration constants:

```
enum colors {black,red,blue,green,white} background_color;
```

Later in the program, a value can be assigned to the object `background_color`:

```
background_color = white;
```

In this example, the compiler automatically assigns the integer values as follows: `black = 0`, `red = 1`, `blue = 2`, `green = 3`, and `white = 4`. Alternatively, explicit values can be assigned during the enumerated type definition:

```
enum colors { black = 5, red = 10, blue, green = 7, white = green+2 };
```

Here, `black` equals the integer value 5, `red` = 10, `blue` = 11, `green` = 7, and `white` = 9. Note that `blue` equals the value of the previous constant (`red`) plus one, and `green` is allowed to be out of sequential order.

Because the ANSI C standard is not strict about assignment to enumerated types, any assigned value not in the predefined list is accepted without complaint.

Typedef in C Language

typedef keyword is used to assign a new name to a type. This is used just to prevent us from writing more.

For example, if we want to declare some variables of type `unsigned int`, we have to write `unsigned int` in a program and it can be quite hectic for some of us. So, we can assign a new name of our choice for `unsigned int` using **typedef** which can be used anytime we want to use `unsigned int` in a program.

```
typedef current_name new_name;
```

```
typedef unsigned int uint;
```

```
uint j,k;
```

Now, we can write *uint* in the whole program instead of unsigned int. The above code is the same as writing:

```
unsigned int j,k;
```

For example,

```
#include<stdio.h>
int main()
{
    typedef unsigned int uint;
    uint j=5, k=9;
    printf("j= %d\n",j);
    printf("k= %d\n",k);
    return 0;
}
```

Check Your Progress 2

- 1) Write a preprocessor directive statement to define a constant PI having the value 3.14.

'A'	0147	0xEFH
077.7	"A"	26.4
"EFH"	'r'	abc

- 2) Classify the examples into Integer, Character and String constants.

'A'	0147	0xEFH
077.7	"A"	26.4
"EFH"	'r'	abc

- 3) Name different categories of constants C programming language.

2.11 EXPRESSIONS AND OPERATORS - AN INTRODUCTION

In the previous sections' we have learnt variables, constants, datatypes and how to declare them in C programming. The next step is to use those variables in expressions. For writing an expression we need operators along with variables. An *expression* is a sequence of operators and operands that does one or a combination of the following:

- specifies the computation of a value
- designates an object or function
- generates side effects.

An *operator* performs an operation (evaluation) on one or more operands. An *operand* is a subexpression on which an operator acts.

This unit focuses on different types of operators available in C including the syntax and use of each operator and how they are used in C.

A computer is different from calculator in a sense that it can solve logical expressions also. Therefore, apart from arithmetic operators, C also contains logical operators. Hence, logical expressions are also discussed in the following sections.

2.12 ASSIGNMENT STATEMENT

In the previous unit, we have seen that variables are basically memory locations and they can hold certain values. But, how to assign values to the variables? C provides an assignment operator for this purpose. The function of this operator is to assign the values or values in variables on right hand side of an expression to variables on the left hand side.

The syntax of the assignment expression is as follows:

variable = constant / variable/ expression;

The data type of the variable on left hand side should match the data type of constant/variable/expression on right hand side with a few exceptions where automatic type conversions are possible. Some examples of assignment statements are as follows:

```
b = a ; /* b is assigned the value of a */
b = 5 ; /* b is assigned the value 5*/
b = a+5; /* b is assigned the value of expr a+5 */
```

The expression on the right hand side of the assignment statement can be:

- an arithmetic expression;
- a relational expression;
- a logical expression;
- a mixed expression.

The above mentioned expressions are different in terms of the type of operators connecting the variables and constants on the right hand side of the variable. Arithmetic operators, relational operators and logical operators are discussed in the following sections.

For example,

```
int a;
float b,c ,avg, t;
avg = (b+c) / 2;           /*arithmetic expression */
a = b && c;              /*logical expression*/
a = (b+c) && (b<c);     /* mixed expression*/
```

2.13 ARITHMETIC OPERATORS

The basic arithmetic operators in C are the same as in most other computer languages, and correspond to our usual mathematical/algebraic symbolism. The following arithmetic operators are present in C:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modular Division

Some of the examples of algebraic expressions and their C notation are given below:

Expression	C notation
$\frac{b * g}{d}$	$(b * g) / d$
$a^3 + cd$	$(a * a * a) + (c * d)$

The arithmetic operators are all binary operators i.e. all the operators have two operands. The integer division yields the integer result. For example, the expression 10/3 evaluates to 3 and the expression 15/4 evaluates to 3. C provides the modulus operator, %, which yields the remainder after integer division. The modulus operator is an integer operator that can be used only with integer operands. The expression $x \% y$ yields the remainder after x is divided by y . Therefore, 10%3 yields 1 and 15%4 yields 3. An attempt to divide by zero is undefined on computer system and generally results in a run-time error. Normally, Arithmetic expressions in C are written in straight-line form. Thus 'a divided by b' is written as a/b .

The operands in arithmetic expressions can be of integer, float, double type. In order to effectively develop C programs, it will be necessary for you to understand the rules that are used for implicit conversion of floating point and integer values in C.

They are mentioned below:

- An arithmetic operator between an integer and integer always yields an integer result.
- Operator between float and float yields a float result.
- Operator between integer and float yields a float result.

If the data type is double instead of float, then we get a result of double data type.

For example,

Operation	Result
5/3	1
5.0/3	1.6666666667
5/3.0	1.6666666667
5.0/3.0	1.6666666667

Parentheses can be used in C expression in the same manner as algebraic expression For example,

$$a * (b + c)$$

It may so happen that the type of the expression and the type of the variable on the left hand side of the assignment operator may not be same. In such a case the value for the expression is promoted or demoted depending on the type of the variable on left hand side of = (assignment operator). For example, consider the following assignment statements:

```
int i;
float b;
i = 4.6;
b = 20;
```

In the first assignment statement, float (4.6) is demoted to int. Hence *i* gets the value 4. In the second assignment statement int (20) is promoted to float, *b* gets 20.0. If we have a complex expression like:

```
float a, b, c;
int s;
s = a * b / 5.0 * c;
```

Where some operands are integers and some are float, then int will be promoted or demoted depending on left hand side operator. In this case, demotion will take place since *s* is an integer.

The rules of arithmetic precedence are as follows:

1. Parentheses are at the “highest level of precedence”. In case of nested parenthesis, the innermost parentheses are evaluated first.

For example,

$((3+4)*5)/6$

The order of evaluation is given below:

$$((3+4)*5)/6$$

↓ ↓ ↓
1 2 3

2. Multiplication, Division and Modulus operators are evaluated next. If an expression contains several multiplication, division and modulus operators, evaluation proceeds from left to right. These three are at the same level of precedence.

For example,

$5*5+6*7$

The order of evaluation is given below.

$$5*5+6*7$$

↓ ↓ ↓
1 2
3

3. Addition, subtraction are evaluated last. If an expression contains several addition and subtraction operators, evaluation proceeds from left to right. Or the associativity is from left to right.

For example,

$8/5-6+5/2$

The order of evaluation is given below.

$$8/5-6+5/2$$

↓ ↓ ↓
1 3 4 2

Apart from these binary arithmetic operators, C also contains two unary operators referred to as increment (++) and decrement (--) operators, which we are going to be discussed below:

The two-unary arithmetic operators provided by C are:

- **Increment operator (++)**
- **Decrement operator (--)**

The increment operator increments the variable by one and decrement operator decrements the variable by one. These operators can be written in two forms i.e. before a variable or after a variable. If an **increment / decrement** operator is written before a variable, it is referred to as

preincrement / predecrement operators and if it is written after a variable, it is referred to as **post increment / postdecrement** operator.

For example,

a++ or ++a is equivalent to a = a+1 and
a-- or - -a is equivalent to a = a -1

The importance of **pre** and **post** operator occurs while they are used in the expressions. **Preincrementing (Predecrementing)** a variable causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears. **Postincrementing (postdecrementing)** the variable causes the current value of the variable is used in the expression in which it appears, then the variable value is incremented (decrement) by 1.

The explanation is given in the table below:

Expression	Explanation
++a	Increment a by 1, then use the new value of a
a++	Use value of a, then increment a by 1
--b	Decrement b by 1, then use the new value of b
b--	Use the current value of b, then decrement by 1

The precedence of these operators is right to left. Let us consider the following examples:

```
int a = 2, b=3;
int c;
c = ++a - b--;
printf ("a=%d, b=%d,c=%d\n",a,b,c);
```

OUTPUT

a = 3, b = 2, c = 0.

Since the precedence of the operators is right to left, first b is evaluated, since it is a post decrement operator, current value of b will be used in the expression i.e. 3 and then b will be decremented by 1. Then, a preincrement operator is used with a, so first a is incremented to 3. Therefore, the value of the expression is evaluated to 0.

Let us take another example,

```
int a = 1, b = 2, c = 3;
int k;
k = (a++)*(++b) + ++a - --c;
printf("a=%d,b=%d, c=%d, k=%d",a,b,c,k);
```

OUTPUT

a = 3, b = 3, c = 2, k = 6

The evaluation is explained below:

$$\begin{aligned}
 k &= (a++) * (++b) + ++a - --c \\
 &= (a++) * (3) + 2 - 2 \quad \text{step1} \\
 &= (2) * (3) + 2 - 2 \quad \text{step2} \\
 &= 6 \quad \text{final result}
 \end{aligned}$$

Check Your Progress 3

1. Give the C expressions for the following algebraic expressions:

i) $\frac{a*4c^2 - d}{m+n}$

ii) $ab - \frac{(e+f)4}{c}$

.....
.....
.....
.....
.....
.....

2. Give the output of the following C code:

```
main()
{
    int a=2,b=3,c=4;
    k = ++b + --a*c + a;
    printf("a= %d b=%d c=%d k=%d\n",a,b,c,k);
}
```

.....
.....
.....
.....
.....
.....

3. Point out the error:

`exp = a**b;`

.....
.....
.....
.....
.....
.....

2.14 RELATIONAL OPERATORS

Executable C statements either perform actions (such as calculations or input or output of data) or make decision. Using relational operators we can compare two variables in the program. The C relational operators are summarized below, with their meanings. Pay particular attention to the equality operator; it consists of two equal signs, not just one. This section introduces a simple version of C's **if** control structure that allows a program to make a decision based on the result of some condition. If the condition is true then the statement in the body of if statement is executed else if the condition is false, the statement is not executed. Whether the body statement

is executed or not, after the if structure completes, execution proceeds with the next statement after the if structure. Conditions in the **if** structure are formed with the relational operators which are summarized in the Table 2.2.

Table 2.2: Relational Operators in C

Relational Operator	Condition	Meaning
==	x==y	x is equal to y
!=	x!=y	x is not equal to y
<	x<y	x is less than y
<=	x<=y	x is less than or equal to y
>	x>y	x is greater than y
>=	x>=y	x is greater or equal to y

Relational operators usually appear in statements which are inquiring about the truth of some particular relationship between variables. Normally, the relational operators in C are the operators in the expressions that appear between the parentheses.

For example,

- i) if (thisNum < minimumSoFar) minimumSoFar = thisNum
- ii) if (job == Teacher) salary == minimumWage
- iii) if (numberOfLegs != 8) thisBug = insect
- iv) if (degreeOfPolynomial < 2) polynomial = linear

Let us see a simple C program given below containing the **if statement** (will be introduced in detail in the next unit). It displays the relationship between two numbers read from the keyboard.

*/*Program to find relationship between two numbers*/*

```
#include <stdio.h>
main()
{
int a, b;
printf("Please enter two integers: ");
scanf ("%d%d", &a, &b);
if (a <= b)
printf(" %d <= %d\n",a,b);
else
printf(" %d > %d\n",a,b);
}
```

OUTPUT

Please enter two integers: 12 17

12 <= 17

We can change the values assigned to a and b and check the result.

2.15 LOGICAL OPERATORS

Logical operators in C, as with other computer languages, are used to evaluate expressions which may be true or false. Expressions which involve logical operations are evaluated and found to be one of two values: **true or false**. So far we have studied simple conditions. If we want to test multiple conditions in the process of making a decision, we have to perform simple tests in separate IF statements (will be introduced in detail in the next unit). C provides logical operators that may be used to form more complex conditions by combining simple conditions.

The logical operators are listed below:

Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

Thus logical operators (AND and OR) combine two conditions and logical NOT is used to negate the condition i.e. if the condition is true, NOT negates it to false and vice versa. Let us consider the following is:

- i) Suppose the grade of the student is ‘B’ only if his marks lie within the range 65 to 75, if the condition would be:

```
if((marks >=65) && (marks <= 75))
printf ("Grade is B\n");
```

- ii) Suppose we want to check that a student is eligible for admission if his PCM is greater than 85% or his aggregate is greater than 90%, then,

```
if((PCM >=85) ||(aggregate >=90))
printf ("Eligible for admission\n");
```

Logical negation (!) enables the programmer to reverse the meaning of the condition. Unlike the && and || operators, which combines two conditions (and are therefore Binary operators), the logical negation operator is a unary operator and has one single condition as an operand. Let us consider an example:

```
if!(grade=='A')
printf ("the next grade is %c\n", grade);
```

The parentheses around the condition grade==A are needed because the logical operator has higher precedence than equality operator. The truth table of the logical AND (&&), OR (||) and NOT (!) are given below.

These table show the possible combinations of zero (false) and nonzero (true) values of x (expression1) and y (expression2) and only one expression in case of NOT operator. The following table 2.3 is the truth table for && operator.

Table 2.3: Truth table for && operator

x	y	x&&y
zero	zero	0
Non zero	zero	0
zero	Non zero	0
Non zero	Non zero	1

The following table 2.4 is the truth table for || operator.

Table 2.4: Truth table for || operator

x	y	x y
zero	zero	0
Non zero	zero	1
zero	Non zero	1
Non zero	Non zero	1

The following table 2.5 is the truth table for ! operator.

Table 2.5: Truth table for ! operator

x	! x
zero	1
Non zero	0

The following table 2.6 shows the operator precedence and associativity

Table 2.6: (Logical operators precedence and associativity)

Operator	Associativity
!	Right to left
&&	Left to right
	Left to right

2.16 COMMA AND CONDITIONAL OPERATORS

Conditional Operator

C provides an called as the conditional operator (?:) or else called as *ternary* operator which is closely related to the **if/else** structure. The conditional operator is C's only ternary operator - it takes three operands. The operands together with the conditional operator form a conditional expression. The first operand is a condition, the second operand represents the value of the entire conditional expression if the condition is true and the third operand is the value for the entire conditional expression if the condition is false.

The syntax is as follows:

(condition)? (expression1): (expression2);

If condition is true, expression1 is evaluated else expression2 is evaluated. Expression1/Expression2 can also be further conditional expression i.e. the case of nested if statement (will be discussed in the next unit).

Let us see the following examples:

i) $x = (y < 20) ? 9 : 10;$

This means, if ($y < 20$), then $x = 9$ else $x = 10$;

ii) $\text{printf}(\text{"%s\n"}, \text{grade} >= 50 ? \text{"Passed"} : \text{"Failed"});$

The above statement will print “passed” if $\text{grade} >= 50$ else it will print “failed”

iii) $(a > b) ? \text{printf}(\text{"a is greater than b\n"}) : \text{printf}(\text{"b is greater than a\n"});$

If a is greater than b , then first printf statement is executed else second printf statement is executed.

Comma Operator

A comma operator is used to separate a pair of expressions. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the value of the type and value of the right operand. All side effects from the evaluation of the left operand are completed before beginning evaluation of the right operand. The left side of comma operator is always evaluated to void. This means that the expression on the right hand side becomes the value of the total comma-separated expression. For example,

$x = (y = 2, y - 1);$

first assigns y the value 2 and then x the value 1. Parenthesis is necessary since comma operator has lower precedence than assignment operator.

Generally, comma operator (,) is used in the for loop (will be introduced in the next unit)

For example,

```
for(i= 0,j= n;i<j; i++,j--)
{
    printf("A");
}
```

In this example **for** is the looping construct (discussed in the next unit). In this loop, $i = 0$ and $j = n$ are separated by comma (,), $i++$ and j —are separated by comma (,). The example will be clear to you once you have learnt for loop (will be introduced in the next unit).

Essentially, the comma causes a sequence of operations to be performed. When it is used on the right hand side of the assignment statement, the value assigned is the value of the last expression in the comma-separated list.

Check Your Progress 4

1. Given $a=3$, $b=4$, $c=2$, what is the result of following logical expressions:
 $(a < --b) \&\& (a == c)$

.....

2. Give the output of the following code:

```
main()
{
    int a=10, b=15,x;
    x = (a<b)?++a:++b;
    printf("x=%d a=%d b=%d\n",x,a,b);
}
```

.....

3. What is the use of comma operator?

.....

2.17 TYPE CAST OPERATOR

We have seen in the previous sections and last unit that when constants and variables of different types are mixed in an expression, they are converted to the same type. That is automatic type conversion takes place. The following type conversion rules are followed:

1. All chars and **short ints** are converted to **ints**. All floats are converted to doubles.
2. In case of binary operators, if one of the two operands is a **long double**, the other operand is converted to **long double**,

else if one operand is **double**, the other is converted to **double**,
else if one operand is **long**, the other is converted to **long**,
else if one operand is **unsigned**, the other is converted to
unsigned,

C converts all operands “up” to the type of largest operand (largest in terms of memory requirement for e.g. **float** requires 4 bytes of storage and **int** requires 2 bytes of storage so if one operand is **int** and the other is **float**, **int** is converted to **float**).

All the above mentioned conversions are automatic conversions, but what if **int** is to be converted to **float**. It is possible to force an expression to be of specific type by using operator called a *cast*. The syntax is as follows:

(type) expression

where *type* is the standard C data type. For example, if you want to make sure that the expression a/5 would evaluate to type **float** you would write it as

(float) a/5

cast is an unary operator and has the same precedence as any other unary operator. The use of *cast* operator is explained in the following example:

```
main()
{
    int num;
    printf("%f %f %f\n", (float)num/2, (float)num/3, float)num/3);
}
```

The *cast* operator in this example will ensure that fractional part is also displayed on the screen.

2.18 SIZE OF OPERATOR

C provides a compile-time unary operator called **sizeof** that can be used to compute the size of any object. The expressions such as:

sizeof object and **sizeof(type name)**

result in an unsigned integer value equal to the size of the specified object or type in bytes. Actually the resultant integer is the number of bytes required to store an object of the type of its operand. An object can be a variable or array or structure. An array and structure are data structures provided in C, introduced in latter units. A type name can be the name of any basic type like **int** or **double** or a derived type like a structure or a pointer.

For example,

sizeof(char) = 1bytes

sizeof(int) = 2 bytes

2.19 C SHORTHAND

C has a special shorthand that simplifies coding of certain type of assignment statements. For example: `a = a+2;`

can be written as: `a += 2;`

The operator `+=`tells the compiler that `a` is assigned the value of `a + 2;`

This shorthand works for all binary operators in C. The general form is:

variable operator = variable / constant / expression;

These operators are listed below:

Operators	Examples	Meaning
<code>+=</code>	<code>a+=2</code>	<code>a=a+2</code>
<code>-=</code>	<code>a-=2</code>	<code>a=a-2</code>
<code>=</code>	<code>a*=2</code>	<code>a = a*2</code>
<code>/=</code>	<code>a/=2</code>	<code>a=a/2</code>
<code>%=</code>	<code>a%=2</code>	<code>a=a%2</code>

Operators	Examples	Meaning
<code>&&=</code>	<code>a&&=c</code>	<code>a=a&&c</code>
<code> =</code>	<code>a =c</code>	<code>a=a c</code>

2.20 PRIORITY OF OPERATORS

Since all the operators we have studied in this unit can be used together in an expression, C uses a certain hierarchy to solve such kind of mixed expressions. The hierarchy and associativity of the operators discussed so far is summarized in Table 2.7. The operators written in the same line have the same priority. The higher precedence operators are written first.

Table 2.7: Precedence of the operators

Operators	Associativity
<code>()</code>	Left to right
<code>! ++ -- (type) sizeof</code>	Right to left
<code>/ %</code>	Left to right
<code>+ -</code>	Left to right
<code>< <= > >=</code>	Left to right
<code>== !=</code>	Left to right
<code>&&</code>	Left to right
<code> </code>	Left to right
<code>? :</code>	Right to left
<code>= += -= *= /= %= &&= =</code>	Right to left
<code>,</code>	Left to right

Check Your Progress 5

1. Give the output of the following C code:

```
main( )  
{  
    int a,b=5;  
    float f;  
    a=5/2;  
    f=(float)b/2.0;  
    (a<f)? b=1:b=0;  
    printf("b = %d\n",b);  
}
```

2. What is the difference between `&&` and `&`. Explain with an example.

3. Use of Bit Wise operators makes the execution of the program.

2.21 SUMMARY

To summarize we have learnt certain basics, which are required to learn a computer language and form a basis for all languages. Character set includes alphabets, numeric characters, special characters and some graphical characters. These are used to form words in C language or names or identifiers. Variable are the identifiers, which change their values during execution of the program. Keywords are names with specific meaning and cannot be used otherwise.

We had discussed four basic data types - int, char, float and double. Some qualifiers are used as prefixes to data types like signed, unsigned, short, and long.

The constants are the fixed values and may be either Integer or Floating point or Character or String type. Symbolic Constants are used to define names used for constant values. They help in using the name rather bothering with remembering and writing the values.

In this unit, we discussed about the different types of operators, namely arithmetic, relational, logical present in C and their use. In the following units, you will study how these are used in C's other constructs like control statements, arrays etc. This unit also focused on type conversions. Type conversions are very important to understand because sometimes a programmer gets unexpected results (logical error) which are most often caused by type conversions in case user has used improper types or if he has not type cast to desired type.

C is referred to as a compact language which is because lengthy expressions can be written in short form. Conditional operator is one of the examples, which is the short form of writing the if/else construct (next unit). Also increment/decrement operators reduce a bit of coding when used in expressions.

Since logical operators are used further in all types of looping constructs and if/else construct (in the next unit), they should be thoroughly understood.

2.22 SOLUTIONS / ANSWERS

Check Your Progress 1

1. **Keywords:** int, union

Valid Identifiers: hello, student_1, max_value

2. int rollno;
- float total_marks, percentage;
3. a) 1 byte b) 2 bytes c) 8 bytes

Check Your Progress 2

1. # define PI 3.14
2. **Integer constant:** 0147

Character constants: 'A', '\r'

String constants: "A", "EFH"

Check Your Progress 3

1. C expression would be
 - i) ((a*4*c*c)-d)/(m+n)
 - ii) a*b-(e+f)*4/c

2. The output would be:
 $a=1 b=4 c=4 k=10$
3. There is no such operator as **.

Check Your Progress 4

1. The expression is evaluated as under:

```
(3 < -4) && (3==2)  
(3 < 3) && (3==2)  
0 && 0  
0
```

Logical false evaluates to 0 and logical true evaluates to 1.

2. The output would be as follows:
 $x=11, a=11, b=16$
3. Comma operator causes a sequence of operators to be performed.

Check Your Progress 5

1. Here a will evaluate to 2 and f will evaluate to 2.5 since type cast operator is used in the latter so data type of b changes to float in an expression. Therefore, output would be b=1.
2. && operator is a logical and operator and & is a bit wise and operator. Therefore, && operator always evaluates to true or false i.e 1 or 0 respectively while & operator evaluates bit wise so the result can be any value. For example:

```
2 && 5 => 1(true)  
2 & 5 => 0(bit-wise anding)
```

3. Use of Bit Wise operators makes the execution of the program faster.

2.23 FURTHER READINGS

1. The C Programming Language, *Kernighan & Ritchie*, PHI Publication.
2. Computer Science A structured programming approach using C, *Behrouza A. Forouzan, Richard F. Gilberg*, Second Edition, Brooks/Cole, Thomson Learning, 2001.
3. Programming with C, *Gottfried*, Second Edition, Schaum Outlines, Tata Mc Graw Hill, 2003.

UNIT 3 DECISION AND LOOP CONTROL STATEMENTS

Structure

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Decision Control Statements
 - 3.2.1 The *if* Statement
 - 3.2.2 The *switch* Statement
- 3.3 Loop Control Statements
 - 3.3.1 The *while* Loop
 - 3.3.2 The *do-while* Statement
 - 3.3.3 The *for* Loop
 - 3.3.4 The Nested Loop
- 3.4 The *Goto* Statement
- 3.5 The *Break* Statement
- 3.6 The *Continue* Statement
- 3.7 Summary
- 3.8 Solutions / Answers
- 3.9 Further Readings

3.0 INTRODUCTION

A *program* consists of a number of statements to be executed by the computer. Not many of the programs execute all their statements in sequential order from beginning to end as they appear within the program. A *C program* may require that a logical test be carried out at some particular point within the program. One of the several possible actions will be carried out, depending on the outcome of the *logical test*. This is called **Branching**. In the **Selection** process, a set of statements will be selected for execution, among the several sets available. Suppose, if there is a need of a group of statements to be executed repeatedly until some logical condition is satisfied, then **looping** is required in the program. These can be carried out using various control statements.

These **Control statements** determine the “*flow of control*” in a program and enable us to specify the order in which the various instructions in a program are to be executed by the computer. Normally, high level procedural programming languages require three basic control statements:

- Sequence instruction
- Selection/decision instruction
- Repetition or Loop instruction

Sequence instruction means executing one instruction after another, in the order in which they occur in the source file. This is usually built into the language as a default action, as it is with C. If an instruction is not a control statement, then the next instruction to be executed will simply be the next one in sequence.

Selection means executing different sections of code depending on a specific condition or the value of a variable. This allows a program to take different courses of action depending on different conditions. C provides three selection structures.

- *if*
- *if...else*
- *switch*

Repetition/Looping means executing the same section of code more than once. A section of code may either be executed a fixed number of times, or while some condition is true. C provides three looping statements:

- *while*
- *do...while*
- *for*

This unit introduces you the decision and loop control statements that are available in C programming language along with some of the example programs.

3.1 OBJECTIVES

After going through this unit you will be able to:

- work with different control statements;
- know the appropriate use of the various control statements in programming;
- transfer the control from within the loops;
- use the *goto*, *break* and *continue* statements in the programs; and
- write programs using branching, looping statements.

3.2 DECISION CONTROL STATEMENTS

In a C program, a decision causes a one-time jump to a different part of the program, depending on the value of an expression. Decisions in C can be made in several ways. The most important is with the *if...else* statement, which chooses between two alternatives. This statement can be used without the *else*, as a simple *if* statement. Another decision control statement, *switch*, creates branches for multiple alternative sections of code, depending on the value of a single variable.

3.2.1 The *if* Statement

It is used to execute an *instruction* or sequence/*block of instructions* only if a *condition* is fulfilled. In *if* statements, expression is evaluated first and then, depending on whether the value of the expression (relation or condition) is “*true*” or “*false*”, it transfers the control to a particular statement or a group of statements.

Different forms of implementation *if*-statement are:

- Simple *if* statement
- *If-else* statement
- *Nested if-else* statement
- *Else if* statement

Simple *if* statement

It is used to execute an instruction or block of instructions only if a condition is fulfilled.

The syntax is as follows:

```
if(condition)
    statement;
```

where *condition* is the expression that is to be evaluated. If this *condition* is *true*, *statement* is executed. If it is *false*, *statement* is ignored (not executed) and the program continues on the next instruction after the conditional statement.

This is shown in the Figure 3.1 given below:

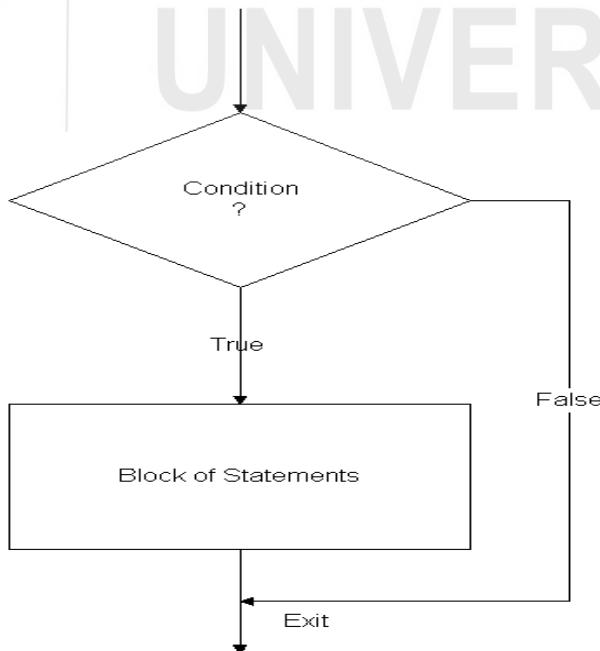


Figure 3.1: Simple *if* statement

If we want more than one statement to be executed, then we can specify a block of statements within the curly braces { }. The syntax is as follows:

```
if(condition)
{
    block of statements;
}
```

Example 3.1

Write a program to calculate the net salary of an employee, if a tax of 15% is levied on his gross-salary if it exceeds Rs. 10,000/- per month.

```
/*Program to calculate the net salary of an employee */
```

```
#include<stdio.h>
main()
{
float gross_salary, net_salary;

printf("Enter gross salary of an employee\n");
scanf("%f",&gross_salary );

if(gross_salary <10000)
    net_salary= gross_salary;
if(gross_salary >= 10000)
    net_salary = gross_salary- 0.15*gross_salary;

printf("\nNet salary is Rs.%2f\n", net_salary);
}
```

OUTPUT

```
Enter gross salary of an employee
9000
Net salary is Rs.9000.00
```

```
Enter gross salary of any employee
10000
Net salary is Rs. 8500.00
```

If ... else statement

If...else statement is used when a different sequence of instructions is to be executed depending on the logical value (*True / False*) of the condition evaluated.

Its form used in conjunction with *if* and the syntax is as follows:

```
if(condition)
    Statement_1;
else
    Statement_2;
statement_3;
```

Or

```
if(condition)
{
    Statements_1_Block;
}
else
{
    Statements_2_Block;
}
Statements_3_Block;
```

If the *condition* is **true**, then the sequence of statements (*Statements_1_Block*) executes; otherwise the *Statements_2_Block* following the *else* part of *if-else* statement will get executed. In both the cases, the control is then transferred to *Statements_3* to follow sequential execution of the program.

This is shown in figure 5.2 given below:

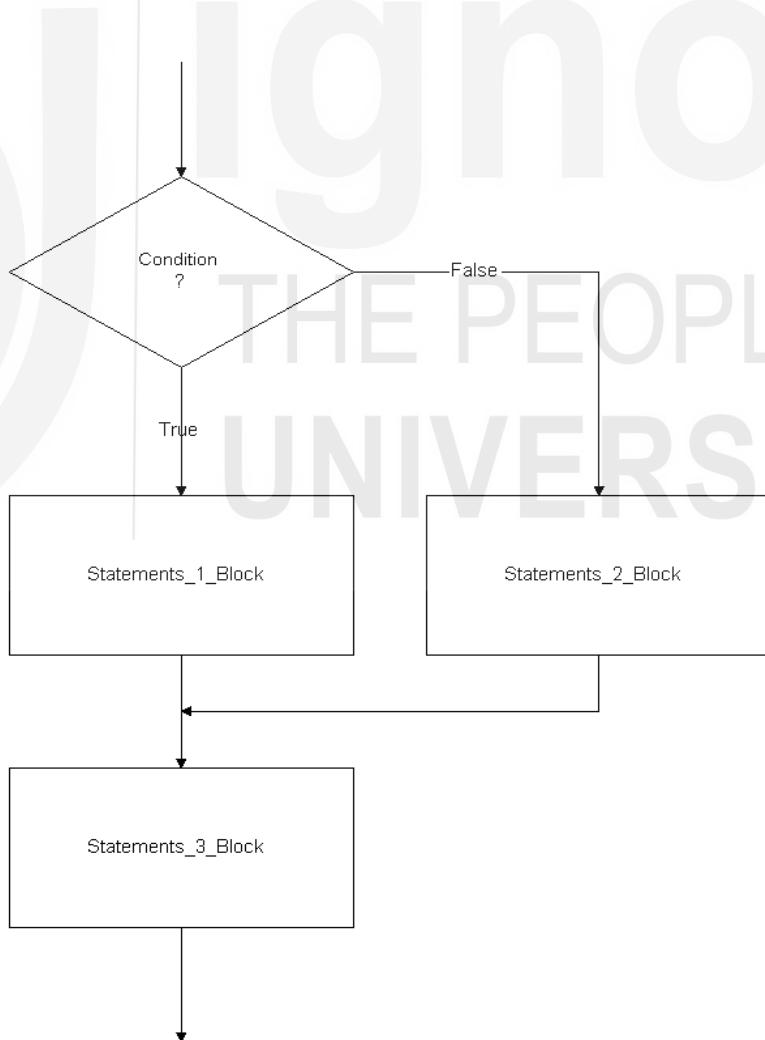


Figure 3.2: If...else statement

Example 3.2

Write a program to print whether the given number is even or odd.

```
/* Program to print whether the given number is even or odd*/
#include<stdio.h>
main()
{
int x;
printf("Enter a number:\n");
scanf("%d",&x);
if(x % 2 == 0)
    printf("\nGiven number is even\n");
else
    printf("\nGiven number is odd\n");
}
```

OUTPUT

Enter a number: 6

Given number is even

Enter a number 7

Given number is odd

Conditional expression using Ternary Operator (?:)

There is another way to express an if-else statement is by introducing the ?: (ternary operator). In a conditional expression the ?: operator has only one statement associated with the if and the else. The syntax is

variable = expression1 ? expression2: expression3;

Example:

```
#include<stdio.h>
main()
{
int x=2;
int y;
y = (x >= 6) ? 6 : x;
printf("y = %d",y);
return 0;
}
```

OUTPUT : y = 2

Nested if...else statement

In *nested if... else statement*, an entire *if...else* construct is written within either the body of the *if* statement or the body of an *else* statement. The syntax is as follows:

```
if(condition_1)
{
```

```

if(condition_2)
{
    Statements_1_Block;
}

else
{
    Statements_2_Block;
}

else
{
    Statements_3_Block;
}

Statement_4_Block;

```

Here, *condition_1* is evaluated. If it is **false** then *Statements_3_Block* is executed and is followed by the execution of *Statements_4_Block*, otherwise if *condition_1* is **true**, then *condition_2* is evaluated. *Statements_1_Block* is executed when *condition_2* is **true** otherwise *Statements_2_Block* is executed and then the control is transferred to *Statements_4_Block*.

This is shown in the figure 3.3 given in the next page:

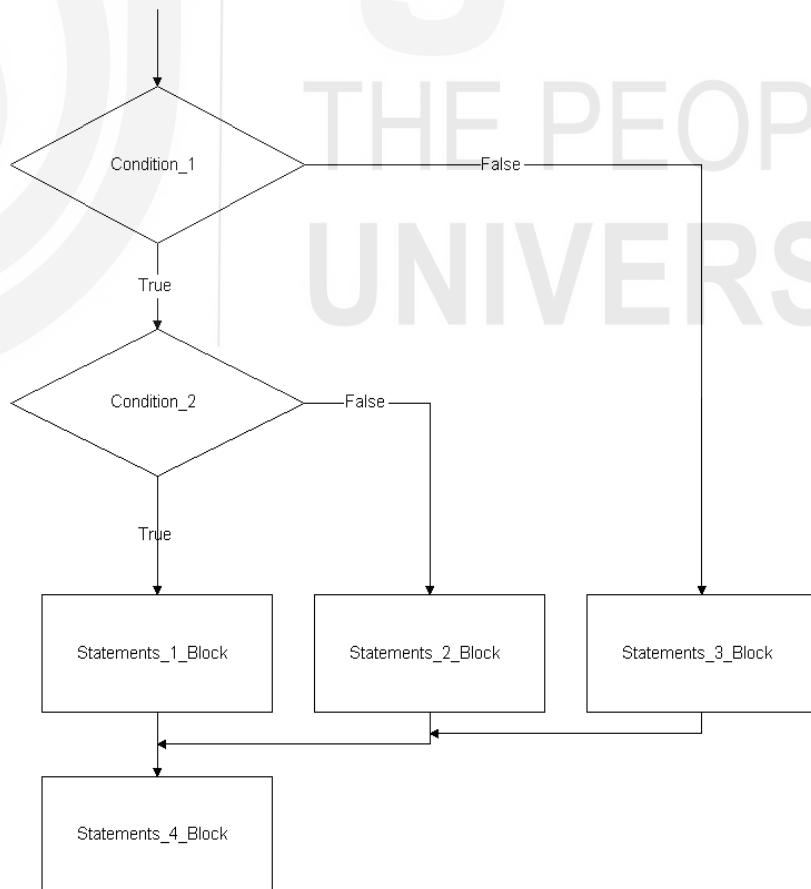


Figure 3.3: Nested if...else statement

Let us consider a program to illustrate Nested if...else statement,

Example 3.3

Write a program to calculate an Air ticket fare after discount, given the following conditions:

- If passenger is below 14 years then there is 50% discount on fare
- If passenger is above 50 years then there is 20% discount on fare
- If passenger is above 14 and below 50 then there is 10% discount on fare.

```
/* Program to calculate an Air ticket fare after discount */
```

```
#include<stdio.h>
main()
{
    int age;
    float fare;
    printf("\n Enter the age of passenger:\n");
    scanf("%d",&age);
    printf("\n Enter the Air ticket fare\n");
    scanf("%f",&fare);
    if(age<14)
        fare=fare-0.5*fare;
    else
        if(age<=50)
    {
        fare=fare-0.1*fare;
    }
    else
    {
        fare=fare-0.2*fare;
    }
    printf("\n Air ticket fare to be charged after discount is %.2f",fare);
    return 0;
}
```

OUTPUT

Enter the age of passenger 12

Enter the Air ticket fare 2000.00

Air ticket fare to be charged after discount is 1000.00

Else if statement

To show a multi-way decision based on several conditions, we use the *else if* statement. This works by cascading of several comparisons. As soon as one of the conditions is true, the statement or block of statements following them is executed and no further comparisons are performed.

The syntax is as follows:

```
if(condition_1)
{
    Statements_1_Block;
```

```

}
else if(condition_2)
{
    Statements_2_Block;
}

-----
else if(condition_n)
{
    Statements_n_Block;
}

else
    Statements_x;

```

Here, the *conditions* are evaluated in order from top to bottom. As soon as any condition evaluates to *true*, then the statement associated with the given condition is executed and control is transferred to *Statements_x* skipping the rest of the conditions following it.

But if all conditions evaluate *false*, then the statement following final *else* is executed followed by the execution of *Statements_x*.

This is shown in the figure 5.4 given below:

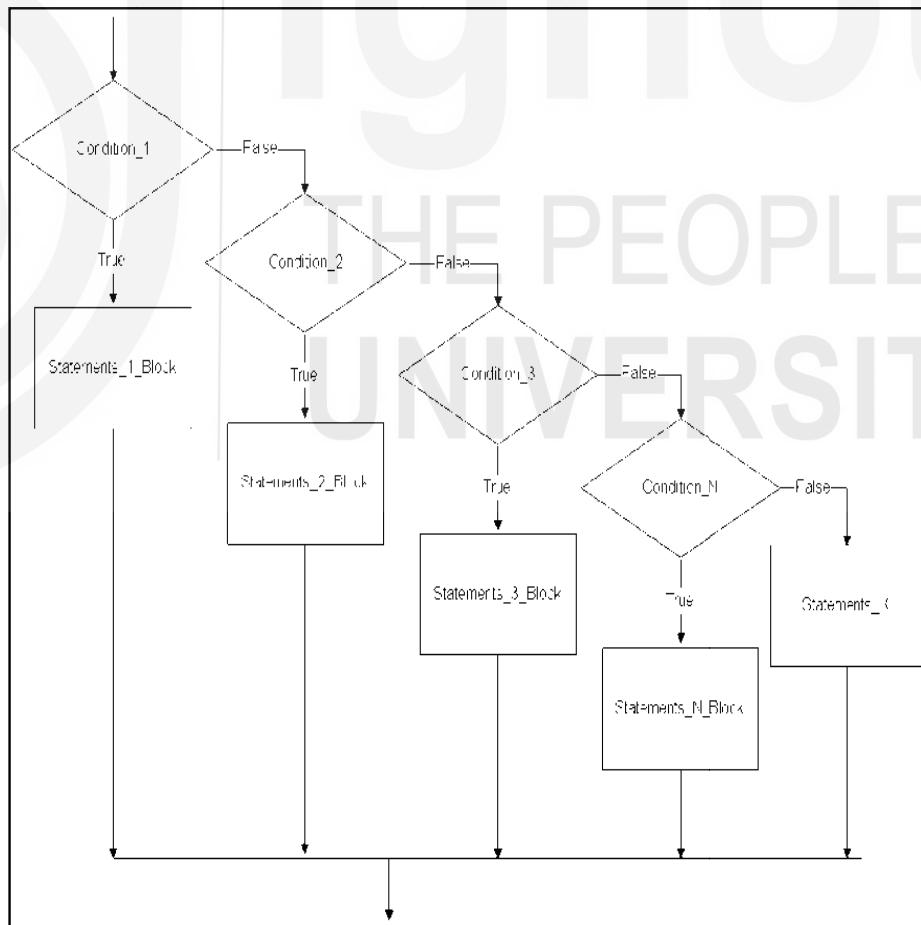


Figure 3.4: Else if statement

Let us consider a program to illustrate *Else if* statement,

Example 3.4

Write a program to award grades to students depending upon the criteria mentioned below:

- Marks less than or equal to 50 are given “D” grade
- Marks above 50 but below 60 are given “C” grade
- Marks between 60 to 75 are given “B” grade
- Marks greater than 75 are given “A” grade.

```
/* Program to award grades */
#include<stdio.h>
main()
{
int result;
printf("Enter the total marks of a student:\n");
scanf("%d",&result);
if(result <= 50)
    printf("Grade D\n");
else if(result <= 60)
    printf("Grade C\n");
else if(result <= 75)
    printf("Grade B\n");
else
    printf("Grade A\n");
}
```

OUTPUT

Enter the total marks of a student:

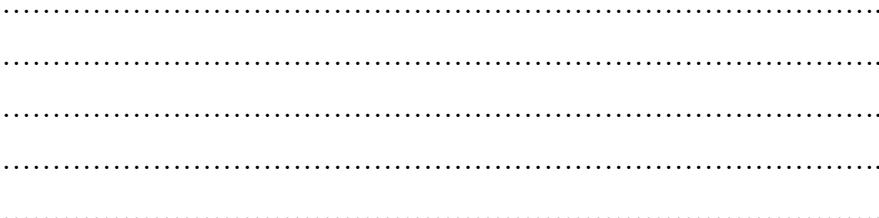
80

Grade A

Check Your Progress 1

1. Find the output for the following program:

```
#include<stdio.h>
main()
{
int a=1, b=1;
if(a==1)
    if(b==0)
        printf("Hi");
    else
        printf("Bye");
}
```



2. Find the output for the following program:

```
#include<stdio.h>
main()
{
    int a,b=0;
    if(a==b)
        printf("hello");
    else
        printf("world");
    return 0;
}
```

.....
.....
.....
.....
.....

3.2.2 The Switch Statement

Its objective is to check several possible constant values for an expression, something similar to what we had studied in the earlier sections, with the linking of several **if** and **else if** statements. When the actions to be taken depending on the value of control variable, are large in number, then the use of control structure **Nested if...else** makes the program complex. There **switch** statement can be used. Its form is the following:

```
switch(expression)
    case expression 1:
        block of instructions 1
        break;
    case expression 2:
        block of instructions 2
        break;
    .
    .
    default:
        default block of instructions
    }
```

It works in the following way: **switch** evaluates expression and checks if it is equivalent to *expression 1*. If it is, it executes *block of instructions 1* until it finds the **break** keyword, moment at finds the control will go to the end of the **switch**. If *expression* was not equal to *expression 1* it will check whether *expression* is equivalent to *expression 2*. If it is, it will execute *block of instructions 2* until it finds the **break** keyword.

Finally, if the value of *expression* has not matched any of the previously specified constants (you may specify as many **case** statements as values you want to check), the program will execute the instructions included in the **default:** section, if it exists, as it is an optional statement.

Let us consider a program to illustrate *Switch* statement,

Example 3.5

Write a program that performs the following, depending upon the choice selected by the user.

- i). calculate the square of number if choice is 1
- ii). calculate the square-root of number if choice is 2 and 4
- iii). calculate the cube of the given number if choice is 3
- iv). otherwise print the number as it is

```
main()
{
int choice,n;
printf("\n Enter any number:\n ");
scanf("%d",&n);
printf("Choice is as follows:\n\n");
printf("1. To find square of the number\n");
printf("2. To find square-root of the number\n");
printf("3. To find cube of a number\n");
printf("4. To find the square-root of the number\n\n");
printf("Enter your choice:\n");
scanf("%d",&choice);
switch(choice)
{
    case 1: printf("The square of the number is %d\n",n*n);
              break;
    case 2:
    case 4: printf("The square-root of the given number is %f",sqrt(n));
              break;
    case 3: printf(" The cube of the given number is %d",n*n*n);
    default: printf("The number you had given is %d",n);
              break;
}
}
```

OUTPUT

Enter any number: 4

Choice is as follows:

1. To find square of the number
2. To find square-root of the number
3. To find cube of a number
4. To find the square-root of the number

Enter your choice: 2

The square-root of the given number is 2

In this section we had discussed and understood various decision control statements. Next section explains you the various loop control statements in C.

3.3 LOOP CONTROL STATEMENTS

Loop control statements are used when a section of code may either be executed a fixed number of times, or while some condition is true. C gives you a choice of three types of loop statements, *while*, *do-while* and *for*.

- The *while* loop keeps repeating an action until an associated *condition* returns *false*. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The *do while* loop is similar, but the *condition* is checked after the loop body is executed. This ensures that the loop body is run at least once.
- The *for* loop is frequently used, usually where the loop will be traversed a fixed number of times.

3.3.1 The *While* Loop

When in a program a single statement or a certain group of statements are to be executed repeatedly depending upon certain test condition, then *while statement* is used. The syntax is as follows:

```
while(test condition)
{
    body_of_the_loop;
}
```

Here, *test condition* is an expression that controls how long the loop keeps running. Body of the loop is a statement or group of statements enclosed in braces and are repeatedly executed till the value of *test condition* evaluates to *true*. As soon as the *condition* evaluates to *false*, the control jumps to the first statement following the *while* statement. If condition initially itself is *false*, the body of the loop will never be executed. *While* loop is sometimes called as *entry-control loop*, as it controls the execution of the body of the loop depending upon the value of the *test condition*. This is shown in the figure 5.5 given below:

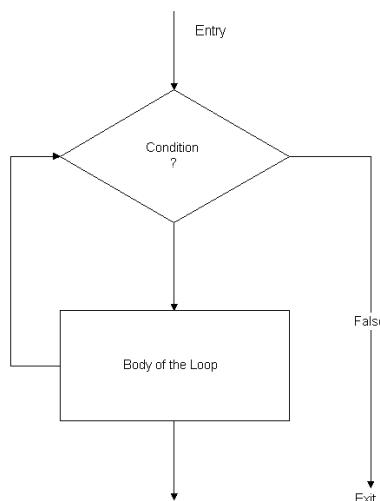


Figure 3.5: The *while* loop statement

Let us consider a program to illustrate *while loop*,

Example 3.6

Write a program to calculate the factorial of a given input natural number.

```
/* Program to calculate factorial of given number */

#include<stdio.h>
#include<math.h>
main()
{
int x;
long int fact = 1;
printf("Enter any number to find factorial:\n"); /*read the number*/
scanf("%d",&x);
while(x > 0)
{
    fact = fact*x; /* factorial calculation*/
    x=x-1;
}
printf("Factorial is %ld",fact);
}
```

OUTPUT

Enter any number to find factorial: 4

Factorial is 24

Here, *condition* in *while* loop is evaluated and body of loop is repeated until *condition* evaluates to *false* i.e., when x becomes zero. Then the control is jumped to first statement following *while* loop and print the value of factorial.

3.3.2 The *do...while* Loop

There is another loop control structure which is very similar to the *while* statement – called as the *do.. while* statement. The only difference is that the expression which determines whether to carry on looping is evaluated at the end of each loop. The syntax is as follows:

```
do
{
    statement(s);
} while(test condition);
```

In *do-while* loop, the body of loop is executed at least once before the *condition* is evaluated. Then the loop repeats body as long as *condition* is *true*. However, in *while* loop, the statement doesn't execute the body of the loop even once, if *condition* is *false*. That is why *do-while* loop is also called *exit-control loop*. This is shown in the figure 3.6 given below.

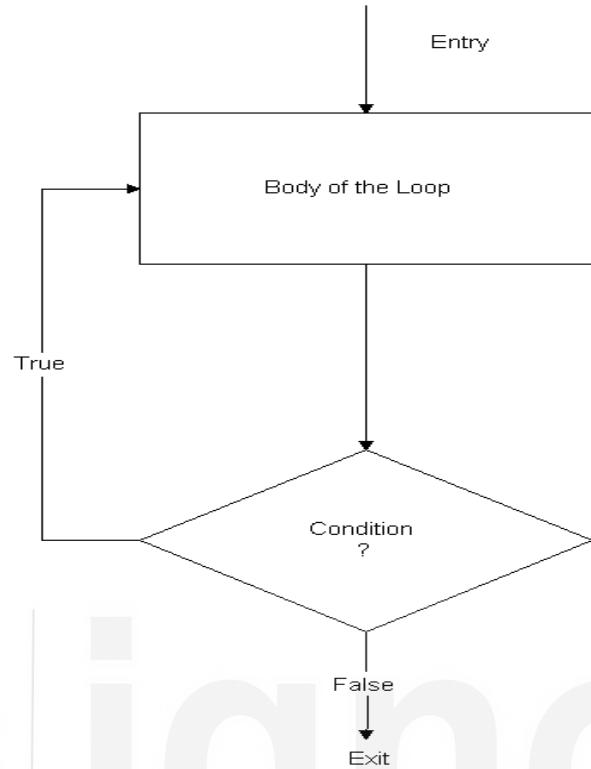


Figure 3.6: The *do...while* statement

Let us consider a program to illustrate *do..while loop*,

Example 3.7

Write a program to print first ten even natural numbers.

```
/* Program to print first ten even natural numbers */
#include<stdio.h>
main()
{
int i=0;
int j=2;
do {
    printf("%d",j);
    j=j+2;
    i=i+1; } while(i<10); }
```

OUTPUT

2 4 6 8 10 12 14 16 18 20

3.3.3 The *for* Loop

for statement makes it more convenient to count iterations of a loop and works well where the number of iterations of the loop is known before the loop is entered. The syntax is as follows:

```
for(initialization; test condition; increment or decrement)
{
    Statement(s);
}
```

The main purpose is to repeat *statement* while *condition* remains true, like the *while* loop. But in addition, *for* provides places to specify an *initialization* instruction and an *increment or decrement of the control variable* instruction. So this loop is specially designed to perform a repetitive action with a counter.

The *for* loop as shown in figure 5.7, works in the following manner:

1. *Initialization* is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. *Condition* is checked, if it is *true* the loop continues, otherwise the loop finishes and *statement* is skipped.
3. *Statement(s)* is/are executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets { }.
4. Finally, whatever is specified in the *increment or decrement of the control variable* field is executed and the loop gets back to step 2.

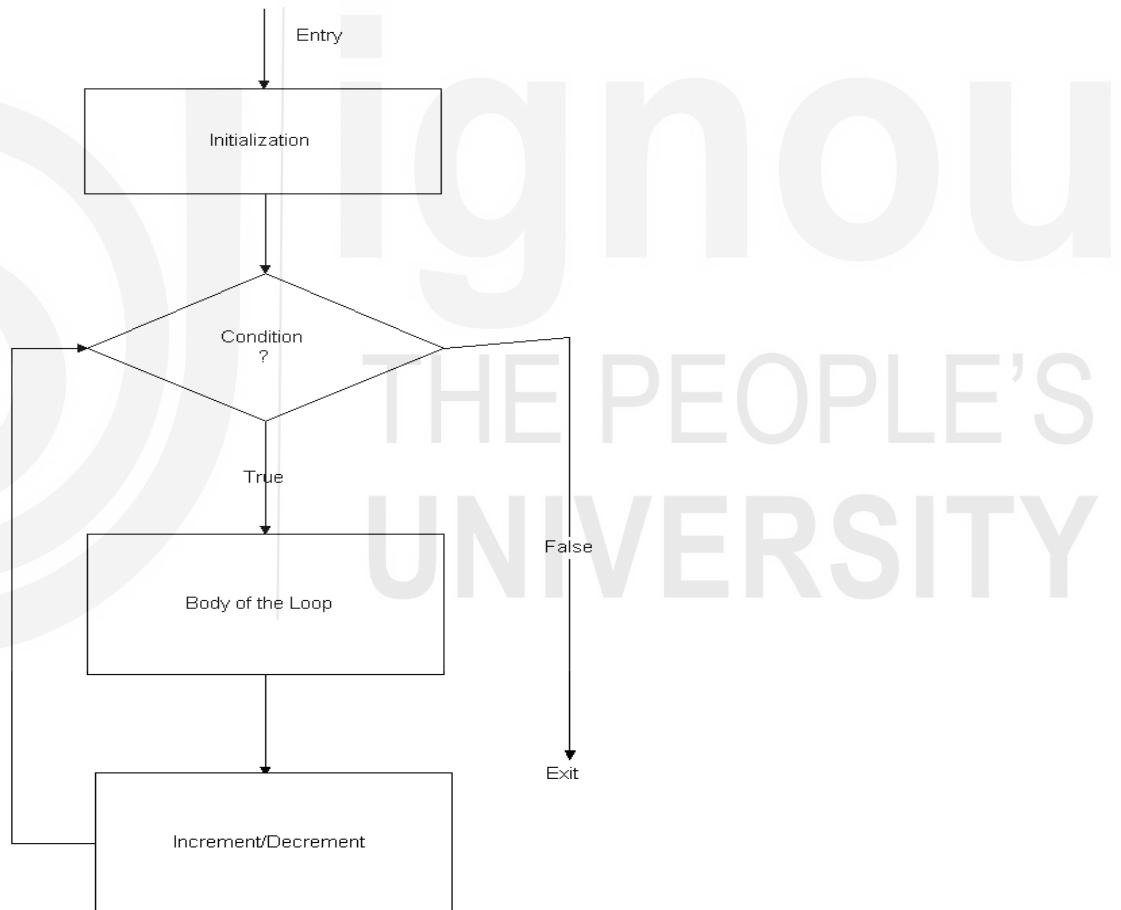


Figure 3.7: The *for* statement

Let us consider a program to illustrate *for loop*,

Example 3.8

Write a program to print first n natural numbers.

```
/* Program to print first n natural numbers */
```

```
#include<stdio.h>
main()
{
int i,n;
printf("Enter value of n \n");
scanf("%d",&n);
printf("\nThe first %d natural numbers are :\n", n);
for(i=1;i<=n;++i)
{
    printf("%d",i);
}
}
```

OUTPUT

```
Enter value of n
6
The first 6 natural numbers are:
1 2 3 4 5 6
```

The three statements inside the braces of a *for* loop usually meant for one activity each, however any of them can be left blank also. More than one control variables can be initialized but should be separated by comma.

Various forms of loop statements can be:

a) *for(;condition;increment/decrement)*

body;

A blank first statement will mean no initialization.

b) *for(initialization;condition;)*

body;

A blank last statement will mean no running increment/decrement.

c) *for(initialization;;increment/decrement)*

body;

A blank second conditional statement means no test condition to control the exit from the loop. So, in the absence of second statement, it is required to test the condition inside the loop otherwise it results in an infinite loop where the control never exits from the loop.

d) *for(;;increment/decrement)*

body;

Initialization is required to be done before the loop and test condition is checked inside the loop.

e) *for(initialization;;)*

body;

Test condition and control variable increment/decrement is to be done inside the body of the loop.

f) `for(;condition;)`

body;

Initialization is required to be done before the loop and control variable increment/decrement is to be done inside the body of the loop.

g) `for(;;)`

body;

Initialization is required to be done before the loop, *test condition* and *control variable* increment/decrement is to be done inside the body of the loop.

3.3.4 The Nested Loops

C allows loops to be *nested*, that is, one loop may be inside another. The program given below illustrates the *nesting* of loops.

Let us consider a program to illustrate *nested loops*,

Example 3.9

Write a program to generate the following pattern given below:

```

1
1   2
1   2   3
1   2   3   4

```

/* Program to print the pattern */

```
#include<stdio.h>
main()
{
int i,j;
for(i=1;i<=4;++i)
{
    printf("%d\n",i);
    for(j=1;j<=i;++j)
        printf("%d\t",j);
}
}
```

Here, an *inner for loop* is written inside the *outer for loop*. For every value of *i*, *j* takes the value from 1 to *i* and then value of *i* is incremented and next iteration of outer loop starts ranging *j* value from 1 to *i*.

Check Your Progress 2

1. Predict the output :

```
#include <stdio.h>
main()
{
    int i;
    for(i=0;i<=10;i++)
        printf("%d",i);
    return 0;
}
```

2. What is the output?

```
#include<stdio.h>
main()
{
    int i;
    for(i=0;i<3;i++)
        printf("%d ",i);
}
```

3. What is the output for the following program?

```
#include<stdio.h>
main()
{
    int i=1;
    do
    {
        printf("%d",i);
    }while(i==1);
}
```

4. Give the output of the following:

```
#include<stdio.h>
main()
{
    int i=3;
    while(i)
    {
        int x=100;
        printf("\n%d..%d",i,x);
        x=x+1;
        i=i+1;
    }
}
```

3.4 THE *goto* STATEMENT

The ***goto*** statement is used to alter the normal sequence of program instructions by transferring the control to some other portion of the program. The syntax is as follows:

goto label;

Here, ***label*** is an identifier that is used to label the statement to which control will be transferred. The targeted statement must be preceded by the unique label followed by colon.

label: statement;

Although ***goto*** statement is used to alter the normal sequence of program execution but its usage in the program should be avoided. The most common applications are:

- i). To branch around statements under certain conditions in place of use of *if- else* statement,
- ii). To jump to the end of the loop under certain conditions bypassing the rest of statements inside the loop in place of *continue* statement,
- iii). To jump out of the loop avoiding the use of *break* statement.

goto can never be used to jump into the loop from outside and it should be preferably used for forward jump.

Situations may arise, however, in which the ***goto*** statement can be useful. To the possible extent, the use of the ***goto*** statement should generally be avoided.

Let us consider a program to illustrate ***goto*** and ***label*** statements.

Example 3.10

Write a program to print first 10 even numbers

```
/* Program to print 10 even numbers */
```

```
#include<stdio.h>
main()
{
    int i=2;
    while(1)
    {
        printf("%d ",i);
        i=i+2;
        if(i>=20)
            goto outside;
    }
    outside : printf("over");
}
```

OUTPUT

```
2 4 6 8 10 12 14 16 18 20 over
```

3.5 THE *break* STATEMENT

Sometimes, it is required to jump out of a loop irrespective of the *conditional test value*. **Break** statement is used inside any loop to allow the control jump to the immediate statement following the loop. The syntax is as follows:

```
break;
```

When nested loops are used, then **break** jumps the control from the loop where it has been used. **Break** statement can be used inside any loop i.e., *while*, *do-while*, *for* and also in *switch* statement.

Let us consider a program to illustrate *break* statement.

Example 3.11

Write a program to calculate the first smallest divisor of a number.

```
/*Program to calculate smallest divisor of a number */
```

```
#include<stdio.h>
main()
{
int div,num,i;
printf("Enter any number:\n");
scanf("%d",&num);
for(i=2;i<=num;++i)
{
if((num % i) == 0)
{
printf("Smallest divisor for number %d is %d",num,i);
break;
}
}
}
```

OUTPUT
Enter any number:
9
Smallest divisor for number 9 is 3

In the above program, we divide the input number with the integer starting from 2 onwards, and print the smallest divisor as soon as remainder comes out to be zero. Since we are only interested in first smallest divisor and not all divisors of a given number, so jump out of the *for* loop using *break* statement without further going for the next iteration of *for* loop.

Break is different from *exit*. Former jumps the control out of the loop while *exit* stops the execution of the entire program.

3.6 THE *continue* STATEMENT

Unlike *break* statement, which is used to jump the control out of the loop, it is sometimes required to skip some part of the loop and to continue the execution with next loop iteration. **Continue** statement used inside the loop helps to bypass the section of a loop and passes the control to the beginning

of the loop to continue the execution with the next loop iteration. The syntax is as follows:

continue;

Let us see the program given below to know the working of the ***continue*** statement.

Example 3.12

Write a program to print first 20 natural numbers skipping the numbers divisible by 5.

```
/* Program to print first 20 natural numbers skipping the numbers divisible by 5 */
```

```
#include<stdio.h>
main()
{
    int i;
    for(i=1;i<=20;++i)
    {
        if((i % 5) == 0)
            continue;
        printf("%d ",i);
    }
}
```

OUTPUT

```
1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19
```

Here, the `printf` statement is bypassed each time when value stored in *i* is divisible by 5.

Check Your Progress 3

1. How many times will hello be printed by the following program?

```
#include<stdio.h>
main()
{
    int i = 5;
    while(i)
    {
        i=i-1;
        if(i==3)
            continue;
        printf("\nhello");
    }
}
```

.....
.....
.....

2. Give the output of the following program segment:

```
#include<stdio.h>
main()
```

```
{
int num,sum;
for(num=2,sum=0;;)
{
    sum = sum + num;
    if(num > 10)
        break;
    num=num+1;
}
printf("%d",sum);
}
```

.....
.....
.....

3. What is the output for the following program?

```
#include<stdio.h>
main()
{
    int i, n = 3;
    for(i=3;n<=20;++n)
    {
        if(n%i == 0)
            break;
        if(i == n)
            printf("%d\n",i);
    }
}
```

.....
.....
.....

3.7 SUMMARY

A *program* is usually not limited to a linear sequence of instructions. During its process it may require to repeat execution of a part of code more than once depending upon the requirements or take decisions. For that purpose, C provides *control* and looping statements. In this unit, we had seen the different looping statements provided by C language namely ***while***, ***do...while and for***.

Using ***break*** statement, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. The ***continue*** statement causes the program to skip the rest of the loop in the present iteration as if the end of the *statement* block would have reached, causing it to jump to the following iteration.

Using the ***goto*** statement, we can make an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation. The destination point is identified by a label,

which is then used as argument for the *goto* instruction. A *label* is made of a valid identifier followed by a colon (:).

3.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1 Bye

2 hello

Check Your Progress 2

1 0 1 2 3 4 5 6 7 8 9 10

2 0 1 2

3 1

4 3..100
2..100
1..100
.....
.....
.....
till infinity

Check Your Progress 3

1 4 times

2 65

3 3

3.9 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI.
2. Programming with C, Second Edition, *Byron Gottfried*, Tata McGraw Hill, 2003.
3. C, The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill,
4. 2002.
5. Computer Science: A Structured Programming Approach Using C, Second Edition, *Behrouz A. Forouzan, Richard F. Gilberg*, Brooks/Cole Thomas Learning, 2001.
6. The C Primer, *Leslie Hancock, Morris Krieger*, Mc Graw Hill, 1983.

UNIT 4 ARRAYS AND STRINGS

Structure

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Array Declaration
 - 4.2.1 Syntax of Array Declaration
 - 4.2.2 Size Specification
- 4.3 Array Initialization
 - 4.3.1 Initialization of Array Elements in the Declaration
 - 4.3.2 Character Array Initialization
- 4.4 Subscript
- 4.5 Processing the Arrays
- 4.6 Multi-Dimensional Arrays
 - 4.6.1 Multi-Dimensional Array Declaration
 - 4.6.2 Initialization of Two-Dimensional Arrays
- 4.7 Introduction to Strings
- 4.8 Declaration and Initialization of Strings
- 4.9 Display of Strings Using Different Formatting Techniques
- 4.10 Array of Strings
- 4.11 String Functions and Applications
- 4.12 Summary
- 4.13 Solutions / Answers
- 4.14 Further Readings

4.0 INTRODUCTION

C language provides four basic data types - *int, char, float and double*. We have learnt about them in Unit 2. These basic data types are very useful; but they can handle only a limited amount of data. As programs become larger and more complicated, it becomes increasingly difficult to manage the data. Variable names typically become longer to ensure their uniqueness. And, the number of variable names makes it difficult for the programmer to concentrate on the more important task of correct coding. Arrays provide a mechanism for declaring and accessing several data items with only one identifier, thereby simplifying the task of data management.

Many programs require the processing of multiple, related data items that have common characteristics like *list* of numbers, marks in a course, or enrolment numbers. This could be done by creating several individual variables. But this is a hard and tedious process. For example, suppose you want to read in five numbers and print them out in reverse order. You could do it the hard way as:

```

main()
{
    int a1,a2,a3,a4,a5;
    scanf("%d %d %d %d %d",&a1,&a2,&a3,&a4,&a5);
    printf("%d %d %d %d %d",a5,a4,a3,a2,a1);
}

```

Does it look good if the problem is to read in 100 or more related data items and print them in reverse order? Of course, the solution is the use of the regular variable names **a1**, **a2** and so on. But to remember each and every variable and perform the operations on the variables is not only tedious a job and disadvantageous too. One common organizing technique is to use arrays in such situations. An **array** is a collection of similar kind of data elements stored in adjacent memory locations and are referred to by a single array-name. In the case of C, you have to declare and define **array** before it can be used. Declaration and definition tell the compiler the name of the array, the type of each element, and the size or number of elements. To explain it, let us consider to store marks of five students. They can be stored using five variables as follows:

```
int ar1, ar2, ar3, ar4, ar5;
```

Now, if we want to do the same thing for 100 students in a class then one will find it difficult to handle 100 variables. This can be obtained by using an array. An array declaration uses its size in [] brackets. For above example, we can define an array as:

```
int ar[100];
```

where *ar* is defined as an array of size 100 to store marks of integer data-type. Each element of this collection is called an *array-element* and an integer value called the *subscript* is used to denote individual elements of the array. An *ar* array is the collection of 200 consecutive memory locations referred as below:



Figure 4.1: Representation of an Array

In the above figure, as each integer value occupies 2 bytes, 200 bytes were allocated in the memory.

This unit explains the use of arrays, types of arrays, declaration and initialization with the help of examples in the first few sections and later on focuses on string handling in C programming language.

4.1 OBJECTIVES

After going through this unit you will be able to:

- declare and use arrays of one dimension;
- initialize arrays;

- use subscripts to access individual array elements;
 - write programs involving arrays;
 - do searching and sorting;
 - handle multi-dimensional arrays;
 - define, declare and initialize a string;
 - discuss various formatting techniques to display the strings; and
 - discuss various built-in string functions and their use in manipulation of strings.
-

4.2 ARRAY DECLARATION

Before discussing how to declare an array, first of all let us look at the characteristic features of an array.

- Array is a data structure storing a group of elements, all of which are of the same data type.
- All the elements of an array share the same name, and they are distinguished from one another with the help of an index.
- Random access to every element using a numeric index(subscript).
- A simple data structure, used for decades, which is extremely useful.
- Abstract Data type(ADT) *list* is frequently associated with the array data structure.

The declaration of an array is just like any variable declaration with additional *size* part, indicating the number of elements of the array. Like other variables, arrays must be declared at the beginning of a function.

The declaration specifies the base type of the array, its name, and its size or dimension. In the following section we will see how an array is declared:

4.2.1 Syntax of Array Declaration

Syntax of array declaration is as follows:

data-type array_name[constant-size];

Data-type refers to the type of elements you want to store
Constant-size is the number of elements

The following are some of declarations for arrays:

```
int char[80];
float farr[500];
static int iarr[80];
char charray[40];
```

There are two restrictions for using arrays in C:

- The amount of storage for a declared array has to be specified at **compile time** before execution. This means that an array has a fixed size.

- The data type of an array applies uniformly to all the elements; for this reason, an array is called a **homogeneous** data structure.

4.2.2 Size Specification

The size of an array should be declared using symbolic constant rather a fixed integer quantity(The subscript used for the individual element is of are integer quantity). The use of a symbolic constant makes it easier to modify a program that uses an array. All reference to maximize the array size can be altered simply by changing the value of the symbolic constant.(Please refer to Unit – 2 for details regarding symbolic constants).

To declare size as 50 use the following symbolic constant, SIZE, defined:

```
#define SIZE 50
```

The following example shows how to declare and read values in an array to store marks of the students of a class.

Example 4.1

Write a program to declare and read values in an array and display them.

```
/* Program to read values in an array*/
#include<stdio.h>
#define SIZE 5 /* SIZE is a symbolic constant */

main()
{
    int i=0; /* Loop variable */
    int stud_marks[SIZE]; /* array declaration */

    /* enter the values of the elements */
    for(i=0;i<SIZE;i++)
    {
        printf("Element no. =%d",i+1);
        printf("Enter the value of the element:");
        scanf("%d",&stud_marks[i]);
    }
    printf("\nFollowing are the values stored in the corresponding array elements: \n\n");
    for( i=0; i<SIZE;i++)
    {
        printf("Value stored in a[%d] is %d\n",i,stud_marks[i]);
    }
}
```

OUTPUT:

```
Element no. = 1 Enter the value of the element = 11
Element no. = 2 Enter the value of the element = 12
Element no. = 3 Enter the value of the element = 13
Element no. = 4 Enter the value of the element = 14
Element no. = 5 Enter the value of the element = 15
```

Following are the values stored in the corresponding array elements:

```
Value stored in a[0] is 11
Value stored in a[1] is 12
Value stored in a[2] is 13
Value stored in a[3] is 14
Value stored in a[4] is 15
```

4.3 ARRAY INITIALIZATION

Arrays can be initialized at the time of declaration. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed within the braces and separated by commas. In the following section, we see how this can be done.

4.3.1 Initialization of Array Elements in the Declaration

The values are assigned to individual array elements enclosed within the braces and separated by comma. Syntax of array initialization is as follows:

```
data type array-name [size] = {val 1, val 2,....., val n};
```

val 1 is the value for the first array element, *val 2* is the value for the second element, and *val n* is the value for the *n* array element. Note that when you are initializing the values at the time of declaration, then there is no need to specify the size. Let us see some of the examples given below:

```
int digits [10]={1,2,3,4,5,6,7,8,9,10};

int digits []={1,2,3,4,5,6,7,8,9,10};

int vector[5]={12,-2,33,21,13};

float temperature[10]={31.2,22.3,41.4,33.2,23.3,32.3,41.1,10.8,11.3,42.3};

double width []={17.33333456,-1.212121213,222.191345 };

int height[10]={60,70,68,72,68 };
```

4.3.2 Character Array Initialisation

The array of characters is implemented as strings in C. Strings are handled differently as far as initialization is concerned. A special character called null character ‘\0’, implicitly suffixes every string. When the external or static string character array is assigned a string constant, the size specification is usually omitted and is automatically assigned; it will include the ‘\0’ character, added at end. For example, consider the following two assignment statements:

```
char thing[3]= “TIN”;
char thing []= “TIN”;
```

In the above two statements the assignments are done differently. The first statement is not a string but simply an array storing three characters ‘T’, ‘I’ and ‘N’ and is same as writing:

```
char thing[3]={‘T’,‘I’,‘N’};
```

whereas, the second one is a four character string TIN\0. The change in the first assignment, as given below, can make it a string.

```
char thing [4] = "TIN";
```

Check Your Progress 1

- What happens if I use a subscript on an array that is larger than the number of elements in the array?

.....
.....
.....

- Give sizes of following arrays.

- char caray [] = "HELLO";
- char caray [5] = "HELLO";
- char caray [] = {'H', 'E', 'L', 'L', 'O'};

.....
.....
.....
.....
.....

- What happens if an array is used without initializing it?

.....
.....
.....
.....
.....

- Is there an easy way to initialize an entire array at once?

.....
.....
.....
.....
.....

- Use a *for* loop to total the contents of an integer array called numbers with five elements. Store the result in an integer called TOTAL.

.....
.....
.....
.....

4.4 SUBSCRIPT

To refer to the individual element in an array, a subscript is used. Refer to the statement we used in the Example 4.1,

```
scanf(" %d",&stud_marks[i]);
```

Subscript is an integer type constant or variable name whose value ranges from 0 to SIZE - 1 where SIZE is the total number of elements in the array. Let us now see how we can refer to individual elements of an array of size 5:

Consider the following declarations:

```
char country[] = "India";
int stud[] = {1,2,3,4,5};
```

Here both arrays are of size 5. This is because the country is a char array and initialized by a string constant "India" and every string constant is terminated by a null character '\0'. And stud is an integer array. country array occupies 5 bytes of memory space whereas stud occupies size of 10 bytes of memory space. The following table: 4.1 shows how individual array elements of *country* and *stud* arrays can be referred:

Table 4.1: Reference of individual elements

Element no.	Subscript	country array		stud array	
		Reference	Value	Reference	Value
1	0	country [0]	'I'	stud [0]	1
2	1	country [1]	'n'	stud [1]	2
3	2	country [2]	'd'	stud [2]	3
4	3	country [3]	'i'	stud [3]	4
5	4	country [4]	'a'	stud [4]	5

Example 4.2

Write a program to illustrate how the marks of 10 students are read in an array and then used to find the maximum marks obtained by a student in the class.

```
/* Program to find the maximum marks among the marks of 10 students*/
```

```
#include< stdio.h>
#define SIZE 10
/* SIZE is a symbolic constant */

main()
{
    int i=0;
    int max=0;
    int stud_marks[SIZE]; /* array declaration */

    /* enter the values of the elements */
    for(i=0;i<SIZE;i++)
    {
        printf("Student no. =%d",i+1);
        printf(" Enter the marks out of 50:");
    }
}
```

```

    scanf("%d",&stud_marks[i]);
}

/* find maximum */
for(i=0;i<SIZE;i++)
{
    if(stud_marks[i]>max)
        max= stud_marks[i];
}
printf("\n\nThe maximum of the marks obtained among all the 10 students is: %d
      ",max);
}

```

OUTPUT

Student no. = 1 Enter the marks out of 50: 10
 Student no. = 2 Enter the marks out of 50: 17
 Student no. = 3 Enter the marks out of 50: 23
 Student no. = 4 Enter the marks out of 50: 40
 Student no. = 5 Enter the marks out of 50: 49
 Student no. = 6 Enter the marks out of 50: 34
 Student no. = 7 Enter the marks out of 50: 37
 Student no. = 8 Enter the marks out of 50: 16
 Student no. = 9 Enter the marks out of 50: 08
 Student no. = 10 Enter the marks out of 50: 37

The maximum of the marks obtained among all the 10 students is: 49

4.5 PROCESSING THE ARRAYS

For certain applications the assignment of initial values to elements of an array is required. This means that the array be defined globally(extern) or locally as a static array.

Let us now see in the following example how the marks in two subjects, stored in two different arrays, can be added to give another array and display the average marks in the below example.

Example 4.3

Write a program to display the average marks of each student, given the marks in 2 subjects for 3 students.

```
/* Program to display the average marks of 3 students */
```

```

#include < stdio.h >
#define SIZE 3
main()
{
int i=0;
float stud_marks1[SIZE]; /* subject 1array declaration */
float stud_marks2[SIZE]; /*subject 2 array declaration */
float total_marks[SIZE];
float avg[SIZE];

printf("\n Enter the marks in subject-1 out of 50 marks: \n");

```

An Introduction to C

```
for(i=0;i<SIZE;i++)
{
    printf("Student no. =%d",i+1);
    printf("Enter the marks= ");
    scanf("%f",&stud_marks1[i]);
}
printf("\nEnter the marks in subject-2 out of 50 marks \n");
for(i=0;i<SIZE;i++)
{
    printf("Student no. =%d",i+1);
    printf("Please enter the marks= ");
    scanf("%f",&stud_marks2[i]);
}

for(i=0;i<SIZE;i++)
{
    total_marks[i]=stud_marks1[i]+ stud_marks2[i];
    avg[i]=total_marks[i]/2;
    printf("Student no.=%d,Average= %f\n",i+1,avg[i]);
}
```

OUTPUT

Enter the marks in subject-1 out of 50 marks:

Student no. = 1 Enter the marks= 23

Student no. = 2 Enter the marks= 35

Student no. = 3 Enter the marks= 42

Enter the marks in subject-2 out of 50 marks:

Student no. = 1 Enter the marks= 31

Student no. = 2 Enter the marks= 35

Student no. = 3 Enter the marks= 40

Student no. = 1 Average= 27.000000

Student no. = 2 Average= 35.000000

Student no. = 3 Average= 41.000000

Let us now write another program to search an element using the linear search.

Example 4.4

Write a program to search an element in a given list of elements using Linear Search.

```
/* Linear Search*/
```

```
# include<stdio.h>
# define SIZE  05
main()
{
int i =0;
int j;
int num_list[SIZE]; /* array declaration */

/* enter elements in the following loop */

printf("Enter any 5 numbers: \n");
for(i=0;i<SIZE;i ++)
```

```

{
    printf("Element no.=%d Value of the element=%d",i+1);
    scanf("%d",&num_list[i]);
}
printf("Enter the element to be searched:");
scanf("%d",&j);

/* search using linear search */
for(i=0;i<SIZE;i++)
{
    if(j == num_list[i])
    {
        printf("The number exists in the list at position: %d\n",i+1);
        break;
    }
}

```

OUTPUT

Enter any 5 numbers:
Element no.=1 Value of the element=23
Element no.=2 Value of the element=43
Element no.=3 Value of the element=12
Element no.=4 Value of the element=8
Element no.=5 Value of the element=5
Enter the element to be searched: 8
The number exists in the list at position: 4

Example 4.5

Write a program to sort a list of elements using the selection sort method

```

/* Sorting list of numbers using selection sort method*/

#include <stdio.h>
#define SIZE 5

main()
{
    int j,min_pos,tmp;
    int i; /* Loop variable */
    int a[SIZE]; /* array declaration */

    /* enter the elements */

    for(i=0;i<SIZE;i++)
    {
        printf("Element no.=%d",i+1);
        printf("Value of the element: ");
        scanf("%d",&a[i]);
    }

    /* Sorting by descending order*/
    for(i=0;i<SIZE;i++)
    {

```

```
min_pos=i;
for(j=i+1;j<SIZE;j++)
    if(a[j] < a[min_pos])
        min_pos= j;
tmp=a[i];
a[i]=a[min_pos];
a[min_pos]=tmp;
}

/* print the result */

printf("The array after sorting:\n");
for(i=0;i<SIZE;i++)
    printf("%d\n",a[i]);
}
```

OUTPUT

```
Element no. = 1 Value of the element: 23
Element no. =2 Value of the element: 11
Element no. =3 Value of the element: 100
Element no. =4 Value of the element: 42
Element no. =5 Value of the element: 50
```

The array after sorting:

```
11
23
42
50
100
```

Check Your Progress 2

1. Name the technique used to pass an array to a function.

.....
.....
.....
.....

2. Is it possible to pass the whole array to a function?

.....
.....
.....
.....
.....

3. List any two applications of arrays.

.....
.....
.....
.....

4.6 MULTI-DIMENSIONAL ARRAYS

Suppose that you are writing a chess-playing program. A chessboard is an 8-by-8 grid. What data structure would you use to represent it? You could use an array that has a chessboard-like structure, i.e. a *two-dimensional array*, to store the positions of the chess pieces. Two-dimensional arrays use two indices to pinpoint an individual element of the array. This is very similar to what is called "algebraic notation", commonly used in chess circles to record games and chess problems.

In principle, there is no limit to the number of subscripts(or dimensions) an array can have. Arrays with more than one dimension are called *multi-dimensional arrays*. While humans cannot easily visualize objects with more than three dimensions, representing multi-dimensional arrays presents no problem to computers. In practice, however, the amount of memory in a computer tends to place limits on the size of an array . A simple four-dimensional array of double-precision numbers, merely twenty elements wide in each dimension, takes up $20^4 * 8$, or 1,280,000 bytes of memory - about a megabyte.

For example, you have ten rows and ten columns, for a total of 100 elements. It's really no big deal. The first number in brackets is the number of rows, the second number in brackets is the number of columns. So, the upper left corner of any grid would be element [0][0]. The element to its right would be [0][1], and so on. Here is a little illustration to help.

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

Three-dimensional arrays(and higher) are stored in the same way as the two-dimensional ones. They are kept in computer memory as a linear sequence of variables, and the last index is always the one that varies fastest(then the next-to-last, and so on).

4.6.1 Multi - Dimensional Array Declaration

You can declare an array of two dimensions as follows:

```
datatype array_name[size1][size2];
```

In the above example, *variable_type* is the name of some type of variable, such as int. Also, *size1* and *size2* are the sizes of the array's first and second dimensions, respectively. Here is an example of defining an 8-by-8 array of integers, similar to a chessboard. Remember, because C arrays are zero-based,

the indices on each side of the chessboard array run 0 through 7, rather than 1 through 8. The effect is the same: a two-dimensional array of 64 elements.

```
int chessboard [8][8];
```

To pinpoint an element in this grid, simply supply the indices in both dimensions.

4.6.2 Initialisation of Two - Dimensional Arrays

If you have an $m \times n$ array, it will have $m * n$ elements and will require $m * n * \text{element-size}$ bytes of storage. To allocate storage for an array you must reserve this amount of memory. The elements of a two-dimensional array are stored row wise. If table is declared as:

```
int table [2] [3]={1,2,3,4,5,6 };
```

It means that element

```
table [0][0]=1;
table [0][1]=2;
table [0][2]=3;
table [1][0]=4;
table [1][1]=5;
table [1][2]=6;
```

The neutral order in which the initial values are assigned can be altered by including the groups in {} inside main enclosing brackets, like the following initialization as above:

```
int table [2][3]={{{1,2,3},{4,5,6}}};
```

The value within innermost braces will be assigned to those array elements whose last subscript changes most rapidly. If there are few remaining values in the row, they will be assigned zeros. The number of values cannot exceed the defined row size.

```
int table [2] [3]={{{1, 2, 3},{4}}};
```

It assigns values as:

```
table[0][0]=1;
table[0][1]=2;
table[0][2]=3;
table[1][0]=4;
table[1][1]=0;
table[1][2]=0;
```

Remember that, C language performs no error checking on array bounds. If you define an array with 50 elements and you attempt to access element 50(the 51st element), or any out of bounds index, the compiler issues no warnings. It is the programmer's task to check that all attempts to access or write to arrays are done only at valid array indexes. Writing or reading past the end of arrays is a common programming bug and is hard to isolate.

1. Declare a multi-dimensioned array of floats called balances having three rows and five columns.

.....
.....
.....
.....
.....

2. Write a *for* loop to total the contents of the multi-dimensioned float array balances.

.....
.....
.....
.....
.....

3. Write a for loop which will read five characters(use scanf) and deposit them into the character based array words, beginning at element 0.

.....
.....
.....
.....
.....

4.7 INTRODUCTION TO STRINGS

In the previous unit, we have discussed numeric arrays, a powerful data storage method that lets you group a number of same-type data items under the same group name. Individual items, or elements, in an array are identified using a subscript after the array name. Computer programming tasks that involve repetitive data processing lend themselves to array storage. Like non-array variables, arrays must be declared before they can be used. Optionally, array elements can be initialized when the array is declared. In the earlier unit, we had just known the concept of *character arrays* which are also called *strings*.

String can be represented as a single-dimensional character type array. C language does not provide the intrinsic string types. Some problems require that the characters within a string be processed individually. However, there are many problems which require that strings be processed as complete entities. Such problems can be manipulated considerably through the use of special string oriented library functions. Most of the C compilers include string library functions that allow string comparison, string copy, concatenation of strings etc. The string functions operate on null-terminated arrays of characters

and require the header <string.h>. The use of some of the string library functions are given as examples in this unit.

4.8 DECLARATION AND INITIALIZATION OF STRINGS

Strings in C are group of characters, digits, and symbols enclosed in quotation marks or simply we can say the string is declared as a “character array”. The end of the string is marked with a special character, the ‘\0’ (*Null character*), which has the decimal value 0. There is a difference between a *character* stored in memory and a *single character string* stored in a memory. The character requires only one byte whereas the single character string requires two bytes (one byte for the character and other byte for the delimiter).

Declaration of Strings

A string in C is simply a sequence of characters. To declare a string, specify the data type as char and place the number of characters in the array in square brackets after the string name. The syntax is shown as below:

char string-name[size];

For example,
char name[20];
char address[25];
char city[15];

Initialization of Strings

The string can be initialized as follows:

char name[8]={‘P’,‘R’,‘O’,‘G’,‘R’,‘A’,‘M’,‘\0’};

Each character of string occupies 1 byte of memory (on 16 bit computing). The size of character is machine dependent, and varies from 16 bit computers to 64 bit computers. The characters of strings are stored in the contiguous (adjacent) memory locations.

| 1 byte |
|--------|--------|--------|--------|--------|--------|--------|--------|
| P | R | O | G | R | A | M | \0 |

1001 1002 1003 1004 1005 1006 1007 1008

The C compiler inserts the NULL (\0) character automatically at the end of the string. So initialization of the NULL character is not essential.

You can set the initial value of a character array when you declare it by specifying a string literal. If the array is too small for the literal, the literal will be truncated. If the literal (including its null terminator) is smaller than the array, then the final characters in the array will be undefined. If you don’t specify the size of the array, but do specify a literal, then C will set the array to the size of the literal, including the null terminator.

```

char str[4]={'u', 'n', 'i', 'x'};
char str[5]={'u', 'n', 'i', 'x', '\0'};
char str[3];
char str[]="UNIX";
char str[4]="unix";
char str[9]="unix";

```

All of the above declarations are legal. But which ones don't work? The first one is a valid declaration, but will cause major problems because it is not *null-terminated*. The second example shows a correct null-terminated string. The special escape character \0 denotes string termination. The fifth example suffers the size problem, the character array 'str' is of size 4 bytes, but it requires an additional space to store '\0'. The fourth example however does not. This is because the compiler will determine the length of the string and automatically initialize the last character to a null-terminator. The strings not terminated by a '\0' are merely a collection of characters and are called as *character arrays*.

String Constants

String constants have double quote marks around them, and can be assigned to char pointers. Alternatively, you can assign a string constant to a char array - either with no size specified, or you can specify a size, but don't forget to leave a space for the null character! Suppose you create the following two code fragments and run them:

```

/* Fragment 1 */
{
    char *s;
    s="hello";
    printf("%s\n",s);
}

/* Fragment 2 */
{
    char s[100];
    strcpy(s," hello");
    printf("%s\n",s);
}

```

These two fragments produce the same output, but their internal behaviour is quite different. In fragment 2, you cannot say **s="hello";**. To understand the differences, you have to understand how the *string constant table* works in C. When your program is compiled, the compiler forms the object code file, which contains your machine code and a table of all the string constants declared in the program. In fragment 1, the statement **s="hello";** causes **s** to point to the address of the string **hello** in the string constant table. Since this string is in the string constant table, and therefore technically a part of the executable code, you cannot modify it. You can only point to it and use it in a read-only manner. In fragment 2, the string **hello** also exists in the constant table, so you can copy it into the array of characters named **s**. Since **s** is not an address, the statement **s="hello";** will not work in fragment 2. It will not even compile.

Example 4.6

Write a program to read a name from the keyboard and display message **Hello** onto the monitor”.

```
/*Program that reads the name and display the hello along with your name*/
#include <stdio.h>
main()
{
char name[10];
printf("\nEnter Your Name :");
scanf("%s", name);
printf("Hello %s\n",name);
}
```

OUTPUT

Enter Your Name: Alex
Hello Alex

In the above example declaration char name [10] allocates 10 bytes of memory space(on 16 bit computing) to array name []. We are passing the base address to scanf function and scanf() function fills the characters typed at the keyboard into array until enter is pressed. The scanf() places ‘\0’ into array at the end of the input. The printf() function prints the characters from the array on to monitor, leaving the end of the string ‘\0’. The %s used in the scanf() and printf() functions is a format specification for strings.

4.9 DISPLAY OF STRINGS USING DIFFERENT FORMATTING TECHNIQUES

The **printf** function with **%s** format is used to display the strings on the screen. For example, the below statement displays entire string:

```
printf("%s", name);
```

We can also specify the accuracy with which character array (string) is displayed. For example, if you want to display first 5 characters from a field width of 15 characters, you have to write as:

```
printf("%15.5s", name);
```

If you include minus sign in the format (e.g. % -10.5s), the string will be printed left justified.

```
printf("% -10.5s", name);
```

Example 4.7

Write a program to display the string “UNIX” in the following format.

```
U
UN
UNI
UNIX
UNIX
UNI
```

```

UN
U
/* Program to display the string in the above shown format*/

#include<stdio.h>
main()
{
int x, y;
static char string[]="UNIX";
printf("\n");
for( x=0; x<4; x++)
{
    y=x + 1;
    /* reserves 4 character of space on to the monitor and minus sign is for left
justified*/
    printf("%-4.*s \n", y, string);

    /* and for every loop the * is replaced by value of y */
    /* y value starts with 1 and for every time it is incremented by 1 until it reaches to 4*/
}

for( x=3; x>=0; x- -)
{
    y=x + 1;
    printf("%-4.*s \n", y, string);
    /* y value starts with 4 and for every time it is decrements by 1 until it reaches to 1*/
}
}
OUTPUT

U
UN
UNI
UNIX
UNIX
UNI
UN
U

```

4.10 ARRAY OF STRINGS

Array of strings are multiple strings, stored in the form of table. Declaring array of strings is same as strings, except it will have additional dimension to store the number of strings. Syntax is as follows:

char array-name[size][size];

For example,

char names[5][10];

where names is the name of the character array and the constant in first square brackets will gives number of string we are going to store, and the value in second square bracket will gives the maximum length of the string.

Example 4.8

char names [3][10]={“martin”,“phil”,“collins”};

It can be represented by a two-dimensional array of size[3][10] as shown below:

0	1	2	3	4	5	6	7	8	9
m	a	r	t	i	n	\0			
p	h	i	l	\0					
c	o	l	l	i	n	s	\0		

Example 4.9

Write a program to initializes 3 names in an array of strings and display them on to monitor

```
/* Program that initializes 3 names in an array of strings and display them on to monitor.*
```

```
#include <stdio.h>
main()
{
    int n;
    char names[3][10]={“Alex”, “Phillip”, “Collins” };
    for(n=0; n<3; n++)
        printf(“%s \n”,names[n]); }
```

OUTPUT

```
Alex
Phillip
Collins
```

Check Your Progress 4

1. Which of the following is a static string?
 - A. Static String;
 - B. “Static String”;
 - C. ‘Static String’;
 - D. char string[100];
-
.....
.....

2. Which character ends all strings?

- A. ‘.’
- B. ‘ ‘
- C. ‘0’
- D. ‘n’

.....
.....
.....

3. What is the Output of the following programs?

```
(a) main()
{
    char name[10] = "IGNOU";
    printf("\n %c", name[0]);
    printf("\n %s", name);
}

(b) main()
{
    char s[] = "hello";
    int j=0;
    while( s[j] != '\0' )
        printf(" %c", s[j++]);
}

(c) main()
{
    char str[] = "hello";
    printf("%10.2s", str);
    printf("%-10.2s", str);
}
```

4 Write a program to read 'n' number of lines from the keyboard using a two-dimensional character array (ie., strings).

```
.....  
.....  
.....
```

4.11 BUILT IN STRING FUNCTIONS AND APPLICATIONS

The header file <string.h> contains some string manipulation functions. The following is a list of the common string managing functions in C.

4.11.1 Strlen Function

The **strlen** function returns the length of a string. It takes the string name as argument. The syntax is as follows:

n=strlen(str);

where **str** is name of the string and **n** is the length of the string, returned by **strlen** function.

Example 4.10

Write a program to read a string from the keyboard and to display the length of the string on to the monitor by using **strlen()** function.

```
/* Program to illustrate the strlen function to determine the length of a string */
```

```
#include <stdio.h>
#include <string.h>
main()
{
char name[80];
int length;
printf("Enter your name: ");
gets(name);
length=strlen(name);
printf("Your name has %d characters\n", length);
}
```

OUTPUT

Enter your name: TYRAN
 Your name has 5 characters

4.11.2 Strcpy Function

In C, you cannot simply assign one character array to another. You have to copy element by element. The string library <string.h> contains a function called **strcpy** for this purpose. The **strcpy** function is used to copy one string to another. The syntax is as follows:

strcpy(str1, str2);

where str1, str2 are two strings. The content of string str2 is copied on to string str1.

Example 4.11

Write a program to read a string from the keyboard and copy the string onto the second string and display the strings on to the monitor by using strcpy() function.

```
/* Program to illustrate strcpy function*/
```

```
#include <stdio.h>
#include <string.h>
main()
{
char first[80], second[80];
printf("Enter a string: ");
gets(first);
strcpy(second, first);
printf("\n First string is : %s, and second string is: %s\n", first, second);
}
```

OUTPUT

Enter a string: ADAMS
 First string is: ADAMS, and second string is: ADAMS

4.11.3 Strcmp Function

The **strcmp** function in the string library function which compares two strings, character by character and stops comparison when there is a difference in the ASCII value or the end of any one string and returns ASCII difference of the characters that is integer. If the return value **zero** means the two strings are equal, a negative value means that first is less than second, and a positive value means first is greater than second. The syntax is as follows:

```
n=strcmp(str1, str2);
```

where **str1** and **str2** are two strings to be compared and **n** is returned value of differed characters.

Example 4.12

Write a program to compare two strings using string compare function.

```
/* The following program uses the strcmp function to compare two strings. */
```

```
#include <stdio.h>
#include <string.h>
main()
{
    char first[80], second[80];
    int value;
    printf("Enter a string: ");
    gets(first);
    printf("Enter another string: ");
    gets(second);
    value=strcmp(first,second);
    if(value ==0)
        puts("The two strings are equal");
    else if(value < 0)
        puts("The first string is smaller ");
    else if(value > 0)
        puts("the first string is bigger");
}
```

OUTPUT

```
Enter a string: MOND
Enter another string: MOHANT
The first string is smaller
```

4.11.4 Strcat Function

The **strcat** function is used to join one string to another. It takes two strings as arguments; the characters of the second string will be appended to the first string. The syntax is as follows:

```
strcat(str1, str2);
```

where **str1** and **str2** are two string arguments, string **str2** is appended to string **str1**.

Example 14.13

Write a program to read two strings and append the second string to the first string.

```
/* Program for string concatenation*/

#include <stdio.h>
#include <string.h>
main()
{
char first[80], second[80];
printf("Enter a string:");
gets(first);
printf("Enter another string: ");
gets(second);
strcat(first, second);
printf("\nThe two strings joined together: %s\n", first);
}
```

OUTPUT

```
Enter a string: BOREX
Enter another string: BANKS
The two strings joined together: BOREX BANKS
```

4.11.5 Strlwr Function

The **strlwr** function converts upper case characters of string to lower case characters. The syntax is as follows:

strlwr(str1);
where str1 is string to be converted into lower case characters.

Example 4.14

Write a program to convert the string into lower case characters using in-built function.

```
/* Program that converts input string to lower case characters */
```

```
#include <stdio.h>
#include <string.h>
main()
{
char first[80];
printf("Enter a string: ");
gets(first);
printf("Lower case of the string is %s", strlwr(first));
}
```

OUTPUT

```
Enter a string: BROOKES
Lower case of the string is brookes
```

4.11.6 Strrev Function

The **strrev** function reverses the given string. The syntax is as follows:

`strrev(str);`
where string **str** will be reversed.

Example 4.15

Write a program to reverse a given string.

```
/* Program to reverse a given string */
```

```
#include <stdio.h>
#include <string.h>
main()
{
char first[80];
printf("Enter a string.");
gets(first);
printf("\n Reverse of the given string is : %s ", strrev(first));
}
```

OUTPUT

Enter a string: ADANY
Reverse of the given string is: YNADA

4.11.7 Strspn Function

The **strspn** function returns the position of the string, where first string mismatches with second string. The syntax is as follows:

```
n=strspn(first, second);
```

where **first** and **second** are two strings to be compared, **n** is the number of character from which first string does not match with second string.

Example 4.16

Write a program, which returns the position of the string from where first string does not match with second string.

```
/*Program which returns the position of the string from where first string does
not match with second string*/
```

```
#include <stdio.h>
#include <string.h>
main()
{
char first[80], second[80];
printf("Enter first string: ");
gets(first);
printf("\nEnter second string: ");
gets(second);
printf("\n After %d characters there is no match",strspn(first, second));
}
```

OUTPUT

Enter first string: ALEXANDER
Enter second string: ALEXSMITH
After 4 characters there is no match

4.11.8 Other String Functions

strncpy function

The **strncpy** function same as *strcpy*. It copies characters of one string to another string up to the specified length. The syntax is as follows:

strncpy(str1, str2, 10);

where **str1** and **str2** are two strings. The **10** characters of string **str2** are copied onto string **str1**.

strcmp function

The **strcmp** function is same as *strcmp*, except it compares two strings ignoring the case(lower and upper case). The syntax is as follows:

n=strcmp(str1, str2);

strncmp function

The **strncmp** function is same as *strcmp*, except it compares two strings up to a specified length. The syntax is as follows:

n=strncmp(str1, str2, 10);

where **10** characters of **str1** and **str2** are compared and **n** is returned value of differed characters.

strchr function

The **strchr** funtion takes two arguments(the string and the character whose address is to be specified) and returns the address of first occurrence of the character in the given string. The syntax is as follows:

cp=strchr(str, c);

where **str** is string and **c** is character and **cp** is character pointer.

strset function

The **strset** funtion replaces the string with the given character. It takes two arguments the string and the character. The syntax is as follows:

strset(first, ch);

where string **first** will be replaced by character **ch**.

strchr function

The **strchr** function takes two arguments(the string and the character whose address is to be specified) and returns the address of first occurrence of the character in the given string. The syntax is as follows:

cp=strchr(str, c);

where **str** is string and **c** is character and **cp** is character pointer.

strncat function

The **strncat** function is the same as *strcat*, except that it appends upto specified length. The syntax is as follows:

strncat(str1, str2, 10);

Arrays and Strings

where 10 character of the str2 string is added into str1 string.

strupr function

The **strupr** function converts lower case characters of the string to upper case characters. The syntax is as follows:

strupr(str1);

where str1 is string to be converted into upper case characters.

strstr function

The **strstr** function takes two arguments address of the string and second string as inputs. And returns the address from where the second string starts in the first string. The syntax is as follows:

cp=strstr(first, second);

where **first** and **second** are two strings, **cp** is character pointer.

Check Your Progress 5

1. Which of the following functions compares two strings?

- A. compare();
 - B. stringcompare();
 - C. cmp();
 - D. strcmp();
-
.....
.....

2. Which of the following appends one string to the end of another?

- A. append();
 - B. stringadd();
 - C. strcat();
 - D. stradd();
-
.....
.....
.....

3. Write a program to concatenate two strings without using the *strcat()* function.

.....
.....
.....
.....

4. Write a program to find string length without using the `strlen()` function.

.....
.....
.....

5. Write a program to convert lower case letters to upper case letters in a given string without using `strupr()`.

.....
.....
.....

4.12 SUMMARY

Like other languages, C uses arrays as a way of describing a collection of variables with identical properties. The group has a single name for all its members, with the individual member being selected by an *index*. We have learnt in this unit, the basic purpose of using an array in the program, declaration of array and assigning values to the arrays and also the string handling functions. All elements of the arrays are stored in the consecutive memory locations. Without exception, all arrays in C are indexed from 0 up to one less than the bound given in the declaration. This is very puzzling for a beginner. Watch out for it in the examples provided in this unit. One important point about array declarations is that they don't permit the use of varying subscripts. The numbers given must be constant expressions which can be evaluated at compile time, not run time. As with other variables, global and static array elements are initialized to 0 by default, and automatic array elements are filled with garbage values. In C, an array of type char is used to represent a character string, the end of which is marked by a byte set to 0(also known as a NULL character).

Whenever the arrays are passed to function their starting address is used to access rest of the elements. This is called – Call by reference. Whatever changes are made to the elements of an array in the function, they are also made available in the calling part. The formal argument contains no size specification except for the rightmost dimension. Arrays and pointers are closely linked in C. Multi-dimensional arrays are simply arrays of arrays. To use arrays effectively it is a good idea to know how to use pointers with them. More about the pointers can be learnt from Unit -7 (Block -2).

Strings are sequence of characters. Strings are to be null-terminated if you want to use them properly. Remember to take into account null-terminators when using dynamic memory allocation. The `string.h` library has many useful functions. Losing the '`\0`' character can lead to some very considerable bugs. Make sure you copy `\0` when you copy strings. If you create a new string, make sure you put `\0` in it. And if you copy one string to another, make sure the receiving string is big enough to hold the source string, including `\0`. Finally, if you point a character pointer to some characters, make sure they end with `\0`.

String Functions	Its Use
<i>strlen</i>	Returns number of characters in string.
<i>strlwr</i>	Converts all the characters in the string into lower case characters
<i>strcat</i>	Adds one string at the end of another string
<i>strcpy</i>	Copies a string into another
<i>strcmp</i>	Compares two strings and returns zero if both are equal.
<i>strupr</i>	Duplicates a string
<i>strchr</i>	Finds the first occurrence of given character in a string
<i>strstr</i>	Finds the first occurrence of given string in another string
<i>strset</i>	Sets all the characters of string to given character or symbol
<i>strrev</i>	Reverse a string

4.13 SOLUTIONS / ANSWERS

Check Your Progress 1

1. If you use a subscript that is out of bounds of the array declaration, the program will probably compile and even run. However, the results of such a mistake can be unpredictable. This can be a difficult error to find once it starts causing problems. So, make sure you're careful when initializing and accessing the array elements.
2. a) 6
b) 5
c) 5
3. This mistake doesn't produce a compiler error. If you don't initialize an array, there can be any value in the array elements. You might get unpredictable results. You should always initialize the variables and the arrays so that you know their content.
4. Each element of an array must be initialized. The safest way for a beginner is to initialize an array, either with a declaration, as shown in this chapter, or with a **for** statement. There are other ways to initialize an array, but they are beyond the scope of this Unit.
5. Use a **for** loop to total the contents of an integer array which has five elements. Store the result in an integer called total.

```
for(loop=0,total=0; loop<5; loop++)
    total=total+numbers[loop];
```

Check Your Progress 2

1. Call by reference.
2. It is possible to pass the whole array to a function. In this case, only the address of the array will be passed. When this happens, the function can change the value of the elements in the array.
3. Two common statistical applications that use arrays are:

- **Frequency distributions:** A frequency array show the number of elements with an identical value found in a series of numbers. For example, suppose we have taken a sample of 50 values ranging from 0 to 10. We want to know how many of the values are 0, how many are 1, how many are 2 and so forth up to 10. Using the arrays we can solve the problem easily . Histogram is a pictorial representation of the frequency array. Instead of printing the values of the elements to show the frequency of each number, we print a histogram in the form of a bar chart.
- **Random Number Permutations:** It is a set of random numbers in which no numbers are repeated. For example, given a random number permutation of 5 numbers, the values of 0 to 5 would all be included with no duplicates.

Check Your Progress 3

1. float balances[3][5];
2. for(row=0, total=0; row < 3; row++)

 for(column=0; column < 5; column++)

 total=total + balances[row][column];
3. for(loop=0; loop < 5; loop++)

 scanf("%c",&words[loop]);

Check Your Progress 4

1. B
2. C
3. (a) I
 IGNOU
(b) hello
(c) hehe

Check Your Progress 5

1. D
2. C
3. /*Program to concatenate two strings without using the strcat() function*/

```
#include<stdio.h>
#include<string.h>
main()
{
    char str1[10];
    char str2[10];
    char output_str[20];
    int i,j,k;
    i=0;
    j=0;
    k=0;
    printf("Input the first string: ");
    gets(str1);
```

```

printf("\nInput the second string: ");
gets(str2);
while(str1[i]!='\0'
output_str[k++]=str1[i++];
while(str2[j]!='\0')
output_str[k++]=str2[j++];
output_str[k]='\0';
puts(output_str);
}
4. /* Program to find the string length without using the strlen() function */

```

```

#include<stdio.h>
#include<string.h>
main()
{
char string[60];
int len,i;
len=0;
i=0;
printf("Input the string : ");
gets(string);
while(string[i++]!='\0')
    len++;
printf("Length of Input String=%d",len);
getchar();
}

```

5. /* Program to convert the lower case letters to upper case in a given string without using strupp() function*/

```

#include<stdio.h>
main()
{
int i=0;
char source[10],destination[10];
printf("Input the string in lower-case");
gets(source);
while(source[i]!='\0')
{
    if((source[i]>=97)&&(source[i]<=122))
        destination[i]=source[i]-32;
    else
        destination[i]=source[i];
    i++;
}
destination[i]=' \0 ';
puts(destination);
}

```

4.14 FURTHER READINGS

1. The C Programming Language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI.
2. C, The Complete Reference, Fourth Edition, *Herbert Schildt*, TMGH, 2002.

3. Computer Science – A Structured Programming Approach Using C, *Behrouz A. Forouzan, Richard F. Gilberg*, Thomas Learning, Second edition, 2001.
4. Programming with ANSI and TURBO C, *Ashok N. Kamthane*, Pearson Education, 2002.
5. Computer Programming in C, *Raja Raman. V*, PHI, 2002.



UNIT 5 FUNCTIONS

Structure

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Definition of a Function
- 5.3 Declaration of a Function
- 5.4 Function Prototypes
- 5.5 The Return Statement
- 5.6 Types of Variables and Storage Classes
 - 5.6.1 Automatic Variables
 - 5.6.2 External Variables
 - 5.6.3 Static Variables
 - 5.6.4 Register Variables
- 5.7 Types of Function Invoking
- 5.8 Call by Value
- 5.9 Recursion
- 5.10 Summary
- 5.11 Solutions / Answers
- 5.12 Further Readings

5.0 INTRODUCTION

To make programming simple and easy to debug, we break a larger program into smaller *subprograms* which perform ‘*well defined tasks*’. These subprograms are called *functions*. So far we have defined a single function *main()*.

After reading this unit you will be able to define many other functions and the *main()* function can call up these functions from several different places within the program, to carry out the required processing.

Functions are very important tools for ***Modular Programming***, where we break large programs into small subprograms or modules (functions in case of C). The use of functions reduces complexity and makes programming simple and easy to understand.

In this unit, we will discuss how functions are defined and how are they accessed from the main program? We will also discuss various types of functions and how to invoke them. And finally you will learn an interesting and important programming technique known as *Recursion*, in which a function calls within itself.

5.1 OBJECTIVES

After going through this unit, you will learn:

- the need of functions in the programming;

- how to define and declare functions in ‘C’ Language;
- different types of functions and their purpose;
- how the functions are called from other functions;
- how data is transferred through parameter passing, to functions and the Return statement;
- recursive functions; and
- the concept of ‘Call by Value’ and its drawbacks.

5.2 DEFINTION OF A FUNCTION

A **function** is a self- contained block of executable code that can be called from any other function .In many programs, a set of statements are to be executed repeatedly at various places in the program and may with different sets of data, the idea of functions comes in mind. You keep those repeating statements in a function and call them as and when required. When a function is called, the control transfers to the called function, which will be executed, and then transfers the control back to the calling function (to the statement following the function call). Let us see an example as shown below:

Example 5.1

```
/* Program to illustrate a function*/
```

```
#include <stdio.h>
main ()
{
void sample();
printf("\n You are in main");
}

void sample()
{
printf("\n You are in sample");
}
```

OUTPUT

```
You are in sample
You are in main
```

Here we are calling a function **sample()** through **main()** i.e. control of execution transfers from **main()** to **sample()** , which means **main()** is suspended for some time and **sample()** is executed. After its execution the control returns back to **main()**, at the statement following function call and the execution of **main()** is resumed.

The syntax of a function is:

```
return data type function_name(list of arguments)
{
    datatype declaration of the arguments;
```

```
executable statements;  
return(expression);  
}
```

where,

- return data type is the same as the data type of the variable that is returned by the function using return statement.
- a function_name is formed in the same way as variable names / identifiers are formed.
- the list of arguments or parameters are valid variable names as shown below, separated by commas: (data type1 var1,data type2 var2,..... data type n var n)
- for example (*int x, float y, char z*).
- arguments give the values which are passed from the calling function.
- the body of function contains executable statements.
- the return statement returns a *single* value to the calling function.

Example 5.2

Let us write a simple function that calculates the square of an integer.

```
/*Program to calculate the square of a given integer*/  
  
/* square() function */  
{  
    int square(int no)          /*passing of argument */  
    int result;                 /* local variable to function square */  
    result = no*no;  
    return(result);             /* returns an integer value */  
}  
  
/*It will be called from main()as follows */  
main()  
{  
    int n ,sq;                /* local variable to function main */  
    printf ("Enter a number to calculate square value");  
    scanf("%d",&n);  
    sq=square(n);             /* function call with parameter passing */  
    printf ("\nSquare of the number is : %d", sq);  
} /* program ends */
```

OUTPUT

```
Enter a number to calculate square value : 5  
Square of the number is : 25
```

5.3 DECLARATION OF A FUNCTION

As we have mentioned in the previous section, every function has its declaration and function definition. When we talk of declaration only, it means

only the function name, its argument list and return type are specified and the function body or definition is not attached to it. The *syntax* of a function declaration is:

return data type function_name(list of arguments);

For example,

```
int square(int no);
float temperature(float c, float f);
```

We will discuss the use of function declaration in the next section.

5.4 FUNCTION PROTOTYPES

In Example 5.1 for calculating square of a given number, we have declared function *square()* before *main()* function; this means before coming to *main()*, the compiler knows about *square()*, as the compilation process starts with the first statement of any program. Now suppose, we reverse the sequence of functions in this program i.e., writing the *main()* function and later on writing the *square()* function, *what happens?* The “C” compiler will give an error. Here the introduction of concept of “function prototypes” solves the above problem.

Function Prototypes require that every function which is to be accessed should be declared in the calling function. The function declaration, that will be discussed earlier, will be included for every function in its calling function . Example 5.2 may be modified using the function prototype as follows:

Example 5.3

```
/*Program to calculate the square of a given integer using the function
prototype*/
#include <stdio.h>
main ()
{
    int n , sq ;
    int square(int) ;           /*function prototype */
    printf ("Enter a number to calculate square value");
    scanf ("%d",&n);
    sq = square(n);           /* function call with parameter passing */
    printf ("\nSquare of the number is : %d", sq);
}

/* square function */
int square(int no)           /*passing of argument */
{
    int result ;             /* local variable to function square */
    result = no*no;
    return(result);          /* returns an integer value */
}
```

Enter a number to calculate square value : 5
Square of the number is: 25

Points to remember:

- *Function prototype* requires that the function declaration must include the return type of function as well as the type and number of arguments or parameters passed.
- The variable names of arguments need not be declared in prototype.
- The major reason to use this concept is that they enable the compiler to check if there is any mismatch between function declaration and function call.

Check Your Progress 1

- 1) Write a function to multiply two integers and display the product.

.....
.....
.....
.....
.....
.....

- 2) Modify the above program, by introducing function prototype in the main function.

.....
.....
.....
.....
.....
.....

5.5 THE *return* STATEMENT

If a function has to return a value to the calling function, it is done through the ***return*** statement. It may be possible that a function does not return any value; only the control is transferred to the calling function. The syntax for the *return* statement is:

return (expression);

We have seen in the *square()* function, the *return* statement, which returns an integer value.

Points to remember:

- You can pass any number of arguments to a function but can return only one value at a time.

For example, the following are the valid *return* statements

- a) `return (5);`
- b) `return (x*y);`

For example, the following are the invalid *return* statements

- c) `return (2, 3);`
- d) `return (x, y);`

- If a function does not return anything, **void** specifier is used in the function declaration.

For example:

```
void square (int no)
{
    int sq;
    sq = no*no;
    printf ("square is %d", sq);
}
```

- All the function's return type is by default is "**int**", i.e. a function returns an integer value, if no type specifier is used in the function declaration.

Some examples are:

- (i) `square (int no);` /* will return an integer value */
- (ii) `int square (int no);` /* will return an integer value */
- (iii) `void square (int no);` /* will not return anything */

- What happens if a function has to return some value other than integer? The answer is very simple: use the particular type specifier in the function declaration.

For example consider the code fragments of function definitions below:

1) **Code Fragment - 1**

```
char func_char( ....... )
{
    char c;
    .....
    .....
    .....
}
```

2) **Code Fragment - 2**

```
float func_float (....)
{
    float f;
    .....
    .....
    .....
```

```

        return(f);
    }

```

Thus from the above examples, we see that you can return all the data types from a function, the only condition being that the value returned using return statement and the type specifier used in function declaration should match.

- A function can have many *return* statements. This thing happens when some condition based returns are required.

For example,

```

/*Function to find greater of two numbers*/
int greater (int x, int y)
{
    if (x>y)
        return (x);
    else
        return (y);
}

```

- And finally, with the execution of return statement, the control is transferred to the calling function with the value associated with it.

In the above example if we take $x = 5$ and $y = 3$, then the control will be transferred to the calling function when the first return statement will be encountered, as the condition ($x > y$) will be satisfied. All the remaining executable statements in the function will not be executed after this returning.

Check Your Progress 2

1. Which of the following are valid return statements?
 - a) `return (a);`
 - b) `return (z,13);`
 - c) `return (22.44);`
 - d) `return;`
 - e) `return (x*x, y*y);`

.....

.....

.....

.....

.....

.....

.....

.....

5.6 TYPES OF VARIABLES AND STORAGE CLASSES

In a program consisting of a number of functions a number of different types of variables can be found.

Global vs. Static variables: Global variables are recognized throughout the program whereas local variables are recognized only within the function where they are defined.

Static vs. Dynamic variables: Retention of value by a local variable means, that in static, retention of the variable value is lost once the function is completely executed whereas in certain conditions the value of the variable has to be retained from the earlier execution and the execution retained.

The variables can be characterized by their **data type** and by their **storage class**. One way to classify a variable is according to its data type and the other can be through its storage class. **Data type** refers to the type of value represented by a variable whereas **storage class** refers to the **permanence** of a variable and its scope within the program i.e. portion of the program over which variable is recognized.

Storage Classes

There are four different storage classes specified in C:

- | | |
|-----------------|----------------|
| 1. Auto (matic) | 2. Extern (al) |
| 3. Static | 4. Register |

The storage class associated with a variable can sometimes be established by the location of the variable declaration within the program or by prefixing keywords to variables declarations.

For example:

```
auto    int    a, b;
static  int    a, b;
extern  float   f;
```

5.6.1 Automatic Variables

The variables local to a function are automatic i.e., declared within the function. The scope of lies within the function itself. The automatic defined in different functions, even if they have same name, are treated as different. It is the default storage class for variables declared in a function.

Points to remember:

- The auto is optional therefore there is no need to write it.
- All the formal arguments also have the auto storage class.
- The initialization of the auto-variables can be done:
 - in declarations

- using assignment expression in a function
- If not initialized the unpredictable value is defined.
- The value is not retained after exit from the program.

Let us study these variables by a sample program given below:

Example 5.4

```
/* To print the value of automatic variables */
```

```
# include <stdio.h>
main ( int argc, char * argv[ ] )
{
int a, b;
double d;
printf("%d", argc);
a = 10;
b = 5;
d = (b * b) - (a/2);
printf("%d, %d, %f", a, b, d);
}
```

All the variables a, b, d, argc and argv [] have automatic storage class.

5.6.2 External (Global) Variables

These are not confined to a single function. Their scope ranges from the point of declaration to the entire remaining program. Therefore, their scope may be the entire program or two or more functions depending upon where they are declared.

Points to remember:

- These are global and can be accessed by any function within its scope. Therefore value may be assigned in one and can be written in another.
- There is difference in external variable definition and declaration.
- External Definition is the same as any variable declaration:
- Usually lies outside or before the function accessing it.
- It allocates storage space required.
- Initial values can be assigned.
- The external specifier is not required in external variable definition.
- A declaration is required if the external variable definition comes after the function definition.
- A declaration begins with an external specifier.
- Only when external variable is defined is the storage space allocated.
- External variables can be assigned initial values as a part of variable definitions, but the values must be constants rather than expressions.
- If initial value is not included then it is automatically assigned a value of zero.

Let us study these variables by a sample program given below:

Example 5.5

```
/* Program to illustrate the use of global variables*/  
  
# include <stdio.h>  
int gv; /*global variable*/  
main ( )  
{  
void function1(); /*function declaration*/  
gv = 10;  
printf ("%d is the value of gv before function call\n", gv);  
function1();  
printf ("%d is the value of gv after function call\n", gv);  
}  
  
void function1 ( )  
{  
gv = 15: }
```

OUTPUT

```
10 is the value of gv before function call  
15 is the value of gv after function call
```

5.6.3 Static Variables

In case of single file programs static variables are defined within functions and individually have the same scope as automatic variables. But static variables retain their values throughout the execution of program within their previous values.

Points to remember:

- The specifier precedes the declaration. Static and the value cannot be accessed outside of their defining function.
- The static variables may have same name as that of external variables but the local variables take precedence in the function. Therefore external variables maintain their independence with locally defined auto and static variables.
- Initial value is expressed as the constant and not expression.
- Zeros are assigned to all variables whose declarations do not include explicit initial values. Hence they always have assigned values.
- Initialization is done only in the first execution.

Let us study this sample program to print value of a static variable:

Example 5.6

```
/* Program to illustrate the use of static variable*/
```

```
#include <stdio.h>
```

```
main()  
{  
int call_static();
```

```
int i,j;  
i=j=0;  
j = call_static();  
printf("%d\n",j);  
j = call_static();  
printf("%d\n",j);  
j = call_static();  
printf("%d\n",j);  
}
```

```
int call_static()  
{  
static int i=1;  
int j;  
j = i;  
i++;  
return(j);  
}
```

OUTPUT

```
1  
2  
3
```

This is because *i* is a static variable and retains its previous value in next execution of function `call_static()`. To remind you *j* is having auto storage class. Both functions main and `call_static` have the same local variable *i* and *j* but their values never get mixed.

5.6.4 Register Variables

Besides three storage class specifications namely, Automatic, External and Static, there is a *register* storage class. *Registers* are special storage areas within a computer's CPU. All the arithmetic and logical operations are carried out with these registers.

For the same program, the execution time can be reduced if certain values can be stored in registers rather than memory. These programs are smaller in size (as few instructions are required) and few data transfers are required. The reduction is there in machine code and not in source code. They are declared by the proceeding declaration by register reserved word as follows:

```
register int m;
```

Points to remember:

- These variables are stored in registers of computers. If the registers are not available they are put in memory.
- Usually 2 or 3 register variables are there in the program.
- Scope is same as automatic variable, local to a function in which they are declared.
- Address operator ‘&’ cannot be applied to a register variable.

- If the register is not available the variable is thought to be like the automatic variable.
- Usually associated integer variable but with other types it is allowed having same size (short or unsigned).
- Can be formal arguments in functions.
- Pointers to register variables are not allowed.
- These variables can be used for loop indices also to increase efficiency.

5.7 TYPES OF FUNCTION INVOKING

We categorize a function's invoking (calling) depending on arguments or parameters and their returning a value. In simple words we can divide a function's invoking into four types depending on whether parameters are passed to a function or not and whether a function returns some value or not.

The various types of invoking functions are:

- With no arguments and with no return value.
- With no arguments and with return value
- With arguments and with no return value
- With arguments and with return value.

Let us discuss each category with some examples:

TYPE 1: With no arguments and have no return value

As the name suggests, any function which **has no arguments and does not return any values to the calling function**, falls in this category. These type of functions are confined to themselves i.e. neither do they receive any data from the calling function nor do they transfer any data to the calling function. So there is no data communication between the calling and the called function are only program control will be transferred.

Example 5.7

```
/* Program for illustration of the function with no arguments and no return value*/  
  
/* Function with no arguments and no return value*/  
  
#include <stdio.h>  
main()  
{  
void message();  
printf("Control is in main\n");  
message(); /* Type 1 Function */  
printf("Control is again in main\n");  
}  
  
void message()  
{  
printf("Control is in message function\n");  
/* does not return anything */
```

Control is in main
 Control is in message function
 Control is again in main

TYPE 2: With no arguments and with return value

Suppose if a function does not receive any data from calling function but does send some value to the calling function, then it falls in this category.

Example 5.8

Write a program to find the sum of the first ten natural numbers.

```
/* Program to find sum of first ten natural numbers */

#include <stdio.h>

int cal_sum()
{
int i, s=0;
for (i=0; i<=10; i++)
s=s + i;
return(s);           /* function returning sum of first ten natural numbers */
}

main()
{
int sum;
sum = cal_sum();
printf("Sum of first ten natural numbers is % d\n", sum);
}
```

OUTPUT

Sum of first ten natural numbers is 55

TYPE 3: With Arguments and have no return value

If a function **includes arguments but does not return anything**, it falls in this category. One way communication takes place between the calling and the called function.

Before proceeding further, first we discuss the *type of arguments or parameters* here. There are two types of arguments:

- Actual arguments
- Formal arguments

Let us take an example to make this concept clear:

Example 5.9

Write a program to calculate sum of any three given numbers.

```
#include <stdio.h>

main()
```

```
{  
int a1, a2, a3;  
void sum(int, int, int);  
printf("Enter three numbers: ");  
scanf ("%d%d%d",&a1,&a2,&a3);  
sum (a1,a2,a3); /* Type 3 function */  
  
/* function to calculate sum of three numbers */  
void sum (int f1, int f2, int f3)  
{  
int s;  
s = f1+ f2+ f3;  
printf ("\nThe sum of the three numbers is %d\n",s);  
}
```

OUTPUT

Enter three numbers: 23 34 45
The sum of the three numbers is 102

Here f1, f2, f3 are **formal arguments** and a1, a2, a3 are **actual arguments**. Thus we see in the function declaration, the arguments are formal arguments, but when values are passed to the function during function call, they are actual arguments.

Note: The actual and formal arguments should match in type, order and number

TYPE 4: With arguments function and with return value

In this category two-way communication takes place between the calling and called function i.e. a function returns a value and also arguments are passed to it. We modify above Example according to this category.

Example 5.10

Write a program to calculate sum of three numbers.

```
/*Program to calculate the sum of three numbers*/  
  
#include <stdio.h>  
main()  
{  
int a1, a2, a3, result;  
int sum(int, int, int);  
printf("Please enter any 3 numbers:\n");  
scanf ("%d %d %d", &a1, &a2, &a3);  
result = sum (a1,a2,a3); /* function call */  
printf ("Sum of the given numbers is : %d\n", result);  
  
/* Function to calculate the sum of three numbers */  
int sum (int f1, int f2, int f3)  
{  
    return(f1+ f2 + f3); /* function returns a value */  
}
```

OUTPUT

Please enter any 3 numbers:

3 4 5

Sum of the given numbers is: 12

Functions

5.8 CALL BY VALUE

So far we have seen many functions and also passed arguments to them, but if we observe carefully, we will see that we have always created new variables for arguments in the function and then passed the values of actual arguments to them. Such function calls are called “*call by value*”.

Let us illustrate the above concept in more detail by taking a simple function of multiplying two numbers:

Example 5.11

Write a program to multiply the two given numbers

```
#include <stdio.h>
main()
{
int x, y, z;
int mul(int, int);
printf("Enter two numbers: \n");
scanf ("%d %d",&x,&y);
z= mul(x, y);           /* function call by value */
printf ("\n The product of the two numbers is : %d", z);
}

/* Function to multiply two numbers */
int mul(int a, int b)
{
int c;
c =a*b;
return(c); }
```

OUTPUT

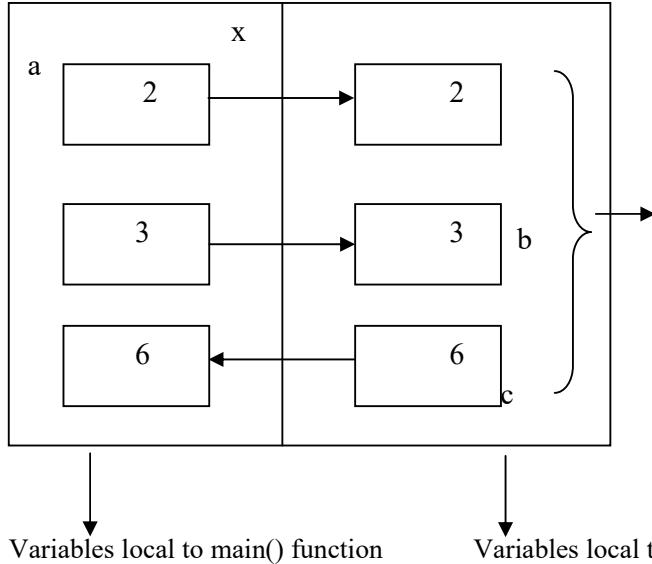
Enter two numbers:

23 2

The product of two numbers is: 46

Now let us see what happens to the actual and formal arguments in memory.

main() function mul() function



The variables are local to the `mul()` function which are created in memory with the function call and are destroyed with the return to the called function

What are meant by local variables? The answer is *local variables are those which can be used only by that function.*

Advantages of Call by value:

The only advantage is that this mechanism is simple and it reduces confusion and complexity.

Disadvantages of Call by value:

As you have seen in the above example, there is separate memory allocation for each of the variable, so unnecessary utilization of memory takes place.

The second disadvantage, which is very important from programming point of view, is that any changes made in the arguments are not reflected to the calling function, as these arguments are local to the called function and are destroyed with function return.

Let us discuss the second disadvantage more clearly using one example:

Example 5.12

Write a program to swap two values.

```
/*Program to swap two values*/
#include <stdio.h>
main()
{
int x = 2, y = 3;
void swap(int, int);

printf ("\n Values before swapping are %d %d", x, y);
swap (x, y);
printf ("\n Values after swapping are %d %d", x, y);
}

/* Function to swap(interchange) two values */
void swap( int a, int b )
{
```

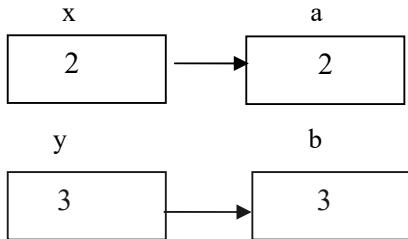
```
int t;  
t = a;  
a = b;  
b = t;  
}
```

Functions

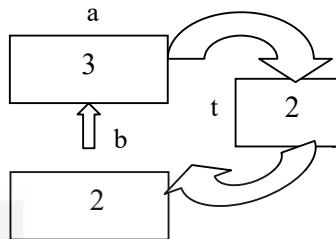
OUTPUT

Values before swap are 2 3
Values after swap are 2 3

But the output should have been 3 2. So what happened?



Values passing from main() to swap() function



Variables in swap() function

Here we observe that the changes which takes place in argument variables are not reflected in the main() function; as these variables namely a, b and t will be destroyed with function return.

- All these disadvantages will be removed by using “*call by reference*”, which will be discussed with the introduction of pointers in UNIT 11.

Check Your Progress 3

1. Write a function to print Fibonacci series upto ‘n’ terms 1,1,2,3,.....n

.....
.....
.....
.....
.....
.....
.....
.....

2. Write a function power (a, b) to calculate a^b

.....
.....
.....
.....
.....
.....
.....

5.9 RECURSION

Within a function body, if the function calls itself, the mechanism is known as '**Recursion**' and the function is known as '**Recursive function**'. Now let us study this mechanism in detail and understand how it works.

- As we see in this mechanism, a chaining of function calls occurs, so it is necessary for a recursive function to stop somewhere or it will result into infinite callings. So the most important thing to remember in this mechanism is that every "recursive function" should have a terminating condition.
- Let us take a very simple example of calculating factorial of a number, which we all know is computed using this formula $5! = 5*4*3*2*1$
- First we will write non – recursive or iterative function for this.

Example 5.13

Write a program to find factorial of a number

```
#include <stdio.h>
main ()
{
int n, factorial;
int fact(int);
printf ("Enter any number:\n" );
scanf ("%d", &n);
factorial = fact ( n); /* function call */
printf ("Factorial is %d\n", factorial);
}

/* Non recursive function of factorial */

int fact (int n)
{
int res = 1, i;
for (i = n; i >= 1; i--)
res = res * i;
return (res);
}
```

OUTPUT

```
Enter any number: 5
Factorial is 120
```

How it works?

Suppose we call this function with $n = 5$

Iterations:

1. $i= 5 \text{ res} = 1*5 = 5$
2. $i= 4 \text{ res} = 5*4 = 20$
3. $i= 3 \text{ res} = 20*4 = 60$
4. $i= 2 \text{ res} = 60*2 = 120$

Now let us write this function **recursively**. Before writing any function recursively, we first have to examine the problem, that it can be implemented through recursion.

For instance, we know $n! = n * (n - 1)!$ (Mathematical formula)

Or fact (n) = n*fact (n-1)
Or fact (5) = 5*fact (4)

That means this function calls itself but with value of argument *decreased by '1'*.

Example 5.14

Modify the program 8 using recursion.

```
/*Program to find factorial using recursion*/
#include<stdio.h>
main()
{
int n, factorial;
int fact(int);
printf("Enter any number: \n");
scanf("%d",&n);
factorial = fact(n); /*Function call */
printf ("Factorial is %d\n", factorial); }

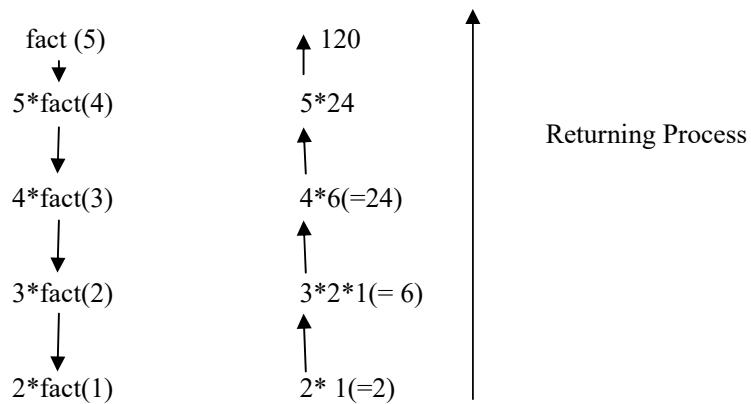
/* Recursive function of factorial */
int fact(int n)
{
int res;
if(n == 1) /* Terminating condition */
    return(1);
else
    res = n*fact(n-1); /* Recursive call */
    return(res); }
```

OUTPUT

Enter any number: 5
Factorial is 120

How it works?

Suppose we will call this function with $n = 5$



Thus a recursive function first proceeds towards the innermost condition, which is the termination condition, and then returns with the value to the outermost call and produces result with the values from the previous return.

Note: This mechanism applies only to those problems, which repeats itself. These types of problems can be implemented either through loops or recursive functions, which one is better understood to you.

Check Your Progress 4

1. Write recursive functions for calculating power of a number ‘a’ raised by another number ‘b’ i.e. a^b



5.10 SUMMARY

In this unit, we learnt about “Functions”: definition, declaration, prototypes, types, function calls datatypes and storage classes, types function invoking and lastly Recursion. All these subtopics must have given you a clear idea of how to create and call functions from other functions, how to send values through arguments, and how to return values to the called function. We have seen that the functions, which do not return any value, must be declared as “void”, return type. A function can return only one value at a time, although it can have many return statements. A function can return any of the data type specified in ‘C’.

Any variable declared in functions are local to it and are created with function call and destroyed with function return. The actual and formal arguments should match in type, order and number. A recursive function should have a terminating condition i.e. function should return a value instead of a repetitive function call.

5.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. /* Function to multiply two integers */

```
int mul( int a, int b )
{
    int c;
    c = a*b;
    return( c );
}
```

```

}

2. #include <stdio.h>
main()
{
    int x, y, z;
    int mul(int, int); /* function prototype */
    printf("Enter two numbers");
    scanf("%d %d", &x, &y);
    z = mul(x, y);           /* function call */
    printf("result is %d", z); }

```

Check Your Progress 2

1. (a) Valid
 (b) Invalid
 (c) Valid
 (d) Valid
 (e) Invalid

Check Your Progress 3

1. /* Function to print Fibonacci Series */

```

void fib(int n)
{
    int curr_term, int count = 0;
    int first = 1;
    int second = 1;
    print("%d %d", curr_term);
    count = 2;
    while(count <= n)
    {
        curr_term = first + second;
        printf("%d", curr_term);
        first = second;
        second = curr_term;
        count++;
    }
}

```

2. /* Non Recursive Power function i.e. pow(a, b) */

```

int pow( int a, int b)
{
    int i, p = 1;
    for(i = 1; i <= b; i++)
        p = p*a;
    return(p);
}

```

Check Your Progress 4

1. /* Recursive Power Function */

```

int pow( int a, int b )
{
    if( b == 0 )
        return(1);
    else
        return(a* pow(a, b-1)); /* Recursive call */
}

```

ignou
THE PEOPLE'S
UNIVERSITY

```
    }  
  
/* Main Function */  
main()  
{  
    int a, b, p;  
    printf(" Enter two numbers");  
    scanf( "%d %d", &a, &b );  
    p = pow(a, b); /* Function call */  
    printf( " The result is %d", p);  
}
```

5.12 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI
2. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill, 2002.
3. Computer Programming in C, *Raja Raman. V*, 2002, PHI.
5. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, TMGH,2002.

THE PEOPLE'S
UNIVERSITY

UNIT 6 STRUCTURES AND UNIONS

Structure

- 6.0 Introduction
- 6.1 Objectives
- 6.2 Declaration of Structures
- 6.3 Accessing the Members of a Structure
- 6.4 Initializing Structure Variables
- 6.5 Structures as Function Arguments
- 6.6 Structures and Arrays
- 6.7 Unions
- 6.8 Initializing an Union
- 6.9 Accessing the Members of an Union
- 6.10 Summary
- 6.11 Solutions / Answers
- 6.12 Further Readings

6.0 INTRODUCTION

We have seen so far how to store numbers, characters, strings, and even large sets of these primitives using arrays, but what if we want to store collections of different kinds of data that are somehow related. For example, a file about an employee will probably have his/her name, age, the hours of work, salary, etc. Physically, all of that is usually stored in someone's filing cabinet. In programming, if you have lots of related information, you group it together in an organized fashion. Let's say you have a group of employees, and you want to make a database! It just wouldn't do to have tons of loose variables hanging all over the place. Then we need to have a single data entity where we will be able to store all the related information together. But this can't be achieved by using the arrays alone, as in the case of arrays, we can group multiple data elements that are of the same data type, and is stored in consecutive memory locations, and is individually accessed by a subscript. That is where the user-defined datatype *Structures* come in.

Structure is commonly referred to as a user-defined data type. C's *structures* allow you to store multiple variables of any type in one place (the structure). A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a *member* of the structure. They can hold any number of variables, and you can make arrays of structures. This flexibility makes structures ideally useful for creating databases in C. Similar to the structure there is another user defined data type called *Union* which allows the programmer to view a single storage in more than one way i.e., a variable declared as union can store within its storage space, the data of different types, at different times. In this unit, we will be discussing the user-defined data type structures and unions.

6.1 OBJECTIVES

After going through this unit you should be able to:

- declare and initialize the members of the structures;
- access the members of the structures;
- pass the structures as function arguments;
- declare the array of structures;
- declare and define union; and
- perform all operations on the variables of type Union.

6.2 DECLARATION OF STRUCTURES

To declare a structure you must start with the keyword **struct** followed by the *structure name* or *structure tag* and within the braces the list of the structure's member variables. Note that the structure declaration does not actually create any variables. The syntax for the structure declaration is as follows:

```
struct structure-tag {  
    datatype variable1;  
    datatype variable2;  
    datatype variable 3;  
    ...  
};
```

For example, consider the student database in which each student has a roll number, name and course and the marks obtained. Hence to group this data with a structure-tag as **student**, we can have the declaration of structure as:

```
struct student {  
    int roll_no;  
    char name[20];  
    char course[20];  
    int marks_obtained;  
};
```

The point you need to remember is that, till this time no memory is allocated to the structure. This is only the definition of structure that tells us that there exists a user-defined data type by the name of student which is composed of the following members. Using this structure type, we have to create the structure variables:

```
struct student stud1, stud2;
```

At this point, we have created two instances or structure variables of the user-defined data type **student**. Now memory will be allocated. The amount of memory allocated will be the sum of all the data members which form part of the structure template.

The second method is as follows:

```
struct {  
    int roll_no;  
    char name[20];  
    char course[20];  
    int marks_obtained;  
} stud1,stud2;
```

In this case, a tag name *student* is missing, but still it happens to be a valid declaration of structure. In this case the two variables are allocated memory equivalent to the members of the structure.

The advantage of having a tag name is that we can declare any number of variables of the tagged named structure later in the program as per requirement.

If you have a small structure that you just want to define in the program, you can do the definition and declaration together as shown below. This will define a structure of type *struct telephone* and declare three instances of it.

Consider the example for declaring and defining a structure for the telephone billing with three instances:

```
struct telephone{  
    int tele_no;  
    int cust_code;  
    char cust_address[40];  
    int bill_amt;  
}tele1, tele2, tele3;
```

The structure can also be declared by using the `typedef` or `typedef`. This can be done as shown below:

```
typedef struct country{  
    char name[20];  
    int population;  
    char language[10];  
}Country;
```

This defines a structure which can be referred to either as `struct country` or `Country`, whichever you prefer. Strictly speaking, you don't need a tag name both before and after the braces if you are not going to use one or the other. But it is a standard practice to put them both in and to give them the same name, but the one after the braces starts with an uppercase letter.

The `typedef` statement doesn't occupy storage: it simply defines a new type. Variables that are declared with the `typedef` above will be of type `struct country`, just like `population` is of type `integer`. The structure variables can be now defined as below:

`Country Mexico, Canada, Brazil;`

6.3 ACCESSING THE MEMBERS OF A STRUCTURE

Individual structure members can be used like other variables of the same type. Structure members are accessed using the *structure member operator* (`.`), also called the *dot operator*, between the structure name and the member name. The syntax for accessing the member of the structure is:

`structurevariable. member-name;`

Let us take the example of the coordinate structure.

```
struct coordinate{  
    int x;  
    int y;  
};
```

Thus, to have the structure named first refer to a screen location that has coordinates $x=50, y=100$, you could write as,

```
first.x = 50;  
first.y = 100;
```

To display the screen locations stored in the structure second, you could write,

```
printf("%d,%d", second.x, second.y);
```

The individual members of the structure behave like ordinary date elements and can be accessed accordingly.

Now let us see the following program to clarify our concepts. For example, let us see, how will we go about storing and retrieving values of the individual data members of the student structure.

Example 6.1

*/*Program to store and retrieve the values from the student structure*/*

```
#include<stdio.h>
struct student {
    int roll_no;
    char name[20];
    char course[20];
    int marks_obtained ;
};
main()
{
student s1 ;
printf("Enter the student roll number:");
scanf("%d",&s1.roll_no);
printf("\nEnter the student name: ");
scanf("%s",s1.name);
printf("\nEnter the student course");
scanf("%s",s1.course);
printf("Enter the student percentage\n");
scanf("%d",&s1.marks_obtained);
printf("\nData entry is complete");
printf( "\nThe data entered is as follows:\n");
printf("\nThe student roll no is %d",s1.roll_no);
printf("\nThe student name is %s",s1.name);
printf("\nThe student course is %s",s1.course);
printf("\nThe student percentage is %d",s1.marks_obtained);
}
```

OUTPUT

```
Enter the student roll number: 1234
Enter the student name: ARUN
Enter the student course: MCA
Enter the student percentage: 84
Date entry is complete
```

```
The data entered is as follows:
The student roll no is 1234
The student name is ARUN
The student course is MCA
The student percentage is 84
```

Another way of accessing the storing the values in the members of a structure is by initializing them to some values at the time when we create an instance of the data type.

6.4 INITIALIZING STRUCTURE VARIABLES

Like other C variable types, structures can be initialized when they're declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values is separated by commas and enclosed in braces. For example, look at the following statements for initializing the values of the members of the *mysale* structure variable.

Example 6.2

```
struct sale {  
    char customer[20];  
    char item[20];  
    float amt;  
} mysale = { "XYZ Industries",  
            "toolkit",  
            600.00  
};
```

In a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Here's an example that expands on the previous one:

Example 6.3

```
struct customer {  
    char firm[20];  
    char contact[25];  
}  
  
struct sale {  
    struct customer buyer1;  
    char item [20];  
    float amt;  
} mysale = {  
    { "XYZ Industries", "Tyran Adams"},  
    "toolkit",  
    600.00  
};
```

These statements perform the following initializations:

- the structure member *mysale.buyer1.firm* is initialized to the string "XYZ Industries".
- the structure member *mysale.buyer1.contact* is initialized to the string "Tyran Adams".
- the structure member *mysale.item* is initialized to the string "toolkit".
- the structure member *mysale.amount* is initialized to the amount 600.00.

For example let us consider the following program where the data members are initialized to some value.

Example 6.4

Write a program to access the values of the structure initialized with some initial values.

```
/* Program to illustrate to access the values of the structure initialized with some
initial values*/
#include<stdio.h>
struct telephone{
    int tele_no;
    int cust_code;
    char cust_name[20];
    char cust_address[40];
    int bill_amt;
};
main()
{
    struct telephone tele = {2314345,
                            5463,
                            "Ram",
                            "New Delhi",
                            2435    };

    printf("The values are initialized in this program.");
    printf("\nThe telephone number is %d",tele.tele_no);
    printf("\nThe customer code is %d",tele.cust_code);
    printf("\nThe customer name is %s",tele.cust_name);
    printf("\nThe customer address is %s",tele.cust_address);
    printf("\nThe bill amount is %d",tele.bill_amt);
}
```

OUTPUT

The values are initialized in this program.
The telephone number is 2314345
The customer code is 5463
The customer name is Ram
The customer Address is New Delhi
The bill amount is 2435

Check Your Progress 1

1. What is the difference between the following two declarations?

```
struct x1{.....};  
typedef struct{.....}x2;
```

.....
.....
.....
.....
.....

2. Why can't you compare structures?

.....
.....

3. Why does size of report a larger size than, one expects, for a structure type, as if there were padding at the end?

.....
.....
.....
.....
.....

4. Declare a structure and instance together to display the date.

.....
.....
.....
.....
.....

6.5 STRUCTURES AS FUNCTION ARGUMENTS

C is a structured programming language and the basic concept in it is the modularity of the programs. This concept is supported by the functions in C language. Let us look into the techniques of passing the structures to the functions. This can be achieved in primarily two ways: Firstly, to pass them as simple parameter values by passing the structure name and secondly, through pointers. We will be concentrating on the first method in this unit and passing using pointers will be taken up in the next unit. Like other data types, a structure can be passed as an argument to a function. The program listing given below shows how to do this. It uses a function to display data on the screen.

Example 6.5

Write a program to demonstrate passing a structure to a function.

```
/*Program to demonstrate passing a structure to a function.*/  
  
#include <stdio.h>  
  
/*Declare and define a structure to hold the data.*/  
  
struct data{  
    float amt;  
    char fname [30];  
    char lname [30];  
} per;  
  
main()  
{  
void print_per(struct data x);  
printf("Enter the donor's first and last names separated by a space:");  
scanf("%s %s", per.fname, per.lname);  
printf("\nEnter the amount donated in rupees:");
```

```
scanf("%f", &per.amt);
print_per(per);
return 0;
}

void print_per(struct data x)
{
    printf("\n %s %s gave donation of amount Rs.%2f.\n", x.fname, x.lname, x.amt);
}
```

OUTPUT

Enter the donor's first and last names separated by a space: RAVI KANT

Enter the amount donated in rupees: 1000.00

RAVI KANT gave donation of the amount Rs. 1000.00.

You can also pass a structure to a function by passing the structure's address(that is, a pointer to the structure which we will be discussing in the next unit). In fact, in the older versions of C, this was the only way to pass a structure as an argument. It is not necessary now, but you might see the older programs that still use this method. If you pass a pointer to a structure as an argument, remember that you must use the indirect membership operator (\rightarrow) to access structure members in the function.

Please note the following points with respect to passing the structure as a parameter to a function.

- The return value of the called function must be declared as the value that is being returned from the function. If the function is returning the entire structure then the return value should be declared as *struct* with appropriate tag name.
- The actual and formal parameters for the structure data type must be the same as the *struct* type.
- The return statement is required only when the function is returning some data.
- When the return values of type is *struct*, then it must be assigned to the structure of identical type in the calling function.

Let us consider another example as shown in the Example 6.6, where *structure salary* has three fields related to an employee, namely - *name*, *no_days_worked* and *daily_wage*. To accept the values from the user we first call the function *get_data* that gets the values of the members of the structure. Then using the *wages* function we calculate the salary of the person and display it to the user.

Example 6.6

Write a program to accept the data from the user and calculate the salary of the person using concept of functions.

```
/* Program to accept the data from the user and calculate the salary of the person*/
```

```
#include<stdio.h>
main()
{
    struct sal    {
        char name[30];
        int no_days_worked;
        int daily_wage;    };
        struct sal salary;
        struct sal get_dat(struct);    /* function prototype*/
        float wages(struct);    /*function prototype*/
        float amount_payable;    /* variable declaration*/
```

```

salary = get_data(salary);
printf("The name of employee is %s",salary.name);
printf("Number of days worked is %d",salary.no_daya_worked);
printf("The daily wage of the employees is %d",salary.daily_wage);
amount_payable = wages(salary);
printf("The amount payable to %s is %f",salary.name,amount_payable);
}

struct sal get_data(struct sal income)
{
    printf("Please enter the employee name:\n");
    scanf("%s",income.name);
    printf("Please enter the number of days worked:\n");
    scanf("%d",&income.no_days_worked);
    printf('Please enter the employee daily wages:\n");
    scanf("%d",&income.daily_wages);
    return(income);
}

float wages(struct)
{
    struct sal amt;
    int total_salary ;
    total_salary = amt.no_days_worked * amt.daily_wages;
    return(total_salary); }
```

Check Your Progress 2

- How is structure passing and returning implemented?

.....

- How can I pass constant values to functions which accept structure arguments?

.....

- What will be the output of the program?

```
#include<stdio.h>
main()
{
    struct pqr{
        int x ;
    };
    struct pqr pqr ;
```

```
pqr.x =10 ;  
printf("%d", pqr.x);  
}
```

.....
.....
.....

6.6 STRUCTURES AND ARRAYS

Thus far we have studied as to how the data of heterogeneous nature can be grouped together and be referenced as a single unit of structure. Now we come to the next step in our real world problem. Let's consider the example of students and their marks. In this case, to avoid declaring various data variables, we grouped together all the data concerning the student's marks as one unit and call it student. The problem that arises now is that the data related to students is not going to be of a single student only. We will be required to store data for a number of students. To solve this situation one way is to declare a structure and then create sufficient number of variables of that structure type. But it gets very cumbersome to manage such a large number of data variables, so a better option is to declare an array.

So, revising the array for a few moments we would refresh the fact that an array is simply a collection of homogeneous data types. Hence, if we make a declaration as:

```
int temp[20];
```

It simply means that *temp* is an array of twenty elements where each element is of type integer, indicating homogenous data type. Now in the same manner, to extend the concept a bit further to the structure variables, we would say,

```
struct student stud[20] ;
```

It means that *stud* is an array of twenty elements where each element is of the type *struct student* (which is a user-defined data type we had defined earlier). The various members of the *stud* array can be accessed in the similar manner as that of any other ordinary array.

For example,

struct student stud[20], we can access the *roll_no* of this array as

```
stud[0].roll_no;  
stud[1].roll_no;  
stud[2].roll_no;  
stud[3].roll_no;  
...  
...  
...  
stud[19].roll_no;
```

Please remember the fact that for an array of twenty elements the subscripts of the array will be ranging from 0 to 19 (a total of twenty elements). So let us now start by seeing how we will write a simple program using array of structures.

Example 6.7

Write a program to read and display data for 20 students.

```
/*Program to read and print the data for 20 students*/
```

```
#include <stdio.h>
struct student { int roll_no;
                char name[20];
                char course[20];
                int marks_obtained ;
            };
main()
{
    struct student stud [20];
    int i;
    printf("Enter the student data one by one\n");
    for(i=0; i<=19; i++)
    {
        printf("Enter the roll number of %d student",i+1);
        scanf("%"d",&stud[i].roll_no);
        printf("Enter the name of %d student",i+1);
        scanf("%"s",stud[i].name);
        printf("Enter the course of %d student",i+1);
        scanf("%"d",stud[i].course);
        printf("Enter the marks obtained of %d student",i+1);
        scanf("%"d",&stud[i].marks_obtained);
    }
    printf("the data entered is as follows\n");
    for(i=0;i<=19;i++)
    {
        printf("The roll number of %d student is %d\n",i+1,stud[i].roll_no);
        printf("The name of %d student is %s\n",i+1,stud[i].name);
        printf("The course of %d student is %s\n",i+1,stud[i].course);
        printf("The marks of %d student is %d\n",i+1,stud[i].marks_obtained);
    }
}
```

The above program explains to us clearly that the array of structure behaves as any other normal array of any data type. Just by making use of the subscript we can access all the elements of the structure individually.

Extending the above concept where we can have arrays as the members of the structure. For example, let's see the above example where we have taken a structure for the student record. Hence in this case it is a real world requirement that each student will be having marks of more than one subject. Hence one way to declare the structure, if we consider that each student has 3 subjects, will be as follows:

```
struct student {
    int roll_no;
    char name[20];
    char course[20];
    int subject1 ;
    int subject2;
    int subject3;
};
```

The above described method is rather a bit cumbersome, so to make it more efficient we can have an array inside the structure, that is, we have an array as the member of the structure.

```
struct student {
```

```
int roll_no;
char name[20];
char course[20];
int subject[3] ;
};
```

Hence to access the various elements of this array we can the program logic as follows:

Example 6.8

/*Program to read and print data related to five students having marks of three subjects each using the concept of arrays */

```
#include<stdio.h>
struct student {
    int roll_no;
    char name[20];
    char course[20];
    int subject[3] ;
};

main()
{
    struct student stud[5];
    int i,j;
    printf("Enter the data for all the students:\n");
    for(i=0;i<=4;i++)
    {
        printf("Enter the roll number of %d student",i+1);
        scanf("%d",&stud[i].roll_no);
        printf("Enter the name of %d student",i+1);
        scanf("%s",stud[i].name);
        printf("Enter the course of %d student",i+1);
        scanf("%s",stud[i].course);
        for(j=0;j<=2;j++)
        {
            printf("Enter the marks of the %d subject of the student %d:\n",j+1,i+1);
            scanf("%d",&stud[i].subject[j]);
        }
    }
    printf("The data you have entered is as follows:\n");
    for(i=0;i<=4;i++)
    {
        printf("The %d th student's roll number is %d\n",i+1,stud[i].roll_no);
        printf("The %d the student's name is %s\n",i+1,stud[i].name);
        printf("The %d the student's course is %s\n",i+1,stud[i].course);
        for(j=0;j<=2;j++)
        {
            printf("The %d the student's marks of %d I subject are %d\n",i+1,j+1,
            stud[i].subject[j]);
        }
    }
    printf("End of the program\n");
}
```

Hence as described in the example above, the array as well as the arrays of structures can be used with efficiency to resolve the major hurdles faced in the real world programming environment.

6.7 UNIONS

Structures are a way of grouping homogeneous data together. But it often happens that at any time we require only one of the member's data. For example, in case of the support price of shares you require only the latest quotations. And only the ones that have changed need to be stored. So if we declare a structure for all the scripts, it will only lead to crowding of the memory space. Hence it is beneficial if we allocate space to only one of the members. This is achieved with the concepts of the *UNIONS*. *UNIONS* are similar to *STRUCTURES* in all respects but differ in the concept of storage space.

A *UNION* is declared and used in the same way as the structures. Yet another difference is that only one of its members can be used at any given time. Since all members of a Union occupy the same memory and storage space, the space allocated is equal to the largest data member of the Union. Hence, the member which has been updated last is available at any given time.

For example a union can be declared using the syntax shown below:

```
union union-tag {  
    datatype variable1;  
    datatype variable2;  
    ...  
};
```

For example,

```
union temp{  
    int x;  
    char y;  
    float z;  
};
```

In this case a float is the member which requires the largest space to store its value hence the space required for float(4 bytes) is allocated to the union. All members share the same space. Let us see how to access the members of the union.

Example 6.9

Write a program to illustrate the concept of union.

```
/* Declare a union template called tag */  
union tag {  
    int nbr;  
    char character;  
}  
/* Use the union template */  
union tag mixed_variable;  
/* Declare a union and instance together */  
union generic_type_tag {  
    char c;  
    int i;  
    float f;  
    double d;  
} generic;
```

6.8 INITIALIZING AN UNION

Let us see, how to initialize a Union with the help of the following example:

Example 6.10

```
union date_tag {  
    char complete_date [9];  
    struct part_date_tag {  
        char month[2];  
        char break_value1;  
        char day[2];  
        char break_value2;  
        char year[2];  
    } parrt_date;  
}date = {"01/01/05"};
```

6.9 ACCESSING THE MEMBERS OF AN UNION

Individual union members can be used in the same way as the structure members, by using the member operator or dot operator(.). However, there is an important difference in accessing the union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time. Trying to access the previously stored values will result in erroneous output.

Check Your Progress 3

1. What will be the output?

```
#include<stdio.h>  
main()  
{  
union{  
    struct{  
        char x;  
        char y;  
        char z;  
        char w;  
    }xyz;  
  
    struct{  
        int p;  
        int q ;  
    }pq;  
    long a ;  
    float b;  
    double d;  
}prq;  
printf ("%d",sizeof(prq));  
}
```

6.10 SUMMARY

In this unit, we have learnt how to use structures, a data type that you design to meet the needs of a program. A structure can contain any of C's data types, including other structures, pointers, and arrays. Each data item within a structure, called a *member*, is accessed using the structure member operator (.) between the structure name and the member name. Structures can be used individually, and can also be used in arrays.

Unions were presented as being similar to structures. The main difference between a union and a structure is that the union stores all its members in the same area. This means that only one member of a union can be used at a time.

6.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. The first form declares a *structure tag*; the second declares a *typedef*. The main difference is that the second declaration is of a slightly more abstract type - users do not necessarily know that it is a structure, and the keyword struct is not used while declaring an instance.
2. There is no single correct way for a compiler to implement a structure comparison consistent with C's low-level flavor. A simple byte-by-byte comparison could detect the random bits present in the unused "holes" in the structure (such padding is used to keep the alignment of later fields correct). A field-by-field comparison for a large structure might require an inordinate repetitive code.
3. Structures may have this padding (as well as internal padding), to ensure that alignment properties will be preserved when an array of contiguous structures is allocated. Even when the structure is not part of an array, the end padding remains, so that *sizeof* can always return a consistent size.
4.

```
struct date {  
    char month[2];  
    char day[2];  
    char year[4];  
} current_date;
```

Check Your Progress 2

1. When structures are passed as arguments to functions, the entire structure is typically pushed on the stack, using as many words. (Programmers often choose to use pointers instead, to avoid this overhead). Some compilers merely pass a pointer to the structure, though they may have to make a local copy to preserve pass-by value semantics.

Structures are often returned from functions in a pointed location by an extra, compiler-supplied "hidden" argument to the function. Some older compilers used a special, static location for structure returns, although this made structure-valued functions non-reentrant, which ANSI C disallows.

2. C has no way of generating anonymous structure values. You will have to use a temporary structure variable or a little structure - building function.
3. 10

Check Your Progress 3

1. 8

6.12 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication, 2002.
2. Computer Science A structured programming approach using C, *Behrouza Forouzan, Richard F. Gilberg*, Second Edition, Brooks/Cole, Thomson Learning, 2001.
3. Programming with C, Schaum Outlines, Second Edition, *Gottfried*, Tata McGraw Hill, 2003.



UNIT 7 POINTERS

Structure

- 7.0 Introduction
- 7.1 Objectives
- 7.2 Pointers and their Characteristics
- 7.3 Address and Indirection Operators
- 7.4 Pointer Type Declaration and Assignment
 - 7.4.1 Pointer to a Pointer
 - 7.4.2 Null Pointer Assignment
- 7.5 Pointer Arithmetic
- 7.6 Passing Pointers to Functions
 - 7.6.1 A Function Returning More than One Value
 - 7.6.2 Function Returning a Pointer
- 7.7 Arrays and Pointers
- 7.8 Array of Pointers
- 7.9 Pointers and Strings
- 7.10 Summary
- 7.11 Solutions / Answers
- 7.12 Further Readings

7.0 INTRODUCTION

If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers. One of those things, beginners in C find difficult is the concept of pointers. The purpose of this unit is to provide an introduction to pointers and their efficient use in the C programming. Actually, the main difficulty lies with the C's pointer terminology than the actual concept.

C uses pointers in three main ways. First, they are used to create *dynamic data structures*: data structures built up from blocks of memory allocated from the heap at run-time. Second, C uses pointers to handle *variable parameters* passed to functions. And third, pointers in C provide an alternative means of accessing information stored in arrays, which is especially valuable when you work with strings.

A normal variable is a location in memory that can hold a value. For example, when you declare a variable *i* as an integer, four bytes of memory is set aside for it. In your program, you refer to that location in memory by the name *i*. At the machine level, that location has a memory address, at which the four bytes can hold one integer value. A *pointer* is a variable that points to another variable. This means that it holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. It points to that other variable by holding its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That address points to a value. There is the pointer and the value pointed to. As long as you're careful to ensure that the pointers in your programs always point to valid memory locations, pointers can be useful, powerful, and relatively trouble-free tools.

We will start this unit with a basic introduction to pointers and the concepts surrounding pointers, and then move on to the three techniques described above. Thorough knowledge of the *pointers* is very much essential for your future courses like the *data structures etc..*

7.1 OBJECTIVES

After going through this unit you should be able to:

- understand the concept and use pointers;
- address and use of indirection operators;
- make pointer type declaration, assignment and initialization;
- use null pointer assignment;
- use the pointer arithmetic;
- handle pointers to functions;
- see the underlying unit of arrays and pointers; and
- understand the concept of dynamic memory allocation.

7.2 POINTERS AND THEIR CHARACTERISTICS

Computer's memory is made up of a sequential collection of storage cells called bytes. Each byte has a number called an address associated with it. When we declare a variable in our program, the compiler immediately assigns a specific block of memory to hold the value of that variable. Since every cell has a unique address, this block of memory will have a unique starting address. The size of this block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PC's integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. If you want to know the size of the various data types on your system, running the following code given in the Example 7.1 will give you the information.

Example 7.1

Write a program to know the size of the various data types on your system.

```
# include <stdio.h>
main()
{
    printf("n Size of a int = %d bytes", sizeof(int));
    printf("n Size of a float = %d bytes", sizeof(float));
    printf("n Size of a char = %d bytes", sizeof(char));
}
```

OUTPUT

```
Size of int = 2 bytes
Size of float = 4 bytes
Size of char = 1 byte
```

An *ordinary variable* is a location in memory that can hold a value. For example, when you declare a variable *num* as an integer, the compiler sets aside 2 bytes of memory (depends up the PC) to hold the value of the integer. In your program, you refer to that location in memory by the name *num*. At the machine level that location has a memory address.

We can access the value 100 either by the name num or by its memory address. Since addresses are simply digits, they can be stored in any other variable. Such variables that hold addresses of other variables are called *Pointers*. In other words, a *pointer* is simply a variable that contains an address, which is a location of another variable in memory. A pointer variable “points to” another variable by holding its address. Since a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That address points to a value. There is a pointer and the value pointed to. This fact can be a little confusing until you get comfortable with it, but once you get familiar with it, then it is extremely easy and very powerful. One good way to visualize this concept is to examine the figure 7.1 given below:

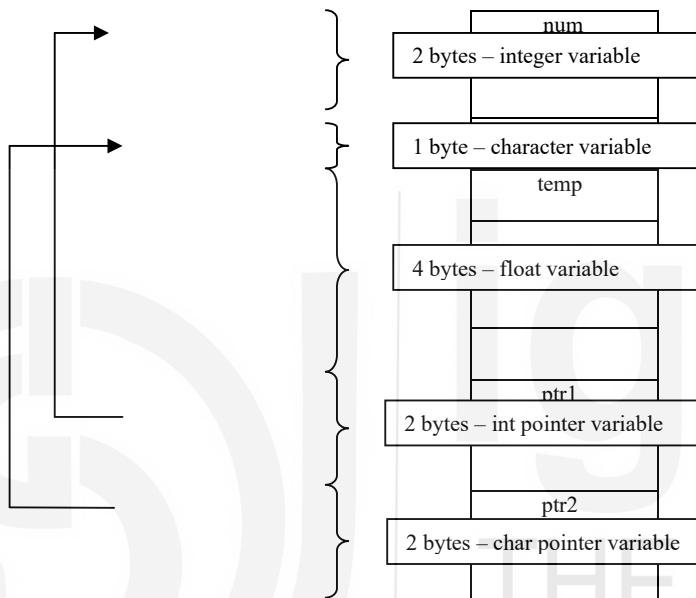


Figure 7.1: Concept of pointer variables

Let us see the important features of the pointers as follows:

Characteristic features of Pointers:

With the use of pointers in programming,

- i. The program execution time will be faster as the data is manipulated with the help of addresses directly.
- ii. Will save the memory space.
- iii. The memory access will be very efficient.
- iv. Dynamic memory is allocated.

7.3 THE ADDRESS AND INDIRECTION OPERATORS

Now we will consider how to determine the address of a variable. The operator that is available in C for this purpose is “&” (*address of*) operator. The operator & and the immediately preceding variable returns the address of the variable associated with it. C’s other unary pointer operator is the “*”, also called as *value at address* or *indirection operator*. It returns a value stored at that address. Let us look into the illustrative example given below to understand how they are useful.

Example 7.2

Write a program to print the address associated with a variable and value stored at that address.

```
/* Program to print the address associated with a variable and value stored at that address*/
```

```
# include <stdio.h>
main()
{
    int qty = 5;
    printf("Address of qty = %u\n",&qty);
    printf("Value of qty = %d \n",qty);
    printf("Value of qty = %d",*(&qty));
}
```

OUTPUT

```
Address of qty = 65524
Value of qty = 5
Value of qty = 5
```

Look at the *printf* statement carefully. The format specifier *%u* is taken to increase the range of values the address can possibly cover. The system-generated address of the variable is not fixed, as this can be different the next time you execute the same program. Remember unary operator operates on single operands. When *&* is preceded by the variable *qty*, has returned its address. Note that the *&* operator can be used only with simple variables or array elements. It cannot be applied to expressions, constants, or register variables.

Observe the third line of the above program. **(&qty)* returns the value stored at address 65524 i.e. 5 in this case. Therefore, *qty* and **(&qty)* will both evaluate to 5.

7.4 POINTER TYPE DECLARATION AND ASSIGNMENT

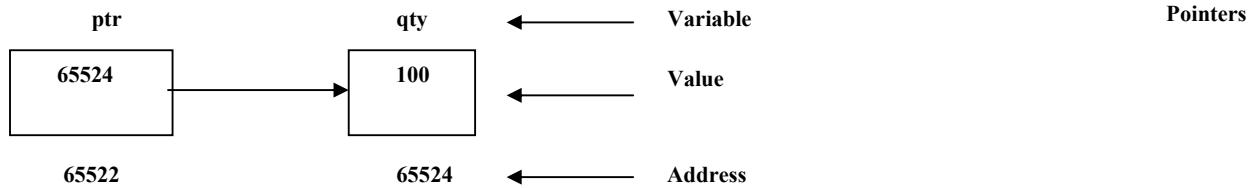
We have seen in the previous section that *&qty* returns the address of *qty* and this address can be stored in a variable as shown below:

```
ptr = &qty;
```

In C, every variable must be declared for its data type before it is used. Even this holds good for the pointers too. We know that *ptr* is not an ordinary variable like any integer variable. We declare the data type of the pointer variable as that of the type of the data that will be stored at the address to which it is pointing to. Since *ptr* is a variable, which contains the address of an integer variable *qty*, it can be declared as:

```
int *ptr;
```

where *ptr* is called a *pointer variable*. In C, we define a pointer variable by preceding its name with an asterisk(*). The “*” informs the compiler that we want a pointer variable, i.e. to set aside the bytes that are required to store the address in memory. The *int* says that we intend to use our pointer variable to store the address of an integer. Consider the following memory map:



Let us look into an example given below:

Example 7.3

```
/* Program below demonstrates the relationships we have discussed so far */
```

```
# include <stdio.h>
main()
{
    int qty = 5;
    int *ptr;      /* declares ptr as a pointer variable that points to an integer variable */
/*
ptr = &qty; /* assigning qty's address to ptr -> Pointer Assignment */

printf("Address of qty = %u \n", &qty);
printf("Address of qty = %u \n", ptr);
printf("Address of ptr = %u \n", &ptr);
printf("Value of ptr = %d \n", ptr);
printf("Value of qty = %d \n", qty);
printf("Value of qty = %d \n", *(&qty));
printf("Value of qty = %d", *ptr);
}
```

OUTPUT

```
Address of qty = 65524
Address of ptr = 65522
Value of ptr = 65524
Value of qty = 5
Value of qty = 5
Value of qty = 5
```

Try this as well:

Example 7.4

```
/* Program that tries to reference the value of a pointer even though the pointer is
uninitialized */
```

```
# include <stdio.h>
main()
{
    int *p; /* a pointer to an integer */
    *p = 10;
    printf("the value is %d", *p);
    printf("the value is %u", p);
}
```

This gives you an error. The pointer *p* is uninitialized and points to a random location in memory when you declare it. It could be pointing into the system stack, or the global variables, or into the program's code space, or into the operating system.

When you say `*p=10;` the program will simply try to write a 10 to whatever random location `p` points to. The program may explode immediately. It may subtly corrupt data in another part of your program and you may never realize it. Almost always, an uninitialized pointer or a bad pointer address causes the fault.

This can make it difficult to track down the error. Make sure you initialize all pointers to a valid address before dereferencing them.

Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable. Remember the variable whose address is assigned to the pointer variable must be declared earlier in the program. In the example given below, let us assign the pointer `p` with an address and also a value 10 through `*p`.

Example 7.5

Let us say,

```
int x; /* x is initialized to a value 10*/
p = &x; /* Pointer declaration & Assignment */
*p=10;
```

Let us write the complete program as shown below:

```
# include <stdio.h>
main()
{
    int *p; /* a pointer to an integer */
    int x;
    p = &x;
    *p=10;
    printf("The value of x is %d",*p);
    printf("\nThe address in which the x is stored is %d",p);
}
```

OUTPUT

```
The value of x is 10
The address in which the x is stored is 52004
```

This statement puts the value of 20 at the memory location whose address is the value of `px`. As we know that the value of `px` is the address of `x` and so the old value of `x` is replaced by 20. This is equivalent to assigning 20 to `x`. Thus we can change the value of a variable *indirectly* using a pointer and the *indirection operator*.

7.4.1 Pointer to a Pointer

The concept of pointer can be extended further. As we have seen earlier, a pointer variable can be assigned the address of an ordinary variable. Now, this variable itself could be another pointer. This means that a pointer can contain address of another pointer. The following program will makes you the concept clear.

Example 7.6

```
/* Program that declares a pointer to a pointer */
```

```
# include<stdio.h>
main()
{
```

```

int i = 100;
int *pi;
int **pii;
pi = &i;
pii = &pi;

printf ("Address of i = %u \n", &i);
printf("Address of i = %u \n", pi);
printf("Address of i = %u \n", *pii);
printf("Address of pi = %u \n", &pi);
printf("Address of pi = %u \n", pii);
printf("Address of pii = %u \n", &pii);
printf("Value of i = %d \n", i);
printf("Value of i = %d \n", *(&i));
printf("Value of i = %d \n", *pi);
printf("Value of i = %d", **pii);
}

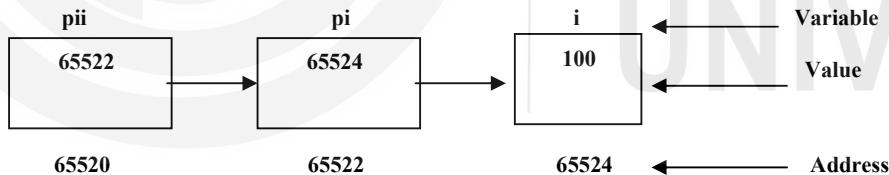
```

OUTPUT

Address of i = 65524
 Address of i = 65524
 Address of i = 65524
 Address of pi = 65522
 Address of pi = 65522
 Address of pii = 65520

Value of i = 100
 Value of i = 100
 Value of i = 100
 Value of i = 100

Consider the following memory map for the above shown example:



7.4.2 Null Pointer Assignment

It does make sense to assign an integer value to a pointer variable. An exception is an assignment of 0, which is sometimes used to indicate some special condition. A macro is used to represent a null pointer. That macro goes under the name *NULL*. Thus, setting the value of a pointer using the *NULL*, as with an assignment statement such as *ptr = NULL*, tells that the pointer has become a *null* pointer. Similarly, as one can test the condition for an integer value as zero or not, like *if(i == 0)*, as well we can test the condition for a null pointer using *if(ptr == NULL)* or you can even set a pointer to *NULL* to indicate that it's no longer in use. Let us see an example given below.

Example 7.7

```
# include<stdio.h>
# define NULL 0
```

```
main()
{
    int *pi = NULL;
    printf("The value of pi is %u", pi);
}
```

OUTPUT

The value of pi is 0

Check Your Progress 1

1. How is a pointer variable being declared? What is the purpose of data type included in the pointer declaration?

.....
.....
.....

2. What would be the output of following programs?

(i) void main()
{
 int i = 5;
 printf("Value of i = %d Address of i = %u", i, &i);
 &i = 65534;
 printf("\n New value of i = %d New Address of i = %u", i, &i);
}

(ii) void main()
{
 int *i, *j;
 j = i * 2;
 printf("j = %u", j);
}

.....
.....
.....

3. Explain the effect of the following statements:

(i) int x = 10, *px = &x;

(ii) char *pc;

(iii) int x;
void *ptr = &x;
*(int *) ptr = 10;

.....
.....
.....

7.5 POINTER ARITHMETIC

Pointer variables can also be used in arithmetic expressions. The following operations can be carried out on pointers:

1. Pointers can be incremented or decremented to point to different locations like

```
ptr1 = ptr2 + 3;
ptr++;
--ptr;
```

However, `ptr++` will cause the pointer `ptr` to point the next address value of its type. For example, if `ptr` is a pointer to float with an initial value of 65526, then after the operation `ptr ++` or `ptr = ptr+1`, the value of `ptr` would be 65530. Therefore, if we increment or decrement a pointer, its value is increased or decreased by the length of the data type that it points to.

2. If `ptr1` and `ptr2` are properly declared and initialized pointers, the following operations are valid:

```
res = res + *ptr1;
*ptr1 = *ptr2 + 5;
prod = *ptr1 * *ptr2;
quo = *ptr1 / *ptr2;
```

Note that there is a blank space between `/` and `*` in the last statement because if you write `/*` together, then it will be considered as the beginning of a comment and the statement will fail.

3. Expressions like `ptr1 == ptr2`, `ptr1 < ptr2`, and `ptr2 != ptr1` are permissible provided the pointers `ptr1` and `ptr2` refer to same and related variables. These comparisons are common in handling arrays.

Suppose `p1` and `p2` are pointers to related variables. The following operations cannot work with respect to pointers:

1. Pointer variables cannot be added. For example, `p1 = p1 + p2` is not valid.
2. Multiplication or division of a pointer with a constant is not allowed. For example, `p1 * p2` or `p2 / 5` are invalid.
3. An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block. Suppose `p` and `q` are two pointers. If we say, `p = q`; when `q` is uninitialized. The pointer `p` will then become uninitialized as well, and any reference to `*p` is an invalid pointer reference.

7.6 PASSING POINTERS TO FUNCTIONS

As we have studied in the FUNCITONS that arguments can generally be passed to functions in one of the two following ways:

1. Pass by value method
2. Pass by reference method

In the first method, when arguments are passed by value, a copy of the *values* of actual arguments is passed to the calling function. Thus, any changes made to the variables inside the function will have no effect on variables used in the actual argument list.

However, when arguments are passed by reference (i.e. when a pointer is passed as an argument to a function), the *address* of a variable is passed. The contents of that address can be accessed freely, either in the called or calling function. Therefore, the function called by reference can change the value of the variable used in the call.

Here is a simple program that illustrates the difference.

Example 7.8

Write a program to swap the values using the pass by value and pass by reference methods.

```
/* Program that illustrates the difference between ordinary arguments, which are passed by value, and pointer arguments, which are passed by reference */
```

```
# include <stdio.h>
main()
{
    int x = 10;
    int y = 20;
    void swapVal( int, int );      /* function prototype */
    void swapRef( int *, int * ); /*function prototype*/
    printf("PASS BY VALUE METHOD\n");
    printf("Before calling function swapVal  x=%d y=%d",x,y);
    swapVal(x, y);             /* copy of the arguments are passed */
    printf("\nAfter calling function swapVal  x=%d y=%d",x,y);
    printf("\n\nPASS BY REFERENCE METHOD");
    printf("\nBefore calling function swapRef x=%d y=%d",x,y);
    swapRef(&x,&y);           /*address of arguments are passed */
    printf("\nAfter calling function swapRef x=%d y=%d",x,y);
}

/* Function using the pass by value method*/
void swapVal(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
    printf("\nWithin function swapVal      x=%d y=%d",x,y);
    return;
}

/*Function using the pass by reference method*/
void swapRef(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
    printf("\nWithin function swapRef      *px=%d *py=%d",*px,*py);
    return;
}
```

OUTPUT

PASS BY VALUE METHOD

```
Before calling function swapVal  x=10      y=20
Within function swapVal        x=20      y=10
After calling function swapVal x=10      y=20
```

PASS BY REFERENCE METHOD

```
Before calling function swapRef  x=10      y=20
Within function swapRef        *px=20    *py=10
After calling function swapRef x=20      y=10
```

This program contains two functions, *swapVal* and *swapRef*.

In the function *swapVal*, arguments *x* and *y* are passed by *value*. So, any changes to the arguments are local to the function in which the changes occur. Note the values of *x* and *y* remain unchanged even after exchanging the values of *x* and *y* inside the function *swapVal*.

Now consider the function *swapRef*. This function receives two *pointers* to integer variables as arguments identified as pointers by the indirection operators that appear in argument declaration. This means that in the function *swapRef*, arguments *x* and *y* are passed by *reference*. So, any changes made to the arguments inside the function *swapRef* are reflected in the function *main()*. Note the values of *x* and *y* is interchanged after the function call *swapRef*.

7.6.1 A Function returning more than one value

Using *call by reference* method we can make a function return more than one value at a time, which is not possible in the *call by value* method. The following program will makes you the concept very clear.

Example 7.9

Write a program to find the perimeter and area of a rectangle, if length and breadth are given by the user.

```
/* Program to find the perimeter and area of a rectangle*/
#include <stdio.h>
void main()
{
float len,br;
float peri, ar;
void periarea(float length, float breadth, float *, float *);
printf("\nEnter the length and breadth of a rectangle in metres: \n");
scanf("%f %f",&len,&br);
periarea(len,br,&peri,&ar);
printf("\nPerimeter of the rectangle is %f metres", peri);
printf("\nArea of the rectangle is %f sq. metres", ar);
}

void periarea(float length, float breadth, float *perimeter, float *area)
{
*perimeter = 2 *(length +breadth);
*area = length * breadth;
}
```

OUTPUT

Enter the length and breadth of a rectangle in metres:

23.0 3.0

Perimeter of the rectangle is 52.000000 metres

Area of the rectangle is 69.000000 sq. metres

Here in the above program, we have seen that the function *periarea* is returning two values. We are passing the values of *len* and *br* but, addresses of *peri* and *ar*. As we are passing the addresses of *peri* and *ar*, any change that we make in values stored at addresses contained in the variables **perimeter* and **area*, would make the change effective even in *main()* also.

7.6.2 Function returning a pointer

A function can also return a pointer to the calling program, the way it returns an int, a float or any other data type. To return a pointer, a function must explicitly mention in the calling program as well as in the function prototype. Let's illustrate this with an example:

Example: 7.10

Write a program to illustrate a function returning a pointer.

```
/*Program that shows how a function returns a pointer */
```

```
# include<stdio.h>

void main()
{
    float *a;
    float *func(); /* function prototype */
    a = func();
    printf("Address = %u", a);
}
float *func()
{
    float r = 5.2;
    return(&r);
}
```

OUTPUT

Address = 65516

This program only shows how a function can return a pointer. This concept will be used later while handling arrays.

Check Your Progress 2

1. Tick mark(✓)whether each of the following statements are true or false.

- | | | |
|---|-------------------------------|--------------------------------|
| (i) An integer is subtracted from a pointer variable. | <input type="checkbox"/> True | <input type="checkbox"/> False |
| (ii) Pointer variables can be compared. | <input type="checkbox"/> True | <input type="checkbox"/> False |
| (iii) Pointer arguments are passed by value. | <input type="checkbox"/> True | <input type="checkbox"/> False |

- (iv) Value of a local variable in a function can be changed by another function. True False
- (v) A function can return more than one value. True False
- (vi) A function can return a pointer. True False

7.7 ARRAYS AND POINTERS

Pointers and arrays are so closely related. An array declaration such as `int arr[5]` will lead the compiler to pick an address to store a sequence of 5 integers, and `arr` is a name for that address. The array name in this case is the *address* where the sequence of integers starts. Note that the value is not the first integer in the sequence, nor is it the sequence in its entirety. The value is just an address.

Now, if `arr` is a one-dimensional array, then the address of the first array element can be written as `&arr[0]` or simply `arr`. Moreover, the address of the second array element can be written as `&arr[1]` or simply `(arr+1)`. In general, address of array element $(i+1)$ can be expressed as either `&arr[i]` or as `(arr + i)`. Thus, we have two different ways for writing the address of an array element. In the latter case i.e., expression `(arr + i)` is a symbolic representation for an address rather than an arithmetic expression. Since `&arr[i]` and `(arr + i)` both represent the address of the i^{th} element of `arr`, so `arr[i]` and `*(arr + i)` both represent the contents of that address i.e., the value of i^{th} element of `arr`.

Note that it is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as `arr`, `(arr + i)` and `arr[i]` cannot appear on the left side of an assignment statement. Thus we cannot write a statement such as:

```
&arr[0] = &arr[1]; /* Invalid */
```

However, we can assign the value of one array element to another through a pointer, for example,

```
ptr = &arr[0]; /* ptr is a pointer to arr[ 0 ] */
arr[1] = *ptr; /* Assigning the value stored at address to arr[1 ] */
```

Here is a simple program that will illustrate the above-explained concepts:

Example 7.11

```
/* Program that accesses array elements of a one-dimensional array using pointers */

#include<stdio.h>
main()
{
    int arr[ 5 ] = {10, 20, 30, 40, 50};
    int i;

    for(i = 0; i < 5; i++)
    {
        printf("i=%d\t arr[i]=%d\t *(arr+i)=%d\n", i, arr[i], *(arr+i));
        printf("&arr[i]=%u\t arr+i=%u\n", &arr[i],(arr+i));
    }
}
```

OUTPUT:

i=0	arr[i]=10	$*(arr+i)=10$	&arr[i]=65516	arr+i=65516
i=1	arr[i]=20	$*(arr+i)=20$	&arr[i]=65518	arr+i=65518
i=2	arr[i]=30	$*(arr+i)=30$	&arr[i]=65520	arr+i=65520
i=3	arr[i]=40	$*(arr+i)=40$	&arr[i]=65522	arr+i=65522
i=4	arr[i]=50	$*(arr+i)=50$	&arr[i]=65524	arr+i=65524

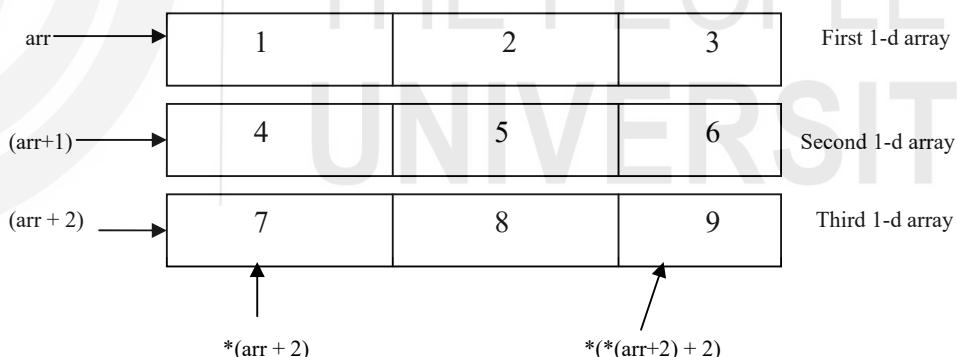
Note that i is added to a pointer value (address) pointing to integer data type (i.e., the array name) the result is the pointer is increased by i times the size (in bytes) of integer data type. Observe the addresses 65516, 65518 and so on. So if ptr is a char pointer, containing addresses a , then $ptr+1$ is $a+1$. If ptr is a float pointer, then $ptr+1$ is $a+4$.

Pointers and Multidimensional Arrays

C allows multidimensional arrays, lays them out in memory as contiguous locations, and does more behind the scenes address arithmetic. Consider a 2-dimensional array.

```
int arr[ 3 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

The compiler treats a 2 dimensional array as an array of arrays. As you know, an array name is a pointer to the first element within the array. So, `arr` points to the first 3-element array, which is actually the first row (i.e., row 0) of the two-dimensional array. Similarly, `(arr + 1)` points to the second 3-element array (i.e., row 1) and so on. The value of this pointer, `*(arr + 1)`, refers to the entire row. Since row 1 is a one-dimensional array, `(arr + 1)` is actually a pointer to the first element in row 1. Now add 2 to this pointer. Hence, `(*arr + 1) + 2` is a pointer to element 2 (i.e., the third element) in row 1. The value of this pointer, `*(*arr + 1) + 2`, refers to the element in column 2 of row 1. These relationships are illustrated below:



7.8 ARRAY OF POINTERS

The way there can be an array of integers, or an array of float numbers, similarly, there can be array of pointers too. Since a pointer contains an address, an array of pointers would be a collection of addresses. For example, a multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays.

Two-dimensional array can be defined as a one-dimensional array of integer pointers by writing:

```
int *arr[3];
```

rather than the conventional array definition,

```
int arr[3][5];
```

Similarly, an n-dimensional array can be defined as (n-1)-dimensional array of pointers by writing

```
data-type *arr[subscript 1] [subscript 2].... [subscript n-1];
```

The subscript1, subscript2 indicate the maximum number of elements associated with each subscript.

Example 7.12

Write a program in which a two-dimensional array is represented as an array of integer pointers to a set of single-dimensional integer arrays.

```
/* Program calculates the difference of the corresponding elements of two table of integers */
```

```
# include <stdio.h>
# include <stdlib.h>
# define MAXROWS 3
void main()
{
    int *ptr1[MAXROWS], *ptr2 [MAXROWS], *ptr3 [MAXROWS];
    int rows, cols, i, j;
    void inputmat(int *[ ], int, int);
    void dispmat(int *[ ], int, int);
    void calcdiff(int *[ ], int *[ ], int *[ ], int, int);

    printf("Enter no. of rows & columns \n");
    scanf("%d%d", &rows, &cols);

    for(i = 0; i < rows; i++)
    {
        ptr1[ i ]=(int *) malloc(cols * sizeof(int));
        ptr2[ i ]=(int *) malloc(cols * sizeof(int));
        ptr3[ i ]=(int *) malloc(cols * sizeof(int));
    }

    printf("Enter values in first matrix \n");
    inputmat(ptr1, rows, cols);
    printf("Enter values in second matrix \n");
    inputmat(ptr2, rows, cols);
    calcdiff(ptr1, ptr2, ptr3, rows, cols);
    printf("Display difference of the two matrices \n");
    dispmat(ptr3, rows, cols);
}

void inputmat(int *ptr1[MAXROWS], int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++)
    {
        for(j = 0; j < n; j++)
        {
            scanf("%d",*(ptr1 + i) + j));
        }
    }
}
```

```
        }
    }
    return;
}

void dispmat(int *ptr3[ MAXROWS ], int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++)
    {
        for(j = 0; j < n; j++)
        {
            printf("%d ", *(*(ptr3 + i) + j));
        }
        printf("\n");
    }
    return;
}

void calcdiff(int *ptr1[ MAXROWS ], int *ptr2 [ MAXROWS ],
              int *ptr3[MAXROWS], int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++)
    {
        for(j = 0; j < n; j++)
        {
            *(*(ptr3 + i) + j) = *(*(ptr1 + i) + j) - *(*(ptr2 + i) + j);
        }
    }
    return;
}
```

OUTPUT

Enter no. of rows & columns

3 3

Enter values in first matrix

2 6 3

5 9 3

1 0 2

Enter values in second matrix

3 5 7

2 8 2

1 0 1

Display difference of the two matrices

-1 1 -4

3 1 1

0 0 1

In this program, *ptr1*, *ptr2*, *ptr3* are each defined as an array of pointers to integers. Each array has a maximum of MAXROWS elements. Since each element of *ptr1*, *ptr2*, *ptr3* is a pointer, we must provide each pointer with enough memory for each row of integers. This can be done using the library function *malloc* included in *stdlib.h* header file as follows:

```
ptr1[ i ] =(int *) malloc( cols * sizeof(int));
```

This function reserves a block of memory whose size(in bytes) is equivalent to `cols * sizeof(int)`. Since `cols = 3`, so $3 * 2$ (size of int data type) i.e., 6 is allocated to each `ptr1[1]`, `ptr1[2]` and `ptr1[3]`. This `malloc` function returns a pointer of type `void`. This means that we can assign it to any type of pointer. In this case, the pointer is type-casted to an integer type and assigned to the pointer `ptr1[1]`, `ptr1[2]` and `ptr1[3]`. Now, each of `ptr1[1]`, `ptr1[2]` and `ptr1[3]` points to the first byte of the memory allocated to the corresponding set of one-dimensional integer arrays of the original two-dimensional array.

The process of calculating and allocating memory at run time is known as *dynamic memory allocation*. The library routine `malloc` can be used for this purpose.

Instead of using conventional array notation, pointer notation has been used for accessing the address and value of corresponding array elements which has been explained to you in the previous section. The difference of the array elements within the function `calcdiff` is written as

$$\ast(\ast(ptr3 + i) + j) = \ast(\ast(ptr1 + i) + j) - \ast(\ast(ptr2 + i) + j);$$

7.9 POINTERS AND STRINGS

As we have seen in strings, a string in C is an array of characters ending in the null character(written as '`\0`'), which specifies where the string terminates in memory. Like in one-dimensional arrays, a string can be accessed via a pointer to the first character in the string. The value of a string is the(constant) address of its first character. Thus, it is appropriate to say that a string is a constant pointer.

A string can be declared as a character array or a variable of type `char *`. The declarations can be done as shown below:

```
char country[ ] = "INDIA";
char *country = "INDIA";
```

Each initialize a variable to the string "INDIA". The second declaration creates a pointer variable `country` that points to the letter I in the string "INDIA" somewhere in memory.

Once the base address is obtained in the pointer variable `country`, `*country` would yield the value at this address, which gets printed through,

```
printf("%s", *country);
```

Here is a program that dynamically allocates memory to a character pointer using the library function `malloc` at run-time. An advantage of doing this way is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional character array.

Example 7.13

Write a program to test whether the given string is a palindrome or not.

```
/* Program tests a string for a palindrome using pointer notation */
```

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

main()
```

```
{  
    char *palin, c;  
    int i, count;  
  
    short int palindrome(char,int); /*Function Prototype */  
    palin=(char *) malloc(20 * sizeof(char));  
    printf("\nEnter a word: ");  
    do  
    {  
        c = getchar();  
        palin[i] = c;  
        i++;  
    }while(c != '\n');  
  
    i = i-1;  
    palin[i] = '\0';  
    count = i;  
  
    if(palindrome(palin,count) == 1)  
        printf("\nEnterd word is not a palindrome.");  
    else  
        printf("\nEnterd word is a palindrome");  
    }  
  
short int palindrome(char *palin, int len)  
{  
    short int i = 0, j = 0;  
    for(i=0 , j=len-1; i < len/2;i++,j--)  
    {  
        if(palin[i] == palin[j])  
            continue;  
        else  
            return(1);  
    }  
    return(0);  
}
```

OUTPUT

Enter a word: malayalam

Entered word is a palindrome.

Enter a word: abcdab

Entered word is not a palindrome.

Array of pointers to strings

Arrays may contain pointers. We can form an array of strings, referred to as a string array. Each entry in the array is a string, but in C a string is essentially a pointer to its first character, so each entry in an array of strings is actually a pointer to the first character of a string. Consider the following declaration of a string array:

```
char *country[ ] = {  
    "INDIA", "CHINA", "BANGLADESH", "PAKISTAN", "U.S"  
};
```

The `*country[]` of the declaration indicates an array of five elements. The `char*` of the declaration indicates that each element of array `country` is of type “pointer to `char`”. Thus, `country[0]` will point to `INDIA`, `country[1]` will point to `CHINA`, and so on.

Thus, even though the array `country` is fixed in size, it provides access to character strings of any length. However, a specified amount of memory will have to be allocated for each string later in the program, for example,

```
country[ i ] = (char *) malloc(15 * sizeof(char));
```

The `country` character strings could have been placed into a two-dimensional array but such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when a large number of strings are stored with most strings shorter than the longest string.

As individual strings can be accessed by referring to the corresponding array element, individual string elements be accessed through the use of the indirection operator. For example, `(*(*country + 3) + 2)` refers to the third character in the fourth string of the array `country`. Let us see an example below.

Example 7.14

Write a program to enter a list of strings and rearrange them in alphabetical order, using a one-dimensional array of pointers, where each pointer indicates the beginning of a string:

```
/* Program to sort a list of strings in alphabetical order using an array of pointers */
```

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# include <string.h>

void readinput(char *[ ], int);
void writeoutput(char *[ ], int);
void reorder(char *[ ], int);

main()
{
    char *country[ 5 ];
    int i;
    for(i = 0; i < 5; i++)
    {
        country[ i ] = (char *) malloc(15 * sizeof(char));
    }
    printf("Enter five countries on a separate line\n");
    readinput(country, 5);
    reorder(country, 5);
    printf("\nReordered list\n");
    writeoutput(country, 5);
    getch();
}

void readinput(char *country[ ], int n)
{
    int i;
```

```
for(i = 0; i < n; i++)
    {   scanf("%s", country[ i ]);   }
return;
}

void writeoutput(char *country[ ], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {   printf("%s", country[ i ]);
        printf("\n");
    }
return;
}

void reorder(char *country[ ], int n)
{
    int i, j;
    char *temp;
    for(i = 0; i < n-1; i++)
    {
        for(j = i+1; j < n; j++)
        {
            if(strcmp(country[ i ], country[ j ]) > 0)
            {
                temp = country[ i ];
                country[ i ] = country[ j ];
                country[ j ] = temp;
            }
        }
    }
return;
}
```

OUTPUT

Enter five countries on a separate line
INDIA
BANGLADESH
PAKISTAN
CHINA
SRILANKA

Reordered list
BANGLADESH
CHINA
INDIA
PAKISTAN
SRILANKA

The limitation of the string array concept is that when we are using an array of pointers to strings we can initialize the strings at the place where we are declaring the array, but we cannot receive the strings from keyboard using *scanf()*.

Check Your Progress 3

1. What is meant by array of pointers?

2. How the indirection operator can be used to access a multidimensional array element.

3. A C program contains the following declaration.

```
float temp[ 3 ][ 2 ] = { {13.4, 45.5}, {16.6, 47.8}, {20.2, 40.8} };
```

- (i) What is the meaning of temp ?
 - (ii) What is the meaning of $(\text{temp} + 2)$?
 - (iii) What is the meaning of $*(\text{temp} + 1)$?
 - (iv) What is the meaning of $*(\text{temp} + 2) + 1$?
 - (v) What is the meaning of $*(*(\text{temp}) + 1) + 1$?
 - (vi) What is the meaning of $*(*(\text{temp} + 2))$?

7.10 SUMMARY

In this unit we have studied about pointers, pointer arithmetic, passing pointers to functions, relation to arrays and the concept of dynamic memory allocation. A pointer is simply a variable that contains an address which is a location of another variable in memory. The unary operator `&`, when preceded by any variable returns its address. C's other unary pointer operator is `*`, when preceded by a pointer variable returns a value stored at that address.

Pointers are often passed to a function as arguments by reference. This allows data items within the calling function to be accessed, altered by the called function, and then returned to the calling function in the altered form. There is an intimate relationship between pointers and arrays as an array name is really a pointer to the first element in the array. Access to the elements of array using pointers is enabled by adding the respective subscript to the pointer value (i.e. address of zeroth element) and the expression preceded with an indirection operator.

As pointer declaration does not allocate memory to store the objects it points at, therefore, memory is allocated at run time known as *dynamic memory allocation*. The library routine *malloc* can be used for this purpose.

7.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. Refer to section 7.4. The data type included in the pointer declaration, refers to the type of data stored at the address which we will be storing in our pointer.
 2. (i) Compile-time Error : Lvalue Required. Means that the left side of an assignment operator must be an addressable expression that include a variable or an indirection through a pointer.

- (ii) Multiplication of a pointer variable with a constant is invalid.
3. (i) Refer section 7.4
(ii) Refer section 7.4
(iii) This means pointers can be of type void but can't be de-referenced without explicit casting. This is because the compiler can't determine the size of the object the pointer points to.

Check Your Progress 2

- 1 (i) True.
(ii) True.
(iii) False.
(iv) True.
(v) True.
(vi) True.

Check Your Progress 3

1. Refer section 7.4.
2. Refer section 7.4 to comprehend the convention followed.
3. (i) Refers to the base address of the array temp.
(ii) Address of the first element of the last row of array temp i.e. address of element 20.2.
(iii) Will give you 0. To get the value of the last element of the first array i.e. the correct syntax would be `*(*(temp+0)+1)`.
(iv) Address of the last element of last row of array temp i.e. of 40.8.
(v) Displays the value 47.8 i.e., second element of last row of array temp.
(vi) Displays the value 20.2 i.e., first element of last row of array temp.

7.12 FURTHER READINGS

1. Programming with C, Second Edition, *Gotfried Byron S*, Tata McGraw Hill, India.
2. The C Programming Language, Second Edition, *Brian Kernighan and Dennis Richie*, PHI, 2002.
3. Programming in ANSI C, Second Edition, *Balaguruswamy E*, Tata McGraw Hill, India, 2002.
4. How to Solve it by Computer, *R.G.Dromey*, PHI, 2002.
5. C Programming in 12 easy lessons, *Greg Perry*, SAMS, 2002.
6. Teach Yourself C in 21 days, Fifth Edition, *Peter G*, Fifth edition,SAMS, 2002.

UNIT 8 FILE HANDLING

Structure

- 8.0 Introduction
- 8.1 Objectives
- 8.2 File Handling in C Using File Pointers
 - 8.2.1 Open a file using the function `fopen()`
 - 8.2.2 Close a file using the function `fclose()`
- 8.3 Input and Output using file pointers
 - 8.3.1 Character Input and Output in Files
 - 8.3.2 String Input / Output Functions
 - 8.3.3 Formatted Input / Output Functions
 - 8.3.4 Block Input / Output Functions
- 8.4 Sequential Vs Random Access Files
- 8.5 Positioning the File Pointer
- 8.6 The Unbuffered I/O - The UNIX like File Routines
- 8.7 Summary
- 8.8 Solutions / Answers
- 8.9 Further Readings

8.0 INTRODUCTION

The examples we have seen so far in the previous units deal with standard input and output. When data is stored using variables, the data is lost when the program exits unless something is done to save it. This unit discusses methods of working with files, and a data structure to store data. C views file simply as a sequential stream of bytes. Each file ends either with an *end-of-file* marker or at a specified byte number recorded in a system maintained, administrative data structure. C supports two types of files called *binary files* and *text files*.

The difference between these two files is in terms of storage. In *text files*, everything is stored in terms of text *i.e.* even if we store an integer 54; it will be stored as a 3-byte string - “54\0”. In a text file certain character translations may occur. For example a *newline(\n)* character may be converted to a carriage return, linefeed pair. This is what Turbo C does. Therefore, there may not be one to one relationship between the characters that are read or written and those in the external device. A *binary file* contains data that was written in the same format used to store internally in main memory.

For example, the integer value 1245 will be stored in 2 bytes depending on the machine while it will require 5 bytes in a text file. The fact that a numeric value is in a standard length makes binary files easier to handle. No special string to numeric conversions is necessary.

The disk I/O in C is accomplished through the use of library functions. The ANSI standard, which is followed by TURBO C, defines one complete set of I/O functions. But since originally C was written for the UNIX operating system, UNIX standard defines a second system of routines that handles I/O operations. The first method, defined by both standards, is called a buffered file system. The second is the unbuffered file system.

In this unit, we will first discuss buffered file functions and then the unbuffered file functions in the following sections.

8.1 OBJECTIVES

After going through this unit you will be able to:

- define the concept of file pointer and file storage in C;
- create text and binary files in C;
- read and write from text and binary files;
- deal with large set of Data such as File of Records; and
- perform operations on files such as count number of words in a file, search a word in a file, compare two files etc.

8.2 FILE HANDLING IN C USING FILE POINTERS

As already mentioned in the above section, a sequential stream of bytes ending with an *end-of-file* marker is what is called a *file*. When the file is opened the stream is associated with the file. By default, three files and their streams are automatically opened when program execution begins - the *standard input*, *standard output*, and the *standard error*. Streams provide communication channels between files and programs.

For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables to write data on the screen. Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) that contains information, such as size, current file pointer position, type of file etc., to perform operations on the file. This structure also contains an integer called a *file descriptor* which is an index into the table maintained by the operating system namely, the *open file table*. Each element of this table contains a block called *file control block (FCB)* used by the operating system to administer a particular file.

The standard input, standard output and the standard error are manipulated using file pointers *stdin*, *stdout* and *stderr*. The set of functions which we are now going to discuss come under the category of buffered file system. This file system is referred to as buffered because, the routines maintain all the disk buffers required for reading / writing automatically.

To access any file, we need to declare a pointer to FILE structure and then associate it with the particular file. This pointer is referred as a *file pointer* and it is declared as follows:

```
FILE *fp;
```

8.2.1 Open A File Using The Function *fopen()*

Once a file pointer variables has been declared, the next step is to open a file. The *fopen()* function opens a stream for use and links a file with that stream. This function returns a file pointer, described in the previous section. The syntax is as follows:

```
FILE *fopen(char *filename, *mode);
```

where *mode* is a string, containing the desired open status. The filename must be a string of characters that provide a valid file name for the operating system and may include a path specification. The legal mode strings are shown below in the table 12.1:

Table 8.1: Legal values to the *fopen()* mode parameter

MODE	MEANING
“r” / “rt”	opens a text file for read only access
“w” / “wt”	creates a text file for write only access
“a” / “at”	text file for appending to a file
“r+t”	open a text file for read and write access
“w+t”	creates a text file for read and write access,
“a+t”	opens or creates a text file and read access
“rb”	opens a binary file for read only access
“wb”	create a binary file for write only access
“ab”	binary file for appending to a file
“r+b”	opens a binary or read and write access
“w+b”	creates a binary or read and write access,
“a+b”	open or binary file and read access

The following code fragment explains how to open a file for reading.

Code Fragment 1

```
#include <stdio.h>

main()
{
    FILE *fp;
    if ((fp=fopen("file1.dat", "r"))==NULL)
    {
        printf("FILE DOES NOT EXIST\n");
        exit(0);
    }
}
```

The value returned by the *fopen()* function is a file pointer. If any error occurs while opening the file, the value of this pointer is *NULL*, a constant declared in *<stdio.h>*. Always check for this possibility as shown in the above example.

8.2.2 Close A File Using The Function *Fclose()*

When the processing of the file is finished, the file should be closed using the *fclose()* function, whose syntax is:

```
int fclose(FILE *fptr);
```

This function flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, and then closes the stream. The return value is 0 if the file is closed successfully or a constant *EOF*, an end-of file marker, if an error occurred. This constant is also defined in *<stdio.h>*. If the function *fclose()* is not called explicitly, the operating system normally will close the file when the program execution terminates.

The following code fragment explains how to close a file.

Code Fragment 2

```
# include <stdio.h>
main()
{
FILE *fp;
if ((fp=fopen("file1.dat", "r"))==NULL)
{
    printf("FILE DOES NOT EXIST\n");
    exit(0);
}
.....
.....
.....
.....
/* close the file */
fclose(fp);
}
```

Once the file is closed, it cannot be used further. If required it can be opened in same or another mode.

Check Your Progress 1

1. How does fopen() function links a file to a stream?

.....
.....
.....

2. Differentiate between text files and binary files.

.....
.....
.....

3. What is EOF and what is its value?

.....
.....
.....

8.3 INPUT AND OUTPUT USING FILE POINTERS

After opening the file, the next thing needed is the way to read or write the file. There are several functions and macros defined in `<stdio.h>` header file for reading and writing the file. These functions can be categorized according to the form and type of data read or written on to a file. These functions are classified as:

- Character input/output functions
- String input/output functions
- Formatted input/output functions

- Block input/output functions.

8.3.1 Character Input and Output in Files

ANSI C provides a set of functions for reading and writing character by character or one byte at a time. These functions are defined in the standard library. They are listed and described below:

- `getc()`
- `putc()`

`getc()` is used to read a character from a file and `putc()` is used to write a character to a file. Their syntax is as follows:

```
int putc(int ch, FILE *stream);
int getc(FILE *stream);
```

The file pointer indicates the file to read from or write to. The character **ch** is formally called an integer in `putc()` function but only the low order byte is used. On success `putc()` returns a character(in integer form) written or EOF on failure. Similarly `getc()` returns an integer but only the low order byte is used. It returns *EOF* when end-of-file is reached. `getc()` and `putc()` are defined in `<stdio.h>` as macros not functions.

fgetc() and fputc()

Apart from the above two macros, C also defines equivalent functions to read / write characters from / to a file. These are:

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

To check the end of file, C includes the function `feof()` whose prototype is:

```
int feof(FILE *fp);
```

It returns **1** if end of file has been reached or **0** if not. The following code fragment explains the use of these functions.

Example 8.1

Write a program to copy one file to another.

```
/*Program to copy one file to another */

#include <stdio.h>
main()
{
    FILE *fp1;
    FILE *fp2;
    int ch;
    if((fp1=fopen("f1.dat","r")) == NULL)

    {
        printf("Error opening input file\n");
        exit(0);
    }
    if((fp2=fopen("f2.dat","w")) == NULL)
{
```

```
printf("Error opening output file\n");
      exit(0);
}

while (!feof(fp1))
{
    ch=getc(fp1);
    putc(ch,fp2);
}
fclose(fp1);
fclose(fp2);
}
```

OUTPUT

If the file "f1.dat" is not present, then the output would be:

Error opening input file

If the disk is full, then the output would be:

Error opening output file

If there is no error, then "f2.dat" would contain whatever is present in "f1.dat" after the execution of the program, if "f2.dat" was not empty earlier, then its contents would be overwritten.

8.3.2 String Input/Output Functions

If we want to read a whole line in the file then each time we will need to call character input function, instead C provides some string input/output functions with the help of which we can read/write a set of characters at one time. These are defined in the standard library and are discussed below:

- *fgets()*
- *fputs()*

These functions are used to read and write strings. Their syntax is:

```
int fputs(char *str, FILE *stream);
char *fgets(char *str, int num, FILE *stream);
```

The integer parameter in *fgets()* is used to indicate that at most num-1 characters are to be read, terminating at end-of-file or end-of-line. The end-of-line character will be placed in the string *str* before the string terminator, if it is read. If end-of-file is encountered as the first character, EOF is returned, otherwise str is returned. The *fputs()* function returns a non-negative number or EOF if unsuccessful.

Example 8.2

Write a program read a file and count the number of lines in the file, assuming that a line can contain at most 80 characters.

```
/*Program to read a file and count the number of lines in the file */
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
    FILE *fp;
```

```

int cnt=0;
char str[80];

/* open a file in read mode */

if ((fp=fopen("lines.dat","r"))==NULL)
{   printf("File does not exist\n");
    exit(0);
}
/* read the file till end of file is encountered */
while(!(feof(fp)))
{ fgets(str,80,fp); /*reads at most 80 characters in str */
    cnt++;           /* increment the counter after reading a line */
}
/* print the number of lines */
printf("The number of lines in the file is :%d\n",cnt);
fclose(fp);
}

```

OUTPUT

Let us assume that the contents of the file “*lines.dat*” are as follows:

This is C programming.
I love C programming.

To be a good programmer one should have a good logic. This is a must.
C is a procedural programming language.

After the execution the output would be:

The number of lines in the file is: 4

8.3.3 Formatted Input/Output Functions

If the file contains data in the form of digits, real numbers, characters and strings, then character input/output functions are not enough as the values would be read in the form of characters. Also if we want to write data in some specific format to a file, then it is not possible with the above described functions. Hence C provides a set of formatted input/output functions. These are defined in standard library and are discussed below:

fscanf() and *fprintf()*

These functions are used for formatted input and output. These are identical to *scanf()* and *printf()* except that the first argument is a file pointer that specifies the file to be read or written, the second argument is the format string. The syntax for these functions is:

```

int fscanf(FILE *fp, char *format, . . .);
int fprintf(FILE *fp, char *format, . . .);

```

Both these functions return an integer indicating the number of bytes actually read or written.

Example 8.3

Write a program to read formatted data (account number, name and balance) from a file and print the information of clients with zero balance, in formatted manner on the screen.

```
/* Program to read formatted data from a file */

#include<stdio.h>
main()
{
    int account;
    char name[30];
    double bal;
    FILE *fp;

    if((fp=fopen("bank.dat","r"))== NULL)
        printf("FILE not present \n");
    else
        do{
            fscanf(fp,"%d%s%lf",&account,name,&bal);
            if(!feof(fp))
            {
                if(bal==0)
                    printf("%d %s %lf\n",account,name,bal);
            }
        }while(!feof(fp));
}
```

OUTPUT

This program opens a file “**bank.dat**” in the read mode if it exists, reads the records and prints the information (account number, name and balance) of the zero balance records.

Let the file be as follows:

```
101    nuj    1200
102    Raman  1500
103    Swathi 0
104    Ajay   1600
105    Udit   0
```

The output would be as follows:

```
103    Swathi 0
105    Udit   0
```

8.3.4 Block Input/Output Functions

Block Input / Output functions read/write a block (specific number of bytes from/to a file. A block can be a record, a set of records or an array. These functions are also defined in standard library and are described below.

- *fread()*
- *fwrite()*

These two functions allow reading and writing of blocks of data. Their syntax is:

File Handling

```
int fread(void *buf, int num_bytes, int count, FILE *fp);  
int fwrite(void *buf, int num_bytes, int count, FILE *fp);
```

In case of *fread()*, buf is the pointer to a memory area that receives the data from the file and in *fwrite()*, it is the pointer to the information to be written to the file. *num_bytes* specifies the number of bytes to be read or written. These functions are quite helpful in case of binary files. Generally these functions are used to read or write array of records from or to a file. The use of the above functions is shown in the following program.

Example 8.4

Write a program using *fread()* and *fwrite()* to create a file of records and then read and print the same file.

```
/* Program to illustrate the fread() and fwrite() functions*/  
#include<stdio.h>  
#include<conio.h>  
#include<process.h>  
#include<string.h>  
  
void main()  
{  
    struct stud  
    {  
        char name[30];  
        int age;  
        int roll_no;  
    }s[30],st;  
    int i;  
    FILE *fp;  
  
/*opening the file in write mode*/  
    if((fp=fopen("sud.dat","w"))== NULL)  
    {  printf("Error while creating a file\n");  
       exit(0);  }  
  
/* reading an array of students */  
    for(i=0;i<30;i++)  
        scanf("%s %d %d",s[i].name,s[i].age,s[i].roll_no);  
  
/* writing to a file*/  
    fwrite(s,sizeof(struct stud),30,fp);  
    fclose(fp);  
  
/* opening a file in read mode */  
    fp=fopen("stud.dat","r");  
  
/* reading from a file and writing on the screen */  
    while(!feof(fp))  
    {  
        fread(&st,sizeof(struct stud),1,fp);  
        fprintf("%s %d %d",st.name,st.age,st.roll_no);  
    }  
    fclose(fp);    }
```

OUTPUT

This program reads 30 records (name, age and roll_number) from the user, writes one record at a time to a file. The file is closed and then reopened in read mode; the records are again read from the file and written on to the screen.

Check Your Progress 2

1. Give the output of the following code fragment:

```
#include<stdio.h>
#include<process.h>
#include<conio.h>
main()
{
FILE * fp1, * fp2;
double a,b,c;

fp1=fopen("file1", "w");
fp2=fopen("file2", "w");

fprintf(fp1,"1 5.34 -4E02");
fprintf(fp2,"-2\n1.245\n3.234e02\n");
fclose(fp1);
fclose(fp2);

fp1=fopen("file1", "r");
fp2=fopen("file2", "r");

fscanf(fp1,"%lf %lf %lf",&a,&b,&c);
printf("%10lf %10lf %10lf",a,b,c);
fscanf(fp2,"%lf %lf %lf",&a,&b,&c);
printf("%10.1e %10lf %10lf",a,b,c);

fclose(fp1);
fclose(fp2);
}
```

2. What is the advantage of using fread/fwrite functions?
-
.....
.....

3. _____ and _____ functions are used for formatted input and output from a file.
-
.....
.....

8.4 SEQUENTIAL Vs RANDOM ACCESS FILES

We have seen in section 12.0 that C supports two type of files – text and binary files, also two types of file systems – buffered and unbuffered file system. We can also differentiate in terms of the type of file access as Sequential access files and random access files. Sequential access files allow reading the data from the file in sequential manner which means that data can only be read in sequence. All the above examples that we have considered till now in this unit are performing sequential access. Random access files allow reading data from any location in the file. To achieve this purpose, C defines a set of functions to manipulate the position of the file pointer. We will discuss these functions in the following sections.

8.5 POSITIONING THE FILE POINTER

To support random access files, C requires a function with the help of which the file pointer can be positioned at any random location in the file. Such a function defined in the standard library is discussed below:

The function *fseek()* is used to set the file position. Its prototype is:

```
int fseek(FILE *fp, long offset, int pos);
```

The first argument is the pointer to a file. The second argument is the number of bytes to move the file pointer, counting from zero. This argument can be positive, negative or zero depending on the desired movement. The third parameter is a flag indicating from where in the file to compute the offset. It can have three values:

SEEK_SET(or value 0)	the beginning of the file,
SEEK_CUR(or value 1)	the current position and
SEEK_END(or value 2)	the end of the file

These three constants are defined in *<stdio.h>*. If successful *fseek()* returns zero.

Another function *rewind()* is used to reset the file position to the beginning of the file. Its prototype is:

```
void rewind(FILE *fp);
```

A call to *rewind* is equivalent to the call

```
fseek(fp,0,SEEK_SET);
```

Another function *ftell()* is used to tell the position of the file pointer. Its prototype is:

```
long ftell(FILE *fp);
```

It returns -1 on error and the position of the file pointer if successful.

Example 8.5

Write a program to search a record in an already created file and update it. Use the same file as created in the previous example.

```
/*Program to search a record in an already created file*/
```

```
#include<stdio.h>
```

```
#include<conio.h>
#include<stdio.h>
#include<process.h>
#include<string.h>
void main()
{
    int r,found;
    struct stud
    {
        char name[30];
        int age;
        int roll_no;
    }st;
    FILE *fp;
    /* open the file in read/write mode */

    if((fp=fopen("f1.dat","r+b"))==NULL)
    { printf("Error while opening the file \n");
        exit(0);
    }

    /* Get the roll_no of the student */
    printf("Enter the roll_no of the record to be updated\n");
    found=0;
    scanf("%d",&r);

    /* check in the file for the existence of the roll_no */
    while((!feof(fp)) && !(found))
    { fread(&st,sizeof(stud),1,fp);
        if(st.roll_no == r)

    /* if roll_no is found then move one record backward to update it */
        { fseek(fp,- sizeof(stud),SEEK_CUR);
            printf("Enter the new name\n");
            scanf("%s",st.name);
            fwrite(fp,sizeof(stud),1,fp);
            found=1;
        }
    }
    if (!found)
        printf("Record not present\n");
    fclose(fp);
}
```

OUTPUT

Let the input file be as follows:

Geeta	18	101
Leena	17	102
Mahesh	23	103
Lokesh	21	104
Amit	19	105

Let the roll_no of the record to be updated be 106. Now since this roll_no is not present the output would be:

Record not present

If the roll_no to be searched is 103, then if the new name is Sham, the output would be the file with the contents:

Geeta	18	101
Leena	17	102
Sham	23	103
Lokesh	21	104
Amit	19	105

8.6 THE UNBUFFERED I/O – THE UNIX LIKE FILE ROUTINES

The buffered I/O system uses buffered input and output, that is, the operating system handles the details of data retrieval and storage, the system stores data temporarily (buffers it) in order to optimize file system access. The buffered I/O functions are handled directly as system calls without buffering by the operating system. That is why they are also known as low level functions. This is referred to as unbuffered I/O system because the programmer must provide and maintain all disk buffers, the routines do not do it automatically.

The low level functions are defined in the header file `<io.h>`.

These functions do not use file pointer of type FILE to access a particular file, but they use directly the file descriptors, as explained earlier, of type integer. They are also called *handles*.

Opening and closing of files

The function used to open a file is `open()`. Its prototype is:

`int open(char *filename, int mode, int access);`

Here `mode` indicates one of the following macros defined in `<fcntl.h>`.

Mode:

<code>O_RDONLY</code>	Read only
<code>O_WRONLY</code>	Write only
<code>O_RDWR</code>	Read / Write

The `access` parameter is used in UNIX environment for providing the access to particular users and is just included here for compatibility and can be set to zero. `open()` function returns `-1` on failure. It is used as:

Code fragment 2

```
int fd;  
  
if ((fd=open(filename,mode,0)) == -1)  
{   printf("cannot open file\n");  
    exit(1); }
```

If the file does not exist, `open()` the function will not create it. For this, the function `creat()` is used which will create new files and re-write old ones. The prototype is:

`int creat(char *filename, int access);`

It returns a file descriptor; if successful else it returns -1 . It is not an error to create an already existing file, the function will just truncate its length to zero. The *access* parameter is used to provide permissions to the users in the UNIX environment. The function *close()* is used to close a file. The prototype is:

```
int close(int fd);
```

It returns zero if successful and -1 if not.

Reading, Writing and Positioning in File

The functions *read()* and *write()* are used to read from and write to a file. Their prototypes are:

```
int read(int fd, void *buf, int size);
int write(int fd, void *buf, int size);
```

The first parameter is the file descriptor returned by *open()*, the second parameter holds the data which must be typecast to the format needed by the program, the third parameter indicates the number of bytes to transferred. The return value tells how many bytes are actually transferred. If this value is -1 , then an error must have occurred.

Example 8.6

Write a program to copy one file to another to illustrate the use of the above functions. The program should expect two command line arguments indicating the name of the file to be copied and the name of the file to be created.

```
/* Program to copy one file to another file to illustrate the functions*/
# include<stdio.h>
# include<io.h>
# include<process.h>

typedef char arr[80];
typedef char name[30];

main()
{
    arr buf;
    name fname, sname;
    int fd1,fd2,size;

    /* check for the command line arguments */
    if (argc!=3)
    {
        printf("Invalid number of arguments\n");
        exit(0);
    }
    if ((fd1=open(argv[1],O_RDONLY))<0)
    {
        printf("Error in opening file %s \n",argv[1]);
        exit(0);
    }
    if ((fd2=creat(argv[2],0))<0)
    {
        printf("Error in creating file %s \n",argv[2]);
        exit(0);
    }

    open(argv[2],O_WRONLY);
    size=read(fd1,buf,80); /* read till end of file */
```

```

while (size>0)
    { write(fd2,buf,80);
      size=read(fd1,buf,80);
    }
    close(fd1);
    close(fd2);
}

```

OUTPUT

If the number of arguments given on the command line is not correct then output would be:

Invalid number of arguments

One file is opened in the read mode, and another file is opened in the write mode. The output would be as follows if the file to be read is not present (let the file be *f1.dat*):

Error in opening file *f1.dat*

The output would be as follows if the disk is full and the file cannot be created (let the output file be *f2.dat*):

Error in creating file *f2.dat*

If there is no error contents of *f1.dat* will be copied to *f2.dat*.

lseek()

The function *lseek()* is provided to move to the specific position in a file. Its prototype is:

long lseek(int fd, long offset, int pos);

This function is exactly the same as *fseek()* except that the file descriptor is used instead of the file pointer.

Using the above defined functions, it is possible to write any kind of program dealing with files.

Check Your Progress 3

1. Random access is possible in C files using function _____.
 2. Write a proper C statement with proper arguments that would be called to move the file pointer back by 2 bytes.
-
.....

3. Indicate the header files needed to use unbuffered I/O.
-
.....
.....

8.7 SUMMARY

In this unit, we have learnt about files and how C handles them. We have discussed the buffered as well as unbuffered file systems. The available functions in the standard library have been discussed. This unit provided you an ample set of programs to start with. We have also tried to differentiate between sequential access as well as random access file. The file pointers assigned to standard input, standard output and standard error are *stdin*, *stdout*, and *stderr* respectively. The unit clearly explains the different type of modes of opening the file. As seen there are several functions available to read/write from the file. The usage of a particular function depends on the application. After reading this unit one must be able to handle large data bases in the form of files.

8.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1. *fopen()* function links a file to a stream by returning a pointer to a FILE structure defined in *<stdio.h>*. This structure contains an index called file descriptor to a File Control Block, which is maintained by the operating system for administrative purposes.
2. Text files and binary files differ in terms of storage. In text files everything is stored in terms of text while binary files stores exact memory image of the data i.e. in text files 154 would take 3 bytes of storage while in binary files it will take 2 bytes as required by an integer.
3. *EOF* is an end-of-file marker. It is a macro defined in *<stdio.h>*. Its value is -1.

Check Your progress 2

1. The output would be:

1.000000 5.340000 -400.000000 -2.0e+00 1.245000 323.400000

2. The advantage of using these functions is that they are used for block read/write, which means we can read or write a large set of data at one time thus increasing the speed.
3. *fscanf()* and *fprintf()* functions are used for formatted input and output from a file.

Check Your progress 3

1. Random access is possible in C files using function *fseek()*.
2. *fseek(fp, -2L, SEEK_END);*
3. *<io.h>* and *<fcntl.h>*

8.9 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication, 2002.
2. C How to Program, *Deitel & Deitel*, Pearson Education, 2002.
3. Practical C Programming, *Steve Oualline*, O'Reilly Publication, 2003.

APPENDIX-A

THE ASCII SET

The ASCII (American Standard Code for Information Interchange) character set defines 128 characters (0 to 127 decimal, 0 to FF hexadecimal, and 0 to 177 octal). This character set is a subset of many other character sets with 256 characters, including the ANSI character set of MS Windows, the Roman-8 character set of HP systems, and the IBM PC Extended Character Set of DOS, and the ISO Latin-1 character set used by Web browsers. They are not the same as the EBCDIC character set used on IBM mainframes. The first 32 values are non-printing **control characters**, such as *Return* and *Line feed*. You generate these characters on the keyboard by holding down the Control key while you strike another key. For example, Bell is value 7, Control plus G, often shown in documents as ^G. Notice that 7 is 64 less than the value of G (71); the Control key subtracts 64 from the value of the keys that it modifies. The table shown below gives the list of the control and printing characters.

The Control Characters

Char	Oct	Dec	Hex	Control-Key	Control Action
NUL	0	0	0	^@	Null character
SOH	1	1	1	^A	Start of heading, = console interrupt
STX	2	2	2	^B	Start of text, maintenance mode on HP console
ETX	3	3	3	^C	End of text
EOT	4	4	4	^D	End of transmission, not the same as ETB
ENQ	5	5	5	^E	Enquiry, goes with ACK; old HP flow control
ACK	6	6	6	^F	Acknowledge, clears ENQ logon hand
BEL	7	7	7	^G	Bell, rings the bell...
BS	10	8	8	^H	Backspace, works on HP terminals/computers
HT	11	9	9	^I	Horizontal tab, move to next tab stop
LF	12	10	a	^J	Line Feed
VT	13	11	b	^K	Vertical tab
FF	14	12	c	^L	Form Feed, page eject
CR	15	13	d	^M	Carriage Return
SO	16	14	e	^N	Shift Out, alternate character set
SI	17	15	f	^O	Shift In, resume defaultn character set
DLE	20	16	10	^P	Data link escape
DC1	21	17	11	^Q	XON, with XOFF to pause listings; ":okay to send".
DC2	22	18	12	^R	Device control 2, block-mode flow control
DC3	23	19	13	^S	XOFF, with XON is TERM=18 flow control
DC4	24	20	14	^T	Device control 4
NAK	25	21	15	^U	Negative acknowledge
SYN	26	22	16	^V	Synchronous idle
ETB	27	23	17	^W	End transmission block, not the same as EOT
CAN	30	24	17	^X	Cancel line, MPE echoes !!!
EM	31	25	19	^Y	End of medium, Control-Y interrupt
SUB	32	26	1a	^Z	Substitute
ESC	33	27	1b	^[\	Escape, next character is not echoed
FS	34	28	1c	^\\	File separator

GS	35	29	1d	^]	Group separator
RS	36	30	1e	^^	Record separator, block-mode terminator
US	37	31	1f	^_	Unit separator

Printing Characters

Char	Octal	Dec	Hex	Description
SP	40	32	20	Space
!	41	33	21	Exclamation mark
"	42	34	22	Quotation mark (" in HTML)
#	43	35	23	Cross hatch (number sign)
\$	44	36	24	Dollar sign
%	45	37	25	Percent sign
&	46	38	26	Ampersand
'	47	39	27	Closing single quote (apostrophe)
(50	40	28	Opening parentheses
)	51	41	29	Closing parentheses
*	52	42	2a	Asterisk (star, multiply)
+	53	43	2b	Plus
,	54	44	2c	Comma
-	55	45	2d	Hyphen, dash, minus
.	56	46	2e	Period
/	57	47	2f	Slant (forward slash, divide)
0	60	48	30	Zero
1	61	49	31	One
2	62	50	32	Two
3	63	51	33	Three
4	64	52	34	Four
5	65	53	35	Five
6	66	54	36	Six
7	67	55	37	Seven
8	70	56	38	Eight
9	71	57	39	Nine
:	72	58	3a	Colon
;	73	59	3b	Semicolon
<	74	60	3c	Less than sign (< in HTML)
=	75	61	3d	Equals sign
>	76	62	3e	Greater than sign (> in HTML)
?	77	63	3f	Question mark
@	100	64	40	At-sign
A	101	65	41	Uppercase A
B	102	66	42	Uppercase B
C	103	67	43	Uppercase C
D	104	68	44	Uppercase D
E	105	69	45	Uppercase E
F	106	70	46	Uppercase F
G	107	71	47	Uppercase G
H	110	72	48	Uppercase H
I	111	73	49	Uppercase I
J	112	74	4a	Uppercase J
K	113	75	4b	Uppercase K

L	114	76	4c	Uppercase L
M	115	77	4d	Uppercase M
N	116	78	4e	Uppercase N
O	117	79	4f	Uppercase O
P	120	80	50	Uppercase P
Q	121	81	51	Uppercase Q
R	122	82	52	Uppercase R
S	123	83	53	Uppercase S
T	124	84	54	Uppercase T
U	125	85	55	Uppercase U
V	126	86	56	Uppercase V
W	127	87	57	Uppercase W
X	130	88	58	Uppercase X
Y	131	89	59	Uppercase Y
Z	132	90	5a	Uppercase Z
[133	91	5b	Opening square bracket
\	134	92	5c	Reverse slant (Backslash)
]	135	93	5d	Closing square bracket
^	136	94	5e	Caret (Circumflex)
_	137	95	5f	Underscore
`	140	96	60	Opening single quote
a	141	97	61	Lowercase a
b	142	98	62	Lowercase b
c	143	99	63	Lowercase c
d	144	100	64	Lowercase d
e	145	101	65	Lowercase e
f	146	102	66	Lowercase f
g	147	103	67	Lowercase g
h	150	104	68	Lowercase h
i	151	105	69	Lowercase i
j	152	106	6a	Lowercase j
k	153	107	6b	Lowercase k
l	154	108	6c	Lowercase l
m	155	109	6d	Lowercase m
n	156	110	6e	Lowercase n
o	157	111	6f	Lowercase o
p	160	112	70	Lowercase p
q	161	113	71	Lowercase q
r	162	114	72	Lowercase r
s	163	115	73	Lowercase s
t	164	116	74	Lowercase t
u	165	117	75	Lowercase u
v	166	118	76	Lowercase v
w	167	119	77	Lowercase w
x	170	120	78	Lowercase x
y	171	121	79	Lowercase y
z	172	122	7a	Lowercase z
{	173	123	7b	Opening curly brace
	174	124	7c	Vertical line

ignou
THE PEOPLE'S
UNIVERSITY

{	175	125	7d	Closing curly brace
~	176	126	7e	Tilde (approximate)
DEL	177	127	7f	Delete (rubout), cross-hatch box



UNIT 9 INTRODUCTION TO PYTHON

Structure

- 9.0 Introduction
- 9.1 Objectives
- 9.2 History of Python
- 9.3 Need of Python
- 9.4 Packages for Cross platform application of Python
- 9.5 Getting started with Python
- 9.6 Program structure in python
- 9.7 Running the First program
- 9.8 Summary

9.0 INTRODUCTION

Python programming is widely used in Artificial Intelligence, Machine Learning, Neural Networks and many other advanced fields of Computer Science. Ideally, It is designed for rapid prototyping of complex applications. Python has interfaces with various Operating system calls and libraries, which are extensible to C, C++ or Java. Many large companies like NASA, Google, YouTube, Bit Torrent, etc. uses the Python programming language for the execution of their valuable projects.

To build the carrier path the skill of programming can be a fun and profitable way, but before starting the learning of this skill, one should be clear about the choice of programming language. Before learning any programming language, one should figure out which language suits best to the learner. As in our case the comparison of C and Python programming languages may help the learners to analyze and generate a lot of opinions about their choice of programming language. In this unit, I have tried to compile a few of them to give you a clear picture.

Metrics	Python	C
Introduction	It is a high-level, general-purpose & interpreted programming language.	C is general-purpose procedural programming language.
Speed	Being Interpreted programming language its execution speed is slower then that of the compiled programming language (i.e. C).	Being compiled programming language its execution speed is faster then that of the interpreted programming language (i.e. Python).
Usage	Number of lines of code written in Python is quite less in comparison to C	Program syntax of C is quite complicated in comparison to Python.

<i>Declaration of variables</i>	Declaration of Variable type is not required, they are un-typed and a given variable can be stuck different types of values at different instances during the execution of any python program.	In C, declaration of variable type is must, and it is done at the time of its creation, and only values of that declared type must be assigned to the variables.
<i>Error Debugging</i>	Error debugging is simple, as it takes only one instruction at a time. Compilation and execution is performed simultaneously. Errors are instantly shown, and execution stops at that instruction only.	Being compiler dependent language, error debugging is difficult in C i.e. it takes the entire source code, compiles it and finally all errors are shown.
<i>Function renaming mechanism</i>	the same function can be used by two different names i.e. it supports the mechanism of function renaming	Function renaming mechanism is not supported by C i.e. the same function cannot be used by two different names.
<i>Complexity</i>	Syntax of Python programs is Quite simple, easy to read, write and learn	The syntax of a C program is comparatively harder than the syntax of Python.
<i>Memory-management</i>	It supports automatic garbage collection for memory management.	In C, the Programmer has to explicitly do the memory management.
<i>Applications</i>	Python is a General-Purpose programming language can be used for Microcontrollers also.	C is generally used for hardware related applications.
<i>Built-in functions</i>	Library of built-in functions is quite large in Python.	Library of built-in functions is quite limited in C
<i>Implementing Data Structures</i>	Gives ease of implementing data structures with built-in insert, append functions.	Explicit implementation of functions is requited for the implementation of datastructures
<i>Pointers</i>	Pointers functionality is not available in Python.	Pointers are frequently used in C.

We learned from the comparison of C and Python, that Python is a high-level, general-purpose, interpreted programming language. It is dynamically typed and garbage-collected, and supports multiple programming paradigms like structured (particularly, procedural,) object-oriented, and functional programming, and due to its comprehensive standard library Python is often described as a "batteries included" language

In this course you are given exposure to both programming languages i.e. C and Python, based on your requirement you can choose your option to build your career in programming.

9.1 OBJECTIVES

After going through this unit you will be able to:

- Understand the need of Python as a programming language

- Describe the cross platform applications
- Understand the structure of python program
- Write and execute your first code in python

9.2 HISTORY OF PYTHON

Python was conceived in the late 1980s as a successor to the ABC language, it was created by Guido van Rossum in 1989 and its first release was in 1991. Later, Python 2.0, released in 2000, this version includes features like list comprehensions and a garbage collection system, which was capable of collecting reference cycles. Python 2 implemented the technical details of Python Enhancement Proposal(PEP), and thus simplified the code development complexities of the earlier versions. But due to some stability issues with Python 2.x versions, the development of Python 2.7 (last version in 2.x, released in 2010) will be discontinued in 2020. The developers were resolving the issues with Python 2 in parallel, and in 2008, Python 3.0 was released. It was a major revision of the language, this version i.e. Python 3 was mainly released to fix problems which exist in Python 2. As python 3 was not completely backward-compatible, thus much of the Python 2 code are not running unmodified on Python 3. To make the migration process easy in Python 3, some features of Python 3 have been backported to Python 2.x versions.

Thus, to migrate a project from python 3 to python 2.x, a lot of changes were required, which are not only related to the projects and its applications but also to the libraries that form the part of the Python environment.

You can freely download the latest version of Python from www.python.org, where various Python interpreters are available for many operating systems. A global community of programmers develops and maintains C-Python, which is an open source reference implementation. A non-profit organization, the Python Software Foundation, manages and directs resources for Python and C-Python development.

So, what is C-python? It is the default or reference implementation of the Python programming language, and as the name suggests C-python is written in C language. Some of the implementations which are based on C-Python runtime core but with extended behavior or features in some aspects are Stack-less Python and Micro-Python; Stack-less Python, relates to C-Python with an emphasis on concurrency using tasklets and channels (used by ds-python for the Nintendo DS), and Micro-Python relates to working with microcontrollers.

C-python compiles the python source code into intermediate bytecode, which is executed by the C-python virtual machine. It is to be noted that Python is dynamic programming language, but it is said to be slow, because the default C-Python implementation compiles the python source code in bytecode which is slow as compared to machine code(native code).C-Python is distributed with a large standard library written in a mixture of C and Python,

C-Python provides the highest level of compatibility with Python packages and C extension modules.

Further, Jython, IronPython and PyPy are also the implementation of Python as a programming language, they also provide a good level of compatibility with Python packages. These implementations are based on Java, .NET and Python, respectively. We will be briefly discussing about these implementations, starting with Jython.

Jython is actually the Java platform based implementation of the Python programming language, so that you can run Python programs on the Java platform. The programs written in Jython use Java classes and not the Python modules. The Jython compiler, compiles the Jython programs into Java byte code, which can then be run by Java virtual machine. Jython enables the use of Java class library functions from the Python program. In comparison to C-Python, Jython is slow, and it is not that much compatible with the libraries of CPython. However, the compatibility in Iron Python and PyPy is quite good.

IronPython is the implementation of Python written in C# (Microsoft's .NET framework). As Jython used Java Virtual Machine (JVM), IronPython uses .Net Virtual Machine i.e. Common Language Runtime. The .NET Framework and Python libraries are used efficiently by the IronPython. Further, due to the availability of Just In Time (JIT) compiler and absence of Global Interpreter Lock, the IronPython performs better for the Python programs where use threads or multiple cores is highly required.

Guido van Rossum (creator of Python) said "If you want your code to run faster, you should probably just use PyPy." —PyPy is an implementation of the Python programming language written in Python itself. The PyPy Interpreter is written in RPython, which is a subset of Python. The PyPy interpreter uses JIT compiler to compile the source code into native machine code which makes it very fast and efficient. PyPy also comes with default support for stackless mode, providing micro-threads for massive concurrency.

☛ Check Your Progress 1

1. Compare between C and Python programming languages
-
.....
.....
.....

2. Discuss different variants of python viz. C-python, Jython, IronPython, PyPy
-
.....
.....

9.3 NEED OF PYTHON

We learned from past sections 9.0 and 9.1, that Python is free and simple to learn. The primary features of Python are, that it is an high-level, Interpreted and dynamically typed programming language. This encourages the rapid development of application prototypes, makes debugging of errors easy and identifying itself as the language to code with.

Now, you might be in the position to appreciate the simplicity and capability of Python as a programming Language, it is a general purpose language and have applicability in almost every domain of software development, be it web development or Scientific or Business or any other application, you can ever think off. It has wider coverage of applications, and it is complying with the current needs of software industry, Thus it assures a promising future to the learners.

The learners will understand the need of Python, once they understand the application areas of Python. You might not be knowing that various applications like YouTube, BitTorrent, DropBox etc. uses Python to achieve their functionality. The reason behind the choice of python, for these applications is the ability of python to be compatible for cross-platform operating systems (a cross-platform application may run on any Operating System, be it Microsoft Windows or Linux, or macOS or any other.), which simplifies the development of applications. Now, its time to start our discussion for the various types of applications that can be developed by using Python, below are few, there may be many more.

- Artificial Intelligence(AI) and Machine Learning(ML) are need of the times, as they yield the most promising careers for the learners. The field of AI and ML relates to the incorporation of intelligence in to machines by the process of self-learning from the data stored in to the system, various algorithms are available for this purpose. But, to bring AI and ML into action, Python is the first choice. Why?, because various well equipped libraries like Pandas, Scikit-Learn, NumPy etc. are available, to facilitate the engineers and scientists. Learn the algorithm, use the library and you have your solution to the problem. It is that simple.
- Data Science and Data Visualization: Data Scientists is just another promising career, if you know how to extract relevant information from the data source then world is yours. But, its not easy, various algorithms and techniques are required to work with. To simplify your work, again Python emerges to help you to study the data you have, perform operations and extract the required information. Libraries such as Pandas, NumPy help you in extracting information. Further, Matplotlib and Seaborn are the data visualization libraries, which are quite helpful in plotting and visualization of data. This is how Python helps you to become a Data Scientist.
- Mobile Applications: In general people think that Android and iOS are for mobile development, but what about using Python for mobile app

development ? Historically, Python didn't have a strong story when it came to writing mobile GUI applications, and that was quite questionable, because Python is projected as one solution for a variety of applications. But, with the passage of time, lot of development in Python has happened and now we are having Python as one of the option for Mobile App development. Now, we'll take a look at two frameworks i.e. Kivy and BeeWare, as options for mobile application development with Python.

- **Embedded Applications:** We learned from earlier sections of this unit that Python is based on C. Thus, it can be used to create software for embedded applications. The Micro-Python is a software implementation of Python 3, written in C. In fact, Micro-Python is a Python compiler that runs on the hardware of micro-controller, it includes modules to provide access to the low-level hardware. One of the renowned embedded application is Raspberry Pi which uses Python for its computing. It can be used as a computer or like a simple embedded board to perform high-level computations, you can use it for IoT or Mobile applications, thus may produce your smart gadgets.
- *Web Development:* Python is frequently used for web development, the reason behind is the availability of the full-stack frameworks for web development. There are many frameworks but Django, Flask and Pyramid are quite famous, among the frameworks for web development. They include common-backend logic and libraries to integrate protocols like HTTPS,FTP,SSL etc., and also supports processing of JSON,XML, E-Mail etc.
- **Game Development :** Gaming is one of the most upcoming software industry, user can build various interactive games by using various modules of Python, like Pygame and Pyglet, also the Python libraries like PySoy are available, to develop 3D games. Some of the renowned games like Civilization-IV, Disney's Toontown Online, Vega Strike etc. are developed by using Python.

Python has a variety of applications where it can be used. No matter what field you take up, Python is rewarding. So I hope you have understood the Python Applications and what sets Python apart from every other programming language.

9.4 PACKAGES FOR CROSS PLATFORM APPLICATION OF PYTHON

Python is the most popular programming language, and it marked its presence by contributing actively to every emerging field in computer science. It has vast set of libraries for almost all fields such as Machine Learning (Numpy, Pandas, Matplotlib), Artificial intelligence (Pytorch, TensorFlow), and Game development (Pygame,Pyglet), and many more.

After going through the section 9.2 of this unit, you understood the meaning of Cross-platform applications and you are now aware of the potential of Python as a cross-platform programming language, you also came to know

that the development of different type of applications require different types of packages, libraries, modules, frameworks etc. Now you might be confused that what is the difference between these terms, are they same or different. Let's Clear your confusion first and then we will briefly discuss about the functionality of different packages, used for the development of various Python applications.

Python: Framework Vs Library Vs Package Vs Module

Framework is a collection of libraries. This is the architecture of the program.

Library is a collection of packages. (*We may understand that, Python library or framework is a pre-written program that is ready to use on common coding tasks.*)

Package is a collection of modules. It must contains an `__init__.py` file as a flag so that the python interpreter processes it as such. The `__init__.py` could be an empty file without causing issues. A package, in essence, is like a directory holding subpackages and modules. While we can create our own packages, we can also use one from the Python Package Index (PyPI) to use for our projects. To import a package, we type `import Game.Sound.load`. Or We can also import it giving it an alias: `import Game.Sound.load as load game`.

Module is a file which contains various Python functions and global variables. It is simply just `.py` extension file which has python executable code. We put similar code together in one module. This helps us modularize our code, and make it much easier to deal with. And not only that, a module grants us reusability. With a module, we don't need to write the same code again for a new project that we take up.

let's see how Module and Package differ:

- 1) A module is a file containing Python code. A package, however, is like a directory that holds sub-packages and modules.
- 2) A package must hold the file `__init__.py`. This does not apply to modules.
- 3) To import everything from a module, we use the wildcard `*`. But this does not work with packages.

We will learn more about them, as we proceed in this course, don't worry. To start with we will discuss about various Frameworks and Libraries first, you will be learning about their usage and also the usage of methods and packages, later.

TensorFlow: TensorFlow is an end-to-end python machine learning library for performing high-end numerical computations. TensorFlow can handle deep neural networks for image recognition, handwritten digit classification, recurrent neural networks, NLP (Natural Language Processing)

Keras : is a leading open-source Python library used for development of neural networks and machine learning projects. It simplifies the process of designing and development of neural networks for the beginners of machine learning. Keras also deals with convolution neural networks(CNN) and Recurrent Neural Networks (RNN), highly required in the field of Deep Learning. It includes algorithms for normalization, optimization, and activation layers. Instead of being an end-to-end Python machine learning library, Keras acts as a user-friendly, extensible interface that enhances modularity & total expressiveness.

Theano : is a library for scientific computation, it allows you to define, optimize as well as evaluate the complex mathematical expressions, which deals with multidimensional arrays. The repetitive computation of a tricky mathematical expression is the basis of several ML and AI applications. Theano quickly performs the data-intensive calculation, the rate of execution is almost hundred times faster than when executing on our CPU alone. It aims to reduce the development and execution time of ML apps, particularly in deep learning algorithms. Only one drawback of Theano in front of TensorFlow is that its syntax is quite hard for the beginners.

Scikit-learn : it is a prominent open-source library for machine learning through Python, it includes a wide range of algorithms like DBSCAN, gradient boosting, random forests, vector machines, and k-means etc., which are generally used to implement various concepts of Machine Learning i.e. Classification, Clustering, Regression etc. It can interoperate with numeric and scientific libraries of Python like NumPy and SciPy. Scikit-learn supports both supervised as well as unsupervised ML.

PyTorch : it is a production-ready library for machine-learning, supported with excellent examples, applications and use cases, also supported by a strong community. It supports GPU and CPU computations, thus provides performance optimization and scalability in research as well as production. The two high-end features of the PyTorch are Deep neural networks and Tensor computation, which are boosted because of the machine learning compiler “Glow”, specially designed to improve the performance of deep learning frameworks.

NumPy : also known as Numerical Python, a library to perform scientific computations. Almost all Python machine-learning tools like Matplotlib, SciPy, Scikit-learn, etc rely on this library to a reasonable extent. It comes with functions for dealing with complex mathematical operations like linear algebra, Fourier transformation, random number and features that work with matrices and n-arrays in Python. It is widely used in handling sound waves, images, and other binary functions.

SciPy : It is library, that works for all type of scientific programming projects. Its main functionality is built upon NumPy, and it includes modules for linear algebra, Integration, optimization, and statistics too. SciPy is supported with extensive documentation, which makes its working really easy.

Pandas : it is an open-source library of python that offers a wide range of tools for data analysis & manipulation, with Pandas, the data from a broad range of sources like CSV, SQL databases, JSON files, and Excel, can be read. The data analysis & manipulation is a pre-requisite for most of the machine learning projects, where a significant amount of time is spent to analyse the trends and patterns hidden in the datasets. Using Pandas one can manage complex data operations with just one or two commands, it comes with several inbuilt methods for data handling, and it also serves as the starting point to create more focused and powerful data tools.

Matplotlib : this library is specifically meant for Data Science, it helps to generate data visualization like 2D diagrams and graphs like histograms, scatter plots, and even graphs for non-cartesian coordinates. It is equipped with the object oriented API to embed plots into the applications, directly. It is because of this library, the Python is able to compete with scientific tools like MATLAB or Mathematica. It is quite compatible with the popular plotting libraries, but developers are required to write more code than usual, while using this library for generating advanced visualizations.

Seaborn : it is an unparalleled visualization library, based on Matplotlib's foundations. Seaborn offers high-level dataset based interface to make amazing statistical graphics. With Seaborn, it is simple to create certain types of plots like time series, heat maps, and violin plots. The functionalities of Seaborn go beyond Pandas and matplotlib with the features to perform statistical estimation at the time of combining data across observations, plotting and visualizing the suitability of statistical models to strengthen dataset patterns.

Scrapy : It is the most widely used library of python, for the purpose of data sciences, specifically data mining. It is used to build crawling programs i.e spider bots, which are used to retrieve structured data (example URLs or Contact Information) from the web. Developers use it for gathering data from APIs.

Pygame : is a python programming language library for making multimedia applications like video games, animations etc. It includes libraries related to computer graphics and sound, which are designed to be used with the Python programming language. Pygame is suitable to create client-side applications that can be potentially wrapped in a standalone executable.

Django : it a web framework build using Python, it encourages rapid development of web applications. Developers can focus on writing app only, It's primary goal is to ease the creation of complex, database-driven websites. Django emphasizes on code reusability and pluggability of components, thus it supports less code, low coupling, and rapid development of websites.

MicroPython : is a software implementation of Python 3, written in C. Infact, MicroPython is a Python compiler that runs on the hardware of micro-controller, it includes modules to provide access to the low-level hardware. One of the renowned embedded application is Raspberry Pi which uses Python for its computing. It can be used as a computer or like a simple

embedded board to perform high-level computations, you can use it for IoT or Mobile applications, thus may produce your smart gadgets.

BeeWare – is a Python GUI and mobile development framework, to develop native Python Mobile Apps. “BeeWare” project, provides a set of tools and an abstraction layer, which can be used to write native-looking mobile and desktop applications using Python.

Kivy – is an opensource Python Framework used as a Cross-platform Python GUI for Mobile App development, it allows you to write pure-Python graphical applications that run on both i.e. the main desktop platforms (Windows, Linux, macOS and Raspberry Pi) and also on iOS & Android. Both Kivy and BeeWare are worth considering. So far as maturity goes, Kivy seems to be the more mature platform right now

☛ Check Your Progress 2

1. Compare Framework, Library, Package and Modules in python
2. Identify the prominent tools of python for following:
 - a. Artificial Intelligence application
 - b. Mobile application development
 - c. Embedded programming
 - d. Web development
 - e. Game development
3. Discuss the utility of following:
 - a. Tensor Flow
 - b. Keras
 - c. Theano
 - d. Pytorch
 - e. NumPy
 - f. SciPy
 - g. Pandas
 - h. Matplotlib
 - i. Kivy
 - j. Django
 - k. Pygame

9.6 PROGRAM STRUCTURE IN PYTHON

This section relates to the discussion over the programming structure of Python programming, i.e. the way you divide a program in your source files and mix parts from various libraries or modules or packages in to a single program, i.e. directly or indirectly, a single program includes multiple

files from different modules. Modules are in fact the library tools, which are implemented to make a collection of top-level files i.e. the high level files may use tools, defined in modules, which may use files, defined in other modules. Coming to our point, in python a file takes a module to get access to the tools it defines, and also to the tools defined in other modules included in programme. In python high level file has important path of control of your program, where from you can start your application. Just refer to the figure given below

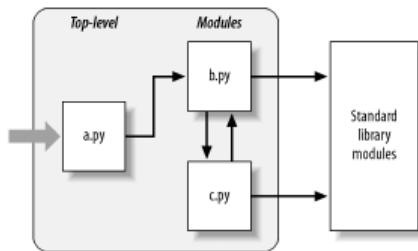


Figure : Structure Python Program

To understand the structure of Python program, say there exist three files a.py,b.py and c.py. The file model a.py is chosen for high level file . it is known as a simple text file of statements. Files b.py and c.py are modules. They are also text files of statements but they are generally not started directly, but they are invoked by a.py i.e high level file.

This is the general discussion over the structure of Python program, although Python includes all the components as they are in any other language such as **data types, conditional statements, looping constructs**, functions, file handling, Classes, Exception handling, Libraries, Modules, packages etc. We will discuss a few of them over here and the rest will be discussed in the subsequent units in this course

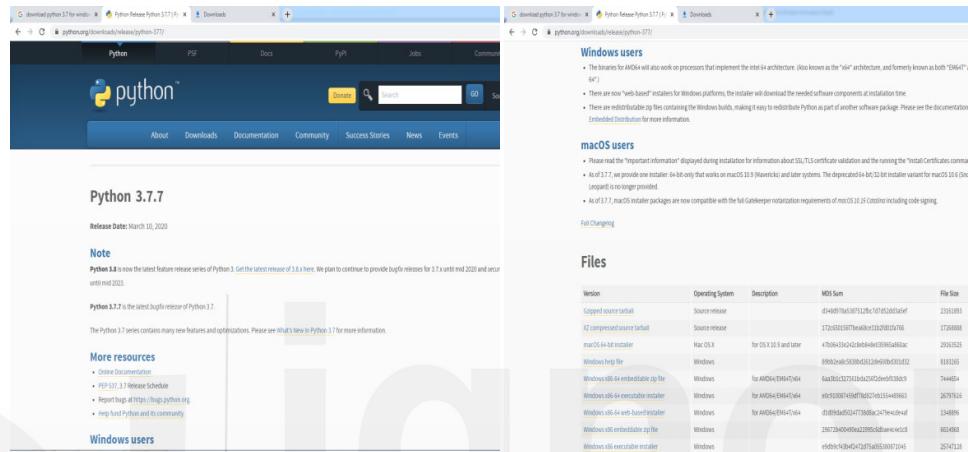
9.5 GETTING STARTED WITH PYTHON

Now, we are having a bit of clarity about the Python as a programming language, from the past sections we learned about the various libraries and frameworks of Python. Now, we need to work on the IDEs (Integrated Development Environments) of Python, there are many IDEs like Jupyter Notebook, Spyder, VS Code, R Programming etc., all are collectively available in Anaconda (Anaconda is a free and open-source distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.)), or you may also go for the cloud versions Like Google Colab Notebook, where you need not to have high configuration hardwares, only internet is required and through your gmail account you can work on Python using Jupyter notebook.

We will discuss in brief, some of the ways to work with Python, you may choose any or try all and other options too.

- 1) Just browse for <https://www.python.org> and perform following steps :

- Download the latest version of Python for the operating system installed on your computer, as in my case its windows 64 bit, so I downloaded python-3.7.7 (python-3.7.7-amd64.exe) from <https://www.python.org/downloads/release/python-377/>
- Now Run this exe file and install the Python, just click next and go ahead, till the setup installation is finished
- Finally you are having an interface for Python programming



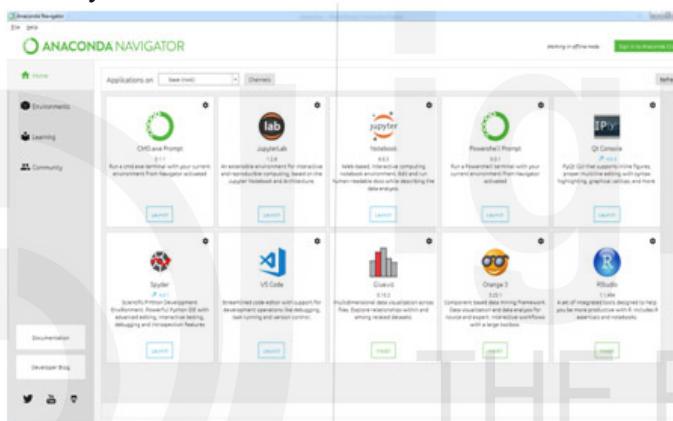
There is a variety of IDEs for python, Like Jupyter Notebook, Spyder, VS Code etc, and all are available at Anaconda (a free and open-source distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.)). To start with Anaconda Just perform following steps.

- Just browse <https://www.anaconda.com/> and perform following steps:
 - Click the Download button on the webpage of <https://www.anaconda.com/>
 - the distribution section <https://www.anaconda.com/distribution/> will open,
 - click the download option given on this page <https://www.anaconda.com/distribution/>
 - <https://www.anaconda.com/distribution/#download-section> will open, the option of 64bit and 32bit graphic installer for Python 3.x (currently 3.7) and 2.x (currently 2.7) are given under Anaconda 2020.02 for windows installer.
 - It is recommended to download 64bit version of Python 3.x (currently 3.7),
 - Anaconda3-2020.02-Windows-x86_64.exe will be downloaded.
 - Now, just run the setup of this Anaconda3-2020.02-Windows-x86_64.exe , and click next-next, till the installation is completed.
 - Finally, you will find Anaconda Navigators shortcut on your desktop, click on it and you can start working with any IDE be it

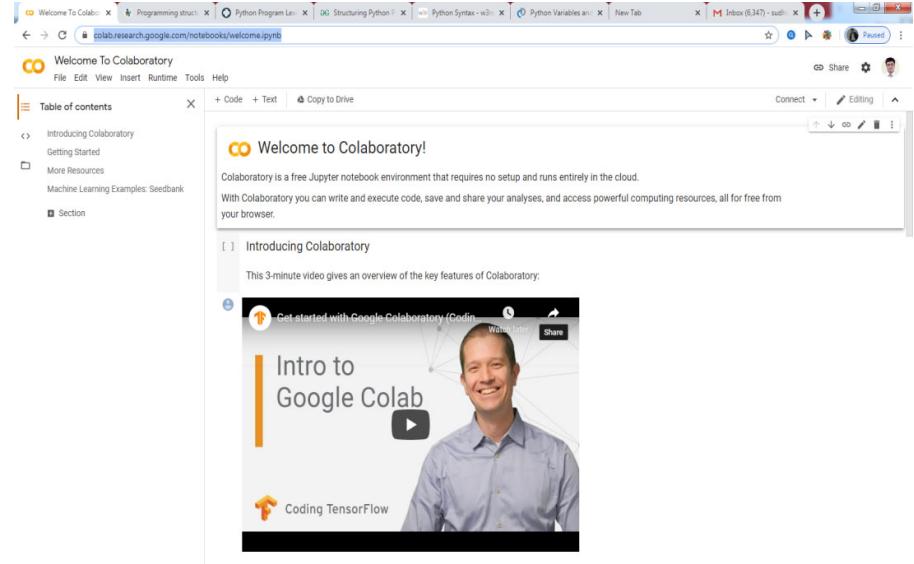
Jupyter Notebook, Spyder, VS Code etc., even you can work with R-Programming.

Important: Before working with IDEs you need to understand how to work with them and which one is suitable, following are observations, currently:

- a) When you start Jupyter Notebook on windows (by clicking on the jupyter section, given in the anaconda Navigator), a browser will open in internet explorer, many a times Jupyter Notebook won't work here, then just copy the URL from the Internet Explorer and paste it in the Google Chrome, you will find that Jupyter Notebook starts working, other details regarding the writing, saving, execution of program, will be discussed later.
- b) Program execution in Jupyter is line by line and in VS Code and Spyder it goes sideways, even errors can also be seen sideways. Any ways all are good to work with Python.

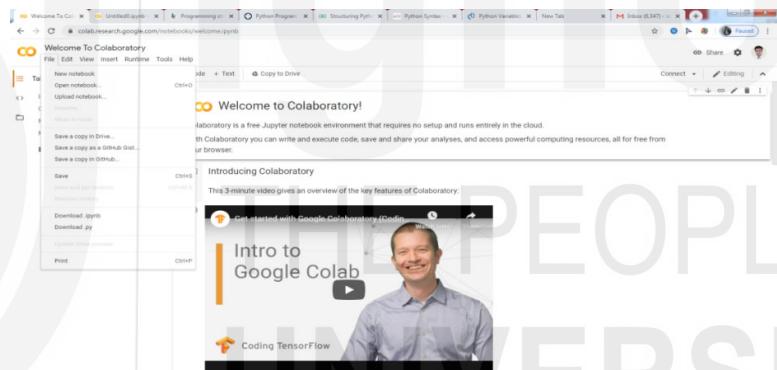


- 3) Many a times the learners may not be equipped with the systems having latest hardware configurations, as desired for the installation of Python, or their might be compatibility issues with operating system or may be due to any reason you are not able to install and start your work with Python. Under such circumstances the solution is Google Colab Notebook (<https://colab.research.google.com/notebooks/welcome.ipynb>), use this and just login with your gmail account and start your work on Jupyter Notebook, as simple as that.

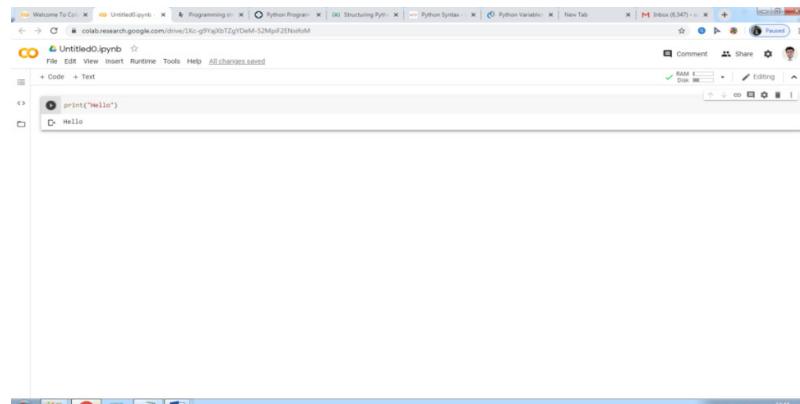


9.7 RUNNING THE FIRST PROGRAM

Just click file option and select new workbook, and new Jupyter notebook will open in Google Colab, now you may start your work



As here, I wrote my first program (`print("hello")`) to say “hello” to all of you , and executed it by simply pressing the play button, you may see just before the print command I wrote, and the output comes just beneath the statement `print("hello")`.



The conceptual fundamentals of python programming language were discussed in this unit, after going through this unit the learner will be equipped not only with the historical understanding of python, but also with the various application areas and the tools relevant to explore the concerned application area.

SOLUTION TO CHECK YOUR PROGRESS

Check your progress 1

- 1) refer to section 9.0
- 2) refer to section 9.2

Check your progress 2

- 1) refer to section 9.4
- 2) refer to section 9.3
- 3) refer to section 9.4

UNIT 10 DATA STRUCTURES AND CONTROL STATEMENTS IN PYTHON

Data Structures and
Control Statements
in Python

Structure

- 10.1 Introduction
 - 10.2 Objectives
 - 10.3 Identifiers and Keywords
 - 10.4 Statements and Expressions
 - 10.4 Variables
 - 10.5 Operators
 - 10.6 Data Types
 - 10.7 Data Structures
 - 10.8 Control Flow Statements
 - 10.9 Summary
-

10.1 INTRODUCTION

A **Python** is a general-purpose, high level programming language which is widely used by programmers. It was created by Guido Van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code. To learn how to code in any language, it is, therefore, very important to learn its basic components. This is what is the objective of this chapter. In this chapter, we will learn about the basic constituents of Python starting from its identifiers, variables, operators and also about how to combine them to form expressions and statements. We will also study about the data types available in Python along with the control flow statements.

10.2 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic building blocks of Python programming Language
 - Understand various Data Structures
 - Understand usage of control flow statements
-

10.3 IDENTIFIERS AND KEYWORDS

An **identifier** is a name given to a variable, function, class or module. Identifiers may be one or more characters in the following format:

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myIndia, other_1 and mca_course, all are valid examples. A Python identifier can begin with an uppercase or a lowercase alphabet or (_).
- An identifier cannot start with a digit but is allowed everywhere else. 1plus is invalid, but plus1 is perfectly fine.
- Keywords (listed in TABLE1) cannot be used as identifiers.
- One cannot use spaces and special symbols like !, @, #, \$, % etc. as identifiers.
- Identifier can be of any length. (However, it is preferred to keep it short and meaningful).

Examples of valid identifiers are: marksPython, Course, MCA301 etc.

Keywords are a list of reserved words that have predefined meaning to the Python interpreter. These are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name. Attempting to use a keyword as an identifier name will cause an error. As Python is case sensitive, keywords must be written exactly as given in TABLE1.

TABLE 1 :List of keywords in Python

and	async	not
assert	finally	or
break	for	pass
class	from	nonlocal
continue	global	raise
def	import	try
elif	in	while
else	is	with
except	lambda	yield
False	True	None
del	if	Return
As	await	

Check your progress - 1:

Q1. Is Python case sensitive when dealing with Identifiers?

- Yes
- No

Q2. What is the maximum possible length of an identifier?

- 32 characters
- 64 characters
- 76 characters
- None of the above

Q3. All keywords in Python are in _____

- a) lower case
- b) UPPER CASE
- c) Capitalized
- d) None of the above

10.4 STATEMENTS AND EXPRESSIONS

A **statement** is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment. When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one. A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statement executes.

For example, the script

```
Print(1)  
x = 2  
print(x)
```

produces the output

```
1  
2
```

The assignment statement produces no output.

An **expression** is an arrangement of values and operators which are evaluated to make a new value. Expressions are statements as well. A value is the representation of some entity like a letter or a number that can be manipulated by a program. A single value **25** or a single variable **x** or a combination of a variable, operator and value **z + 25** are all examples of expressions. An expression, when used in interactive mode is evaluated by the interpreter and result is displayed instantly. For example,

```
In[]: 7 + 3  
Out[]: 10
```

But in script, an expression all by itself doesn't show any output altogether. You need to explicitly print the result.

Check your progress-2 :

Q4. What is the output of the following Python expression?: $36 / 4$

- a) 9.0
- b) 9

Q5. What is the output of following statement? : $y = 3.14$

- a) 3.14
- b) y
- c) None of the above

Q6. An expression can contain:

- a) Values
- b) Variables
- c) Operators
- d) All of the above
- e) None of the above

Q7. In the Python statement $x = a + 5 - b$:

a and b are _____

$a + 5 - b$ is _____

- a) Operands, an equation
- b) Operands, an expression
- c) Terms, a group
- d) Operators, a statement

10.5 VARIABLES

A **variable** is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution. Python is a Dynamically Typed Language. There is no need to declare a variable explicitly by specifying whether the variable is an integer or a float or any other type. To define a new variable in Python, we simply assign a value to a name.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letter; but it is a good idea to begin variable names with a lowercase letter. Variable names are case-sensitive. E.g., IGNOU and Ignou are different variables.

The underscore character can appear in a name. It is often used in names with multiple words, such as `my_name` or `marks_in_maths`. Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

The general format for assigning values to variables is as follows:

`variable_name = expression`

The equal sign (=) also known as simple assignment operator is used to assign values to variables. In the general format, the operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the expression which can be a value or any code snippet that results in a value. That value is stored in the variable on the execution of the assignment statement. Assignment operator should not be confused with the = used in algebra to denote equality. E.g.,

```
In[]: number = 100           #integer type value is assigned to a  
variable number
```

```
In[]: name = "Python"       #string type value is assigned to  
variable name
```

In Python, not only the value of a variable may change during program execution but also the type of data that is assigned. You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the variable. A new assignment overrides any previous assignments.

```
In[]: year = 2020           #an integer value is assigned to year  
variable
```

```
In[]: year
```

```
Out[]: 2020
```

```
In[]:year ="twenty twenty" #then an string value is assigned to year  
variable
```

```
In[]: year
```

```
Out[]: 'twenty twenty'
```

Python allows you to assign a single value to several variables simultaneously. E.g.

```
In[]: x = y = z = 5          #an integer value is assigned to  
variables x, y, z simultaneously.
```

Check your progress -3 :

Q8. Which of the following is a valid variable name in Python?

- a) do it
- b) do+1
- c) 1do
- d) All of the above
- e) None of the above

Q9. What is the value of 'a' here?

a = 25

b = 35

a = b

- a) 25

- b) 35

- c) It will print error
- d) None of the above

Q10. Which of the following is true for variable names in Python?

- a) unlimited length
- b) variable names are not case sensitive
- c) underscore and ampersand are the only two special characters allowed
- d) none of the mentioned

Q11. What error occurs when you execute the following Python Code snippet?

```
apple = mango
```

- a) SyntaxError
- b) NameError
- c) ValueError
- d) TypeError

10.6 OPERATORS

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands. Python language supports a wide range of operators. They are:

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators

TABLE 2 : List of Arithmetic Operators

Operator	Operator Name	Description	Example (try in lab) n1 = 5, n2 = 6
+	Addition operator	Adds two operands, producing their sum	In[]: n1 + n2 Out[]: 11
-	Subtraction operator	Subtracts the two operands, producing their difference	In[]: n1 - n2 Out[]: -1
*	Multiplication operator	Produces the product of the operands	In[]: n1 * n2 Out[]: 30
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor	In[]: n1 / n2 Out[]: 0.833333

%	Modulus operator	Divides left hand operand by the right hand operand and returns a remainder	In[]: n1 % n2 Out[]: 5
**	Exponent operator	Performs exponential (power) calculation on operators	In[]: n1 ** n2 Out[]: 15625
//	Floor division operator	Returns the integral part of the quotient	In[]: n1 // n2 Out[]: 0

TABLE 3 : List of Assignment Operators

Operator	Operator Name	Description	Example (try in lab) (‘~’ stand for is equivalent to)
=	Assignment	Assigns values from right side operands to left side operand	m = n1 + n2 In[]: m = n1 + n2 In[]: m Out[]: 11
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand	m += n1 ~ m = m + n1 In[]: m += n1 In[]: m Out[]: 16
-=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand	m -= n1 ~ m = m - n1 In[]: m -= n1 In[]: m Out[]: 11
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand	m *= n1 ~ m = m * n In[]: m *= n1 In[]: m Out[]: 55
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand	m /= n1 ~ m = m / n1 In[]: m /= n1 In[]: m Out[]: 11.0
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand	m ** n1 ~ m = m ** n1 In[]: m **= n1 In[]: m Out[]: 161051.0
//=	Floor Division	Produces the integral part	m //= n1 ~ m = m // n1

	Assignment	of the quotient of its operands where the left operand is the dividend and the right operand is the divisor	m // n1 In[]: m // = n1 In[]: m Out[]: 32210.0
%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	m %= n1 ~ m = m % n1 In[]: m %= n1 In[]: m Out[]: 0.0

TABLE 4 : List of Relational Operators

Operator	Operation	Description	Example (Try in Lab) n1 = 10, n2 = 0, s1 = "Hello", s2="World"
==	Equals to	If values of two operands are equal, then the condition is True, otherwise it is False.	In[]: n1 == n2 Out[]: False In[] : s1 == s2 Out[]: False
!=	Not equal to	If values of two operands are not equal, then condition is True, otherwise it is False.	In[] : n1 != n2 Out[]: True In[]: s1 != s2 Out[]: False
>	Greaer than	If the value of the left operand is greater than the value of the right operand, then the condition is True, otherwise it is False.	In[]: n1 > n2 Out[]: True In[]: s1 > s2 Out[]: True
<	Less than	If the value of the left operand is less than the value of the right operand, the condition is True otherwise it is False.	In[]: n1 < n2 Out[]: False
>=	Greater than or equal to	If the value of the left operand is greater than or equal to the right operand, the condition is True otherwise it is False.	In[]: n1 >= n2 Out[]: True
<=	Less than or equal to	If the value of the left operand is less than or equal to the right operand, the condition is True otherwise it is False.	In []: n1 <= n2 Out[]: False

Note :For strings, the characters in both the strings are compared one by one based on their Unicode value. The character with the lower Unicode value is considered to be smaller.

TABLE 5 : List of Logical Operator

Operator	Operation	Description	Example (Try in Lab) n1 = 10, n2 = -20
And	Logical AND	If both operands are True, then condition becomes True.	In[]: n1 == 10 and n2 == -20 Out[]: True
Or	Logical OR	If any of the two operands are True, then condition becomes True.	In[]: n1 >= 10 or n2 <-20 Out[]: True
Not	Logical NOT	Used to reverse the logical state of its operand.	In[]: not(n1 ==20) Out[]: True

TABLE 6 : Boolean Logic Truth Table

n1	n2	n1 and n2	n1 or n2	not n1
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

TABLE 7 : List of Bitwise Operators

Operator	Operator Name	Description	Example (try in Lab) n1 = 60, n2 = 13
&	Binary AND	Result is one in each bit position for which the corresponding bits of both operands as 1s.	In[]: n1 & n2 Out[]: 12 (means 0000 1100)
	Binary OR	Result is one in each bit position for which the corresponding bits of either or both operands are 1s.	In[]: n1 n2 Out[]: 61 (means 0011 1101)
^	Binary XOR	Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.	In[]: n1 ^ n2 Out[]: 49 (means 0011 0001)

\sim	Binary Ones Complement	Inverts the bits of its operand.	In[]: $\sim n1$ Out[]: -61 (means 1100 0011 in 2's complement form due to a signed binary number)
$<<$	Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.	In[]: $n1 << 2$ Out[]: 240 (means 1111 0000)
$>>$	Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	In[]: $n1 >> 2$ Out[]: 15 (means 0000 1111)

TABLE 8 : Bitwise Truth Table

n1	n2	n1 & n2	n1 n2	n1 ^ n2	$\sim n1$
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

TABLE 9 : Operator Precedence in Python

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor Division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=	Comparisons
Not	Logical NOT
And	Logical AND
Or	Logical OR

Check your progress – 4 :

Q12. 4 is 100 in binary and 11 is 1011. What is the output of the following bitwise operators?

$$a = 4$$

$$b = 11$$

```
print(a | b)
print(a >> 2)
a) 15, 1
b) 14, 1
c) 15, 2
d) 14, 2
```

Q13. What is the output of `print(2 ** 3 ** 2)`?

- a) 64
- b) 128
- c) 256
- d) 512

Q14. What is the output of the following assignment operator?

```
y = 10
x = y += 2
print(x)
a) 12
b) 10
c) SyntaxError
```

Q15. What is the output of following code?

```
a = 100
b = 200
print(a and b)
a) 100
b) 200
c) True
d) False
```

Q16. Which of the following operators has the lowest precedence?

- a) %
- b) +
- c) **
- d) not
- e) and

10.7 DATA TYPES

Data types specify the type of data like numbers and characters to be stored and manipulated within a program. Basic data types of Python are:

- 10.7.1 Number
- 10.7.2 String
- 10.7.3 None

- 10.7.4 List
- 10.7.5 Tuple
- 10.7.6 Dictionary
- 10.7.7 Set

10.7.1 Number

In Numbers or Numerical Data Types, we mainly have numbers. These numbers are also of four types : Integer, Float, Complex, Boolean.

TABLE 10 : Data Types in Python

Type / Class	Description	Examples
Int	Integer numbers	-12, -3, 0, 123, 2
float	Floating point numbers	-2.04, 4.0, 14.23
complex	Complex numbers	3+4j, 2-2j
bool	Boolean numbers	True, false

In Interactive mode:

Note :`type()` function can be used to check the data type of the variables

Examples:

`In[]: i = 23` #variable i assigned with integer value

`In[]: type(i)`

`Out[]: int`

`In[]: c = 2+5j` #variable c assigned with a complex number

`In[]: type(c)`

`Out[]: complex`

`In[]: b = 10>7`

`In[]: type(b)` #to display the data type of variable b

`Out[]: bool`

`In[]: b` #to display the value of b

`Out[]: True`

10.7.2 String

A **string** is an ordered sequence of characters. These characters may be alphabets, digits or special characters including spaces. String values are enclosed in single quotation marks (e.g. “hello”) or in double quotation marks (e.g., “python course”). The quotes are not a part of the string, they are used to mark the beginning and end of the string for the interpreter. In Python, there is no character data type, a character is a string of length one. It is represented by **str** class. Few of the characteristics of strings are:

- Numerical operations can not be performed on strings, even when the string contains a numeric value like str2 defined below.
- A string has a length. Get the length with the len() built-in function.
- A string is indexable. Get a single character at a position in a string with the square bracket operator, for example str1[4]. Indexing always start with 0.
- One can retrieve a slice (sub-string) of a string with a slice operation, for example str1[4:8].
- Strings are immutable, which means you can't change an existing string.

In[]: **str1 = ‘Hello Friend’** #variable str1 of string type is declared

In[]: **str2 = “452”** #variable str2 of string type is declared

In[]: **len(str1)** #length of string str1

Out[]: **12**

In[]: **str1[4]** #to access 5th character in the string

Out[]: **‘o’**

In[]: **str1[-1]** #negative indices can be used, which count backward from the end of string. [-1] yields the last letter.

Out[]: **‘d’**

In []: **str[4:8]** # Operator returns the part of the string from the first index to the second index, including the first but excluding the last. So, sub-string starting from 5 character to 8 character is extracted.

Out[]: **‘o Fr’**

If one omits the first index (before the colon), the slice starts at the beginning of the string and if second index is omitted, the slice goes to the end of the string.

In[]: **str1[:3]**

Out[]: ‘hel’

In[]: str1[3:]

Out[]: ‘lo Friend’

In []: str1[2] = ‘a’

TypeError: ‘str’ object does not support item assignment.

In[]: str2 = ‘ Narender’

In[]: str1 + str2 #for concatenating two strings, “+” operator is used

Out[]: ‘hello Friend Narender’

In[]: str1 * 3 #for creating multiple concatenated copies of a string, “*” operator is used

Out[]: ‘hello FriendhelloFriendhello Friend’

10.7.3 None

A **none** is another special data type in Python. A none is frequently used to represent the absence of a value. For example,

In[]: money = None #None value is assigned to variable money

Some literatures on Python consider additional data types like List, Tuple, Set &Dictionary whereas some others consider them as the built-in data structures. We will put them in the category of data structures and discuss considering them in same but considering them as data types is also not wrong.

10.7.4 List

A **list** is a sequence of items separated by commas and items enclosed in square brackets []. These are similar to arrays available with other programming languages but unlike arrays items in the list may be of different data types. Lists are mutable, and hence, they can be altered even after their creation. Lists in Python are ordered and have a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and creditability. It is represented by **list** class.

In[]: list1 = [5, 3.4, ‘IGNOU’, ‘Part 3’, 45] #to create a list

In[]: list1 #to print the elements of the list

list1

```
Out[]: [5, 3.4, 'IGNOU', 'Part 3', 45]
In[]: list1[0]                                #accessing first element of the
list
Out[]: 5
In[]: list1[3]                                #accessing 4th element of the list
Out[]: Part 3
In[]: list1[-1]                               #accessing last element of the
list
Out[]: 45
In[]: list1[-3]                               #accessing last 3rd element of the
list
Out[]: 'IGNOU'
```

10.7.5 Tuple

A **tuple** is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma and enclosed in parentheses.

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple and it is immutable.

```
In[]: t = ('hello','world',2)                  #creating tuple t
In[]: t                                         # printing tuple t
Out[]: ('hello', 'world', 2)
In[]: t[1]                                      #accessing specific element of the tuple
say second element
Out[]: 'world'
```

10.7.6 Dictionary

A **dictionary** in Python holds data items in key-value pairs of items. The items in the dictionary are separated with the comma and enclosed in the curly brackets {}. Dictionaries permit faster access to data. Every key is separated from its value using a colon(:) sign. The key value pairs of a dictionary can be accessed using the key. Keys are usually of string type and their values can be of any data type. In order to access any value in the dictionary, we have to specify its key in square brackets [].

```
#Creating dictionary dict1
In[]: dict1 = {'Fruit':'Apple','Climate':'Cold','Price(Kg)':120}

In[]: dict1                                #printing the contents of
      dictionary dict1

Out[]: {'Fruit': 'Apple', 'Climate': 'Cold', 'Price(Kg)': 120}

In[]: dict1['Climate']                      #Getting value by specifying the
      key

Out[]: Cold

In[]: dict1.keys()                          #printing all the Keys in the
      dictionary

Out[]: dict_keys(['Fruit', 'Climate', 'Price(Kg)'])

In[]: dict1.values()                        #printing all the values in the
      dictionary

Out[]: dict_values(['Apple', 'Cold', 120])
```

10.7.7 Set

In Python, a set is an ordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by ‘comma’. A set contains only unique elements but at the time of set creation, multiple duplicate values can also be passed. The order of elements in a set is undefined and is unchangeable. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

```
In[]: set1 = set("IGNOU MCA BATCH")          #creating
      set set1

In[]: set1
      #displaying contents of set1

Out[]: {' ', 'A', 'B', 'C', 'G', 'H', 'T', 'M', 'N', 'O', 'T', 'U'}

In[]: set2 = set([2, 3, 2, 'MCA', 'IGNOU', 1, 'IGNOU']) #set with
      the use of mixed values

In[]: set2

Out[]: {1, 2, 3, 'IGNOU', 'MCA'}
```

Check your progress - 5:

Q17. Which of these is not a core data type?

- a) Lists
- b) Dictionary
- c) Tuples
- d) Class

Q18. What data type is the object below?

L = [1, 23, 'hello', 1]

- a) List
- b) Dictionary
- c) Tuple
- d) Array

Q19. What will be the output of the following Python code?

In[]:str="hello"

In[]: str[:2]

- a) he
- b) lo
- c) olleh
- d) hello

Q20. In python we do not specify types, it is directly interpreted by the compiler. So consider the following operation to be performed:

x = 13 ? 2

objective is to make sure x has a integer value, select all that apply

- a) x = 13 // 2
- b) x = int(13 / 2)
- c) x = 13 % 2
- d) all of the above

Q21. What is the result of print(type({})) is set

- a) True
- b) False

Q22. What is the data type of print (type(10))?

- a) float
- b) integer
- c) int

Q23. If we convert one data type to another, then it is called

- a) Type Conversion
- b) Type Casting
- c) Both of the above

d) None of the above

Q24. In which data type, indexing is not valid?

- a) list
- b) string
- c) dictionary
- d) None of the above

Q25. In which of the following data type duplicate items are not allowed?

- a) list
- b) set
- c) dictionary
- d) None of the above

Q26. Which statement is correct?

- a) List is immutable && Tuple is mutable
- b) List is mutable && Tuple is immutable
- c) Both are mutable
- d) Both are immutable

10.8 DATA STRUCTURES

Data Structures are used to organize, store and manage data for efficient access and modification. Some literatures on Python categorize List, Tuple, Dictionary and Set as data types and some categorize these as primitive data structures. We had already introduced the above mentioned structures in the data type. Lets discuss them in detail here:

10.8.1 List

A **list** a container data type that acts as a dynamic array. That is to say, a list is a sequence that can be indexed into and that can grow and shrink. A few characteristics of lists are:

- A list has a (current) length – Get the length of a list with len() function.
- A list has an order – the items in a list are ordered, order going from left to right.
- A list is heterogeneous –different types of objects can be inserted into the same list.
- Lists are mutable and one can extend, delete, insert or modify items in the list.

The syntax for creating list is,

```
list_name = [item_1, item_2, item_3, ..... , item_n]
```

Any item can be stored in a list like string, number, object, another variable and even another list. In a list, we can have a mix of different item types and these item types need not have to be homogeneous. For example, we can have a list which is a mix of type numbers, strings and another list itself. The contents of the list variable are displayed by executing the list variable name.

Examples :

In[]: **list_name** = [] #empty list
without any items

#when each item in the list is string

In[]: **stringlist** = ["mahesh", "ramesh", "suresh", "vishnu",
"ganesh"]

#when each item in the list is number

In[]: **numlist** = [4, 6, 7, 10, 15, 13]

#when list contains mixed items

In[]: **mixed_list** = ['cat', 87.23, 65, [9, 8, 1]]

10.8.1.1 Indexing and Slicing in Lists

As an ordered sequence of elements, each item in a list can be individually, through indexing. The expression inside the bracket is called the index. Lists use square brackets [] to access individual items, with the first item at index 0, the second item at index 1 and so on. The index provided within the square brackets indicates the value being accessed.

The syntax for accessing an item in a list is,

list_name [index]

where index should always be an integer value and indicates the item to be selected.

In[]: **stringlist[0]** #to access first item in the stringlist

Out[]: 'mahesh'

In[]: **numlist[4]** #to access fifth item in the numlist

Out[]: 15

In[]: **numlist[6]** #if index value is more than the number of items in the list, it will raise an error

Output would be :

Traceback (most recent call last):

File “<stdin>”, line 1, in <module>

numlist[6]

IndexError : List index out of range

Here, in numlist index numbers range from 0 to 5 as it contains 6 items.

In addition to positive index numbers, one can also access items from the list with a negative index number, by counting backwards from the end of the list, starting at -1. It is useful if the list is long and we want to locate an item towards the end of a list.

In[]: stringlist[-2] #to access second list item from the list

Out[]: ‘vishnu’

The slice operator also works on lists:

In[]: numlist[1:3]

Out[]: [6, 7]

In[]: mixed_list[:3]

Out[]: [‘cat’, 87.23, 65]

In[]: mixed_list[2:]

Out[]: [65, [9, 8, 1]]

In[]: numlist[:]

Out[]: [4, 6, 7, 10, 15, 13]

If the first index is omitted, the slice starts at the beginning and if the second index is omitted, the slice goes to the end. So, if both indexes are omitted, the slice is a copy of the whole list.

10.8.1.2 List Operations

The + operator concatenates lists:

In[]: a = [1, 2, 3]

In[]: b = [4, 5, 6]

In[]: a + b

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

Similarly, the * operator repeats a list given a number of times:

```
In[]: c = ['pass']
In[]: c * 3
Out[]: ['pass', 'pass', 'pass']
```

`==` operator is used to compare the two lists.

```
In[]: a == b
Out[]: False
```

`in` and `not in` membership operator are used to check for the presence of an item in the list.

```
In[]: 2 in a
Out[]: True
In[]: 4 not in [a]
Out[]: True
```

10.8.1.3 The list() function

The built-in-list() function is used to create a list. The syntax for list() function is,

list([sequence])

where the sequence can be a string, tuple or list itself. If the optional sequence is not specified then an empty list is created.

```
In[]: greet = "How are you doing?"           #string declaration
In[]: greet
Out[]: 'How are you doing?'
In[]: str_to_list = list(greet)                #string converted to list
using list()
In[]: str_to_list
Out[]: ['H', 'o', 'w', ' ', 'a', 'r', 'e', ' ', 'y', 'o', 'u', ' ', 'd', 'o', 'i', 'n', 'g', '?']
In[]: friend = "nidhi"                         #string declaration
In[]: str_to_list + friend                     #list concatenated with
string
```

TypeError: can only concatenate list (not "str") to list

```
In[]: str_to_list + list(friend)             #list concatenated with
list
```

```
Out[]:[‘H’,’o’,’w’,’ ’,’a’,’r’,’e’,’ ’,’y’,’o’,’u’,’ ’
,’d’,’o’,’i’,’n’,’g’,’?’,’n’,’i’,’d’,’h’,’i’]
```

10.8.1.4 Modifying items in Lists

Lists are mutable in nature as the list items can be modified after a list has been created. Also, a list can be modified by replacing the older item with a newer item in its place and without assigning the list to a completely new variable.

```
In[]: stringlist[1] = “umesh”
```

```
In[]: stringlist
```

```
Out[]: [‘mahesh’, ‘umesh’, ‘suresh’, ‘vishnu’, ‘ganesh’]
```

When an existing list variable is assigned to a new variable, an assignment ($=$) on lists does not make a new copy. Instead, assignment makes both the variables names point to the same list in memory. That's why any change in one list will reflect in other list also.

```
In[]: listofstrings = stringlist
```

```
In[]: listofstrings[1] = “ramesh”
```

```
In[]: listofstrings
```

```
Out[]:[‘mahesh’, ‘ramesh’, ‘suresh’, ‘vishnu’, ‘ganesh’]
```

```
In[]: stringlist
```

```
Out[]:[‘mahesh’, ‘ramesh’, ‘suresh’, ‘vishnu’, ‘ganesh’]
```

10.8.1.5 Traversing a list

The most common way to traverse the elements of a list is with a for loop.

```
In[]: for names in stringlist :
```

```
    print(names)
```

```
Out[]: mahesh
```

```
    ramesh
```

```
    suresh
```

```
    vishnu
```

```
    ganesh
```

This works well if one needs to read the elements of the list. But if one wants to write or update the elements, one needs the indices. For this one needs to combine the functions range and len:

```
In[]: for i in range(len(numlist)) :  
        numlist[i] = numlist[i] * 2
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 30, 26]
```

This loop traverses the list and updates each element. The command `len` returns the number of elements in the list. The command `range` returns a list of indices from 0 to $n-1$, where n is the length of the list. Each time through the loop, the variable `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

10.8.1.6 List Methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
In[]: numlist.append(50)
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 30, 26, 50]
```

`extend` takes a list as an argument and appends all of the elements:

```
In[]: nlist = [25, 35, 45]
```

```
In[]: numlist.extend(nlist)
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 30, 26, 50, 25, 35, 45]
```

`sort` arranges the elements of the list from low to high:

```
In[]: numlist.sort()
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 25, 26, 30, 35, 45, 50]
```

`pop` is used to delete elements from a list if the index of the element is known:

```
In[]: numlist.pop(5)
```

```
Out[]: 26
```

```
In[]: numlist
```

```
Out[]: [8, 12, 14, 20, 25, 30, 35, 45, 50]
```

pop modifies the list and returns the element that was removed. If the removed element is not required then one can use the **del** operator:

```
In[]: del numlist[0]
```

```
In[]: numlist
```

```
Out[]: [12, 14, 20, 25, 30, 35, 45, 50]
```

To remove more than one element, **del** can be used with a slice index

```
In[]: del numlist[0:2]
```

```
In[]: numlist
```

```
Out[]: [20, 25, 30, 35, 45, 50]
```

If element from the list is known which needs to be removed and not the index, **remove** can be used:

```
In[]: numlist.remove(30)
```

```
In[]: numlist
```

```
Out[]: [20, 25, 35, 45, 50]
```

The return value for remove is none.

To get a list of all the methods associated with the list, pass the list function to **dir()**

```
In[]: dir(list)
```

TABLE 11 : Various List Methods

List Methods	Syntax	Description
append()	list.append(item)	The append() method adds a single item to the end of the list. This method does not return new list and it just modifies the original.
count()	list.count(item)	The count() method counts the number of times the item has occurred in the list and returns it.
insert()	list.insert(index, item)	The insert() method inserts the item at the given index, shifting items to the right.
extend()	list.extend(list2)	The extend() method adds the items in list2 to the end of the list.

index()	list.index(item)	The index() method searches for the given item from the start of the list and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the list then ValueError is thrown by this method.
remove()	list.remove(item)	The remove() method searches for the first instance of the given item in the list and removes it. If the item is not present in the list then ValueError is thrown by this method.
sort()	list.sort()	The sort() method sorts the items in place in the list. This method modifies the original list and it does not return a new list.
reverse()	list.reverse()	The reverse() method reverses the items in place in the list. This method modifies the original list and it does not return a new list.
pop()	list.pop([index])	The pop() method removes and returns the item at the given index. This method returns the rightmost item if the index is omitted.

Note: Replace the word "list" mentioned in the syntax with your actual list name in your code.

10.8.1.7 Built-In Functions Used on Lists

There are many built-in functions for which a list can be passed as an argument.

TABLE 12 : Built – In Functions Used on Lists

Built-In Functions	Description
len()	The len() function returns the numbers of items in a list.
sum()	The sum() function returns the sum of numbers in the list.
any()	The any() function returns True if any of the Boolean values in the list is True.
all()	The all() function returns True if all the Boolean values

	in the list are True, else returns False.
sorted()	The sorted() function returns a modified copy of the list while leaving the original list untouched.

In[]: **len(stringlist)**

Out[]: **5**

In[]: **sum(numlist)**

Out[]: **175**

In[]: **max(numlist)**

Out[]: **50**

In[]: **min(numlist)**

Out[]: **20**

In[]: **any([1, 1, 0, 0, 1, 0])**

Out[]: **True**

In[]: **all([1, 1, 1, 1])**

Out[]: **True**

In[]: **sorted_stringlist = sorted(stringlist)**

In[]: **sorted_stringlist**

Out[]: **['ganesh', 'mahesh', 'ramesh', 'suresh', 'vishnu']**

10.8.1.8 Nested List

A list inside another list is called a **nested list** and you can get the behavior of nested lists in Python by storing lists within the elements of another list. The syntax for nested lists is:

```
nested_list_name = [[item_1, item_2, item_3],
[item_4, item_5, item_6],
[item_7, item_8, item_9]]
```

Each list inside another list is separated by a comma. One can traverse through the items of nested lists using the for loop.

```
In[]: asia = [[“India”, “Japan”, “Korea”],
[“Srilanka”, “Myanmar”, “Thailand”],
```

[“Cambodia”, “Vietnam”, “Israel”]]

In[]: asia[0]

Out[]: [‘India’, ‘Japan’, ‘Korea’]

In[]: asia[0][1]

Out[]: ‘Japan’

In[]: asia[1][2]

Out[]: ‘Thailand’

Program 10.1 : Write a Program to Find the Transpose of a Matrix

```
1. matrix = [[10, 20],  
2.             [30, 40],  
3.             [50, 60]]  
4. matrix_transpose = [[0, 0, 0],  
5.                      [0, 0, 0]]  
6. for rows in range(len(matrix)):  
7.     for columns in range(len(matrix[0])):  
8.         matrix_transpose[columns][rows] = matrix[rows][columns]  
9. print("Transposed Matrix is")  
10. for items in matrix_transpose:  
11.     print(items)
```

Fig 1 : Screen Shot of execution of Program 10.1

```
[2] matrix = [[10, 20],  
              [30, 40],  
              [50, 60]]  
matrix_transpose = [[0, 0, 0],  
                   [0, 0, 0]]  
for rows in range(len(matrix)):  
    for columns in range(len(matrix[0])):  
        matrix_transpose[columns][rows] = matrix[rows][columns]  
print("Transposed Matrix is")  
for items in matrix_transpose:  
    print(items)
```

⇨ Transposed Matrix is
[10, 30, 50]
[20, 40, 60]

Check your progress – 6 :

Q27. Empty list in python is made by?

- a) l = []
- b) l = list()

- c) Both of the above
- d) None of the above

Q28. If we try to access the item outside the list index, then what type of error it may give?

- a) List is not defined
- b) List index out of range
- c) List index out of bound
- d) No error

Q29. $l = [1, 2, 3, 4]$, then $l[-2]$ is?

- a) 1
- b) 2
- c) 3
- d) 4

Q30. The marks of a student on 6 subjects are stored in a list, $list1 = [80, 66, 94, 87, 99, 95]$. How can the students average marks be calculated?

- a) `print(avg(list1))`
- b) `print(sum(list1)/len(list1))`
- c) `print(sum(list1)/sizeof(list1))`
- d) `print(total(list1)/len(list1))`

Q31. What will be the output of the following Python code?

```
list1=["Python", "Java", "c", "C", "C++"]
print(min(list1))
```

- a) c
- b) C++
- c) C
- d) Min function can't be used on string elements

Q32. The elements of a list are arranged in descending order:

Which of the following two will give same outputs?

- a) `print(list_name.sort())`
- b) `print(max(list_name))`
- c) `print(list_name.reverse())`
- d) `print(list_name[-1])`

i, ii

i, iii

ii, iii

iii, iv

Q33. What will be the output of below python code?

```
list1=["tom", "mary", "simon"]
list1.insert(5,8)
print(list1)
```

- a) [“tom”, “mary”, “simon”, 5]
- b) [“tom”, “mary”, “simon”, 8]
- c) [8, “tom”, “marry”, “simon”]
- d) Error

Q34. What will be the output of below python code?

```
list1=[1,3,5,2,4,6,2]
```

```
list1.remove(2)
```

```
print(sum(list1))
```

- a) 18
- b) 19
- c) 21
- d) 22

Q35. What will be the output of below python code?

```
list1=[3,2,5,7,3,6]
```

```
list1.pop(3)
```

```
print(list1)
```

- a) [3,2,5,3,6]
- b) [2,5,7,3,6]
- c) [2,5,7,6]
- d) [3,2,5,7,3,6]

10.8.2 Tuple

In mathematics, a tuple is a finite ordered list (sequence) of elements. A **tuple** is defined as a data structure that comprises an ordered, finite sequence of immutable, heterogeneous elements that are of fixed sizes. Often, we may want to return more than one value from a function. Tuples solve this problem. They can also be used to pass multiple values through one function parameter.

10.8.2.1 Creating Tuples

A tuple is a finite ordered list of values of possibly different types which is used to bundle related values together without having to create a specific type to hold them. Tuples are immutable. Once a tuple is created, one cannot change its values. A tuple is defined by putting a comma-separated list of values inside parentheses(). Each value inside a tuple is called an item. The syntax for creating tuples is,

```
tuple_name = (item_1, item_2, item_3, ..... , item_n)
```

Syntactically, a tuple is a comma-separated list of values:

```
In[]: t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when Python code is looked at:

```
In[]: t = ('a', 'b', 'c', 'd', 'e')
```

It is actually the comma that forms a tuple making the commas significant and not the parentheses.

To create a tuple with a single element, one must include the final comma:

```
In[]: t1 = ('a',)
```

```
In[]: type(t1)
```

```
Out[]: tuple
```

In Python, the tuple type is tuple. Without the comma Python treats ('a') as an expression with a string in parentheses that evaluates to a string:

```
In[]: t2 = ('a')
```

```
In[]: type(t2)
```

```
Out[]: str
```

One can create an empty tuple without any values. The syntax is,

```
tuple_name = ()
```

```
In[]: empty_tuple = ()
```

```
In[]: empty_tuple
```

```
Out[]: ()
```

One can store any type of type string, number, object, another variable, and even another tuple itself. One can have a mix of different types of items in tuples, and they need not be homogeneous.

```
In[]: socis = ('mca','6 sem','3-6 yrs',54000)
```

```
In[]: ignou = ('soe','soh',socis,'soms')
```

```
In[]: ignou
```

```
Out[]: ('soe', 'soh', ('mca', '6 sem', '3 - 6 yrs', 54000), 'soms')
```

10.8.2.2 Basic Tuple Operations

Most list operators also work on tuples. The bracket operator indexes an element:

```
In[]: t[0]
```

```
Out[]: 'a'
```

And the slice operator selects a range of elements.

```
In[]: t[1:3]
```

```
Out[]: ('b', 'c')
```

But if you try to modify one of the elements of the tuples, one will get an error:

```
In[]: t[0] = 'A'
```

```
TypeError: 'tuple' object does not support item assignment
```

We can't modify the elements of a tuple, but can replace one tuple with another:

```
In[]: t = ('A',) + t[1:]
```

```
In[]: t
```

```
Out[]: ('A', 'b', 'c', 'd', 'e')
```

Here, + operator is used to concatenate two tuples together. Similarly, * operator can be used to repeat a sequence of tuple items.

```
In[]: t * 2
```

```
Out[]: ('A', 'b', 'c', 'd', 'e', 'A', 'b', 'c', 'd', 'e')
```

in and **not in** membership operator are used to check for the presence of an item in a tuple.

Comparison operators like <, <=, >, >=, == and != are used to compare tuples.

Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered even if they are really big.

```
In[]: (0, 1, 2) < (0, 3, 4)
```

```
Out[]: True
```

```
In[]: (0, 1, 500000) < (0, 3, 4)
```

```
Out[]: True
```

10.8.2.3 The tuple() Function

The built-in tuple() function is used to create a tuple. The syntax for the tuple() function is:

tuple([sequence])

where sequence can be a number, string or tuple itself. If the optional sequence is not specified, then an empty tuple is created.

```
In[]: t3 = tuple()
```

```
In[]: t3
```

```
Out[]: ()
```

If the argument is a sequence (string, list or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
In[]: t3 = tuple('IGNOU')
```

```
In[]: t3
```

```
Out[]: ('I', 'G', 'N', 'O', 'U')
```

```
In[]: t4 = (1, 2, 3, 4)
```

```
In[]: nested_t = (t3, t4)
```

```
In[]: nested_t
```

```
Out[]: (('I', 'G', 'N', 'O', 'U'), (1, 2, 3, 4))
```

10.8.2.4 Built-In Functions Used on Tuples

There are many built-in functions as listed in TABLE for which a tuple can be passed as an argument.

TABLE 13: Built-In functions Used on Tuples

Built-In Functions	Description
len()	The len() function returns the number of items in a tuple.
sum()	The sum() function returns the sum of numbers in the tuple.
sorted()	The sorted() function returns a sorted copy of the tuple as a list while leaving the original tuple untouched.

```
In[]: len(t3)
```

```
Out[]: 5
```

```
In[]: sum(t4)
```

```
Out[]: 10
```

```
In[]: t5 = sorted(t3)
```

```
In[]: t5
Out[]: ['G', 'I', 'N', 'O', 'U']
```

10.8.2.5 Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows one to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

```
In[]: m = ['good', 'luck']
```

```
In[]: x, y = m
```

```
In[]: x
```

```
Out[]: 'good'
```

```
In[]: y
```

```
Out[]: 'luck'
```

Python roughly translates the tuple assignment syntax to be the following:

```
In[]: m = ['good', 'luck']
```

```
In[]: x = m[0]
```

```
In[]: y = m[1]
```

Stylistically, when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
In[]: (x, y) = m
```

10.8.2.6 Relation between Tuples and Lists

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing. Lists are mutable, and their items are accessed via indexing. Items cannot be added, removed or replaced in a tuple.

If an item within a tuple is mutable, then you can change it. Consider the presence of a list as an item in a tuple, then any changes to the list get reflected on the overall items in the tuple.

```
In[]: ignou = ['soe', 'soms', 'socis']
```

```
In[]: univ = ('du', 'ggsipu', 'jnu', ignou)
```

In[]: **univ**

Out[]: ('du', 'ggsipu', 'jnu', ['soe', 'soms', 'socis'])

In[]: **univ[3].append('soss')**

In[]: **univ**

Out[]: ('du', 'ggsipu', 'jnu', ['soe', 'soms', 'socis', 'soss'])

10.8.2.7 Tuple Methods

To get a list of all the methods associated with the tuple, pass the tuple function to dir().

Various methods associated with tuple are listed in the TABLE.

TABLE 14: Various Tuple Methods

Tuple Methods	Syntax	Description
count()	tuple_name.count(item)	The count() method counts the number of times the item has occurred in the tuple and returns it.
index()	tuple-name.index(item)	The index() method searches for the given item from the start of the tuple and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the tuple, then ValueError is thrown by this method.

Note: Replace the word “tuple_name” mentioned in the syntax with your actual tuple name in the code.

In[]: **channels = ("dd", "star", "sony", "zee", "dd", "sab")**

In[]: **channels.count("dd")**

Out[]: 2

In[]: **channels.index("dd")**

Out[]: 0

In[]: **channels.index("zee")**

Out[]: 3

In[]: channels.count("sab")

Out[]: 1

PROGRM 10.2 : Program to Populate with User-Entered Items

```
1. tuple_items = ()  
2. total_items = int(input("Enter the total number of items: "))  
3. for i in range(total_items):  
4.     user_input = int(input("Enter a number: "))  
5.     tuple_items += (user_input,)  
6. print(f"Items added to tuple are {tuple_items}")  
7. list_items = []  
8. total_items = int(input("Enter the total number of items: "))  
9. for i in range(total_items):  
10.    item = input("Enter an item to add: ")  
11.   list_items.append(item)  
12. items_of_tuple = tuple(list_items)  
13. print(f"Tuple items are {items_of_tuple}")
```

IGNOU
THE PEOPLE'S
UNIVERSITY

Fig 2 : Screen Shot of execution of Program 10.2

```
tuple_items = ()  
total_items = int(input("Enter the total number of items: "))  
for i in range(total_items):  
    user_input = int(input("Enter a number: "))  
    tuple_items += (user_input,)  
print(f"Items added to tuple are {tuple_items}")  
list_items = []  
total_items = int(input("Enter the total number of items: "))  
for i in range(total_items):  
    item = input("Enter an item to add: ")  
    list_items.append(item)  
items_of_tuple = tuple(list_items)  
print(f"Tuple items are {items_of_tuple}")
```

```
Enter the total number of items: 5  
Enter a number: 8  
Enter a number: 2  
Enter a number: 9  
Enter a number: 1  
Enter a number: 6  
Items added to tuple are (8, 2, 9, 1, 6)  
Enter the total number of items: 4  
Enter an item to add: 2  
Enter an item to add: 4  
Enter an item to add: 6  
Enter an item to add: 8  
Tuple items are ('2', '4', '6', '8')
```

Items are inserted into the tuple using two methods: using continuous concatenation `+=` operator and by converting list items to tuple items. In the code, `tuple_items` is of tuple type. In both the methods, the total number of items are specified which will be inserted to the tuple beforehand. Based on this number, the for loop is iterated using the `range()` function. In the first method, the user entered items are continuously concatenated to the tuple using `+=` operator. Tuples are immutable and are not supposed to be changed. During each iteration, each `original_tuple` is replaced by `original_tuple + (new_element)`, thus creating a new tuple. Notice a comma after `new_element`. In the second method, a list is created. For each iteration, the user entered value is appended to the `list_variable`. This list is then converted to tuple type using `tuple()` function.

Program 10.3 : Program to Swap Two Numbers without Using Intermediate/Temporary Variable. Prompt the User for Input.

1. `a = int(input("Enter a value for the first number "))`
2. `b = int(input("Enter a value for the second number "))`
3. `b, a = a, b`
4. `print("After Swapping")`
5. `print(f"Value for first number {a}")`
6. `print(f"Value for second number {b}")`

Fig. 3 : Screen Shot of Program 10.3

```
a = int(input("Enter a value for the first number "))
b = int(input("Enter a value for the second number "))
b, a = a, b
print("After Swapping")
print(f"Value for first number {a}")
print(f"Value for second number {b}")
```

```
Enter a value for the first number 5
Enter a value for the second number 9
After Swapping
Value for first number 9
Value for second number 5
```

The contents of variables a and b are reversed. The tuple variables are on the left side of the assignment operator and, on the right side, are the tuple values. The number of variables on the left and the number of values on the right has to be the same. Each value is assigned to its respective variable.

Check your progress – 7 :

Q36. A python tuple can also be created without using parentheses

- a) False
- b) True

Q37. What is the output of the following?

```
aTuple = "Yellow", 20, "Red"
```

```
a, b, c = aTuple
```

```
print(a)
```

- a) ('Yellow', 20, 'Red')
- b) TypeError
- c) Yellow

Q38. Choose the correct way to access value 20 from the following tuple

```
aTuple = ("Orange", [10, 20, 30], (5, 15, 25))
```

- a) aTuple[1:2][1]
- b) aTuple[1:2](1)
- c) aTuple[1:2][1]
- d) aTuple[1][1]

Q39. Select which is true for python tuple

- a) A tuple maintains the order of items
- b) A tuple is unordered
- c) we cannot change the tuple once created
- d) we can change the tuple once created

Q40. What will be the output of below python code:

```
tuple1=(2, 4, 3)
tuple2=tuple1*2
print(tuple2)
a) (4, 8, 6)
b) (2, 4, 3, 2, 4, 3)
c) (2, 2, 4, 4, 3, 3)
d) Error
```

Q41. What will be the output of below python code:

```
tupl = ("annie", "hena", "sid")
print(tupl[-3:0])
a) ("annie")
b) ()
c) None
d) Error as slicing is not possible in tuple
```

Q42. Which of the following options will not result in an error when performed on tuples in python where tupl = (5, 2, 7, 0, 3)?

- a) tupl[1] = 2
- b) tupl.append(2)
- c) tupl1 = tupl + tupl
- d) tupl.sort()

10.8.3 Dictionaries

In the real world, you have seen your Contact-list in your phone. It is practically impossible to memorize the mobile number of everyone you come across. In the Contact-list, you store the name of the person as well as his number. This allows you to identify the mobile number based on a person's name. One can think of a person's name as the key that retrieves his mobile number, which is the value associated with the key. So, dictionary can be thought of :

- an un-ordered collection of key-value pairs, with the requirement that the keys are unique within a dictionary.
- as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.
- has a length, specifically the number of key-value pairs.
- provides fast look up by key.

The keys of the dictionary must be immutable object types and are case sensitive. Keys can be either a string or a number. But lists can not be used as keys. A value in the dictionary can be of any data type including string, number, list or dictionary itself.

Dictionaries are constructed using curly braces {}, wherein a list of key:value pairs get separated by commas. There is a colon(:) separating each of these keys and value pairs, where the words to the left of the colon operator are the keys and the words to the right of the colon operator are the values.

The syntax for creating a dictionary is :

```
dictionary_name = {key_1:value_1, key_2:value_2, key_3:value_3, .....
, key_n : value_n}
```

```
In[]: eng2sp = {'one' : 'uno', 'two' : 'dos', 'three' : 'tres'}
```

```
In[]: eng2sp
```

```
Out[]:{'one' : 'uno', 'two' : 'dos', 'three' : 'tres'}
```

With Python 3.6 version, the output of dictionary statements is ordered key:value pairs. Here, ordered means “insertion ordered”, i.e, dictionaries remember the order in which the key:value pairs were inserted. The elements of a dictionary are never indexed with integer indices. Instead, the keys are used to look up the corresponding values:

```
In[]: eng2sp['two']
```

```
Out[]: 'dos'
```

The key ‘two’ always maps to the value ‘dos’ so the order of the items doesn’t matter. If the key isn’t in the dictionary, one get an exception:

```
In[]: eng2sp['four']
```

```
Out[]: KeyError: 'four'
```

Slicing in dictionaries is not allowed since they are not ordered like lists.

10.8.3.1 Built –In Functions Used on Dictionaries

There are many built-in functions for which a dictionary can be passed as an argument. The main operations on a dictionary are storing a value with some key and extracting the value for a given key.

TABLE 15: Built-In Functions Used on Dictionaries

Built – in Functions	Description
len()	The len() function returns the number of items (key:value pairs) in a dictionary.
all()	The all() function returns Boolean True value if all the keys in the dictionary are True else retuens False.
any()	The any() function returns Boolean True value if any

	of the key in the dictionary is True else returns False.
sorted()	The sorted() function by default returns a list of items, which are sorted based on dictionary keys.

In[]: **len(eng2sp)**

Out[]: 3

len() function can be used to find the number of key:value pairs in the dictionary eng2sp. In Python, any non-zero integer value is True, and zero is interpreted as False. With **all()** function, if all the keys are Boolean True values, then the output is True else it is False.

In[]: **dict_func = {0 : True, 1 : False, 4 : True}**

In[]: **all(dict_func)**

Out[]: **False**

For **any()** function, if any one of the keys is True then it results in a True Boolean value else False Boolean value.

In[]: **any(dict_func)**

Out[]: **True**

The **sorted()** function returns the sorted list of keys by default in ascending order without modifying the original key:value pairs. For list of keys sorted in descending order by passing the second argument as **reverse = True**.

In[]: **sorted(eng2sp)**

Out[]: ['one', 'three', 'two']

In[]: **sorted(eng2sp, reverse=True)**

Out[]: ['two', 'three', 'one']

The **in** operator works on dictionaries; it tells whether something appears as a key in the dictionary (appearing as a value is not good enough).

In[]: **'one' in eng2sp**

Out[]: **True**

In[]: **'uno' in eng2sp**

Out[]: **False**

The **in** operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As a list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python

uses an algorithm called a hash table, so, the insertion takes about the same amount of time no matter how many items there are in a dictionary.

10.8.3.2 Dictionary Methods

Various methods associated with dictionary are listed below:

TABLE 16: Various Dictionary Methods

Dictionary Methods	Syntax	Description
clear()	dictionary_name.clear()	The clear() method removes all the key:value pairs from the dictionary.
fromkeys()	dictionary_name.fromkeys(seq[, value])	The fromkeys() method creates a new dictionary from the given sequence of elements with a value provided by the user.
get()	dictionary_name.get(key[, default])	The get() method returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to None, so that this method never raises a KeyError.
items()	dictionary_name.items()	The items() method returns a new view of dictionary's key and value pairs as tuples.
keys()	dictionary_name.keys()	The keys() method returns a new view consisting of all the keys in the dictionary.
pop()	dictionary_name.pop(key[, default])	The pop() method removes the key from the dictionary and returns its value. If the key is not present, then it returns the default value. If the default is not given and the key is not in the dictionary, then it results in KeyError.
popitem()	dictionary_name.popitem()	The popitem() method removes and returns an arbitrary (key, value) tuple pair from the dictionary. If the dictionary is empty, then calling popitem() results in KeyError.
setdefault()	dictionary_name.setdefault(key[, default])	The setdefault() method returns a value for the key present in the dictionary. If the key is not present, then insert the key into the dictionary with a default value and return the default value. If key is present, default defaults to None, so that this method never raises a KeyError.
update()	dictionary_name.update([o])	The update() method updates the dictionary with the key:value pairs from other dictionary object and it

	ther])	returns None.
values()	dictionary_name.values()	The values() method returns a new view consisting of all the values in the dictionary.

Note: Replace the word “dictionary_name” mentioned in the syntax with the actual dictionary name.

```
In[]: million_dollar = {"sanju" : 2018, "tiger zindahai" : 2017,  
"baahubali 2" : 2017, "dangal" : 2016, "bajrangibhaijaan" : 2015}
```

```
In[]: bolwd_million_dollar =  
million_dollar.fromkeys(million_dollar, "1,00,00,000")
```

```
In[]: bolwd_million_dollar
```

```
Out[]: {'sanju': '1,00,00,000',  
  
'tiger zindahai': '1,00,00,000',  
  
'baahubali 2': '1,00,00,000',  
  
'dangal': '1,00,00,000',  
  
'bajrangibhaijaan': '1,00,00,000'}
```

```
In[]: million_dollar.get("bahubali 1")
```

```
In[]: million_dollar.get("bahubali 1", 2015)
```

```
Out[]: 2015
```

```
In[]: million_dollar.keys()
```

```
Out[]: dict_keys(['sanju', 'tiger zindahai', 'baahubali 2', 'dangal',  
'bajrangibhaijaan'])
```

```
In[]: million_dollar.values()
```

```
Out[]: dict_values([2018, 2017, 2017, 2016, 2015])
```

```
In[]: million_dollar.items()
```

```
Out[]: dict_items([('sanju', 2018), ('tiger zindahai', 2017),  
('baahubali 2', 2017), ('dangal', 2016), ('bajrangibhaijaan', 2015)])
```

```
In[]: million_dollar.update({"bahubali 1" : 2015})
```

```
In[]: million_dollar
```

```
Out[15]:
```

```
{'sanju': 2018,  
  
'tiger zindahai': 2017,  
  
'bahubali 1': 2015}
```

'baahubali 2': 2017,
'dangal': 2016,
'bajrangibhaijaan': 2015,
'bahubali 1': 2015}

One of the common ways of populating dictionaries is to start with an empty dictionary {}, then use the **update()** method to assign a value to the key using assignment operator. If the key doesnot exist, then the key:value pairs will be created automatically and added to the dictionary.

```
In[]: states ={}  
In[]: states.update({"Haryana":"Chandigarh"})  
In[]: states.update({"Bihar":"Patna"})  
In[]: states.update({"west bengal" : "kolkata"})  
In[]: states  
Out[23]: {'Haryana': 'Chandigarh', 'Bihar': 'Patna', 'west bengal': 'kolkata'}
```

If a dictionary is used as the sequence in a for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
In[] : for key in states :  
        print(key, states[key])  
Out[]:  
Haryana Chandigarh  
Bihar Patna  
west bengalkolkata
```

Program 10.4 : Write a Program that Accepts a Sentence and Calculate the Number of Digits, Uppercase and Lowercase Letters

1. sentence = input("Enter a sentence : ")
2. construct_dictionary = {"digits":0, "lowercase":0, "uppercase":0}
3. for each_character in sentence :
4. if each_character.isdigit() :
5. construct_dictionary["digits"] += 1
6. elif each_character.isupper() :
7. construct_dictionary["uppercase"] += 1
8. elif each_character.islower() :
9. construct_dictionary["lowercase"] += 1
10. print("The number of digits, lowercase and uppercase letters are")

11. print(construct_dictionary)

Fig. 4: Screen Shot of execution of Program 10.4

```

sentence = input("Enter a sentence : ")
construct_dictionary = {"digits":0, "lowercase":0, "uppercase":0}
for each_character in sentence :
    if each_character.isdigit() :
        construct_dictionary["digits"] += 1
    elif each_character.isupper() :
        construct_dictionary["uppercase"] += 1
    elif each_character.islower() :
        construct_dictionary["lowercase"] += 1
print("The number of digits, lowercase and uppercase letters are")
print(construct_dictionary)

```

```

Enter a sentence : Ask not what your country can do for you; ask what you can do for your country, John Kennedy, 1961
The number of digits, lowercase and uppercase letters are
{'digits': 4, 'lowercase': 69, 'uppercase': 3}

```

To delete the key:value pair, use the **del** statement followed by the name of the dictionary along with the key you want to delete.

In[]: **del** states[“west bengal”]

In[]: **states**

Out[]: {‘Haryana’: ‘Chandigarh’, ‘Bihar’ : ‘Patna’}

10.8.3.3 Relation between Tuples and Dictionaries

Tuples can be used as key:value pairs to build dictionaries.

In[]:pm_year
 (('jln',1947),('lbs',1964),('ig',1966),('rg',1984),('pvn',1991),('abv',1998),('nm',2014))

In[]: pm_year

Out[]:

((‘jln’, 1947),
 (‘lbs’, 1964),
 (‘ig’, 1966),
 (‘rg’, 1984),
 (‘pvn’, 1991),
 (‘abv’, 1998),
 (‘nm’, 2014))

In[]: pm_year_dict = dict(pm_year)

In[]: pm_year_dict

Out[]:

```
{'jln': 1947,  
 'lbs': 1964,  
 'ig': 1966,  
 'rg': 1984,  
 'pvn': 1991,  
 'abv': 1998,  
 'nm': 2014}
```

The tuples can be converted to dictionaries by passing the tuple name to the **dict()** function. This is achieved by nesting tuples within tuples, wherein each nested tuple item should have two items in it. The first item becomes the key and the second item as its value when the tuple gets converted to a dictionary.

Check your progress - 8:

Q43. In Python, Dictionaries and its keys both are immutable.

- a) a)True, True
- b) True, False
- c) False, True
- d) False, False

Q44. Select correct ways to create an empty dictionary:

- a) sampleDict = {}
- b) sampleDict = dict()
- c) sampleDict = dict{}

Q45. Items are accessed by their position in a dictionary and all the keys in a dictionary must be of the same type.

- a) True
- b) False

Q46. To obtain the number of entries in dictionary, d which command do we use?

- a) d.size()
- b) len(d)
- c) size(d)
- d) d.len()

Q47. Which one of the following is correct?

- a) In python, a dictionary can have two same keys with different values.
- b) In python, a dictionary can have two same values with different keys.
- c) In python, a dictionary can have two same keys or same values but cannot have two same key-value pair.
- d) In python, a dictionary can neither have two same keys nor two same

values.

Q48. What will be the output of above Python code?

```
d1={"abc":5,"def":6,"ghi":7}
```

```
print(d1[0])
```

- a) abc
- b) 5
- c) {"abc":5}
- d) Error

Q49. What will the above Python code do?

```
dict={"Phy":94,"Che":70,"Bio":82,"Eng":95}
```

```
dict.update({'Che':72,"Bio":80})
```

- a) It will create new dictionary as dict={"Che":72,"Bio":80} and old dict will be deleted.
- b) It will throw an error as dictionary cannot be updated.
- c) It will simply update the dictionary as dict={"Phy":94,"Che":72,"Bio":80,"Eng":95}
- d) It will not throw any error but it will not do any changes in dict

Q50. Select all correct ways to remove the key ‘marks’ from a dictionary

```
student = {  
    "name" : "Emma",  
    "class" : 9,  
    "marks" : 75  
}
```

- a) student.pop("marks")
- b) del student["marks"]
- c) student.popitem()
- d) dict1.remove("key2")

Q51. Select all correct ways to copy a dictionary in Python

- a) dict2 = dict1.copy()
- b) dict2 = dict(dict1.items())
- c) dict2 = dict(dict1)
- d) dict2 = dict1

10.8.4 Sets

A **set** is an unordered collection with no duplicate items. Primary uses of sets include membership testing and eliminating duplicate entries. Sets also support mathematical operations, such as union, intersection, difference, and symmetric difference.

Curly braces {} or the set() function can be used to create sets with a comma-separated list of items inside curly brackets {}. Note: to create an empty set you have to use set() and not {} as the latter creates an empty dictionary.

Sets are mutable. Indexing is not possible in sets, since set items are unordered. One cannot access or change an item of the set using indexing or slicing.

```
In[]: basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
```

```
In[]: basket
```

```
Out[]: {'apple', 'banana', 'orange', 'pear'}
```

A set is a collection of unique items. Duplicate items are removed from the set basket. Here, the set will contain only one item of ‘orange’ and ‘apple’.

```
In[]: 'orange' in basket
```

```
Out[]: True
```

```
In[]: 'grapes' in basket
```

```
Out[]: False
```

One can test for the presence of an item in a set using `in` and `not in` membership operators.

```
In[]: len(basket)
```

```
Out[]: 4
```

```
In[]: sorted(basket)
```

```
Out[]: ['apple', 'banana', 'orange', 'pear']
```

Total number of items in the set basket is found using the `len()` function. The `sorted()` function returns a new sorted list from items in the set.

```
In[]: a = set('abracadabra')
```

```
In[]: a
```

```
Out[]: {'a', 'b', 'd', 'k', 'r'}
```

```
In[]: b = set('alexander')
```

```
In[]: b
```

```
Out[]: {'a', 'd', 'e', 'l', 'n', 'r', 'x'}
```

```
In[]: a-b #letters present in set a, but not in set b
```

```
Out[]: {'b', 'k'}
```

```
In[]: a | b #Letters present in set a, set b, or both
```

```
Out[]: {'a', 'b', 'd', 'e', 'k', 'l', 'n', 'r', 'x'}
```

```
In[]: a & b #letters present in both set a and set b
```

```
Out[]: {'a', 'd', 'r'}
```

```
In[]: a ^ b #letters present in set a or set b, but not both
```

```
Out[]: {'b', 'e', 'k', 'l', 'n', 'x'}
```

10.8.4.1 Set Methods

A list of all the methods associated with the set can be obtained by passing the set function to dir().

Various methods associated with set are listed in the TABLE.

TABLE 17 : Various Set Methods

Set Methods	Syntax	Description
add()	set_name.add(item)	The add() method adds an item to the set set_name.
clear()	set_name.clear()	The clear() method removes all the items from the set set_name.
difference()	set_name.difference(*others)	The difference() method returns a new set with items in the set set_name that are not in the others sets.
discard()	set_name.discard(item)	The discard() method removes an item from the set set_name if it is present.
intersection()	set_name.intersection(*others)	The intersection() method returns a new set with items common to the set set_name and all other sets.
isdisjoint()	set_name.isdisjoint(other)	The isdisjoint() method returns True if the set set_name has no items in common with other set. Sets are disjoint if and only if their intersection is the empty set.
issubset()	set_name.issubset(other)	The issubset() method returns True if every item in the set set_name is in the other set.
issuperset()	set_name.issuperset(other)	The issuperset() method returns True if every element in other set is in the set set_name.
pop()	set_name.pop()	The method pop() removes and returns an arbitrary item from the set set_name. It raises KeyError if the set is empty.
remove()	set_name.remove(item)	The method remove() removes an item from the set set_name. It raises KeyError if

		the item is not contained in the set.
symmetric_difference()	set_name.symmetric_difference(other)	The method symmetric_difference() returns a new set with items from the set set_name and all other sets.
union()	set_name.union(*others)	The method union() returns a new set with items from the set set_name and all others sets.
update()	set_name.update(*others)	Update the set set_name by adding items from all others sets

Note: Replace the words “set_name”, “other” and “others” mentioned in the syntax with your actual set names in your code.

```
In[]: flowers1 = {'sunflower', 'roses', 'lavender', 'tulips',
'goldcrest'}
```

```
In[]: flowers2 = {'roses', 'tulips', 'lilies', 'daisies'}
```

```
In[]: flowers2.add('orchids')
```

```
In[]: flowers2.difference(flowers1)
```

```
Out[]: {'daisies', 'lilies', 'orchids'}
```

```
In[]: flowers2.intersection(flowers1)
```

```
Out[]: {'roses', 'tulips'}
```

```
In[]: flowers2.isdisjoint(flowers1)
```

```
Out[]: False
```

```
In[]: flowers2.issuperset(flowers1)
```

```
Out[]: False
```

```
In[]: flowers2.issubset(flowers1)
```

```
Out[]: False
```

```
In[]: flowers2.symmetric_difference(flowers1)
```

```
Out[]: {'daisies', 'goldcrest', 'lavender', 'lilies', 'orchids',
'sunflower'}
```

```
In[]: flowers2.union(flowers1)
```

```
Out[]:
```

```
{'daisies',
'goldcrest',
'lavender',
```

```
'lilies',
'orchids',
'roses',
'sunflower',
'tulips'}
```

```
In[]: flowers2.update(flowers1)
```

```
Out[]: flowers2
```

```
Out[]:
{'daisies',
'goldcrest',
'lavender',
'lilies',
'orchids',
'roses',
'sunflower',
'tulips'}
```

```
In[]: flowers2.discard("roses")
```

```
In[]: flowers2
```

```
Out[]:
{'daisies',
'goldcrest',
'lavender',
'lilies',
'orchids',
'sunflower',
'tulips'}
```

```
In[]: flowers1.pop()
```

```
Out[]: 'roses'
```

```
In[]: flowers2.clear()
```

```
In[]: flowers2
```

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

10.8.4.2 Traversing of Sets

One can iterate through each item in a set using a for loop.

```
In[]: for flower in flowers1 :
```

```
print(f"flower} is a flower")
```

```
Out[]: lavender is a flower
```

```
tulips is a flower
```

```
goldcrest is a flower
```

```
sunflower is a flower
```

10.8.4.3Frozenset

A **frozenset** is basically the same as a set, except that it is immutable. Once a frozenset is created, its items cannot be changed. Since they are immutable, they can be used as members in other sets and as dictionary keys. The frozensets have the same functions as normal sets, except none of the functions that change the contents (update, remove, pop, etc.) are available.

```
In[]: fs = frozenset(["g","o","o","d"])
#creating/declaring frozenset
```

```
In[]: fs
of frozenset
#displaying the contents
```

```
Out[]: frozenset({'d', 'g', 'o'})
```

```
In[]: pets = set([fs, "horse", "cow"])
#Frozenset type
is used within a set
```

```
In[]: pets
```

```
Out[]: {'cow', frozenset({'d', 'g', 'o'}), 'horse'}
```

```
In[]: lang_used = {"english":59, "french":29, "spanish":21}
```

```
In[]: frozenset(lang_used)
#keys in a dictionary are
returned when a dictionary is
passed as an argument to
frozenset() function.
```

```
Out[]: frozenset({'english', 'french', 'spanish'})
```

```
In[]: frs = frozenset(["german"])
```

```
#Frozenset is used as a key in dictionary
```

```
In[]: lang_used = {"english":59, "french":29, "spanish":21, frs:6}
```

```
In[]: lang_used
```

```
Out[]: {'english': 59, 'french': 29, 'spanish': 21, frozenset({'german'}): 6}
```

Check your progress - 9:

Q52. Select which is true for Python set:

- a) Sets are unordered.
- b) Set doesn't allow duplicate
- c) sets are written with curly brackets {}
- d) set Allows duplicate
- e) set is immutable
- f) a set object does support indexing

Q53. What is the output of the following union operation?:

```
set1 = {10, 20, 30, 40}
```

```
set2 = {50, 20, "10", 60}
```

```
set3 = set1.union(set2)
```

```
print(set3)
```

- a) {40, 10, 50, 20, 60, 30}
- b) {40, '10', 50, 20, 60, 30}
- c) {40, 10, '10', 50, 20, 60, 30}
- d) SyntaxError: Different types cannot be used with sets

Q54. What is the output of the following set operation?:

```
set1 = {"Yellow", "Orange", "Black"}
```

```
set2 = {"Orange", "Blue", "Pink"}
```

```
set3 = set2.difference(set1)
```

```
print(set3)
```

- a) {'Yellow', 'Black', 'Pink', 'Blue'}
- b) {'Pink', 'Blue'}
- c) {'Yellow', 'Black'}

Q55. What is the output of the following set operation?:

```
set1 = {"Yellow", "Orange", "Black"}
```

```
set1.discard("Blue")
```

```
print(set1)
```

- a) {'Yellow', 'Orange', 'Black'}
- b) KeyError: 'Blue'

Q56. The isdisjoint() method returns True if none of the items are present in both sets, otherwise, it returns False.

- a) True
- b) False

Q57. Select all the correct ways to copy two sets

- a) set2 = set1.copy()
- b) set2 = set1
- c) set2 = set(set1)
- d) set2.update(set1)

Q58. The symmetric_difference() method returns a set that contains all items from both sets, but not the items that are present in both sets.

- a) False
- b) True

10.9 CONTROL FLOW STATEMENTS

Python supports a set of control flow statements that you can integrate into your program. The statements inside your Python program are generally executed sequentially from top to bottom in the order that they appear. Apart from sequential control flow statements, you can employ decision making and looping control flow statements to break up the flow of execution thus enabling your program to conditionally execute particular block of code. The term control flow details the direction the program takes.

The control flow statements in Python Programming Language are:

10.9.1 Sequential Control Flow Statements :This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program.

10.9.2 Decision Control Flow Statements :Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if...else and if...elif...else).

10.9.3 Loop Control Flow Statements : This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (for loop and while loop). Loop Control Flow statements are also called Repetition statements or Iteration Statements.

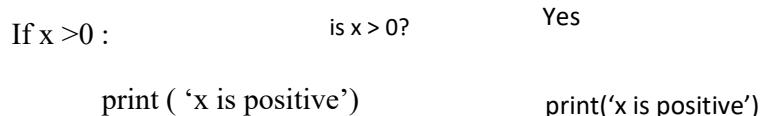
10.9.2.1 The if Decision Control Flow Statement

In order to write useful programs, we almost need the ability to check conditions and change the behavior of the program accordingly. The Decision Control Flow Statements give us this ability. The simplest form is the if statement.

The syntax for if statement is:

```
if Boolean_Expression :  
    statement(s)
```

The Boolean expression after the if statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the if statement are indented. The if statement decides whether to run statement (s) or not depending upon the value of the Boolean expression. If the Boolean expression evaluates to True then indented statements in the if block will be executed, otherwise if the result is False then none of the statements are executed. E.g.,

**Fig. 5: If Logic**

Here, depending on the value of x, print statement will be executed i.e. only if $x > 0$.

There is no limit on the number of statements that can appear in the body, but there must be at least one. In Python, the if block statements are determined through indentation and the first unindented statement marks the end. You don't need to use the `==` operator explicitly to check if the variable's value evaluates to True as the variable name can itself be used as a condition.

Program 10.5:Program reads your age and prints a message whether you are eligible to vote or not???

1. `age = int(input("Enter your age : "))`
2. `if age >= 18 :`
3. `print ("Congratulations!!! you can vote")`
4. `print ("!!! Thanks !!!")`
5. `print ("Voting is your birth right. Use judicially")`

Fig. 6: Screen Shot of execution of Program 10.5

```

age = int(input("Enter your age : "))
if age >= 18 :
    print ("Congratulations!!! you can vote")
    print ("!!! Thanks !!!")
print ("Voting is your birth right. Use judicially")

Enter your age : 22
Congratulations!!! you can vote
!!! Thanks !!!
Voting is your birth right. Use judicially

```

Here, condition or Boolean expression is true, indented block would be executed

```

Enter your age : 17
Voting is your birth right. Use judicially

```

Here, condition is False, so indented block would not be executed and only the unindented statement after the if block would be executed.

10.9.2.2 The if...else Decision Control Flow Statement

An if statement can also be followed by an else statement which is optional. An else statement does not have any condition. Statements in the if block are

executed if the Boolean_Expression is True. Use the optional else block to execute statements if the Boolean_Expression is False. The if...else statements allow for a two – way decision.

The syntax for if...else statement is:

```
if Boolean_Expression :
    statement_blk_1
else
    statement_blk_2
```

If the Boolean_Expression evaluates to True, then statement_blk_1 (single or multiple statements) is executed, otherwise it is evaluated to False then statement_blk_2 (single or multiple statements) is executed. Indentation is used to separate the blocks. After the execution of either statement_blk_1 or statement_blk_2, the control is transferred to the next statement after the if statement. Also, if and else keywords should be aligned at the same column position.

Program 10.6 : Program to find if a given number is odd or even

1. num = int(input("Enter a number : "))
2. if num % 2 == 0 :
3. print (num," is even number")
4. Else:
5. print (num," is odd number")

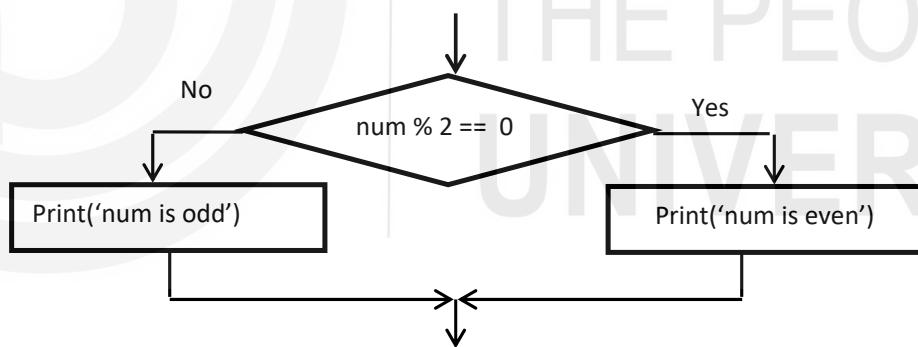


Fig. 6: IF – ELSE LOGIC

A number is read and stored in the variable named num. The num is checked using modulus operator to determine whether the num is perfectly divisible by 2 or not. Num entered is 15 which makes the evaluated expression False, so the else statement is executed and num is odd.

Fig. 7: Screen Shot of execution of Program 10.6

```
num = int(input("Enter a number : "))
if num % 2 == 0:
    print (num," is even number")
else:
    print (num," is odd number")
```

```
Enter a number : 15
15  is odd number
```

10.9.2.3 The if...elif...else Decision Control Statement

The if...elif...else is also called as multi-way decision control statement. When you need to choose from several possible alternatives, an elif statement is used along with an if statement. The keyword ‘elif’ is short for ‘else if’ and is useful to avoid excessive indentation. The else statement must always come last, and will again act as the default action.

The syntax for if...elif...else statement is,

```
if Boolean_Expression_1 :
    statement_blk_1
elif Boolean_Expression_2 :
    statement_blk_2
elif Boolean_Expression_3 :
    statement_blk_3
    :
    :
    :
else :
    statement_blk_last
```

This if...elif...else decision control statement is executed as follows:

- In the case of multiple Boolean expression, only the first logical Boolean expression which evaluates to True will be executed.
- If Boolean_Expression_1 is True, then statement_blk_1 is executed.
- If Boolean_Expression_1 is False and Boolean_Expression_2 is True, then statement_blk_2 is executed.
- If Boolean_Expression_1 and Boolean_Expression_2 is False and Boolean_Expression_3 is True, then statement_blk_3 is executed and so on.
- If none of the Boolean_Expression is True, then statement_blk_last is executed.

Program 10.7 : Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error. If the score is between 0.0 and 1.0, print a grade using the following table

Score	Grade
≥ 0.9	A
≥ 0.8	B
≥ 0.7	C
≥ 0.6	D
< 0.6	F

```

1. score = float(input("Enter your score : "))
2. if score < 0 or score >1 :
3.     print('Wrong Input')
4. elif score >= 0.9 :
5.     print('Your Grade is "A"')
6. elif score >= 0.8 :
7.     print('Your Grade is "B"')
8. elif score >= 0.7 :
9.     print('Your Grade is "C"')
10. elif score >= 0.6 :
11.    print('Your Grade is "D"')
12. else :
13.    print('Your Grade is "F"')

```

Fig. 9: Screen Shot of execution of Program 10.7

```

score = float(input("Enter your score : "))
if score < 0 or score > 1 :
    print('Wrong Input')
elif score >= 0.9 :
    print('Your Grade is "A"')
elif score >= 0.8 :
    print('Your Grade is "B"')
elif score >= 0.7 :
    print('Your Grade is "C"')
elif score >= 0.6 :
    print('Your Grade is "D"')
else :
    print('Your Grade is "F"')

```

Enter your score : 0.82
Your Grade is "B"

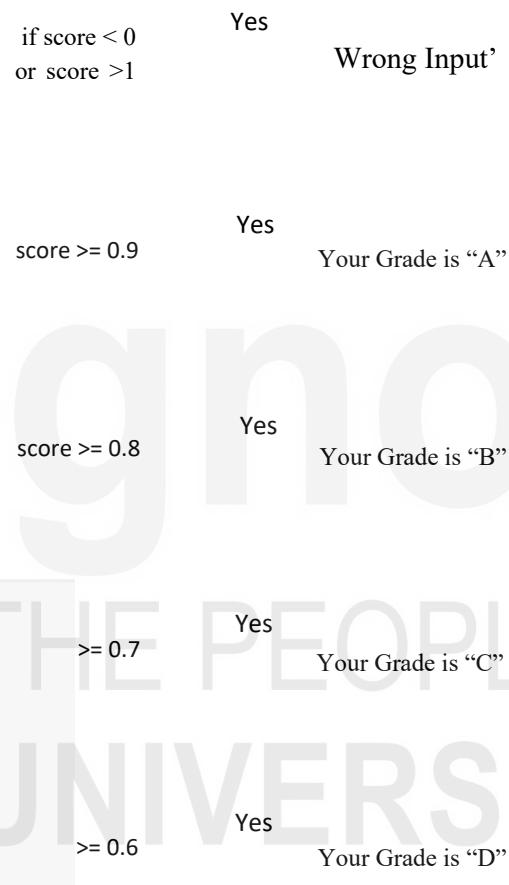


Fig.8 : IF – ELIF - ELSE LOGIC

10.9.2.4 Nested if Statement

In some situations, you have to place an if statement inside another statement. An if statement that contains another if statement either in its if block or else block is called a Nested if statement.

The syntax of the nested if statement is,

```

if Boolean_Expression_1 :
    if Boolean_Expression_2 :
        statement_blk_1
    else :
        statement_blk_2
else :
    statement_blk_3

```

If the Boolean_Expression_1 is evaluated to True, then the control shifts to Boolean_Expression_2 and if the expression is evaluated to True, then statement_blk_1 is executed. If the Boolean_Expression_2 is evaluated to False then the statement_blk_2 is executed. If the Boolean_Expression_1 is evaluated to False, then the statement_blk_3 is executed.

A three – branch example could be written as :

```

if x == y :
    print('x and y are equal')
else :
    if x < y :
        print('x is less than y')
    else :
        print('x is greater than y')

```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

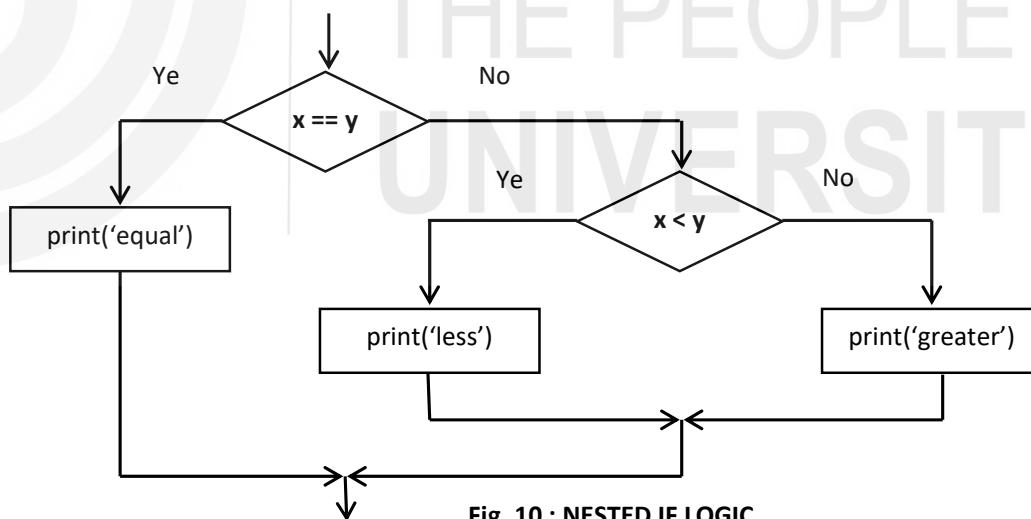


Fig. 10 : NESTED IF LOGIC

PROGRAM 10.8 : Program to check if a given year is a leap year

1. `year = int(input('Enter a year : '))`
2. `if year % 4 == 0 :`
3. `if year % 100 == 0 :`
4. `if year % 400 == 0 :`
5. `print(f'{year} is a Leap Year')`
6. `else :`
7. `print(f'{year} is not a Leap Year')`

```

8. else :
9.         print(f'{year} is a Leap Year')
10. else :
11.     print(f'{year} is not a Leap Year')

```

Fig. 11: Screen Shot of execution of Program 10.8

```

year = int(input('Enter a year : '))
if year % 4 == 0:
    if year % 100 == 0 :
        if year % 400 == 0 :
            print(f'{year} is a Leap Year')
        else :
            printf(f'{year} is not a Leap Year')
    else :
        print(f'{year} is a Leap Year')
else :
    print(f'{year} is not a Leap Year')

```

```

Enter a year : 2014
2014 is not a Leap Year

```

All years which are perfectly divisible by 4 are leap years except for century years (years ending with 00) which is a leap year only if it is perfectly divisible by 400. For example, years like 2012, 2004, 1968 are leap years but 1971, 2006 are not leap years. Similarly, 1200, 1600, 2000, 2400 are leap years but 1700, 1800, 1900 are not.

Although the indentation of the statements makes the structure apparent, nested conditional becomes difficult to read very quickly. In general, it is a good idea to avoid them when you can.

10.9.3.1 The while loop

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making error is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the while statement. The syntax for while loop is :

```

while Boolean_Expression :
    statement(s)

```

Program 10.9 : Program that counts down from five and then says “Blastoff!”

1. n = 5
2. while n > 0 :

3. print(n)
4. n = n - 1
5. print('Blastoff!')

Fig. 12: Screen Shot of execution of Program 10.9

```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Blastoff!')
```

```
5
4
3
2
1
Blastoff!
```

You can almost read the while statement as if it were English. It means, “While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!”

More formally, the flow of execution for a while statement is :

1. Evaluate the condition or the Boolean_Expression, yielding True or False.
2. If the Boolean_Expression is false, exit the while statement and then continue execution at the next statement.
3. If the Boolean_Expression is true, execute the body and then go back.

This type of flow is called a loop because the third step loops back around to the top. We call each time we execute the body of the loop an iteration. for the above loop, we would say, “It had five iterations”, which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition or the Boolean_Expression becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

PROGRAM 10.10 : Program to display the Fibonacci Sequences up to nth term where n is provided by the user

1. nterms = int(input('How many terms?'))
2. current = 0
3. previous = 1

```

4. next_term = 0
5. count = 0
6. if nterms<= 0 :
7.     print('Please enter a positive number')
8. elif nterms == 1 :
9.     print('Fibonacci Sequence')
10.    print('0')
11. else :
12.     print("Fibonacci Sequence")
13.     while count < nterms :
14.         print(next_term)
15.         current = next_term
16.     next_term = previous + current
17.     previous = current
18.     count += 1

```

Fig. 13: Screen Shot of execution of Program 10.10

```

nterms = int(input('How many terms? '))
current = 0
previous = 1
next_term = 0
count = 0
if nterms <= 0 :
    print('Please enter a positive number')
elif nterms == 1 :
    print('Fibonacci Sequence')
    print('0')
else :
    print("Fibonacci Sequence")
    while count < nterms :
        print(next_term)
        current = next_term
        next_term = previous + current
        previous = current
        count += 1

```

```

How many terms? 5
Fibonacci Sequence
0
1
1
2
3

```

In a Fibonacci sequence, the next number is obtained by adding the previous two numbers. The first two numbers of the Fibonacci sequence are 0 and 1. The next number is obtained by adding 1 and 1 which is 1. Again, the next number is obtained by adding 1 and 1 which is 2 and so on. Get a number from user up to which you want to generate Fibonacci sequence. Assign

values to variables current, previous, next_term and count. The variable count keeps track of number of times the while block is executed. User is required to enter a positive number to generate a single number in the sequence, then print zero. The next_term is obtained by adding the previous and current variables and the statements in the while block are repeated until while block conditional expression becomes False.

10.9.3.2 The for loop

Sometimes we want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a definite loop using a for statement. We call the while statement an indefinite loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax for the for loop is :

for iteration_variable in sequence :

statement(s)

The for loop starts with for keyword and ends with a colon. The first item in the sequence gets assigned to the iteration variable iteration_variable. Here, iteration_variable can be any valid variable name. Then the statement block is executed. This process of assigning items from the sequence to the iteration_variable and then executing the statement continues until all the items in the sequence are completed.

The for loop is incomplete without the use of **range()** function which is a built-in function. It is very useful in demonstrating for loop. The range() function generates a sequence of numbers which can be iterated through using for loop. the syntax for range() function is,

range([start ,] stop [, step])

Both start and step arguments are optional and is represented with the help of square brackets and the range argument value should always be an integer.

start → value indicates the beginning of the sequence. If the start argument is not specified, then the sequence of numbers start from zero by default.

stop → generates numbers up to this value but not including the number itself.

step → indicates the difference between every two consecutive numbers in the sequence. The step value can be both negative and positive but not zero.

PROGRAM 10.11 : Program to find the sum of all odd and even numbers up to a number specified by the user.

1. `number = int(input("Enter a number"))`
2. `even = 0`

```

3. odd = 0
4. for i in range(number) :
5.     if i % 2 == 0 :
6.         even = even + i
7. else :
8.     odd = odd + i
9. print(f'Sum of Even numbers are {even} and Odd numbers are {odd}')

```

Fig. 14: Screen Shot of execution of Program 10.11

```

number = int(input("Enter a number"))
even = 0
odd = 0
for i in range(number) :
    if i % 2 == 0 :
        even = even + i
    else :
        odd = odd + i
print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")

```

```

Enter a number10
Sum of Even numbers are 20 and Odd numbers are 25

```

A range of numbers are generated using range() function. As only stop value is indicated in the range() function, so numbers from 0 to 9 are generated. Then the generated numbers are segregated as odd or even by using the modulus operator in the for loop. All the even numbers are added up and assigned to even variable and odd numbers are added up and assigned to odd variable and print the result. The for loop will be executed / iterated number of times entered by user which is 10 in this case.

PROGRAM 10.12 : Program to find the factorial of a number.

```

1. number = int(input('Enter a number'))
2. factorial = 1
3. if number <0 :
4.     print("Factorial doesn't exist for negative numbers")
5. elif number == 0 :
6.     print("The factorial of 0 is 1")
7. else :
8.     for i in range(1, number + 1) :
9.         factorial = factorial * i
10.    print(f'The factorial of number {number} is {factorial}')

```

Fig. 15: Screen Shot of execution of Program 10.12

```
number = int(input('Enter a number'))
factorial = 1
if number < 0 :
    print("Factorial doesn't exist for negative numbers")
elif number == 0 :
    print("The factorial of 0 is 1")
else :
    for i in range(1, number + 1) :
        factorial = factorial * i
print(f"The factorial of number {number} is {factorial}")
```

```
Enter a number5
The factorial of number 5 is 120
```

Read the number from user. A value of 1 is assigned to variable factorial. To find the factorial of a number it has to be checked for a non – negative integer. If the user entered number is zero then the factorial is 1. To generate numbers from 1 to the user entered number range() function is used. Every number is multiplied with the factorial variable and is assigned to the factorial variable inside the for loop. The for loop block is repeated for all the numbers starting from 1 up to the user entered number. Finally, the factorial value is printed.

10.9.3.3 The continue and break Statements

The break and continue statements provide greater control over the execution of code in a loop. Whenever the break statement is encountered, the execution control immediately jumps to the first instruction following the loop. To pass control to the next iteration without exiting the loop, use the continue statement. Both continue and break statements can be used in while and for loops.

PROGRAM 10.13 : Program that prints integers from zero to 5.

1. count = 0
2. while True :
3. count += 1
4. if count > 5 :
5. break
6. print (count)

Fig. 16: Screen Shot of execution of Program 10.13

```
count = 0
while True :
    count += 1
    if count > 5 :
        break
    print (count)
```

```
1
2
3
4
5
```

In the while loop, as soon as count value reaches 6, because of break statement print statement will not be executed and control will come out of the while loop.

PROGRAM 10.14 : Program that processes only odd integers from 0 to 10.

1. count = 0
2. while count < 10 :
3. count += 1
4. if count % 2 == 0 :
5. continue
6. print (count)

Fig. 17: Screen Shot of execution of Program 10.14

```
count = 0
while count < 10 :
    count += 1
    if count % 2 == 0 :
        continue
    print (count)
```

```
1
3
5
7
9
```

The continue statement in the while loop will not let the print statement to execute if the count value is even and control will go for next iteration. Here, loop will go through all the iterations and the continue statement will affect only the statements in the loop to be executed or not occurring after the continue statement. Whereas, the break statement will not let the loop to complete its iterations depending on the condition specified and control is transferred to the statement outside the loop.

PROGRAM 10.15 : Program to check whether a number is prime or not

```
1. number = int(input('Enter a number: '))
2. prime = True
3. for i in range(2, number) :
4.     if number % i == 0 :
5.         prime = False
6.         break
7. if prime :
8.     print(f'{number} is a prime number')
9. else :
10.    print(f'{number} is not a prime number')
```

Fig. 18: Screen Shot of Program 10.15

```
number = int(input('Enter a number: '))
prime = True
for i in range(2, number) :
    if number % i == 0 :
        prime = False
    break
if prime :
    print(f'{number} is a prime number')
else :
    print(f'{number} is not a prime number')

Enter a number: 9
9 is a prime number
```

Check your progress-10 :

Q59. In a python program, a control structure:

- a) Defines program-specific data structures
- b) Directs the order of execution of the statements in the program
- c) Dictates what happens before the program starts and after it terminates
- d) None of the above

Q60. Which of the following is False regarding loops in python?

- a) Loops are used to perform certain tasks repeatedly
- b) While loop is used when multiple statements are to be executed repeatedly until the given condition becomes False
- c) While loop is used when multiple statements are to be executed repeatedly until the given condition becomes True.
- d) for loop can be used to iterate through the elements of lists.

Q61. We can write if/else into one line in python.

- a) True
- b) False

Q62. Given the nested if-else below, what will be the value x when the code executes successfully?

x = 0

```
a = 5
b = 5
if a > 0:
    if b < 0:
        x = x + 5
    elif a > 5:
        x = x + 4
    else:
        x = x + 3
else:
    x = x + 2
print(x)
a) 0
b) 4
c) 2
d) 3
```

Q63. if -3 will evaluate to true

- a) True
- b) False

Q64. What is the output of the following nested loop

numbers = [10, 20]

items = ["Chair", "Table"]

for x in numbers:

 for y in items:

print(x, y)

- a) 10 Chair

10 Table

20 Chair

20 Table

- b) 10 Chair

20 Chair

10 Table

20 Table

Q65. What is the value of x?:

X = 0

While (x < 100):

 x+=2

print(x)

- a) 101

- b) 99

- c) None of the above, this is an infinite loop

- d) 100

Q66. What is the value of the var after the for loop completes its execution?

```
var = 10
for i in range(10):
    for j in range(2, 10, 1):
        if var % 2 == 0:
            continue
        var += 1
    var+=1
else:
    var+=1
print(var)
a) 20
b) 21
c) 10
d) 30
```

Q67. Find the output of the following program:

```
a = {i : i * i for i in range(6)}
print (a)
```

Dictionary comprehension doesn't exist

- a) {0:0, 1:1, 2:4, 3:9, 4:16, 5:25, 6:36}
- b) {0:0, 1:1, 4:4, 9:9, 16:16, 25:25}
- c) {0:0, 1:1, 2:4, 3:9, 4:16, 5:25}

10.10 SUMMARY

After the completion of this chapter, I am sure, you would be able to learn and understand the basics of Python language. Using the contents explained in this chapter, you would be able to turn your logic into codes where identifiers and variables would help you to form statements and expressions using operators, lists, tuples, sets and dictionaries which are the back bone of Python language which makes it unique and easy to program with. All the basic structures of Python can be bound together with the control flow statements which ultimately form a Python program. However, this is not all; there is still a long way to go. Python has many more unique features which make it a programmer's language. So, Happy Programming!

SOLUTIONS TO CHECK YOUR PROGRESS

Data Structures and
Control Statements
in Python

Answers to Check your Progress 1 to 10 :

1. a	2. d	3. d	4. a	5. c	6. d
7. b	8. e	9. b	10. a	11. b	12. a
13. d	14. c	15. b	16. e	17. d	18. a
19. a	20. d	21. b	22. c	23. c	24. c
25. b	26. b	27. b	28. b	29. c	30. b
31. c	32. b	33. b	34. c	35. a	36. b
37. c	38. d	39. a,c	40. b	41. b	42. c
43. c	44. a,b	45. b	46. b	47. b	48. d
49. c	50. a,b,c	51. a,b,c	52. a,b,c	53. c	54. b
55. a	56. a,c,d	57. a,c,d	58. b	59. b	60. b
61. a	62. d	63. a	64. a	65. d	66. b
67. d					

IGNOU
THE PEOPLE'S
UNIVERSITY

UNIT 11 FUNCTIONS AND FILE HANDLING IN PYTHON

**Functions and Files
Handling in Python**

Structure

- 11.0 Introduction
 - 11.1 Objectives
 - 11.2 Function definition and calling
 - 11.3 Function Scope
 - 11.4 Function arguments
 - 11.5 Returning from a function
 - 11.6 Function objects
 - 11.7 Lambda / Anonymous Functions
 - 11.8 File Operations
 - 11.9 Summary
-

11.0 INTRODUCTION

Python provides a way of organizing the tasks into more manageable units called functions. Functions make the code modular which is one of the characteristics of an object oriented programming language (OOPs). Modularity is the process of decomposing a problem into set of sub-problems so as to reduce the complexity of a problem. It also makes the code reusable. File handling is another important aspect discussed in this unit. File handling includes- creation, deletion and manipulations of files using python programming.

11.1 OBJECTIVES

After completing this unit, you will be able to

- Perform functions definition and calling
 - Understand scope of functions
 - Define function objects
 - Create lambda functions
 - Understand basic file operations
-

11.2 FUNCTION DEFINITION AND CALLING

A function is block of code that performs a specific task. When the size of a program increases, its complexity also increases. Hence, it becomes important to organize the program into more manageable units. Further, it makes the code reusable.

There are three types of functions in python-

1. Built-in functions – these are the functions which are already defined in the language. Example print(), max(), input(), etc.
2. User Defined functions- these are the functions that can be created or defined by the users according to their needs.
3. Anonymous functions

11.2.1 Creating user defined functions

A function can be defined using `def` statement and giving suitable name to a function by following the rules of identifiers. The process of defining a function is called *function definition*.

Syntax of function definition

```
Def function_name( list_of_parameters ) :  
Statement (s)
```

Where,

`function_name()` - is the name of the function. A function name must be followed by a set of parenthesis. Any name can be given to a function following the rules of identifiers.

`List_of_parameters` - is an optional field which is used to pass values or inputs to a function. It can be none or a comma separated list of variables.

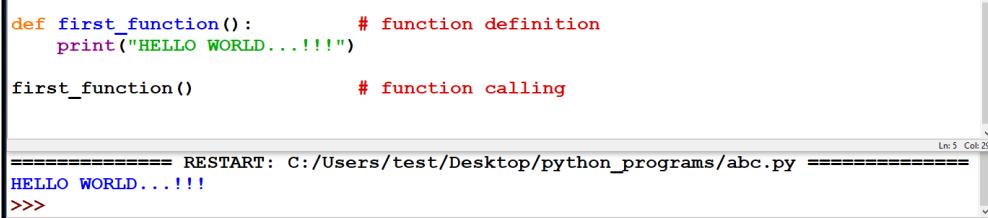
`Statements(s)` – is a set of statements or commands within the function. Each time the function is called, all the statements will be executed. These statements together form the body of a function.

Even if the function is defined, it can never be executed till the time it is called. Hence, for using a function, it must be called using *function call statement*. A function can be called by its name with set of parenthesis and optional list of arguments.

Syntax of Function Calling

```
function_name(list_of_arguments)
```

Example1. A function to display HELLO WORLD



```
def first_function():      # function definition
    print("HELLO WORLD...!!!")

first_function()          # function calling

=====
===== RESTART: C:/Users/test/Desktop/python_programs/abc.py ======
HELLO WORLD...!!!
>>>
```

Function needs to be defined only once, but it can be called any number of times by using calling statements. They can not only be called in the same program, but they can also be called in different programs. This will further be discussed in the next chapter.

Another example shown below is the program having a function to calculate factorial of a number.

Example2. A function to display factorial of a number.

```
def factorial():      # function definition
    fact=1
    n=int(input("Enter a number :"))
    for i in range(1,n+1):
        fact=fact*i
    print("Factorial of ",n," is ",fact)

factorial()          # function calling

=====
RESTART: C:/Users/test/Desktop/python_programs/factorial.py =====
Enter a number :4
Factorial of 4 is 24
>>>
```

11.3 FUNCTION SCOPE

Part of a code in which a variable can be accessed is called *Scope of a variable*. Scope of a variable can be global or local.

Local variables are the variable created within a function's body or function's scope. They cannot be accessed outside the function. Their scope is only limited to the function in which they are created.

Global variables are the variable created outside any function. Their scope is global, hence can be used anywhere. Global variables can also be created within function using keyword *global*.

Example 3. Use of local and global variables.

```
x=1                      # global variable
def test():
    y=2                  # local variable
    print("Global inside function x=",x) # both local and global can be used here
    print("Local inside function y=",y)

test()
print("Global outside function x=",x)
print("Local outside function y=",y) # shows error:local variable cannot be used here

=====
RESTART: C:/Users/test/Desktop/python_programs/test.py =====
Global inside function x= 1
Local inside function y= 2
Global outside function x= 1
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/test.py", line 9, in <module>
    print("Local outside function y=",y)      # shows error: local variable cannot be used here
NameError: name 'y' is not defined
>>>
```

In Example 3, variable created x is global, hence, can be used both inside and outside any function. Variable y is created inside function test(), therefore, its scope is only limited to this function and hence cannot be used outside. Global variables can be created in functions using *global* keyword as shown in example 4.

Example 4: Program to create global variable inside function.

```
x=1                                # global variable
def test():
    global x                         # to access global variable for modification
    print("inside function before modification x=",x)
    x=x+2
    print("inside function after modification x=",x)

test()
print("outside function x=",x)
=====
===== RESTART: C:/Users/test/Desktop/python_programs/test.py ======
inside function before modification x= 1
inside function after modification x= 3
outside function x= 3
Ln: 10
```

If we create a variable inside function having same name as that of global variable, then a separate variable of same name gets created. Function in this case can access local variable and hence, cannot refer to global variable as shown in example 5.

Example 5: Program showing use of Local and global variables of same name.

```
x=1                                # global variable
def test():
    x=2                         # local variable
    print("inside function x=",x)

test()
print("outside function x=",x)
=====
===== RESTART: C:/Users/test/Desktop/python_programs/test.py ======
inside function x= 2
outside function x= 1
>>>
Ln: 6 Col: 0
```

In python, global variable can be accessed in function directly (as shown in Example 3) but they cannot be directly modified inside the function (shown in example 6). Then how can we modify global variables inside function? Answer to this question is through *global* keyword. It can not only be used for creating global variables inside functions, but it can also be used for modifying global variables inside functions. See example 7.

Example 6: Program to show modification of global variable is not allowed inside function

```
x=1                                # global variable
def test():
    x=x+2      # shows error: modification of global variable is not allowed
    print("inside function x=",x)

test()
print("outside function x=",x)
=====
===== RESTART: C:/Users/test/Desktop/python_programs/test.py ======
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/test.py", line 6, in <module>
    test()
  File "C:/Users/test/Desktop/python_programs/test.py", line 3, in test
    x=x+2      # shows error: modification of global variable is not allowed
UnboundLocalError: local variable 'x' referenced before assignment
>>>
Ln: 8 Col: 0
```

Example 7: Program to show modification of global variable using global keyword

```
x=1                                # global variable
def test():
    global x                      # to access global variable for modification
    print("inside function before modification x=",x)
    x=x+2
    print("inside function after modification x=",x)

test()
print("outside function x=",x)
```

===== RESTART: C:/Users/test/Desktop/python_programs/test.py =====

inside function before modification x= 1
inside function after modification x= 3
outside function x= 3

Check your Progress 1

Ex1. What are the benefits of using functions? Also differentiate between function definition and function calling.

Ex2. What is Scope of a variable? Explain local and global variables with example.

Ex3. Write a function named *pattern* to display the pattern given below-

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

11.4 FUNCTION ARGUMENTS

Arguments are a way of passing values or input to a function. Arguments are passed to a function during function calls. Their values replace the functions parameters in the function definition.

Python provides various mechanisms of passing arguments to a function.

1. Required arguments
2. Default arguments
3. Keyword arguments
4. Arbitrary arguments

1. Required Arguments or Positional Arguments

In required arguments, number of arguments should match number of function parameters and should be given in same order. If number of arguments is not the same, then error will be shown and if correct order is not followed then output can be incorrect.

Example 8: Passing arguments to a function.

```
def calculator(a,b):
    print("Addition      : ",a+b)
    print("Subtraction   : ",a-b)
    print("Multiplication: ",a*b)
    print("Division      : ",a/b)

calculator(5,2)

=====
RESTART: C:/Users/test/Desktop/python_programs/arguments.py =====
Addition      : 7
Subtraction   : 3
Multiplication: 10
Division      : 2.5
>>>
```

2. Default arguments

Default values of variables can be given in the parameters itself. So if any argument is not given at the calling time, it will be replaced by the default value. Default values can be given for any number of arguments. It must be taken care that default argument must follow non-default arguments.

Example 9: Use of Default Arguments

```
def simple_interest(p,r=2,t=5):
    interest=(p*r*t)/100
    print(" Simple Interest is:",interest)

simple_interest(1000)      # p=1000, r=2 (default) ,t=5(default)
simple_interest(1000,5)    # p=1000, r=5 ,t=5(default)
simple_interest(1000,5,10) # p=1000, r=2 ,t=5

=====
RESTART: C:\Users\test\Desktop\python_programs\arguments.py =====
Simple Interest is: 100.0
Simple Interest is: 250.0
Simple Interest is: 500.0
>>>
```

3. Keyword Arguments

We have seen above that arguments must be given in the order in which parameters are defined otherwise the result will be effected. Keyword arguments are one way by which we can give arguments in any order. They are given by specifying the parameter name with each argument during function call. In this case position does not matter but name matters hence they are also called named arguments.

Example 10: Use of Keyword Arguments

```
def compound_interest(p,r,t):
    amount = p*pow(1+ (r/100) ,t)
    interest = amount-p
    print("Compound Interest is:",interest)

compound_interest(r=13,p=100000,t=5)      # passing keyword arguments
compound_interest(100000,13,5)          # passing positional arguments

=====
RESTART: C:/Users/test/Desktop/python_programs/compound_interest.py ====
Compound Interest is: 84243.51792999991
Compound Interest is: 84243.51792999991
>>>
```

4. Arbitrary Arguments

Sometimes it is required to pass different number of arguments to a same function or it is not known at function definition time that how many arguments could be used at calling time. Python provides the feature of arbitrary arguments to deal with this issue. Arbitrary arguments can be declared using * symbol.

Example11: Use of Arbitrary Arguments

```
def sum(*a):           # declaration of arbitrary argument
    sum=0
    for i in a:
        sum=sum+i
    print("Sum of Series",a," is:",sum)

sum(1,2,5,7,8,10,2)    # passing arbitrary arguments
sum(10,20,25)

=====
RESTART: C:/Users/test/Desktop/python_programs/Arbitrary.py ====
Sum of Series (1, 2, 5, 7, 8, 10, 2)  is: 35
Sum of Series (10, 20, 25)  is: 55
>>>
```

11.5 RETURNING FROM A FUNCTION

As we can pass values as input to function parameters, similarly we can also return or extract values out of a function. This can be done using *return* statement. By default when we do not include return statement, value None is returned from the function.

Syntax of returning from function

```
def function_name( arg1,arg2.... ) :
    .....
    .....
    return value
```

Example12: Function with return statement

```
def example(a,b,c):      # function with return statement
    avg=(a+b+c)/3
    return avg

result=example(2,3,4)      # returned value passed to result variable

print("AVG OF NUMBERS:",result) # display return value by print statement

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
AVG OF NUMBERS: 3.0
>>>
```

There can be only one return statement in a function. It doesn't mean that we can return only one value. It means that the return statement can return only one object and object in python can contain single (variable) or multiple values (List, tuple). Hence, we can return multiple values with a single return statement.See example13.

Example 13: Function returning more than one value

```
def example(a,b,c):
    sum=a+b+c
    avg=sum/3
    return sum,avg      # function returning two values with single statement

result1,result2=example(2,3,4)
print("SUM OF NUMBERS:",result1)
print("AVG OF NUMBERS:",result2)

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
SUM OF NUMBERS: 9
AVG OF NUMBERS: 3.0
>>>
```

Various benefits of using return statement -

1. Values returned by function can be used again in the rest of the program.
2. They can also be passed as argument to other functions providing better flow of control. See example 14.
3. They can also be used to break out of function in the middle.

Example 14: Function returning more than one value

```
def first(a,b):
    sum=a+b
    return sum

def second(a,b):
    avg=a/b
    return avg

value1=float(input("Enter first input:"))
value2=float(input("Enter second input:"))
result1=first(value1,value2) # result1 contains value return from first() function
result2=second(result1,2)   # result1 is passed as argument to second() function
print("AVG OF NUMBERS:",result2) # display return value by print statement

print("10 / 2 =:",second(10,2)) # function with return statement can also be
                                # directly called with print statement

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
Enter first input:3
Enter second input:6
AVG OF NUMBERS: 4.5
10 / 2 =: 5.0
```

It is also possible to break out of a function in the middle of statements in function using return. The statements following the return statement will never be executed. In example 15, inside a function, there is loop ranging from 1 to 9, but it will only be executed 4 times, since the function will return when value of i reaches 5, hence remaining iterations will be skipped.

Example 15: Function returning in between the loop

```
def example():
    for i in range(1,10):
        if(i==5):
            return      # function returns when i=5, no further execution
        else:
            print(i)

example()

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
1
2
3
4
>>>
```

Check your Progress 2

Ex 1. What are the various ways of passing arguments to a function?

Ex 2. Write a function which takes list of numbers as argument and returns a list of unique elements from it.

Ex 3. Write a function to display all prime numbers between a range which is passed as arguments.

11.6 FUNCTION OBJECTS

In python, data is represented as objects. Like Lists, Strings etc, functions are also treated as objects. Functions in python are first class objects. Since functions are objects, it provides various features with additional to the existing ones. These are-

1. functions can be assigned to a variable
2. functions can be used as elements of data structures
3. functions can be sent as arguments to another function
4. functions can be nested

1. Functions assigned to variable

As functions are object, they can be assigned to a variable (object). Hence, function can then be called using variable and set of parenthesis. Assigning function to a variable creates a reference to the same function or a function with two names. If one function or variable is deleted, function can still be referenced by another name. Following is the example of function assigned to a variable. (example 16)

Example 16: Function assigned to variable

```

def example():
    print("Testing function")

a = example      # function assigned to variable
a()              # function call using variable

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py ======
Testing function
>>>

```

In the following console window it can be clearly seen that after deleting a function, it cannot be accessed with the same name. But it can still be called using another name as shown in example 17.

Example 17: Deleting reference to a function

```

>>> del example
>>> example()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    example()
NameError: name 'example' is not defined
>>> a()
Testing function
>>> |

```

2. Functions as element of data structure

Functions can be passed as an element to any data structure. This is very useful at times when we want to apply different functions to same input. Below is the example 18 to show that. We are using various built-in functions to demonstrate that.

Example 18: Function assigned as elements to List

```

abc=[str.upper,str.lower,len]  # list of functions

for i in abc:
    print(i,i("function as element"))# applying different function to same input

===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py ======
<method 'upper' of 'str' objects> FUNCTION AS ELEMENT
<method 'lower' of 'str' objects> function as element
<built-in function len> 19
>>>

```

3. Function as argument to another function

Objects can be passed as arguments to a function. Since functions are objects in python, they can also be passed as arguments to functions. This technique allows application of multiple operations on a particular set of inputs. Given below is an example of function calling another function as arguments.

Example 19: Function calling another function as argument

```

def add(n):
    sum=0
    for i in n:
        sum=sum+i
    return sum

def display(l,func):
    print("Addition of elements:",func(l))

display([1,2,3,4,5,6,7],add)      # function display() calling function add()

```

```

Ln: 13 Col: 0
===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
Addition of elements: 28
>>>
Ln: 212 Col: 34

```

4. Nested function

Functions created within another function are called nested function. This is one of the unique properties in python since functions are objects. The inner function which is created within some outer function has access to all the variables of enclosing scope. Inner functions can never be directly called outside outer function. Hence, it provides security feature called Encapsulation. It can only be called by the use of enclosing function.

Example 20: Nested function

```

def outer(text):          # outer function
    print(str.upper(text))

    def inner():           # nested or inner function
        print(str.lower(text))

    inner()    # nested function can only be called inside outer function

outer("Nested Function") # call to outer function makes indirect call to inner
inner("Nested Function") # shows error: nested function cannot be called outside

```

```

Ln: 10 Col: 80
===== RESTART: C:/Users/test/Desktop/python_programs/return_example.py =====
NESTED FUNCTION
nested function
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/return_example.py", line 12, in <module>
    inner("Nested Function") # shows error: nested function cannot be called outside
NameError: name 'inner' is not defined
>>>

```

11.7 LAMBDA / ANONYMOUS FUNCTIONS

Python allows to create functions without names called Anonymous functions. They are not declared with `def` keyword, instead `lambda` keyword is used to create these functions. Hence, these functions are also called **lambda functions**. Like other

functions, these functions can also contain arguments but there can be only one statement in the body of these functions. Statement is evaluated and the result is returned. There are situations when a function needs to be created for a short operation (similar to macros in C language) or usable for a short period of time, in that case lambda functions are best suited.

Syntax of Anonymous function

```
Variable = lambda arg : statement
```

Here,arg is one or more arguments passed to a function and statement consists of operation performed by a function.

Given below is example 21 in which two lambda functions are created. First function calculates cube of a given argument and second function gives addition to two given arguments. Lambda function is then assigned to a variable (function assigned to variable, discussed in previous section), which can then be used to call function with arguments and set of parenthesis.

Example 21: Use of Lambda function

```
cube = lambda x : x*x*x      # lambda function to calculate cube
add = lambda x,y: x+y        # lambda function to calculate sum

print("cube of 3:",cube(3))   # first lambda function called
print("addition of 10 & 20:",add(10,20 # second lambda function called

=====
RESTART: C:/Users/test/Desktop/python_programs/lambda.py =====
cube of 3: 27
addition of 10 & 20: 30
>>>
```

Lambda functions are mostly used as argument to other high-order function like map () and filter () function.

map() function : map () is a built-in function which takes two arguments, first argument given is function (can be lambda function), second argument is given as a list. map () function returns a list of elements obtained by applying given function to each element of a list.

filter() function : filter() is another built-in function which takes two arguments. First argument is a function and second is list of elements, just as map() function. Here, given function is applied to each of the elements of list and returns a list of items for which the function evaluates to True.

Syntax of map and filter function

```
map ( function , [ list ] )
```

```
filter ( function , [ list ] )
```

Example 22: Application of map() function with lambda function

```
def factorial(n):
    fact = 1
    for i in range(1,n+1):
        fact = fact*i
    return fact

a=[1,2,3,4,5]

b = list(map(lambda x : x*x*x,a)) # map() applied on lambda function
c = list(map(factorial,a))         # map() applied on user defined function

print("CUBE OF SERIES",b)
print("FACTORIAL OF SERIES",c)

===== RESTART: C:/Users/test/Desktop/python_programs/lambda.py =====
CUBE OF SERIES [1, 8, 27, 64, 125]
FACTORIAL OF SERIES [1, 2, 6, 24, 120]
>>>
```

Example 22: Application of filter() function with lambda function

```
a=[1,2,3,4,5,6,7,8,9,10]

b = list(filter(lambda x : x%2==0,a)) # filter() applied on lambda function

print("EVEN NUMBER SERIES",b)

===== RESTART: C:/Users/test/Desktop/python_programs/lambda.py =====
EVEN NUMBER SERIES [2, 4, 6, 8, 10]
>>>
```

Check your Progress 3

Ex 1. Write a program to find cube of numbers in a list using lambda function.

Ex 2. Write a program to add two given lists using map function.

Ex 3. Write a program to display vowels from a given list of characters using filter function.

11.8 FILE OPERATIONS

Main memory is volatile in nature, hence, nothing can be stored permanently. File handling is a very important functionality which must be provided by any language to deal with this problem. Inputs can be taken from files instead of users, output can be displayed and saved permanently in files which can further be accessed later and appended or updated. All these functionalities come under file handling. Like other languages, python also provides various built-in functions for file handling operations.

There are two types of files supported by python- Binary and Text.

Binary files – These are the files that can be represented as 0's and 1's. These files can be processed by applications knowing about the file's structure. Image files are example of binary files.

Text files – These files are organized as sequence of characters. Here, each line is separated by a special end of line character.

Any file handling operation can be performed in three steps-

1. Opening a file
2. Operating on file – read , write , append etc
3. Closing a file

1. Opening a file

A file can be opened in a Python using built-in function *open()*. By default the files are opened in read text ('r') file mode.

Syntax of open() function

```
file = open ( "file_name.ext ","mode_of_operation")
```

Where,

File_name.ext - is the name of the file to be opened and ext is the extension of file.

Mode_of_operation - is the mode in which file is needed to be opened.

file – is the object returned by the function. This object can be used for further operations on file

The possible mode of operations in python –

Mode of opening file	Functionality
'r'	Opens a file for reading. It is default mode
'w'	Opens a file for writing. Overwrites a file if already exists. Creates new file if does not exist.
'a'	Opens a file for appending.
'r+'	Opens a file for both reading and writing.
'w+'	Same as 'r+' but creates new file if does not exist and overwrites if exists
'a+'	Opens a file for appending and reading
'x'	Creates new file. Fails if already exists. Added in python 3.
'rb'	Opens a file for reading in binary mode.
'wb'	Same as 'w' except data is in binary
'ab'	Same as 'a' except data is in binary

2. Operating on file

There are various operations -

2.1 Reading from a file

There are many ways to read from a file. Given below are the functions available for reading from a file.

In the below table assume that *file* is the object returned from `open()` function.

Function	Syntax	Description
<code>read()</code>	<code>file.read()</code>	Returns entire file contents as a string
	<code>file.read(n)</code>	Returns n characters from beginning of file as string
<code>readline()</code>	<code>file.readline()</code>	Returns single line of file at a time. First line in this case.
<code>readlines()</code>	<code>file.readlines()</code>	Returns a list of all lines

2.2 Writing to a file

Similar to read operations, functions are there in python to write data to file.

Function	Syntax	Description
<code>write()</code>	<code>file.write("text")</code>	Writes text or string to a file

2.3 Operations on file

Python allows many other operations to be done in with files. Listed below are some functions used for file handling.

Function	Syntax	Description
<code>tell()</code>	<code>file.tell()</code>	Returns the current cursor position in file
<code>seek()</code>	<code>file.seek(pos)</code>	Places the file cursor to the position pos
<code>os.stat()</code>	<code>os.stat(filename)</code>	This method is used to get display status of the file given as argument.

os.path.exists()	os.path.exists('arg')	Returns true if the file or directory of name 'arg' exist
os.path.isfile()	os.path.isfile('arg')	Returns true if the file of name 'arg' exists
os.path.isdir()	os.path.isdir('arg')	Returns true if the directory of name 'arg' exists
shutil.copy()	shutil.copy(src,dst)	Copies a file from src to des file.
os.path.getsize()	os.path.getsize(filename)	Returns the size of file

3. Closing a file

After completing all the operations on file, it must be closed properly. This step frees up the resources and allows graceful termination of file operations.

Syntax of closing a file

```
file.close()
```

11.8.1 Reading data from a file

There are various ways to read a file by following the steps explained in the above section. For reading data from the file, it must be existing. Below is given the program (Example 23) to read a file named "first.txt" in the directory files within the present directory.

Example 23: Program to display file contents

```
file = open ("files\\first.txt","r") # step 1: opening file for reading
s = file.read()                      # step 2: reading entire file
print(s)                             # displaying file contents
file.close()                          # step 3: closing file

===== RESTART: C:/Users/test/Desktop/python_programs/file.py ======
Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.[28]

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural,) object-oriented, and functional programming. Python is often described as a "batteries included"
```

It may be noted that the file that we want to read or write, may be present in some other directory or folder. In that case, we can give absolute or relative path of that file along with its name, shown in the example below. The path separated by \ must

be proceeded by one more \ (backslash) to turn-off special feature of this character. If the file to be read or written is in the same folder, in which python program is present, we need not give its path.

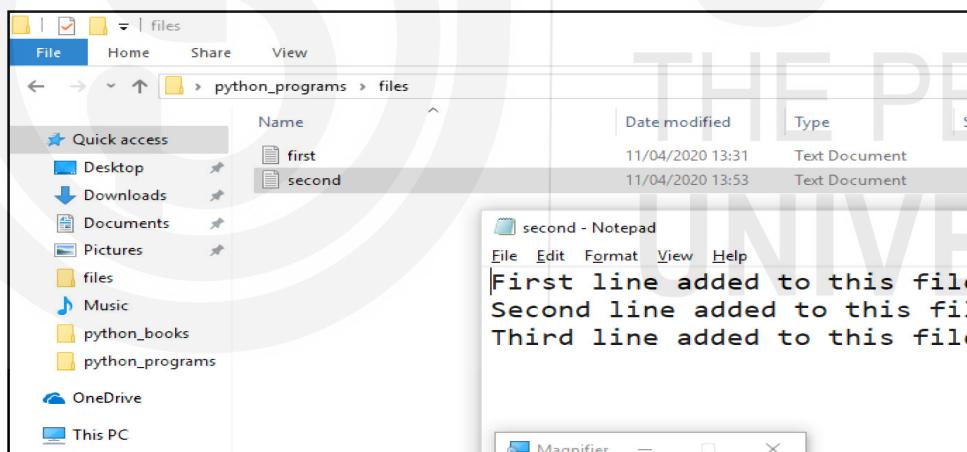
11.8.2 Creating a file

For creation of new file, the file can be opened in ‘w’ mode. If the file already exists, the contents can be overwritten. Also, if we want to be sure that no existing file gets overwritten, then ‘x’ mode can be used to create new file. If the file already exists, it will show error message. Given below is the example to open a file and write contents to it.

Example24 : Program to create file and add contents to it.

```
file = open ("files\\second.txt","w") # step 1: opening file for reading  
  
file.write("First line added to this file\n")# step 2: writing contents to file  
file.write("Second line added to this file\n")  
file.write("Third line added to this file\n")  
file.close() # step 3: closing file  
print("file created successfully")  
  
===== RESTART: C:/Users/test/Desktop/python_programs/file.py ======  
file created successfully  
>>>
```

After executing this program a file name second.txt gets created in the mentioned drive with the added contents.



Reading, writing, appending with *with* statement

The methods of dealing with files used in the above section may not be safe sometimes. It may happen that an exception occurs as a result of operations on file and the code exits without closing the file. To make it more convenient python introduced another statement called *with* statement which ensures that file is closed when the code within *with* is exited. Call to close() function is not required explicitly. The syntax of how to use *with* statement for reading, writing and appending file is given below.

Syntax of read,write and append with *with* statement:

```
with open("file_name","r") as file: # to read file
    file.read()

with open("file_name","w") as file: # to write file
    file.write("contents to add")

with open("file_name","a") as file: # to append file
    file.write("\n contents to append")
```

11.8.3 Copying a file

To copy a file from one to another various methods can be used. One such way of copying file is by importing a module named ‘shutil’ which contains a built-in function to create copy of a file.

Example 25 : Program to copy a file from ‘second.txt’ to ‘third.txt’.

```
import shutil
shutil.copy("files\\second.txt","files\\third.txt")
```

11.8.4 Deleting a file or folder

A file can be deleted using remove() function from os module. Before using this function you should move to the directory in which the file to be removed exists. Similarly, to delete a folder or directory, os.rmdir() function can be used. The directory to be removed must be empty or otherwise error will be shown. In the example given below, we want to delete a file name “delete_123.py”. os.listdir() function is used to display the list of files present in a directory.

Example 26: Deleting a file.

```
>>> import os
>>> os.chdir("C:\\\\Users\\\\test\\\\Desktop\\\\python_programs")
>>> os.listdir()
['abc.py', 'Arbitrary.py', 'arguments.py', 'arg_test.py', 'compound_interest.py',
 'delete_123.py', 'demo.py', 'demo2.py', 'factorial.py', 'file.py', 'file2.py',
 'files', 'lambda.py', 'modules_test.py', 'module_2.py', 'pack', 'path.py', 'rand_
test.py', 'return_example.py', 'series.py', 'simple_interest.py', 'test.py', 't
est_2.py', '__pycache__']
>>> os.remove("delete_123.py")
>>> os.listdir()
['abc.py', 'Arbitrary.py', 'arguments.py', 'arg_test.py', 'compound_interest.py',
 'demo.py', 'demo2.py', 'factorial.py', 'file.py', 'file2.py', 'files', 'lambda
.py', 'modules_test.py', 'module_2.py', 'pack', 'path.py', 'rand_test.py', 'retu
rn_example.py', 'series.py', 'simple_interest.py', 'test.py', 'test_2.py', '__py
cache__']
```

Check your Progress 4

Ex 1. Write a program to display frequency of each word in a file.

Ex 2. Write a program to display first n lines from a file, where n is given by user.

Ex 3. Write a program to display size of a file in bytes.

11.9 SUMMARY

A **Function** is a block of code that performs a specific task. In this chapter, we have discussed various aspects of functions such as how to create functions, their scope, passing arguments to function, and Lambda functions. In addition to these topics, file handling operations are also discussed in detail such as how to interact with file, copying, deleting, etc.

SOLUTIONS TO CHECK YOUR PROGRESS

Check your Progress 1

Ex 1. The various benefits of using functions are –

- i) It decomposes a larger problem into more manageable units
- ii) It allows reusability of code
- iii) It reduces duplication
- iv) It makes code easy to understand, use and debug

The process of defining a function is called *function definition*. It actually describes the working of a function. It includes – function name, list of arguments and function body.

Syntax def function_name():
..... # body of function
.....

Even if the function is defined, it can never be executed till the time it is called. Hence, for using a function, it must be called using *function call statement*. Calling of a function includes function name and list of arguments (no function body).

Syntax function_name()

Ex 2. Part of a code in which a variable can be accessed is called *Scope of a variable*.

Local variables are the variable created within a function's body or function's scope. They cannot be accessed outside the function. Their scope is only limited to the function in which they are created.

Global variables are the variable created outside any function. Their scope is global, hence can be used anywhere. Global variables can also be created within function using keyword *global*.

```
def f():
    s = 1      # s is local variable
    print(s)
    r = "IGNOU" # r is global variable
f()
```

```
print(r)
Ex 3.
```

```
def pattern():
    for i in range(1,6):
        for j in range(1,i+1):
            print(j,end=' ')
    print()

pattern()
```

Check your Progress 2

Ex 1. Python provides various mechanisms of passing arguments to a function.

1. Required arguments
2. Default arguments
3. Keyword arguments
4. Arbitrary arguments

Ex 2.

```
def unique(numbers):
    out = []
    for i in numbers:
        if i not in out:
            out.append(i)
    return out

ans=unique([1,2,1,1,2,2,3,4,6,3])
print(ans)
```

Ex 3.

```
def prime(a,b):
    for i in range(a,b+1):
        if i==0 or i==1:
            continue
        test=0
        for j in range(2,i):
            if i%j==0:
                test=1
                break
        if test==0:
            print(i,end=' ')

prime(1,50)
```

Check your Progress 3

Ex 1.

```
nums = [1, 2, 3, 4, 5]
print("\nCube of numbers:")
cube = list(map(lambda x: x ** 3, nums))
print(cube)
```

Ex 2.

Functions and Files
Handling in Python

```
a = [1, 2, 3]
b = [10, 20, 6]

result = map(lambda x, y: x + y, a, b)
print("\nAddition of two list:")
print(list(result))
```

Ex 3.

```
def vowel(a):
    v=['a','e','i','o','u']
    if a in v:
        return a

l=['a','b','f','o','x','z','y']
print("LIST OF VOWELS:")
print(list(filter(vowel,l)))
```

Check your Progress 4

Ex 1.

```
from collections import Counter
def wordcount(fname):
    with open(fname) as f:
        return Counter(f.read().split())

print("Frequency of words:",wordcount("abc.txt"))
```

Ex 2.

```
n=int(input("enter number of lines:"))
c=1
file=open("test.txt","r")
for i in file:
    if c<=n:
        print(i)
        c+=1
file.close()
```

Ex 3.

```
def fsize(fname):
    import os
    status = os.stat(fname)
    return status.st_size

print("File size in bytes: ",fsize("test.txt"))
```

UNIT 12 MODULES AND PACKAGES

Structure

- 12.0 Introduction
 - 12.1 Objectives
 - 12.2 Module Creation and Usage
 - 12.3 Module Search Path
 - 12.4 Module Vs Script
 - 12.5 Package Creation and Importing
 - 12.6 Standard Library Modules
 - 12.7 Summary
-

12.0 INTRODUCTION

Modules are files that contain various functions, variables or classes which are logically related in some manner. Modules like functions are used to implement modularity feature of OOPs concept. Related operations can be grouped together in a file and can be imported in other files. Package is a collection of modules and other sub-modules. Modules can be well organized and easily accessible if collectively stored in a package.

12.1 OBJECTIVES

Afetr going through this unit, you will be able to :

- Understand usage of Modules
- Create your own Modeules
- Compare Modules and scripts
- Import packages and Create your own packages
- Understand the standard library modules

12.2 MODULE CREATION AND USAGE

Module is a logical group of functions , classes, variables in a single python file saved with .pyextension.In other words, we can also say that a python file is a module. We have seen few examples of built-in modules in previous chapters like os, shutil which are used in the program by import statement. Python provides many built-in modules. We can also create our own module.

A major benefit of a module is that functions, variable or objects defined in one module can be easily used by other modules or files, which make the code re-usable. A module can be created like any other python file. Name of the module is the same as the name of a file. Let us create out first module- series.py.

In this module, we have created three functions- to find factorial of a number, fibonacci series upto given number of terms and a function to display exponential series and it sum. After creating a file, save it with name series.py.

Note: it is important to check where we have saved this file or module. Currently, it is saved in my present working directory. You can check current working directory by using built-in function `getcwd()` under `os` module by using following syntax in console window or python prompt.

```
>>> import os  
>>> os.getcwd()  
'C:\\\\Users\\\\test\\\\Desktop\\\\python_programs'  
>>> |
```

Example 1: Creating module named series.py

```
# creating module name series.py

def factorial(n):                      # function to calculate factorial
    fact=1
    for i in range(1,n+1):
        fact=fact*i
    return fact

def fibonacci(n):                      # function to display fibonacci series
    first=0
    second=1
    print("FIBONACCI SERIES:\n",first, " ",second,end=' ')
    for i in range(1,n-1):
        third=first+second
        print(" ",third,end=' ')
        first=second
        second=third

def exponential(x,n):                  # function to display exponential series upto n
    s=0
    for i in range(n):
        print("(x pow",i,"/",i,"!)+",end=' ')
        s=s+(x**i/factorial(i))
    print("\nExponential series sum:",s )
```

Our module is successfully created. Now let us test our module by importing in some other file and check whether it is working or not. For verifying that, in a new file, two steps are needed to be done-

1. Import the module we have created to make it accessible
2. Call the functions of that module with module name and a dot (.) symbol.

Example 2: Accessing function in module created in Example 1.

```

import series # importing module series.py

print ("Here we are using module series.py")

n=int(input("enter number of terms in fibonacci series"))

series.fibonacci(n) # calling a function present in other module

|                                     Ln: 10 Col: 0
=====
RESTART: C:/Users/test/Desktop/python_programs/demo.py =====
Here we are using module series.py
enter number of terms in fibonacci series 10
FIBONACCI SERIES:
 0  1  1  2  3  5  8  13  21  34
>>>                                     Ln: 67 Col: 0

```

Similar to the above example, we can call another function created in the module fibonacci() by following the same process.

```

import series

Series.fibonacci ( 2, 10 )

```

When a module is imported in a file, a folder named `__pycache__` gets created by interpreter which contains .pyc file of a module imported. This file contains the compiled bytecode of module so that conversion from source code to bytecode can be skipped for subsequent imports and making execution faster.

Importing a module

Importing is the process of loading a module in other modules or files. It is necessary to import a module before using its functions, classes or other objects. It allows users to reference its objects. There are various ways of importing a module.

1. using *import* statement
2. using *from import* statement
3. using *from import ** statement

1. Importing Complete module

In this method, we can import the whole module all together with a single import statement. In this process, after importing the module, each function (or variable, objects etc.) must be called by the name of the module followed by dot (.) symbol and name of the function.

Syntax of function calling within module

```

import module
module.function_name()

```

For example, let us import the built-in module random, and call its function randint(), which generates a random integer between a range given by user.

This can be done by running the code below in console window directly or in a python file.

```
>>> import random  
>>> random.randint(10,100)  
74  
>>> random.randint(10,100)  
95  
>>>
```

In this method, other functions or objects present in module random can be called similarly.

```
>>> import random  
>>> random.random()  
0.62009496454466  
>>> |
```

Here,random () is function present within module random.

2. Importing using *from import* statement

In this method of importing, instead of importing the entire module function or objects, only a particular object needed can be imported. In this method, objects can be directly accessed with its name.

Let us take an example of another module called math. This module contains various functions and variables. One such variable is pi, which contains value of π .

```
>>> from math import pi  
>>> pi  
3.141592653589793  
>>>
```

In this example, only a variable called pi is imported from math module, hence it can be directly accessed with its name. In this case, module name cannot be used for calling its objects, doing this will show NameError. Also other functions within the module math cannot be accessed, since only one variable pi is imported. You will be able to access them only after importing them individually with *from import* statement.

```
>>> from math import pi  
>>> pi  
3.141592653589793  
>>> math.pi  
Traceback (most recent call last):  
  File "<pyshell#4>", line 1, in <module>  
    math.pi  
NameError: name 'math' is not defined  
>>>
```

3. Importing entire module using *from import **

This method can be used to import the entire module using from import * statement. Here,*represents all the functions of a module. Like previous method, an object can be accessed directly with its name.

```
>>> from math import *
>>> pi
3.141592653589793
>>> log(2)
0.6931471805599453
>>> log10(100)
2.0
>>> |
```

Ln: 26 Col: 4

Check your Progress 1

- Ex 1. What are modules in Python and how can we create modules ?
- Ex 2. What are the various ways of importing modules ?
- Ex 3. Name any 3 built-in modules in python.

12.3 MODULE SEARCH PATH

When we use import statements to import a module, it is searched in a list of directories or search paths stored by the environment variable PYTHONPATH.This list of directories can be checked using sys.path variable.

```
>>> import sys
>>> sys.path
['', 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\Lib\\\\idlelib',
 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\python37.zip',
 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\DLLs', 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\lib',
 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\lib\\\\site-packages']
>>>
```

These directories include:

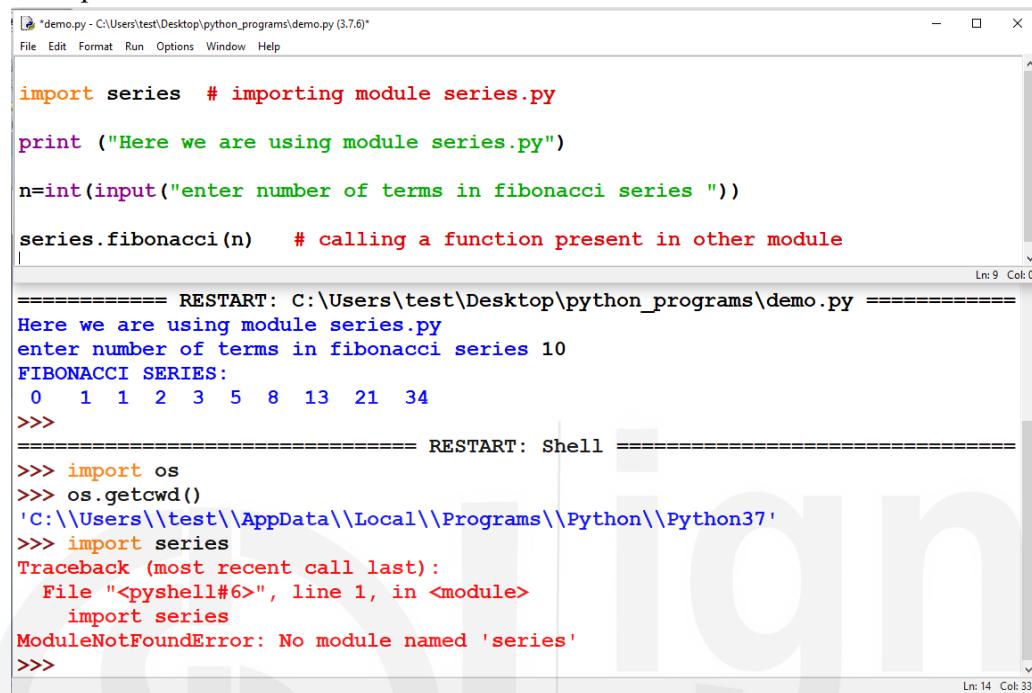
1. The current directory in which user is working [‘ ’]
2. Installation dependent paths
3. Directories stored in variable PYTHONPATH

Upto now, we were able to import our modules without doing anything special because they were all created in the same current directory. But if we move to some other directory, and try to import modules located in previous directories, we will not be able to use it.

In example 1 of this unit, we have created a module named series.py and used this module in a file named demo.py in example2. We were able to import

modules since both of them were in the same directory. But when we re-start shell, we move to python's default location. In this location, we will not be able to import of series.py module. Shown in example 3 below.

Example 3:



```
! "demo.py" - C:\Users\test\Desktop\python_programs\demo.py (3.7.6)*
File Edit Format Run Options Window Help
import series # importing module series.py
print ("Here we are using module series.py")
n=int(input("enter number of terms in fibonacci series "))
series.fibonacci(n) # calling a function present in other module
=====
===== RESTART: C:\Users\test\Desktop\python_programs\demo.py ======
Here we are using module series.py
enter number of terms in fibonacci series 10
FIBONACCI SERIES:
 0  1  1  2  3  5  8  13  21  34
>>>
===== RESTART: Shell ======
>>> import os
>>> os.getcwd()
'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37'
>>> import series
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    import series
ModuleNotFoundError: No module named 'series'
>>>
```

Therefore, any modules created must be located in python's search path for its global identification. This can be done in either of the ways-

1. Creating module in one of the locations already present in search path
2. Adding your module path in the search path using sys.path.
3. Updating PYTHONPATH environment variable.
- 4.

Adding module location to search path

A module path can be added to python's module search path by appending the sys.path variable. This can be done by using the append() function of sys.path. Directory in which your module is located should be appended as shown below example 4.

Example 4: Adding module path to search path

```
>>> import sys
>>> sys.path.append('C:\\\\Users\\\\test\\\\Desktop\\\\python_programs')
>>> sys.path
['', 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\Lib\\\\idlelib',
 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\python37.zip', 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\DLLs', 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\lib', 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37', 'C:\\\\Users\\\\test\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python37\\\\lib\\\\site-packages', 'C:\\\\Users\\\\test\\\\Desktop\\\\python_programs']
>>>
```

As we can see in above example, our directory is now present in the list of search directories. Hence, now we can import series module from any location. This method is not robust since it adds modules only for current session. For each new session, path needs to be added again.

```
>>> import series
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    import series
ModuleNotFoundError: No module named 'series'
>>> import sys
>>> sys.path.append('C:\\\\Users\\\\test\\\\Desktop\\\\python_programs')
>>> import series
>>> series.fibonacci(10)
FIBONACCI SERIES:
 0  1  1  2  3  5  8  13  21  34
>>> |
```

12.4 MODULE VS SCRIPT

For large number of instructions to be used together instead of directly running in shell or console, we used to write the code text files. These files can be modules or scripts. Extension of both the files is .py. Though there are several similarities, there are few differences as well.

SCRIPTS

Scripts are the files with sequence of instructions, which are executed each time the script is executed. There are various ways to execute a script, provided by different IDEs. It can also be executed in the console (shell in Unix/Linux and cmd in windows) using the command given below.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.17134.112]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\test>cd C:\Users\test\Desktop\python_programs

C:\Users\test\Desktop\python_programs>python rand_test.py
0.8414709848078965
2

C:\Users\test\Desktop\python_programs>
```

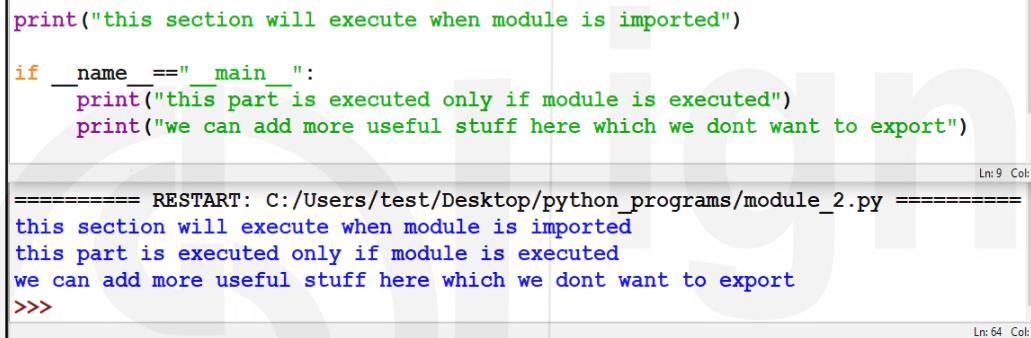
It should be noted that this command should be run in the directory where your python script exists otherwise *no file or directory exists* error will be shown.

MODULES

Functions which can be called from multiple scripts should be created within a module or we can say that a module is a file which is created for the purpose of importing. They are used to organize code in hierarchy. Module after creation should be added to search path.

When a module is imported, it runs the file from top to bottom. But when a module is executed, it runs the entire file and set the `_name_` attribute to the value “`_main_`”. This allows us to put a special code in a particular section which we want and we execute only when the module is executed directly. This section will not be executed during import.

Here, a module named `module_2.py` is created. If this module is executed, output received is given in the below screenshot. The whole file will be executed.

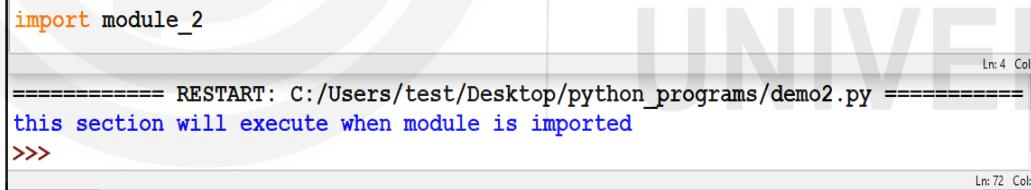


```
print("this section will execute when module is imported")

if __name__ == "__main__":
    print("this part is executed only if module is executed")
    print("we can add more useful stuff here which we dont want to export")

===== RESTART: C:/Users/test/Desktop/python_programs/module_2.py ======
this section will execute when module is imported
this part is executed only if module is executed
we can add more useful stuff here which we dont want to export
>>>
```

But when the above module is imported, the section under `if __name__ == "__main__":` will not be executed as shown below.



```
import module_2

===== RESTART: C:/Users/test/Desktop/python_programs/demo2.py ======
this section will execute when module is imported
>>>
```

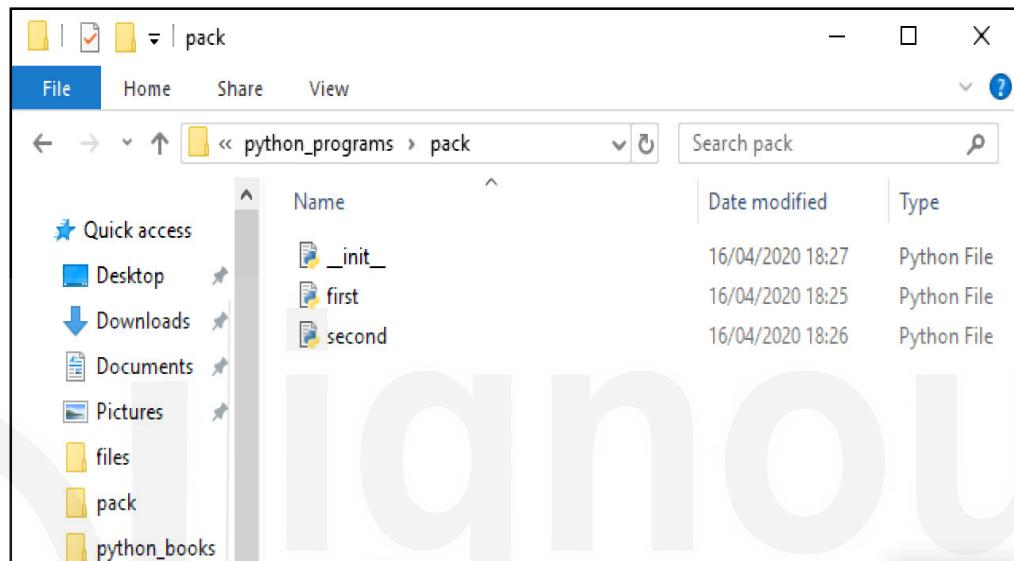
12.5 PACKAGE CREATION AND IMPORTING

Packages like modules are also used to organize the code in a better way. A package is a directory which contains multiple python modules. It is used to group multiple related python modules together. A python package in addition to modules must contain a file called `__init__.py`. This file may be empty or contains data like other modules of package.

File `__init__.py`

It is a file that makes the package importable. When a package is imported in a script, this file is automatically executed. It initializes variables, objects and makes the functions in the package accessible.

Let us create a package named `pack` and within this package create two modules `first.py` and `second.py`.



The `__init__.py` file created is empty. Module one contains function `abc()` and module two contains function `xyz()`.

Packages can be imported in the same way as we import modules.

The various ways in which we can import from package are-

```
import pack.first  
pack.first.abc()
```

```
from pack import first  
first.abc()
```

```
from pack.first import abc  
abc()
```

There are more methods to import. We have used `*` to import all the functions from a module in the previous section. This method can also be used here. But by default importing package modules using `*` will show error.

```
from pack import *
first.abc()

=====
RESTART: C:/Users/test/Desktop/python_programs/test_2.py =====
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/test_2.py", line 3, in <module>
    pack.first.abc()
AttributeError: module 'pack' has no attribute 'first'
>>>
```

This can be made possible using `__all__` variable. This variable when added to `__init__.py` file, can make modules within package accessible outside using `from import *` statement.

Hence, we need to add `__all__` statement in `__init__.py`

```
__all__=['first']
```

The above statement makes module `first.py` accessible using `from import *` statement.

Adding statement `__all__` to `__init__.py` file.

```
__init__.py - C:\Users\test\Desktop\python_programs\pack\__init__.py (3.7.6)
File Edit Format Run Options Window Help
__all__=['first']

=====
RESTART: C:/Users/test/Desktop/python_programs/pack/__init__.py =====
>>>
```

Now another way to import the module is given below:

Syntax to import using `from import *`

```
from pack import *
first.abc()
```

```
from pack import *
first.abc()
second.xyz()

=====
RESTART: C:/Users/test/Desktop/python_programs/test_2.py =====
under module first
Traceback (most recent call last):
  File "C:/Users/test/Desktop/python_programs/test_2.py", line 4, in <module>
    second.xyz()
NameError: name 'second' is not defined
>>>
```

Here, we can clearly see that `first.py` module is now accessible, since we have added it to `__all__` variable. But `second.py` module is not accessible simultaneously, since it was not added to `__all__` attribute in `__init__.py`.

Check your Progress 2

Ex. 1 What are packages ?How are they different from modules ?

Ex. 2 What is module search path ?How can we check it ?State the ways of adding a user defined module to search path.

Ex. 3 Create a package named Area and create 3 module in it named – square, circle and rectangle each having a function to calculate area of square, circle and rectangle respectively. Import the module in separate location and use the functions.

12.6 STANDARD LIBRARY MODULES

Python standard library provides number of built-in modules. They are automatically loaded when an interpreter starts. We have already used few of them in previous chapters. It should be noted that before using any module, it should be imported first. Some of the commonly used library modules are-

- sys
- os
- math
- random
- statistics

Module Attributes

There are some attributes or functions that work for every module whether it is built-in library module or custom module. These attributes help in smooth operations of these modules. Some of them are explained below:

1. help () – it is a function used to display modules available for use in python or to get help on specific module.

```
>>> help('modules')

Please wait a moment while I gather a list of all available modules...

__future__          atexit           html            search
__main__            audioop          http            searchbase
__abc__             autocomplete    hyperparser    searchengine
__ast__              autocomplete_w  idle           secrets
__asyncio__         autoexpand     idle_test      select
__bisect__          base64          idlelib        selectors
__blake2__          bdb             imaplib        setuptools
__bootlocale__      binascii       img HDR       shelve
__bz2__              binhex          imp             shlex
__codecs__          bisect          importlib     shutil
__codecs_cn__       browser         inspect        sidebar
__codecs_hk__       builtins        io              signal
__codecs_iso2022__  bz2            iomenu        site
__codecs_jp__       cProfile       ipaddress     smtpd
__codecs_kr__       calendar       iter tools   smtp lib
__codecs_tw__       calltip        json           snd hdr
__collections__     calltip_w     keyword        socket
__collections_abc__ cgi            lib2to3       socketserver
__compat_pickle__   cgi b          linecache    sqlite3
__compression__     chunk          locale        squeezer
__contextvars__    cmath          logging       sr compile
```

2. dir ()- it is a function which is used to display objects or functions present in a specific module.Before using dir() function, module should be first imported.

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fm
od', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'is
inf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan'
, 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'ta
u', 'trunc']
>>>
```

Ln: 114 Col: 4

3. __name__ attribute

This attribute returns name of the module. By default its value is the same as the name of the module.

When a module or script is executed, its value becomes ‘__main__’. Also, when called without module name, it returns ‘__main__’.

```
>>> import math
>>> math.__name__
'math'
>>> __name__
'__main__'
>>> |
```

Ln: 870 Col: 4

4. __file__ attribute

This attribute returns the location or path of the module.

```
>>> import os
>>> os.__file__
'C:\Users\test\AppData\Local\Programs\Python\Python37\lib\os.py'
>>> |
```

Ln: 892 Col: 4

5. __doc__ attribute

This attribute displays documentation given at the beginning of the module file.

```
>>> math.__doc__
'This module provides access to the mathematical functions\ndefined by the C sta
ndard.'
>>> |
```

Ln: 904 Col: 4

OS MODULE

This is a module responsible for performing many operating system tasks. It provides functionality like- creating directory, removing directory, changing directory, etc.Some of the functions in os modules are given in the table

below. It should be noted that before using these functions, the module should be imported.

importos

function	Description
os.mkdir("location")	Creates new directory in a given location.
os.rmdir("location")	Removes directory given by user. It should be taken care that current working directory cannot be removed and the directory to be removed should be empty.
os.getcwd()	Displays the current working directory.
os.chdir("location")	Changes current working directory to a given location.
os.listdir("location")	Displays list of files and directories in a given location. If location is not given, files of current directory will be displayed.

SYS MODULE

This module contains various variables and functions that can manipulate python runtime environment. Some of them are listed intable given below:

Function	Description
sys.path	Shows list of directories used to search python modules
sys.argv	Displays list of values passed as command line arguments to python program
sys.maxsize	Returns the largest integer value a variable can store
sys.version	Returns string representing python version
sys.getsizeof(object)	Returns size of an object in bytes
sys.exit	Used to exit from a program in case of exception

MATH MODULE

This module provides various mathematical functions and constant variables. It includes logarithmic, trigonometric functions etc. Some of the functions are listed in table below:

Function	Description	Modules and Packages
math.pow(x,y)	Returns x to the power y i.e. $x^{**}y$	
math.sqrt(x)	Returns square root of x i.e. \sqrt{x}	
math.pi	Returns value of π	
math.e	Returns value of e, Euler's number	
math.radians(x)	Converts angle x from degree to radians	
maths.degree(x)	Converts angle x from radians to degree	
math.sin(x)	Returns sin() of angle x in radians	
math.log(x)	Returns natural log of x	
math.log10(x)	Returns log base 10 of x	
math.floor(x)	Returns largest integer $\leq x$	
math.ceil(x)	Returns smallest integer $\geq x$	

STATISTICS MODULE

This module contains various functions used in statistics. These functions are widely used for data analysis or data science.

Function	Description
Statistics.mean(list)	Returns arithmetic mean of list or data given by user
Statistics.median(list)	Returns median value of list given by user
Statistics.mode(list)	Returns mode (highest frequency) value given by user
Statistics.stdev(list)	Returns standard deviation of list given by user
Statistics.variance(list)	Returns variance of list given by user

12.7 SUMMARY

In this unit, we have discussed modules and package creations in details. Modules are python files which can be imported in other files. A package is a folder which can store multiple modules and sub-packages within. Moreover,

built-in modules are also discussed in details that add real power to python programming.

SOLUTION TO CHECK YOUR PROGRESS

check your Progress 1

Ex.1 A **Module** is a logical group of functions , classes, variables in a single python file. A major benefit of a module is that functions, variable or objects defined in one module can be easily used by other modules or files, which make the code re-usable.

A module can be created like any other python file i.e. with .py extension. Name of the module is the same as the name of a file.

Ex. 2 The various methods of importing a module are

1. using *import* statement
2. using *from import* statement
3. using *from import ** statement

1. *import module*
module.function()

This method is used to import the entire module. Individual functions can be used with module name and .symbol.

2. *from module import function*
function()

This method is used to import individual function from a module. In this method, function can be directly called with its name.

3. *From module import **
function()

This method can be used to import the entire module using *from import ** statement. Here, * represents all the functions of a module. Like previous method, an object can be accessed directly with its name.

Ex. 3 The 3 built-in modules in python are –os, math, random.

check your Progress 2

Ex. 1 A package is a directory which contains multiple python modules. It is used to group multiple related python modules together. A python package in addition to modules must contain a file called `__init__.py`. This file may be empty or may contain data like other modules of package.

Ex. 2 When we use import statements to import a module, it is searched in a list of directories or search paths stored by the environment variable PYTHONPATH. This is called **module search path**. This list of directories can be checked using **sys.path** variable. Any modules created must be located in python's search path for its global identification.

Modules can be added to search path by either of the ways-

1. Creating module in one of the locations already present in search path
2. Adding module path in the search path using sys.path.
3. Updating PYTHONPATH environment variable.

Ex. 3 First of all, create a folder named area and place 4 file named – circle.py, square.py, rectangle.py and __init__.py in folder.

circle.py

```
*circle.py - C:\Users\test\Desktop\python_programs\area\circle.py (3.7.6)*
File Edit Format Run Options Window Help

def circle(r):
    import math
    return math.pi*r**2
```

square.py

```
*square.py - C:\Users\test\Desktop\python_programs\area\square.py (3.7.6)*
File Edit Format Run Options Window Help

def square(side):
    return side*side
```

rectangle.py

```
*rectangle.py - C:\Users\test\Desktop\python_programs\area\rectangle.py (3.7.6)*
File Edit Format Run Options Window Help

def rectangle(l,b):
    return l*b
```

Now, we can import the package along with all the modules in any file.

```
from area.circle import *
from area.square import *
from area.rectangle import *

print("Area of Circle:",circle(4))
print("Area of Square:",square(4))
print("Area of Rectangle:",rectangle(3,4))
```

UNIT 13 CLASSES IN PYTHON

Structure

- 13.0 Introduction to Object-Oriented Paradigms
- 13.1 Objectives
- 13.2 Classes and instances
- 13.3 Classes method calls
- 13.4 Inheritance and Compositions
- 13.5 Static and Class Methods
- 13.6 Operator Overloading
- 13.7 Polymorphism
- 13.8 Summary

13.0 INTRODUCTION TO OBJECT ORIENTED PARADIGMS

Object-oriented programming is a programming paradigm which define a class(a group of similar types of objects)and objects (containing both data and methods) to achieve the modularity and portability of the various components of the programs.

In Python programming language, we can define new classes. These classes are customized to a particular application which helps the user to perform its task in an easy way and maintain all the things required in the application.

The reason to define a new class helps in solving the structured application program in terms of object-oriented programming.In structured programming, language functions are defined. These functions are used by the users, but the implementations are hidden by the users. Similarly, a class defines methods and these methods are used by the users but how these methods are implemented are hidden from the users. A customized namespace is associated with every class and objects.

13.1 OBJECTIVES

After going through this unit you will be able to :

- Understand concepts of Object Oriented programming in Python
- Create your own Classes and Objects of classes
- Apply the concept of Inheritance
- Understand the concept of Overloading and Overriding

13.2 CLASSES AND INSTANCES

Classes

A class is a group of similar types of objects. For example, a university is a class, and various objects of the class are open_university, government_university, central_university, state_private_university, private_university, deemed_university.

To create a class we use keyword ‘class’. Following is the syntax of class:

Class class_name:

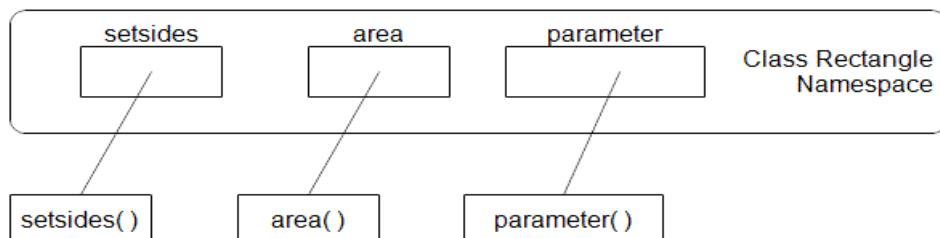
```
    class_variable1= value_of_variable  
    class_variable2= value_of_variable  
    class_variable3= value_of_variable  
    ...  
    def class_method1(self,arg1,arg2,...):  
        method1_code  
    def class_method2(self,arg1,arg2,...):  
        method2_code  
    ...
```

The first class we defined is a Rectangle.

```
>>>  
class Rectangle:  
    def setsides (self,x,y):  
        self.height=x  
        self.width=y  
        print('The sides of the rectangle are {} and  
        {}'.format(self.height,self.width))  
    def area(self):  
        return(self.height*self.width)  
    def parameter(self):  
        return(2*(self.height+self.width))
```

It's a convention to write the name of a class with the first character in a capital letter. But it supports a small letter also. After writing the name of a class, it must be preceded by a colon ‘:’. In C++ or Java, we were using pair brackets () but python supports colon ‘:’ only. The statement is written after the colon ‘:’ will be taken by the Python interpreter as part of the class document.

When we create a class, a namespace is created for the class Rectangle. This namespace stores all the attributes of the class Rectangle. This namespace will specify the names of class Rectangle methods.



Object

In Python, whatever value we are storing everything is taken as an object. For example, the string value 'IGNOU University' or the list ['IGNOU',22] or the integer value 28 all are stored in memory as an object. We can think object as a container which stores value in computer memory (RAM). Objects are the containers used to store the values, and it hides the details of the storage from the programmers and only gives the information which is required during the implementation.

Each object contains two things: *types and values*.

The type specifies what type of values can be assigned to object or the type of operations that can be operated by the objects.

```
>>>object1=Rectangle()
```

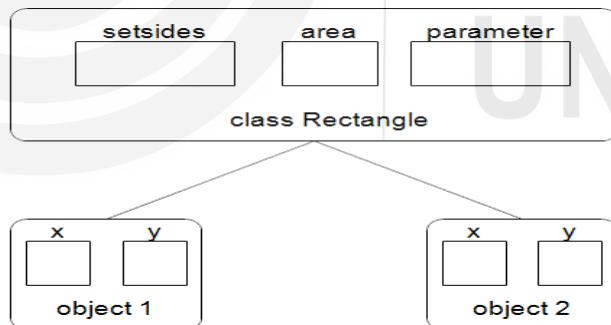
In Python, a namespace is created for each class. Each namespace is specified with a name, and this name is same as class name—all the attributes of the class use this namespace for the storage. Thus in our example namespace Rectangle must contain names of all the class methods.

Whenever an object object1 is created of the class Rectangle(), the separate namespace is created for the object1 also.

Object Object1 namespace

Let us describe the various methods of the class Rectangle:

setsides(x,y) : method to describe the sides of rectangle (height and width).
area() : method which return the area of rectangle.
parameter() : method which return the parameter of rectangle.



The function setsides(), area() and parameter() are defined in the name space. The syntax of method would be:

```
def setsides(self, x,y):  
    # implementation of setsides()  
def area():  
    # implementation of area()  
def parameter():  
    # implementation of parameter()
```

Instance

Variables that point to the object namespace is called instance variables. Each instance variable contains a different value for different objects. All the

variable define inside the `__init__` variable are called as instance variables. And all the variable defined inside the class but outside the `__init__` variable is called class variables. A class variable is also called as static variables.

Example1: Program to calculate area and parameter of a rectangle.

```

class Rectangle:
    def setsides (self,height,width):           # function to set height and width of rectangle
        self.side1=height
        self.side2=width
        print('The sides of the rectangle are {} and {}'.format(self.side1,self.side2))
    def area_of_rectangle( self ):                # function to calculate are of rectangle
        return(self.side1*self.side2)
    def parameter_of_rectangle(self) :           # function to calculate parameter of rectangle
        return(2*(self.side1+ self.side2))

object1=Rectangle()
object1.setsides(4,5)
object2=Rectangle()
object2.setsides(2,6)
#area_of_rectangle=object1.area_of_rectangle()
#paramater_of_rectangle=object1.parameter_of_rectangle()
print('The area of Rectangle 1 is {}'.format(int(object1.area_of_rectangle())))
print('The parametra of Rectangle 1 is {}'.format(int(object1.parameter_of_rectangle())))
print('The area of Rectangle 2 is {}'.format(int(object2.area_of_rectangle())))
print('The parametra of Rectangle 2 is {}'.format(int(object2.parameter_of_rectangle())))

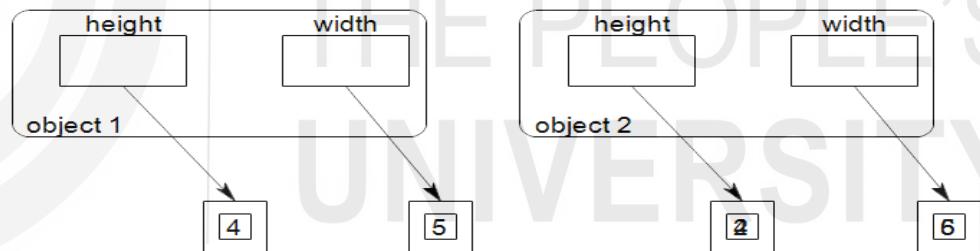
```

Ln: 20 Col: 91

===== RESTART: D:/python/IGNOU/Example 1.py =====

The sides of the rectangle are 4 and 5:
The sides of the rectangle are 2 and 6:
The area of Rectangle 1 is 20
The parametra of Rectangle 1 is 40
The area of Rectangle 2 is 12
The parametra of Rectangle 2 is 24
>>>
>>>

Ln: 12 Col: 4



13.3 CLASSES METHOD CALLS

To call a method of a class first, we have to create an object of the same class. After object creation, we will envoke the method defined in a class with the dot ‘.’ operator.

When we create an object of the class, the constructors declared in the class envoke automatically to which it belongs. In Python, there is a unique method `__init__` to implement constructor of the class in which it is defined. The first argument of the method `__init__` must be `self`. `Self` is a reference to a class to which this object belongs.

Example 2: Program to call a constructor

```
class Rectangle:
    def __init__(self):
        print ("Method invoke without call")
obj1=Rectangle()
obj2=Rectangle()

=====
RESTART: D:\python\IGNOU\Example 2.py =====
Method invoke without call
Method invoke without call
```

When two objects obj1 and obj2 are created `__init__` method is called automatically.

Class	Methods
Predefined word <code>class</code> is used to describe a class.	Predefined word <code>def</code> is used to describe the method of a class.
Each class statement defines a new type with a given name.	A <code>def</code> statement defines a new function with a given name.
Each class name is preceded by a colon ‘ <code>:</code> ’ statement.	Each method name is preceded by a colon ‘ <code>:</code> ’ statement.
Example -- class Rectangle:	Example -- <code>def</code> <code>setSides(self):</code>

Table 13.1 Comparison between class and method

13.4 INHERITANCE AND COMPOSITIONS

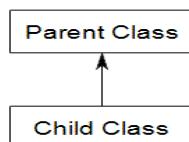
Inheritance is the mechanism by which object acquire the property of another object derived in a hierarchy.

The class which is inherited is called a base class or parent class or superclass, and the inheriting class is called the derived class or child class or subclass. For our reference, we will use the term parent class and child class. A child class acquire all the properties of the parent class, but the vice versa is not true; parent class can't access any features of a child class. The main motto behind the inheritance is code reusability. Once a code is defined in a class if it is required by another class then by inheriting the class code can be reused.

Syntax of inheritance:

`class <Child Class>:`

If this child class inherits the parent class, then the statement will be changed to
`class <Child Class> (<Parent class>):`



Consider a class Parent containing two methods methodA1() and methodA2() and a class Child with method methodB1() and methodB2(). Child class is defined to be the subclass of Parent class. Therefore it inherits both the methods methodA1() and methodA2() of Parent class .

Example 3: Inheritance of a parent class by child class

```
class Parent:
    def methodA1():
        print("Method 1 of Parent class ")
    def methodA2():
        print("Method 2 of Parent class ")
class Child(Parent):
    def methodB1():
        print("Method 1 of Child class ")
    def methodB2():
        print("Method 2 of Child class ")
obj1=Parent
obj1.methodA1()
obj1.methodA2()
```

```
Ln: 6 Col: 20
=====
RESTART: D:/python/IGNOU/Example 3.py =====
Method 1 of Parent class
Method 2 of Parent class
>>>
```

After inheritance class child contains methods methodB1() and methodB2() as well as methods of class Parent , methodA1() and methodA2().

method A1()
method A2()

class A

method B1()
method B2()
method A1()
method A2()

class B

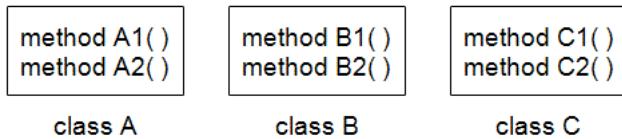
Example 4: Calling method of parent class by object of child class.

```
class Parent:
    def methodA1():
        print("Method 1 of Parent class ")
    def methodA2():
        print("Method 2 of Parent class ")
class Child(Parent):
    def methodB1():
        print("Method 1 of Child class ")
    def methodB2():
        print("Method 2 of Child class ")
obj2=Child
obj2.methodA1()
obj2.methodA2()
obj2.methodB1()
obj2.methodB2()
```

```
Ln: 8 Col: 35
=====
RESTART: D:/python/IGNOU/Example 4.py =====
Method 1 of Parent class
Method 2 of Parent class
Method 1 of Child class
Method 2 of Child class
>>>
```

Multilevel inheritance:

Consider three classes class A, class B and class C

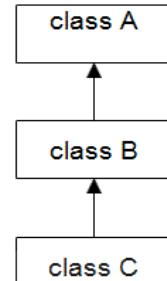
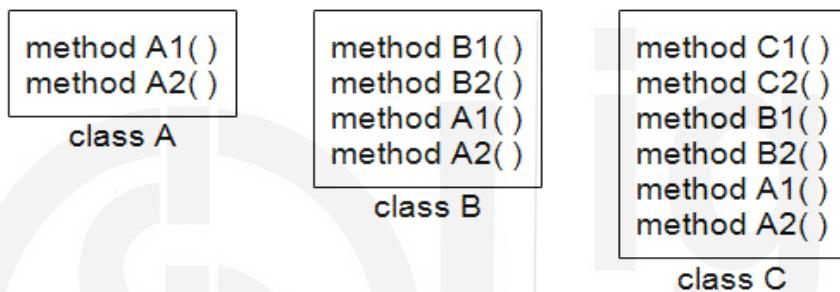


When an inherited class is inherited by another class, this is called multilevel inheritance.

Consider Class B inherits class A, and class C inherits class B.

Class B is called subclass of Class A, and Class C is called subclass of class B.

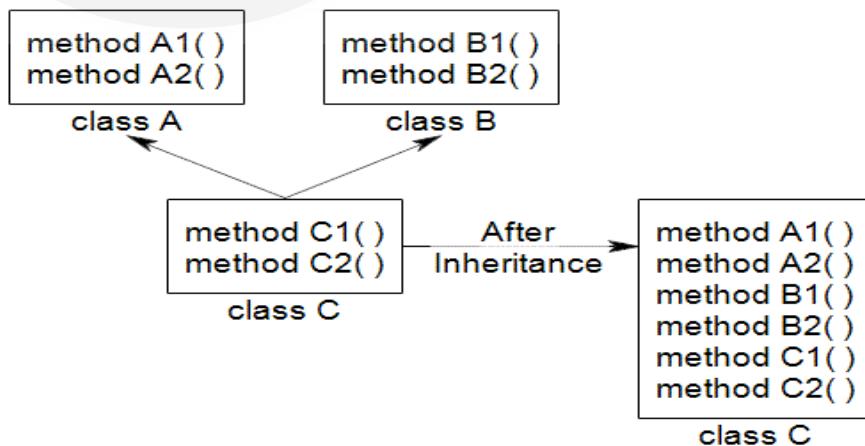
Class C will contain all the attributes and methods of class A and class B.



Multiple Inheritance:

When a single class inherit two or more classes, then it is called multiple inheritance.

Consider two classes, class A with two methods methodA1(), methodA2() and class B with methodB1() and methodB2(). If another class C inherits both class A and class B, then class C will inherit the features of class A and B.



Example 5: Implementation of multiple inheritance

```
class C1:          #Parent class 1
    def methodA1():
        print("Method 1 of class C1")
    def methodA2():
        print("Method 2 of class C1")
class C2:          #Parent class 2
    def methodB1():
        print("Method 1 of class C2")
    def methodB2():
        print("Method 2 of class C2")

class C3(C1,C2):      # Child class of class C1 and Class C2
=====
Ln: 12 Col: 62
===== RESTART: D:/python/IGNOU/Example 5.py =====
Method 1 of class C1
Method 1 of class C2
Method 1 of class C3
>>>
```

The behaviour of constructor in inheritance:

Example 6: Constructor of class

```
Example 6.py - D:\python\IGNOU\Example 6.py (3.8.5) - X
File Edit Format Run Options Window Help
class C1:
    def __init__(self): #Constructor of class C1
        print("Constructor of C1")
    def methodC1():
        print("Method 1 of class C1")
class C2(C1):
    def __init__(self): #Constructor of class C2
        print("Constructor of C2")

obj1=C1()
obj2=C2()
=====
Ln: 3 Col: 41
===== RESTART: D:/python/IGNOU/Example 6.py =====
Constructor of C1
Constructor of C2
>>>
```

Since the creation of an object of a class will automatically invoke the constructor of the class. Here in the example class C1 object will call a constructor of class C1 automatically, and creation of an object of class C2 will call the constructor of class C2 automatically.

Creating an object of child class will first search the `__init__` method of the child class. If `__init__` method is in the child class then it will be executed first if it is not in the child class then it will go to the `__init__` method of the parent class.

Example 7:

```
Example 7.1.py - D:/python/IGNOU/Example 7.1.py (3.8.5)
File Edit Format Run Options Window Help
class C1:
    def __init__(self):
        print("Constructor of C1")
    def method1():
        print("Method 1 of class C1")
class C2(C1):
    def method2():
        print("Method of Class B")
obj1=C2()

=====
Ln: 5 Col: 31
===== RESTART: D:/python/IGNOU/Example 7.1.py =====
Constructor of C1
>>>
```

If we want to call the init method of the parent class also then we can call it with the help of keyword super.

Example 8: Use of super keyword to call a method of the parent class.

```
class C1:
    def __init__(self):          # Constructor of class C1
        print("Constructor of C1")
    def method1():
        print("Method 1 of class C1")
class C2(C1):
    def __init__(self):
        super().__init__()      #Calling parent class C1 constructor
        print ("Constructor of C2")
    def method2():
        print("Method of Class C2")
obj1=C2()

=====
Ln: 2 Col: 0
Constructor of C1
Constructor of C2
>>>
```

Thus when we create an object of the child class, it will call the init method of the child class first. If we have called super, then it will first call init of the parent class then it will call int of the child class.

If we have two classes, A and B inherited by class C. If init method is defined in all three classes then what will happen if call inits of the parent class with the help of super() keyword? Then which class init method will be called?

Example 9: Uses of the super keyword in multiple inheritances.

```

class C1:
    def __init__(self):          #Constructor of class C1
        print("Contractor of C1")
    def methodA1():
        print("Method 1 of class C1")
class C2:
    def __init__(self):          #Constructor of class C2
        print("Contractor of C2")
    def methodB1():
        print("Method 1 of class C2")
class C3(C1,C2):           # Inheriting class C1 and C2
    def __init__(self):
        super().__init__()
        print("Contractor of C3")
obj1=C3()

```

```

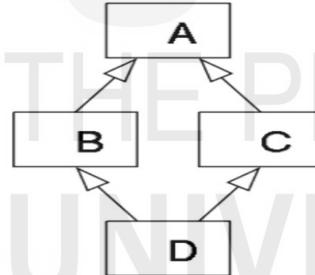
Contractor of C1
Contractor of C3
>>>

```

It will first call the init method of the child class then with the help of super() keyword it will call the init method of the left parent class that is A in our example.

In multiple inheritances, when we inherit the classes, then the method is executed based on the order specified, and this is called Method Resolution Order (MRO).

If class A is inherited by class B and Class C and then class D inheriting class B and class C.



Then according to the Method of Resolution Order (MRO) the order of execution of the methods and attributes are: class D → class B → class C → class A

13.5 STATIC AND CLASS METHODS

A static variable is that variable whose value remains the same for all the objects of the class. It creates one copy of the variable which is shared by all objects of the class. To declare a static variable, it must be declared inside the class. No method is used to declare a static variable it is declared without any method.

To use the static variable, we will use the class name to which this static variable belongs, or we will use the reference of the object. But it is always preferred to use class name in place of reference of the object.

Example 10: Implementation of static variable.

```

class Check:
    a=100      # static variable
    def __init__(self):
        self.b=200
obj1=Check()
obj2=Check()
print("obj1:", obj1.a, obj1.b)
print("obj 2:", obj2.a, obj2.b)
Check.a=888
obj1.b=999
print("t1:", obj1.a, obj1.b)
print("t2:", obj2.a, obj2.b)

Ln: 13 Col: 0

```

```

obj1: 100 200
obj 2: 100 200
t1: 888 999
t2: 888 200
>>>

```

In Python, there are three types of methods: Instance methods, class methods and static methods.

Instance methods:

Example 11: Implementation of an instance method.

```

class MyClass:
    def __init__(self,a1,b2,c3):      # constructor
        self.a1=a1
        self.b2=b2
        self.c3=c3
    def avg(self):                  # instance method
        return((self.a1+self.b2+self.c3)/3)
obj1=MyClass(3,3,6)
print(obj1.avg())

===== RESTART: D:/python/IGNOU/Example 11.py =====
4.0
>>>

```

In the above example first method `__init__` is a constructor of the class. The second method `avg(self)` is the instance method of the class. This method contains one parameter `self`, which points to the instance of the class `MyClass`. 3.0 will be printed as an output.

When the method is called Python, replace the `self` argument with the instance of the object. Thus instance method always works on the object.

Class method

```

class MyClass:
    classvariable= "my class variable"
    @classmethod
    def classmethod(cls):           #class method
        return cls.classvariable
print(MyClass.classmethod())

```

Ln: 10 Col: 0

```

=====
RESTART: D:/python/IGNOU/Example 12.py =====
my class variable
>>>

```

A class method is a method which is bound to the class and not the object of the class. A special symbol called a decorator ‘@’ followed by the keyword classmethod is used to define the class method.

A class method can be called by the class or by the object of the class to which this method belongs. The first parameter of the class method is class itself. Thus as an instance method is used to call the instance variable similarly class method is used to call class variables.

Static method

Suppose we are looking to a method which is not a concern to the instance variable neither to the class variable. In that case, we will use a static method. Static method in Python is used when such methods are called another class or methods are used to perform some mathematical calculations based on the values received as an argument. A special symbol called as a decorator ‘@’ is used followed by the keyword staticmethod is used to define a static method. Thus a static method can be invoked without the use of the object of the class.

```

class MyClass:
    @staticmethod
    def staticmethod():      #static method
        print("Calling of static method")

MyClass.staticmethod()

```

Ln: 6 Col: 22

```

=====
RESTART: D:\python\IGNOU\Example 13.py =====
Calling of static method
>>>

```

Thus if we want to work with the variables other than class variable and instance variable, we will use static.

13.6 OPERATOR OVERLOADING

Consider an operator ‘+’.

>>> 2 + 5

```
>>>[2,3,4]+[5,6]
```

```
[2,3,4,5,6]
```

Here it performs concatenation of two lists.

```
>>> 'IGNO'+ 'U University'
```

```
IGNOU University
```

Here it performs concatenation of two strings.

The operator ‘+’ is said to be an overloaded operator. If an operator is defined for more than one classes, then it is called the overloaded operator. For each class, the implementation of the overloaded operator is different. In our example, the overloaded operator ‘+’ is defined for the in-class,list class and string class.

The Python interpreter will take the operator ‘+’ as `x.__add__(y)`, and this is called method invocation.

```
__add__(..)
```

`x.__add__(y)` equivalent to `x+y`

Thus when we are performing `2+5`, it is

```
>>int(2).__add__(5)
```

```
7
```

```
>>>[2,3,4].__add__[5,6]
```

```
[2,3,4,5,6]
```

```
>>> 'IGNO'.__add__ 'U University.'
```

```
'IGNOU University'
```

Operator	Method	Number	List & String
<code>n1 + n2</code>	<code>n1.__add__(n2)</code>	Adding n1&n2	Concatenation
<code>n1-n2</code>	<code>n1.__sub__(n2)</code>	Subtracting n2 from n1	—
<code>n1 * n2</code>	<code>n1.__mul__(n2)</code>	Multiplying n1 & n2	Self concatenation
<code>n1 / n2</code>	<code>n1.__truediv__(n2)</code>	Dividing n1 by n2	—
<code>n1 // n2</code>	<code>n1.__floordiv__(n2)</code>	Integer division of n1 by n2	—
<code>n1 % n2</code>	<code>n1.__mod__(n2)</code>	Remainder after division of n1 by n2	—
<code>n1 == n2</code>	<code>n1.__eq__(n2)</code>	n1 & n2 both are same	
<code>n1 != n2</code>	<code>n1.__ne__(n2)</code>	n1 & n2 both are different	
<code>n1>n2</code>	<code>n1.__gt__(n2)</code>	n1 is larger than n2	
<code>n1 >= n2</code>	<code>n1.__ge__(n2)</code>	n1 is large than n2 or equal to n2	
<code>n1 < n2</code>	<code>n1.__lt__(n2)</code>	n1 is small than n2	
<code>n1 <= n2</code>	<code>n1.__le__(n2)</code>	n1 is small than or equal to n2	
<code>repr(n1)</code>	<code>n1.__repr__()</code>	Canonical representation of string n1	
<code>str(n1)</code>	<code>n1.__str__()</code>	Informal representation of string n1	
<code>len(n1)</code>	<code>n1.__len__()</code>	Size of n1	
<code><type>(n1)</code>	<code><type>.__init__(n1)</code>	Constructor	

Table 13.2: Overloaded operators

If we want to add a and b where a=2 and b= 'RAM.'

```
>>>a+b
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Because different types are not defined with the operator addition.

Consider a Student class

Example 14: Addition of two objects.

```
class Student:  
    def __init__(self,n1,n2):  
        self.n1=n1  
        self.n2=n2  
s1=Student(82,73)  
s2=Student(56,97)  
s3=s1+s2  
  
===== RESTART: D:/python/IGNOU/Example 14.py ======  
Traceback (most recent call last):  
  File "D:/python/IGNOU/Example 14.py", line 7, in <module>  
    s3=s1+s2  
TypeError: unsupported operand type(s) for +: 'Student' and 'Student'  
>>>
```

When we add two objects s1 & s2, it will show an error. Here it is not defined to the Python interpreter to add two objects. Hence we have to overload operator add.

Example 15: Operator overloading

```
class Student:  
    def __init__(self,n1):  
        self.n1=n1  
  
    def __add__(self,other):  
        return self.n1+other.n1  
  
s1= Student("Ignou")  
s2= Student(" University")  
  
print(s1+s2)  
  
===== RESTART: D:/python/IGNOU/Example 15.py ======  
Ignou University  
>>>
```

Here we overloaded operator add.

Suppose we want to compare s1 and s2

Example 16: Implementation of a comparison operator

```
class Student:  
    def __init__(self,n1):  
        self.n1=n1  
s1= Student(10)  
s2= Student(20)  
if s1>s2:  
    print('s1 wins')  
else:  
    print('s2 wins')  
  
===== RESTART: D:/python/IGNOU/Example 16.py ======  
Traceback (most recent call last):  
  File "D:/python/IGNOU/Example 16.py", line 6, in <module>  
    if s1>s2:  
TypeError: '>' not supported between instances of 'Student' and 'Student'  
>>>
```

The comparison operator is not defined to the Python interpreter for the objects. Hence we have to redefine it or overload it. The function which corresponds to the symbol greater than 'greater than' is `__gt__` (refer to table 13.2). So we have to overload this operator by defining the function.

Example 17: Overloading operator ‘>.’

The screenshot shows a Python terminal window. The code defines a class Student with an __init__ method and an __gt__ method for overloading the > operator. It compares two student objects based on their n1 attribute. If s1.n1 is greater than s2.n1, it prints 's1 wins'; otherwise, it prints 's2 wins'. When run, the program prints 's2 wins' because s2.n1 (20) is greater than s1.n1 (10).

```
class Student:
    def __init__(self,n1):
        self.n1=n1
    def __gt__(self,other):
        p1=self.n1
        p2=other.n1
        if p1>p2:
            return True
        else:
            return False
s1= Student(10)
s2= Student(20)
if s1>s2:
    print('s1 wins')
else:
    print('s2 wins')
=====
RESTART: D:/python/IGNOU/Example 17.py =====
s2 wins
>>>
```

because the value of s2 is higher than s1 hence s2 wins.

Now, what will happen if we want to print object s1

```
>>>print(s1)
```

...

```
<__main__.Student object at 0x00000192B57EE390>
```

It will print the address of the object. Now we want to print the value of the object then we have to redefined the function __str__().

```
>>>def __str__(self):
```

```
    return ('{} , {}'.format(self.n1,self.n2))
```

The value of two objects s1 and s2 printed.

13.7 POLYMORPHISM

Poly means ‘multiple’ and Morph means ‘forms’. Thus polymorphism is multiple forms. For example, human being behaves differently in a different environment. The behaviour of a human in the office is different from his behaviour at home, which is different from his behaviour with friends at a party.

Thus in terms of object orientation due to polymorphic characteristic object behave differently in a different situation.

There are four ways to implement polymorphism in Python:

Duck Typing

Operator loading

Method Overloading

Method Overriding

Duck typing

There is a sentence in the English language “if this is a bird which is walking like a duck,quacking like a duck and swimming like a duck then that bird is a duck”. It means if the behaviour of the bird is like a duck, then we can say it a Duck.

X=4

and

X= ‘Mohit’

In the first statement, X is a variable storing integer value. The type of the X is int. However, in the second statement X= ‘Mohit’ the storage memory taken by the

variable is a string. In Python, we can't specify the type explicitly. During the runtime, whatever value we are storing in a variable the type is considered automatically. And this is called Duck Typing principle.

Example 18: Duck Typing principle



```
Example 18.py - D:/python/IGNOU/Example 18.py (3.8.5)
File Edit Format Run Options Window Help
class Animal_Dog:
    def execute(self) :
        print ("Bow..Bow")
class Bird_Duck:
    def execute(self):
        print ("Quack..Quack")
class Animal:
    def code(self,ide):
        ide.execute()

ide=Bird_Duck()
obj=Animal()
obj.code(ide)

=====
Ln: 17 Col: 0
=====
RESTART: D:/python/IGNOU/Example 18.py =====
Quack..Quack
>>>
```

In the above example, the obj is an object of class Animal. When we call the code of Animal class, we have to pass an object ide. So before passing ide, we have to define it at the object of Duck class, one more ide for the class dog is defined. At the moment when we assign ide as an object of Dog class, then it will execute the dog method.

Example 19:



```
Example 19.py - D:/python/IGNOU/Example 19.py (3.8.5)
File Edit Format Run Options Window Help
class Animal_Dog:
    def execute(self) :
        print ("Bow..Bow")
class Bird_Duck:
    def execute(self):
        print ("Quack..Quack")
class Animal:
    def code(self,ide):
        ide.execute()

ide=Animal_Dog()
obj=Animal()
obj.code(ide)

=====
Ln: 17 Col: 0
=====
RESTART: D:/python/IGNOU/Example 19.py =====
Bow..Bow
>>>
```

So it doesn't matter which class object we are passing the matter is that object must have executed method. And this is called duck typing.

Operator Loading

Consider A=2 and B=5 are two variables. In a programming language, when we are performing a+b, it will perform the addition of two integer numbers. If X= 'Hello' and Y= 'Hi' then x+y will perform addition of two strings.

A=2

B=5

```

>>>print(A+B)
...
7
Python interpreter will take it as print(init.__add__(a,b)), where add() is a method
of the init class.
>>>print(init.__add__(a,b))
7

X= 'Hello'
Y= 'Hi'
>>>print (X+Y)
HelloHi
Python interpreter will take print(X+Y) as print(str.__add__(X,Y)), where add()is a
method osstr class.
>>>print(str.__add__(X,Y))
HelloHi

```

Method Overloading

Consider two classes having methods with the same name, but with a different number of parameters or different types of parameters, then the methods are called overloaded. In overloaded methods, the number of arguments is different, or types of arguments are different.

class student:

```

    marks(a,b)
    marks(a,b,c)

```

here two methods marks are defined one with two parameters and another having three parameters. These two methods are called method overloading. Python does not support method overloading.

If there are multiple methods in a class having the same name then Python will consider only the method which is described at the end of the class.

Example 20: Implementation of method overloading.

```

class MethodOverloading:
    def method1(self):
        print('Methodl without argument')
    def method1(self,x):
        print('Method with only single argument')
    def method1(self,x,y):
        print('Method with two arguments')
obj1=MethodOverloading()
obj1.method1()
obj1.method1(2)
obj1.method1(3,4)

=====
RESTART: D:/python/IGNOU/Example 20.py =====
Traceback (most recent call last):
  File "D:/python/IGNOU/Example 20.py", line 9, in <module>
    obj1.method1()
TypeError: method1() missing 2 required positional arguments: 'x' and 'y'
>>>

```

Method Overriding

In Python, we can't create the same methods with the same name and the same number of parameters. But we can create methods of the same method in the classes

derived in a hierarchy. Thus the child class redefined the method of the parent class, and this is called method overriding.

Example 21: Implementation of method overriding.

```
class University:
    def UnivName(self):
        print('IGNOU University')
    def course(self):
        print('BCA')
class student(University):
    def course(self):      # overrided method
        print('MCA')

stud1=student()
stud1.UnivName()
stud1.course()

===== RESTART: D:/python/IGNOU/Example 21.py =====
IGNOU University
MCA
>>>
```

A method which is overridden in a child class can also access the method of its parent class with the help of keyword super().

Example 22: Method overriding with the super keyword

```
class University:
    def UnivName(self):
        print('IGNOU University')
    def course(self):
        print('BCA')
class student(University):
    def course(self):      # overrided method
        super().course()# calling super class method
        print('MCA')
stud1=student()
stud1.UnivName()
stud1.course()

===== RESTART: D:/python/IGNOU/Example 22.py =====
IGNOU University
BCA
MCA
>>>
```

Check Your Progress - 1

1. Implement the class Circle that represents a circle. The class must contain the following methods:

Circle_setradius(): Takes one number of values as input and sets the radius of the circle.

circle_perimeter(): Returns the perimeter of the circle.

Circle_area(): Returns the area of the circle.

2. Define method overloading and constructor overloading in Python.
-
-
-

3. Select the correct output generated from the following program code:

```
class code1:  
    def __init__(self,st="Welcome to Python World"):  
        self.st=st  
    def output(self):  
        print(self.st)  
obj=code1()  
obj.output()
```

- a) The code result an error because constructor are defined with default arguments
 - b) Output is not displayed
 - c) "Welcome to Python World" is printed
 - d) The code result an error because parameters are not defined in a function
-
-
-

4. What type of inheritance is illustrated in the following Python code?

```
class Class1():  
    pass  
class Class2(Class1):  
    pass  
class Class3(Class2):  
    pass
```

- a) Multilevel inheritance
 - b) Multiple inheritance
 - c) Hierarchical inheritance
 - d) Single-level inheritance
-
-
-

5. What is polymorphism and what is the main reason to use it?
-
-
-

13.8 SUMMARY

In this unit, we discussed the concepts of classes and objects. Concept of a namespace to store object and class in memory is also defined in this unit. Different types of class methods namely static methods ,instance methods and class methods are discussed in this unit.

This unit also focused on inheritance and various types of inheritance: single level inheritance,multilevel inheritance and multiple inheritances.

In this unit, it is described that the same operator can be used for multiple purposes, and this is called operator overloading. A list of operators and corresponding methods, also called as magic methods are also given in the unit.

Polymorphism means it is the ability to behave differently in a different situation. The concept of polymorphism and the implementation of polymorphism with Duck Typing, Operator loading, Method Overloading and Method Overriding are also described in the unit.

SOLUTIONS TO CHECK YOUR PROGRESS

Check Your Progress -1

1. class Circle:

```
defcircle_setradius(self, radius):
    self.r = radius
defcircle_perimeter(self):
    return 2 * 3.142 * self.r
defcircle_area(self):
    return self.x * self.x*3.142
```
2. If two or methods are defined with the same name but they differ in return types or having a different number of arguments or having different types of arguments then these methods are called as method overloading.
Two constructors are overloaded when both are having different no. or different types of arguments.
In the python method overloading and constructor, overloading is not possible.
3. C
4. A
5. Polymorphism is one of the main features of object-oriented programming languages. With this characteristic, we can implement elegant software.

Structure

- 14.0 Introduction
 - 14.1 Objectives
 - 14.2 Default Exception Handler
 - 14.3 Catching Exceptions
 - 14.4 Raise an exception
 - 14.5 User Defined Exceptions
 - 14.6 Summary
-

14.0 INTRODUCTION

Programmers always try to write an error-free program, but sometimes a program written by the programmer generates an error or not execute due to the error generated in the program. Some times the program code is correct, but still, it generates an error due to the malicious data is given as an input to the program. This malicious data may be given by the user or from a file that causes an error generation during the execution. This type of error generally occurs when we use server-side programs such as web programming and gaming servers.

Every developer is writing a code that must not generate an error during execution. We will study various types of errors that generate during the program execution or before the execution of the program code.

14.1 OBJECTIVES

After going through this unit, you will be able to :

- Understand the requirement of exception handling in programming
- Raise and catch the exceptions
- Perform exception handling in python programming

14.2 DEFAULT EXCEPTION HANDLER

Consider the following examples in which syntax is correct (as per the Python statement), but it causes an error.

Example 1.

>>>3/0

Name of Error will be ZeroDivisionError

And the reason for the error is division by zero

Example 2

```
>>> list1=[2, 3, 4, 5, 6]  
>>>list1[5]
```

Name of Error is :IndexError

Reason of Error is : list index out of range

Example 3.

```
>>>x+3
```

Name of Error is : Name Error

Reason of Error is : name 'x' is not defined"

Example 4.

```
>>>int('22.5')
```

Name of Error is :ValueError

Reason of error is : invalid literal for int() with base 10: '22.5'

Example 5.

```
>>> '2'*3'
```

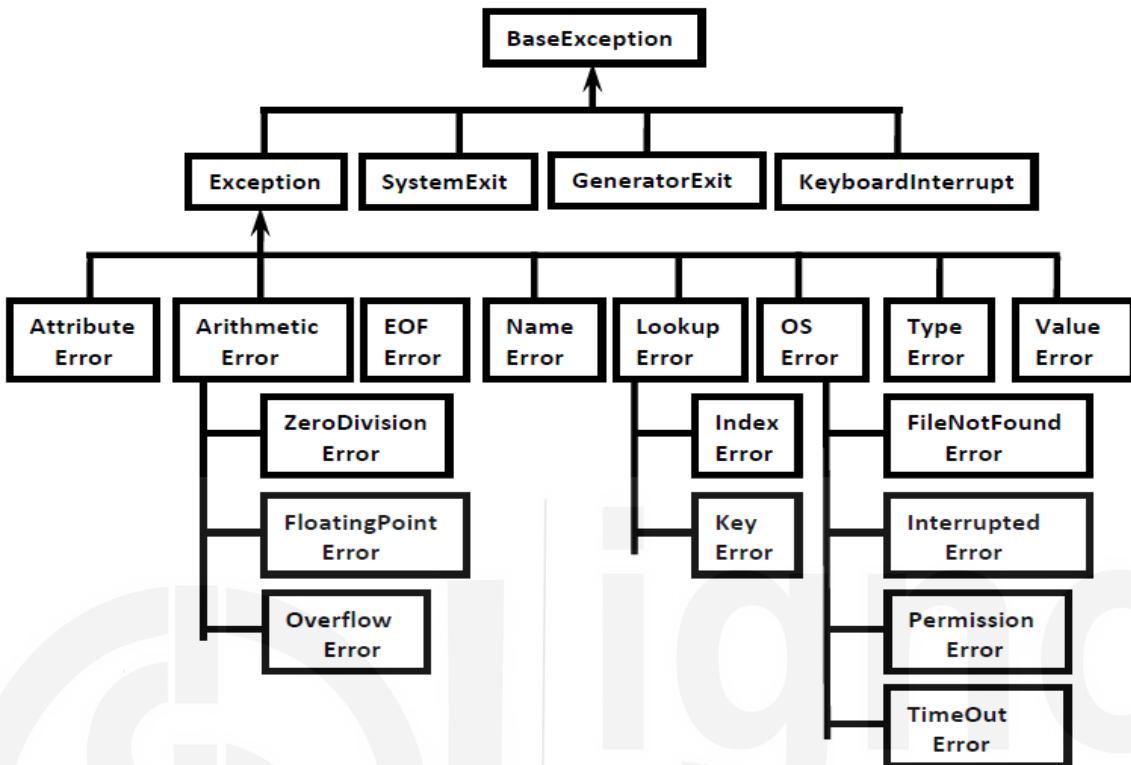
Error name is :TypeError

Reason of Error is : can't multiply sequence by non-int of type 'str'

In each example syntax is correct, but it goes to an invalid state. These are runtime errors. In such a situation, Python interpreter generates an exception for run time errors. This exception generates an object which contains all information related to the error. In Example 5 the error message displayed: what is happening and at which line number it is happening (here line number is one because there is only one statement in the code) and the type of error is also printed.

It is the responsibility of python virtual machine to create the object corresponding to the exception generated to handle the program code. If the code responsible for handling the exception is not there in the program code, then python interpreter will suspend the execution of the program code since an exception is generated the python print the information about the generated exception.

14.3 CATCHING EXCEPTIONS



In Python, every exception is class, and these classes are the derived class of the BaseException class. Thus BaseException class is the topmost class in the hierarchy of exception classes.

Exception handling using try-except:

```
try:  
    risky code(generating exception)  
except:  
    corresponding to risky code(handler)
```

The code which generates an exception must be written in a block with the name specify by 'try' and the code which will handle the exception will be given in 'except' block.

If the three lines code are written in Python

```
print("IGNOU")  
print(87/0)  
print("Welcome to the University")  
>>>  
...
```

Here the first line will be executed, and IGNOU will be printed when Python interpreter will execute 2nd lines of the program then it generates an exception, and the default exception handler will print the error code "division by Zero" then after the program is terminated. Thus the third line of the program will not execute. And this is also called as abnormal termination. If we rewrite the code with the try and except then the code will become

```
print("IGNOU")
```

Example 1: Implementation of try and except block.

```
try:  
    print(10/0)  
except ZeroDivisionError:  
    pass  
print ("Welcome to the University")  
print("IGNOU")|  
===== RESTART: D:/python/IGNOU/Unit 14/Example 1.py ======  
Welcome to the University  
IGNOU  
>>>
```

Here program terminated successfully. We can also print the error message by modifying the code as:

Example 2:

```
Example 2.py - D:/python/IGNOU/Unit 14/Example 2.py (3.8.5)  
File Edit Format Run Options Window Help  
try:  
    print(23/0)  
except ZeroDivisionError as msg1:  
    print("Exception raised : ",msg1)  
print("Welcome to the University")  
|  
===== RESTART: D:/python/IGNOU/Unit 14/Example 2.py ======  
Exception raised : division by zero  
Welcome to the University  
>>>  
Ln: 14 Col: 0
```

In the above code, the first line executed successfully and "IGNOU" will be printed. When the python interpreter executes the print statement "(23/0)" an exception is raised, and a message given in the print statement will be printed with the name of the exception "division by zero". After that, the interpreter executes the last statement, and it will print "Welcome to the University."

If there are more than one except block written in the try block, then the try block will execute the except block which is responsible for the code.

Example 3: Try block with more than one except block.

```
try:
    num1=int(input("Enter First Number: "))
    num2=int(input("Enter Second Number: "))
    print(num1/num2)
except ZeroDivisionError as msg1:
    print("Can't Divide with Zero : ",msg1)
except ValueError as msg1:
    print("Please Enter integer value only : ",msg1)

Ln: 8 Col: 0
=====
RESTART: D:/python/IGNOU/Unit 14/Example 3.py =====
Enter First Number: 3
Enter Second Number: 4
0.75
>>>
=====
RESTART: D:/python/IGNOU/Unit 14/Example 3.py =====
Enter First Number: 4
Enter Second Number: 0
Can't Divide with Zero :  division by zero
>>>
=====
RESTART: D:/python/IGNOU/Unit 14/Example 3.py =====
Enter First Number: 0
Enter Second Number: 4
0.0
>>>
=====
RESTART: D:/python/IGNOU/Unit 14/Example 3.py =====
Enter First Number: 4
Enter Second Number: a
Please Enter integer value only :  invalid literal for int() with base 10: 'a'
```

With the help of single except block, we can handle multiple exceptions:

Example 4: Multiple exception handling with a single try block and multiple except block.

```
try:
    num1=int(input("Enter First Number: "))
    num2=int(input("Enter Second Number: "))
    print(num1/num2)
except ZeroDivisionError as msg1:
    print("Can't Divide with Zero : ",msg1)
except ValueError as msg1:
    print("Please Enter integer value only : ",msg1)
```

```
Ln: 9 Col: 0
=====
RESTART: D:/python/IGNOU/Unit 14/Example 4.py =====
Enter First Number: 4
Enter Second Number: 0
Can't Divide with Zero :  division by zero
>>>
=====
RESTART: D:/python/IGNOU/Unit 14/Example 4.py =====
Enter First Number: 0
Enter Second Number: 3
0.0
>>>
=====
RESTART: D:/python/IGNOU/Unit 14/Example 4.py =====
Enter First Number: 5
Enter Second Number: a
Please Enter integer value only :  invalid literal for int() with base 10: 'a'
>>>
```

Corresponding to each error there will be an except block if corresponding except block is not there then a except block written at the end of all except block is used for the error. And it must be the last except block in all.

Example 5: Implementation of default except for block

```
try:
    num1=int(input("Enter 1st Number: "))
    num2=int(input("Enter 2nd Number: "))
    print(num1/num2)
except ZeroDivisionError as msg:
    print("Plz provide value greater than ZERO value in second number and the error is :")
except:
    print("Default Except:Plz provide valid input")

>>>
=====
RESTART: D:/python/IGNOU/Unit 14/Example 5.py =====
Enter 1st Number: 0
Enter 2nd Number: 4
0.0
>>>
=====
RESTART: D:/python/IGNOU/Unit 14/Example 5.py =====
Enter 1st Number: 4
Enter 2nd Number: 0
Plz provide value greater than ZERO value in second number and the error is : division by zero
>>>
=====
RESTART: D:/python/IGNOU/Unit 14/Example 5.py =====
Enter 1st Number: a
Default Except:Plz provide valid input
>>>
```

Some times exceptions may or may not be raised, and sometimes exceptions may or may not be handled. Irrespective of both of these we still want to execute some code. And such codes are written in finally block.

try:

Code with Error

except:

Code to handle error

finally:

Necessary code

Example 6: Implementation of finally block.

```
try :
    print("Block inside Try")
    print(35/0)
except ZeroDivisionError:
    print("Except Block")
finally:
    print("Necessary Code")
| 

=====
RESTART: D:/python/IGNOU/Unit 14/Example 6.py =====
Block inside Try
Except Block
Necessary Code
>>>
```

Nested try-except block:

If a try block is written inside another try block, then it is called as nested.

The code having higher risk must be defined in the innermost try block. If there is any exception raised by the innermost try block, then it will be handled by the innermost except block. If the innermost except block is not able to handle the error then except block defined outside the inner try block will handle the exception.

Example 7: Implementation of the nested try block

```

try:                                # try block
    print("Outer try block")
    try:                            # nested try block
        print("Inner try block ")
        print(22/0)
    except ZeroDivisionError:
        print("Except block of Inner try block")
    finally:
        print("Finally block of Inner try block")
except:
    print("Except block of Outer try block")
finally:
    print("Finally Block of Outer try block")

=====
===== RESTART: D:/python/IGNOU/Unit 14/Example 7.py =====
Outer try block
Inner try block
Except block of Inner try block
Finally block of Inner try block
Finally Block of Outer try block
>>>

```

A single except block can be used to handle multiple exceptions. Consider the given an example:

Example 8:Single except block handling multiple exceptions.

```

try:
    num1=int(input("Enter first Number: "))
    num2=int(input("Enter Second Number: "))
    print(num1/num2)
except (ZeroDivisionError,ValueError) as printmsg:
    print("Invalid Number & Error is: ",printmsg)

=====
===== RESTART: D:/python/IGNOU/Unit 14/Example 8.py =====
Enter first Number: 4
Enter Second Number: 0
Invalid Number & Error is:  division by zero
>>>
===== RESTART: D:/python/IGNOU/Unit 14/Example 8.py =====
Enter first Number: 4
Enter Second Number: two
Invalid Number & Error is:  invalid literal for int() with base 10: 'two'
>>>

```

In case of different types of error, we can use a default except for block. This block is generally used to display normal error messages. The default except block must be the last block of all except blocks.

Example 9: Implementation of default except for block.

```

try:
    n1=int(input("Enter First Number: "))
    n2=int(input("Enter Second Number: "))
    print(n1/n2)
except ZeroDivisionError:
    print("ZeroDivisionError:Can't divide with zero")
except:
    print("Default Except:Plz provide valid input only")

=====
===== RESTART: D:/python/IGNOU/Unit 14/Example 9.py =====
Enter First Number: 5
Enter Second Number: 6
0.8333333333333334
>>>
===== RESTART: D:/python/IGNOU/Unit 14/Example 9.py =====
Enter First Number: 22
Enter Second Number: 0
ZeroDivisionError:Can't divide with zero
>>>
===== RESTART: D:/python/IGNOU/Unit 14/Example 9.py =====
Enter First Number: 22
Enter Second Number: ten
Default Except:Plz provide valid input only
>>>

```

14.4 RAISE AN EXCEPTION

Some times user want to generate an exception explicitly to inform that this is the exception in this code. Such type of exceptions is called as user-defined exceptions or customized exceptions.

This user-defined exception is a class defined by the user, and this class is derived from the Exception class. Python interpreter does not have any information related to the user-defined exception class. Thus, it must be raised explicitly by the user.

The keyword `raise` is used to raise the class when it is required.

Example 10: Implementation of user-defined exceptions.

```

class EligibleForEntrance (Exception) :
    def __init__(self,arg1) :
        self.msg1=arg1

class NotEligible (Exception) :
    def __init__(self,arg1) :
        self.msg1=arg1

percentage=int (input ("Enter Your PCM Marks percentage"))
if percentage>50:
    raise EligibleForEntrance("You can apply for the entrance examination")
else:
    raise NotEligible("You can't apply for the entrance examination")

Ln: 5 Col: 31
Enter Your PCM Marks percentage 88
__main__.EligibleForEntrance: You can apply for the entrance examination
>>>
===== RESTART: D:/python/IGNOU/Unit 14/Example 10.py =====
Enter Your PCM Marks percentage 44
__main__.NotEligible: You can't apply for the entrance examination

```

14.5 USER-DEFINED EXCEPTIONS

A programmer can create his exception, called a user-defined exception or custom exception. A user-defined exception is also a class derived from exception class.

Thus, class, is the user define a class which is a subclass of the Exception class.

There are two steps in using user-defined exception:

Step 1: By inheriting Exception class create a user-defined exception class.

Step 2: Raise this exception in a program where we want to handle the exception.

Syntax :

```

classMyException(Exception):
    pass
classMyException(Exception)
    def __init__(self,argumnet):
        self.msg=argumnet

```

A *raise* statement is used to raise the statement.

Example 11: Program to raise an exception.

```
class TooYoungException(Exception):
    def __init__(self,arg):
        self.msg=arg

class TooOldException(Exception):
    def __init__(self,arg):
        self.msg=arg

age=int(input("Enter Age:"))
if age>60:
    raise TooYoungException("Plz wait some more time you will get best match soon!!!!")
elif age<18:
    raise TooOldException("Your age already crossed marriage age...no chance of getting marriage")
else:
    print("You will get match details soon by email!!!")

=====
===== RESTART: D:/python/IGNOU/Unit 14/Example 11.py =====
Enter Age:55
Plz wait some more time you will get best match soon!!!
>>>
===== RESTART: D:/python/IGNOU/Unit 14/Example 11.py =====
Enter Age:33
You will get match details soon by email!!!
>>>
```

Check Your Progress

1. Give the name of the error generated by the following python codes.

- a) 4 / 0
- b) (3+4]
- c) lst = [4;5;6]
- d) lst = [14, 15, 16]
 lst[3]
- e) x + 5
- f) '2' * '3'
- g) int('4.5')
- h) 2.0**10000
- i) for i in range(2**100):
 pass

2. Define the minimum number of except statements that can be possible in the try-except block.

3. Describe the conditions in which finally block executed.

4. Select the keyword from the following, which is not an exception handling in Python.

- a) try
- b) except
- c) accept
- d) finally

5. Give the output generated from the following Python code?

```
lst = [1, 2, 3, 4]
lst[4]
```

6. What will be the output generated with the following code?

```
def function1():
    try:
        function2(var1, 22)
    finally:
        print('after function2')
        print('after function2?')

function1()
```

- a) Output will not be generated
 - b) after f?
 - c) error
 - d) after f
-

14.6 SUMMARY

In this unit, it is defined that a program may produce an error even though the programmer is writing error-free code. These are the mistakes that change the behaviour of our program code.

In this unit, it is defined that the default exception handlers are there in Python. How these exception handler works are described in this unit.

By default, there are exception handlers in Python. But a user can also raise the exception which is a customized exception, not a language defined exception.

SOLUTION TO CHECK YOUR PROGRESS

Check Your Progress

1.
 - a) ZeroDivisionError
 - b) SyntaxError
 - c) SyntaxError
 - d) IndexError
 - e) NameError
 - f) TypeError
 - g) ValueError
 - h) OverflowError
 - i) KeyboardInterrupt Error
2. At least one except statement.
3. Finally block is always executed.
4. C
5. IndexError because the maximum index of the list given is 3.
6. C, since function function1 is not defined.

UNIT 15 PYTHON-ADVANCE CONCEPTS

Structure

- 15.0 Introduction
 - 15.1 Objectives
 - 15.2 Decorators
 - 15.3 Iterators
 - 15.4 Generators
 - 15.5 Co-routines
 - 15.6 Summary
-

15.0 INTRODUCTION

In unit number 11 of this course we learned about the functions, we learned that we have inbuilt functions and we also learned to develop our own functions just for the sake of revision, some of the examples are listed below

Say the following lines of code are saved in functions.py

```
# In-Built Function
```

```
Name = "IGNOU-SOCIS"
```

```
print(len(Name))
```

executing the functions.py by running the command \$ python function.py we get the output 11, which is the length of the variable Name. here, len() is the in-built function of python, which can be used to calculate length of any string, we need not to write code again and again to determine the length of any string simply we need to use this built-in function and get the job done.

We also learned to define our own function just to recapitulate lets write our function to add two numbers

```
# User-Defined Functions
```

```
defadd_two(a,b) :
```

```
    return a+b
```

```
total = add_two(5,4)
```

```
print(total)
```

here add_two(a,b) is the user defined function that takes two numbers as input and return their sum, which is stored in the variable total, the result of total is printed subsequently.

We also learned the concept of anonymous functions i.e lambda functions or Lambda expressions, the concept was introduced in python and later it was adopted by many other languages be it C++ or Java. The advantage of using the lambda function is that, they can be defined within the code by writing few lines and they don't need any name, that's amazing. One example of Lambda expressions is sited below, here we will write the lambda expression equivalent for the User-Defined Function add_two, which is given above

```
# Lambda Expressions (Anonymous Functions)
```

```
def add(a,b)
```

```
return a+b
```

```
add2 = lambda a,b : a+b
```

```
print(add2(2,3))
```

here, add2 collects the output of the lambda function, we can see the output by calling lambda function, by writing print(add2(2,3))

Generally lambda functions are not meant for the user defined functions but are used to facilitate the working of various built in functions like, map, reduce, filter, etc.

Modern programming language emphasizes on code reusability, where one need to work with the already written codes, but these codes needs to be customized, i.e. their functionality is required to be enhanced, and to enhance the functionality of already written functions, python has the concept of decorators. We will learn about decorators in our subsequent section 15.2

15.1 OBJECTIVES

After going through this unit, you will be able to:

- Enhance the functionality of functions by using decorators
- Understand the concept of iterators and iterables
- Appreciate the concept of generators
- Understand the concept cooperative multitasking using co-routines

15.2 DECORATORS

Modern programming paradigm recommends the technique of code reusability, where we need to customize the code as per our requirements, without disturbing the actual functionality of the code which is already written. To achieve this, python introduced the concept of decorators, this

concept is used to enhance the functionality of functions, which are already written, for example say we have two functions func1() and func2() as given in the python code given below

```
# Decorators – to enhance functionality of other functions
```

```
deffuncA():
```

```
    print (' this is functionA')
```

```
deffuncB():
```

```
    print (' this is functionB')
```

```
funcA()
```

```
funcB()
```

on executing the above mentioned code by running deco.py we will get output

```
this is function1
```

```
this is function2
```

but without disturbing the existing code if we want that along with, “this is function1” the output should contain the line “this is a wonderful function”, i.e. to enhance the already existing functionality, we need to exercise the concept of decorators.

To understand this concept lets re-write the code given above

```
# Decorators – to enhance functionality of other functions
```

```
defdecorator_funcn (any_funcn):
```

```
    defwrapper_funcn():
```

```
        print('this is a wonderful function')
```

```
        any_funcn()
```

```
        returnwrapper_funcn
```

```
    deffuncA():
```

```
        print (' this is functionA')
```

```
    deffuncB():
```

```
        print (' this is functionB')
```

```
    variable = decorator_funcn (funcA)
```

```
    variable( )
```

Lets understand the concept of Decorators through the above code. Here, decorator_funcn is defined to enhance the functionality of any function (may be funcA or funcnB) from printing “this is function A” to “this is a wonderful function this is function A” or “this is function B” to “this is a wonderful function this is function B”.

Without altering any line of code of already written functions. To achieve this a decorator function `decorator_funcn` is defined, this function takes `any_funcn` as input argument, it can be any function may it be `funcnA` or `funcnB`. Inside the `decorator_funcn`, a wrapper function named `wrapper_funcn()` is defined to wrapup the new features over the existing features of the function taken as argument by the `decorator_funcn()`. The body of the `wrapper_funcn()` includes the additional feature, in our case it is `print('this is a wonderful function')` an then the original function passed as an argument to the `decorator_funcn()` i.e. `any_funcn()` is called, and finally the output of `wrapper_funcn` is returned. To execute the `decorator_funcn`, the `decorator_funcn` with argument `funcA` is called and its return value is collected in variable, then `variable()` is executed and the output received is “this is a wonderful function this is function A” or “this is a wonderful function this is function B”.

Lets run some shortcuts also to work with the concept of decorators, which involves one more concept of syntactic sugar, where we use `@` symbol to use the decorators before executing any function, i.e. to enhance its functionality.

```
# Decorators – to enhance functionality of other functions  
# @ used for decorators  
  
defdecorator_funcn (any_funcn):  
    defwrapper_funcn():  
        print('this is a wonderful function')  
        any_funcn()  
        returnwrapper_funcn  
  
    @decorator_funcn  
    deffuncA():  
        print (' this is functionA')  
        funcA()  
  
    @decorator_funcn  
    deffuncB():  
        print (' this is functionB')  
        funcB()
```

whenever we use `@decorator_funcn` before any function the output of that function preceeds with the output “this is a wonderful function”, later the output of actual function turnsup. As in this code, calling `funcA()` leads to output “this is a wonderful function this is function A” or calling `funcB()` leads to output “this is a wonderful function this is function B”.

In this section we learned, the concept of decorators, as functionality enhancers. Now we will study the concept of Iterators and Iterables.

☛ Check Your Progress 1

- 1) Compare built-in functions with user defined and Lambda functions
 - 2) What are Decorators? Briefly discuss the utility of decorators.
-

15.3 ITERATORS

In Python Iterator is an object that can be iterated i.e. an object which will return data or one element at a time. Iterators exist every where, but they are implemented well in generators, comprehensions and even in for loops; but are generally hidden in plain sights. In this section we will try to understand this concept by exploring the functioning of for loops. In our earlier units we learned about the concepts of Lists, Tuples, Dictionaries, Sets , Strings etc., infact most of these built in containers are collectively called, iterables. An object is called iterable if we can get an iterator from it..In Python any iterator object must follow the iterator protocol i.e. the implementation of iter() and next() functions, the iter function returns an iterator, and an object is called iterable if we can get an iterator from it (for this we use iter()function). To understand what are iterables lets understand the functioning of for loop.

We learned that numbers = [1,2,3,4] is a list and if we want to print the elements of this list, we may use for loop for this purpose, and we will write

```
# Iterables  
numbers = [1,2,3,4]  
  
for i in numbers :  
  
    print (i)
```

now lets understand how for loop works behind the scenes, firstly the for loop calls a function called iter() function, this iter function changes the iterable to iterator i.e. it takes list as argument, so in our case we have iter(numbers), and this is now an iterator. Subsequently the next function is called whose argument is this iterator i.e. next(iter(numbers)), as the loop progresses the next function provides the values from the iterable i.e. the list numbers in our case, first run provides 1, second run provides 2 and so on. i.e to see how the for loop works we write the logic of for loop as follows

```
# Iterables  
numbers = [1,2,3,4]  
  
number_iter = iter(numbers)  
  
print(next(number_iter))     # Output will be 1
```

```

print(next(number_iter))      # Output will be 2

print(next(number_iter))      # Output will be 3

print(next(number_iter))      # Output will be 4

print(next(number_iter))      # Output will be Error as the iterable numbers
is upto 4 only

```

In this way only the for loop works on any of the iterables i.e. List or Tuple or String. So, iterables are the python data types which uses the iter and next functions, but iterator can directly use the next function. The iterables uses the iter() function to generate iterator and then uses next function but iterators don't use iter function they directly use the next function to get the job done. In Python any iterator object must follow the iterator protocol i.e. the implementation of iter() and next() functions, the iter function returns an iterator, and an object is called iterable if we can get an iterator from it (for this we use iter(function))

Now, we will learn about the iterators.

```

# Iterators

numbers = [1,2,3,4]  # Iterables

Squares = map(lambda a : a**2, numbers)  #Iterator

print( next(squares))  # output will be square of 1 i.e. 1

print( next(squares))  # output will be square of 2 i.e. 4

print( next(squares))  # output will be square of 3 i.e. 9

print( next(squares))  # output will be square of 4 i.e. 16

```

In this section we learned about the concept of Iterators and Iterables, now we will extend our discussion to Generators, in the next section.

☛ Check Your Progress 2

- 1) What are Iterators? How iterators differ from iterables?
- 2) Discuss the execution of for loop construct, from iterators point of view

15.4 GENERATORS

Generators are also a kind of iterators, but they are quite memory efficient i.e. needs very less memory hence helps in improving the performance of any programme. We learned in last section that iterators involves production of

any sequence, the generators are also generating the sequence but their modus operandi is quite different. Like List say $L = [1,2,3,4]$ is a sequence but it is an iterable, the generator is also a sequence but it is an iterator not a iterable. You might be thinking that we already have a mechanism to refer a sequence i.e. say List, then why do we need a generator. To understand this we need to understand the memory utilization by list and generator. Say, we are having a list with many numbers, when we create a list then it will take some time , secondly these numbers will get stored in to the memory i.e memory usage will also be on higher side. But, In the case of Generators, at one time only one number is generated and the same is used for further processing, i.e. both time and memory space are saved, they are comparatively quite less in case of generators. So, while processing the list entire list is loaded and processed, but in generators one by one elements are generated and are processed accordingly.

Now you might be thinking that, when to use lists then ? the answer is that when you need to use your sequence again and again (may be to perform some functionality) then list is the best option, but when you simply need to use the sequence for one time only, then its better to go for generators, you will understand this, later in this section only.

Lets learn how to write a generator, for this you may use two techniques, i.e. you may use generator function or generator comprehension, as technique to develop your own generator. Generator comprehension is the technique which is quite similar to list comprehension, which is used to generate list.

Firstly we will discuss about generator function, say we need to define a function which is suppose to take a number as an argument and that function is suppose to print number from 1 to that number say 10

We can write the following code to achieve the task:

```
defnums(n)
for i in range (1, n+1) :      # n+1 because for loop goes upto n-1 th term
    print(i)
nums(10)      # calling the function nums with n = 10
```

this will print the numbers from 1 to 10, here function is not returning any thing, but simply printing the numbers. This was quite simple, as you learned in your earlier units, but if we need to develop our generator to do the same task then you need to replace the print command with yield keyword, and the code will be

```
defnums(n)
```

```
for i in range (1, n+1) :      # n+1 because for loop goes upto n-1 th term
```

```
    yield(i)
```

```
nums(10)
```

This yield keyword will create a generator, on executing this code nothing will be printed, but if in place of nums(10) i.e. the last line of the above code, we write print(nums(10)) then on execution you will come to know a generator object nums is produced.

Note : a) In normal function we either print or return but in generator function we use yield keyword

b) yield is a keyword, and not a function, so we can write ‘yield i’ in place of ‘yield(i)’

Now, lets understand how a generator function works, for this lets again explore the functioning with for loop, refer to the code given below

```
defnums(n)
```

```
for i in range (1, n+1) :      # n+1 because for loop goes upto n-1 th term
```

```
    yield(i)
```

```
numbers = nums(10)
```

```
fornum in numbers :
```

```
    print(num)
```

Executing the above code the sequence of numbers from 1 to 10 will be generated, but if you again execute the for loop then nothing will be printed i.e.

```
defnums(n)
```

```
for i in range (1, n+1) # n+1 because for loop goes upto n-1 th term
```

```
    yield(i)
```

```
numbers = nums(10) # here nums(10) is a generator, we can transform  
nums(10) into list by writing list(nums(10))
```

```
fornum in numbers :
```

```
    print(num)
```

fornum in numbers :

```
print(num)
```

execution of the second for loop will not produce any result because the generators generates the numbers one by one and they are not retained in to the memory as in the case of lists. So the execution of first for loop will produce a sequence from 1 to 10, i.e. the numbers are placed in to the memory one by one, which is there after refreshed, so past instance is lost. Thus the execution of second for loop has no databasecauseenums(n) function only exists before first for loop not after it. If the nums(n) also exists after first for loop then data for second for loop is also available.

```
defnums(n)
```

```
for i in range (1, n+1) # n+1 because for loop goes upto n-1 th term
```

```
    yield(i)
```

```
numbers = list(nums(10)) #here nums(10) is a generator, we can transform  
nums(10) into list by writing list(nums(10))
```

fornum in numbers :

```
print(num)
```

fornum in numbers :

```
print(num)
```

If we re-execute the above code with list and not generator i.e. with list(nums(10)) and not nums(10), then the sequence from 1 to 10 will be printed twice because the content of List persists in the memory, thus the execution of both for loops will produce a separate sequence from 1 to 10.

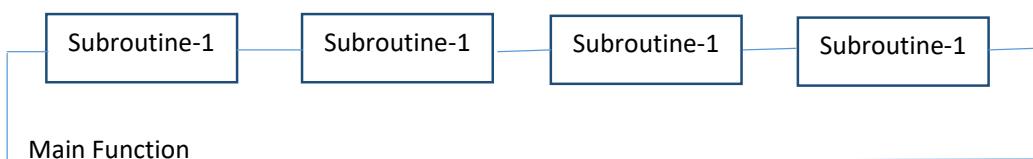
☛ Check Your Progress 3

- 1) What are generators in Python? Briefly discuss the utility of generators in python
- 2) Compare Generators and Lists

15.5 CO-ROUTINES

We learned about functions in our earlier units, and we knew that they are also referred as procedures, subroutines, sub-processes etc. In fact a function is packed unit of instructions, required to perform certain task. In a complex function the logic is to divide its working into several self-contained steps, which themselves are functions, such functions are called subroutines or

helper functions, these subroutines have single entry point. The coordination of these subroutines is performed by the main function.



The generalized co-routines are referred as subroutines, the working of co-routines relates to cooperative multitasking i.e. when a process voluntarily passes on (yield) the control to another process, periodically or when idle. Co-routines are cooperative that means they link together to form a pipeline. One co-routine may consume input data and send it to other which process it. Finally there may be a co-routine to display result.

This feature of co-routine helps for simultaneous processing of multiple applications. The Co-routines are referred as generalized subroutines, but there is a difference between subroutine and co-routine, the same are given below:

<u>Subroutines</u>	<u>Co-Routines</u>
<ol style="list-style-type: none"> 1. Co-routines have many entry points for suspending and resuming execution. 2. Co-routine can suspend its execution and transfer control to other co-routine and can resume again execution from the point it left off. 3. In Co-routines there is no main function to call co-routines in particular order and coordinate the results. 	<ol style="list-style-type: none"> 1. Subroutines have single entry point for suspending and resuming execution 2. Subroutines can't suspend its execution and transfer control to other subroutine and can resume again execution from the point it left off. 3. In Subroutines there is main function to call subroutines in particular order and coordinate the results.

From the above discussion it appears that co-routines are quite similar to threads, both seems to do the same job. But, there is a difference in between the thread and the Co-routine, in case of threads, it is the operating system i.e. the run time environment that performs switching in accordance with scheduler. But, in the case Co-routines the decision making for switching is performed by the programmer and programming language. In co-routines the cooperative multitasking, by suspending and resuming at set points is under the control of programmer.

In Python, co-routines are similar to generators but with few extra methods and slight change in how we use yield statement. Generators produce data for iteration while co-routines can also consume data. A generator is essentially a cut down (asymmetric) coroutine. The difference between a coroutine and generator is that a coroutine can accept arguments after it's been initially called, whereas a generator can't. In Python the Co-routines are declared with the `async` or `await` syntax, it is the preferred way of writing `asyncio` applications.

`asyncio` is a library to write concurrent code using the `async/await` syntax. `asyncio` is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc. To understand co-routine refer to following example code

Example, the following snippet of code (requires Python 3.7+) prints “hello”, waits 1 second, and then prints “world”:

```
import asyncio

asyncdef main():

    print ('hello')

    awaitasyncio.sleep(1)

    print ('world')

asyncio.run(main())
```

It is to be noted that by simply calling a co-routine, it will not be scheduled for the execution. To actually run a co-routine, `asyncio` provides three main mechanisms, listed below:

1. The `asyncio.run()` function to run the top-level entry point “`main()`” function (see the above example.)
2. Awaiting on a co-routine. (see the example given below)

Example - The following snippet of code will print “hello” after waiting for 1 second, and then print “world” after waiting for another 2 seconds:

```
importasyncio

import time

asyncdeftell_after(delay_time, what_to_tell):

    awaitasyncio.sleep(delay_time)

    print(what_to_tell)
```

```
asyncdef main():

    print(f"started at {time.strftime('%X')}")

    await tell_after(1,'hello')

    await tell_after(2,'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Expected output:

Started at 17:11:52

hello

world

finished at 17:11:55

3. The asyncio.create_task() function to run coroutines concurrently as asyncio Tasks.

Let's modify the above example and run two tell_after coroutines concurrently:

```
asyncdef main() :
```

```
    task1 = asyncio.create_task(tell_after(1,'hello'))

    task2 = asyncio.create_task(tell_after(2,'world'))

    print(f"started at {time.strftime('%X')}")

    # wait until both tasks are completed (should take around 2 seconds)

    await tell_after(1,'hello')

    await tell_after(2,'world')

    print(f"finished at {time.strftime('%X')}")
```

Expected Output:

Started at 17:24:32

hello

world

finished at 17:24:34

Note that expected output now shows that the snippet runs 1 second faster than before

☛ Check Your Progress 4

- 1) What are Co-routines? How they support cooperative multi-tasking in python
- 2) Compare Subroutines and Co-routines
- 3) How Co-routines differ from threads

15.6 SUMMARY

In this you learned about decorators, a way to enhance the functionality of already written functions, which ia useful concept for code reusability. Further, the discussion was enhanced to the concepts of iterables and iterators, through the understanding of the execution of For loop. There after the concept of generators was discussed where the concept of yield keyword and print command were mentioned, the comparative analysis between the generator and a function also clars the concept of performance improvement in python programming. Finally, the understanding of co-routines cleared the learners understanding towards the cooperative multitasking.

THE PEOPLE'S
UNIVERSITY

UNIT 16 DATA ACCESS USING PYTHON

Structure

- 16.1 Introduction
 - 16.2 Database Concepts
 - 16.3 Creating Database
 - 16.4 Querying Database
 - 16.5 Using SQL to get more out of Database
 - 16.6 CSV files in Python
 - 16.7 Summary
 - 16.8 Solutions to Check your Progress
-

16.1 INTRODUCTION

Python is the most popular high-level programming language that can be used for real-world programming. It is a dynamic and object-oriented programming language that can be used in a vast domain of applications, especially data handling. Python was designed as a very user-friendly general-purpose language with a collection of modules and packages and gained popularity in the recent times. Whenever we think of machine learning, big data handling, data analysis, statistical algorithms, python is the only name which comes first in our mind. It is a dynamically typed language, which makes it highly flexible. Furthermore, creating a database and linking of the database is very fast and secure. It makes the program to compile and run in an extremely flexible environment can support different styles of programming, including structural and object-oriented. Its ability to use modular components that were designed in other programming languages allows a programmer to embed the code written in python for any interface. Due to its enormous features, python has become the first choice of learners in the field of data science and machine learning. This unit is specially designed for beginners to create and connect database in python. You will learn to connect data through SQL as well as CSV files with the help of real dataset (PIMA Indian Dataset), used most commonly by researchers and academia in data analysis.

16.2 DATABASE CONCEPTS

The database is a collection of an interrelated, organized form of persistent data. It has properties of integrity and redundancy to serve numerous applications. It is a storehouse of all important information, not only for large business enterprises rather for anyone who is handling data even at the micro-level. Traditionally, we used to store data using file systems, which eventually taken by advanced Database Management Systems software, which stores, manipulates and secure the data in the most promising manner. To access the data stored in the database, one must know query language like SQL, MySQL etc., which can retrieve the information from the database server and passes it to the client. Thus, client can easily approach the server

for day to day transactions by hitting the database without worrying about the load on the processor as it is very fast and consumes power to run the desired query only.

The nature of serendipity in the Python environment is an example of data analysis in machine learning and datascience.Python is a bucket of packages and libraries which enables us to perform data classification,prediction and analysis in a very efficient way.

Now the question is, the database is stored in different forms in different software's like Oracle, MS-Access etc.,so how any programming language like python can accessit.For that, we need to learn about database connectivity in python. In this unit you will learn how to create and connect database in python using MySQL along with basic knowledge of database and its features.

Database Management system DBMS

Database management systems are software that serves the special purpose of storing,managing, extracting and modifying data from a database. It contains a set of programs that allows access to data contained in a database. DBMS also interfaces with application programs so that data contained in the database can be used by multiple application and users. It handles all access to the database and responsible for applying the authorization checks and validation procedures. It acts as a bridge between users and data, and the bridge is crossed using special query language like SQL, MYSQL wrote in high-level languages.

A typical DBMS has users with different rights and permissions who use it for different purposes. Some users retrieve data and some back it up. The users of a DBMS can be broadly categorized as follows –

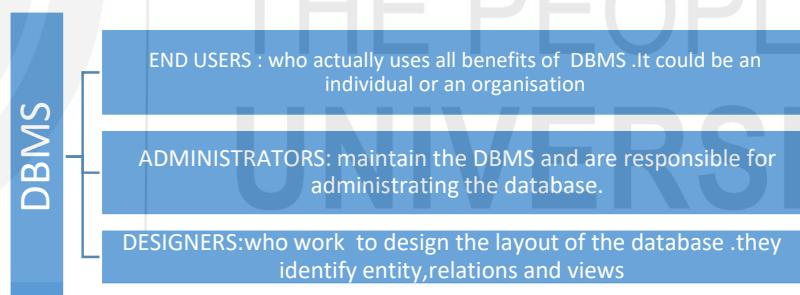


Fig 16.1 (a)Basic structure of users of DBMS

As we discussed no information processing is possible without a database. It is the hub of data in the field of information technology.Some of the major advantages of the database are listed below:

- **Real-world entity** – A modern DBMS is more realistic and uses real-world entities to design its architecture. It uses the behaviour and attributes too. For example, a school database may use students as an entity and their age as an attribute.
- **Reduced data redundancy**: Centralized control of database of avoids unnecessary duplication of data and effectively reduces the total amount of data storage required, unlike the non-database system in which each department is maintaining its own copy of data. Avoiding duplication of data also results in

- the elimination of data inconsistencies that tend to be present in duplicate data files.
- **Data consistency:** Centralized control over data, ensures data consistency as there is no duplication of data. In the non-database system, there are multiple copies of the same data with each department, whenever the data is updated, the changes are not reflected in each department which leads to data inconsistency. For example, change in address of the student is reflected in the teacher's record but not in account department, which results in inconsistent data.
 - **Shared data:** A database allows the sharing of data by multiple application, users and programs like a database of students can be shared by the teachers for making results as well as admin department to keep track of the fees account.
 - **Greater data integrity and independence from applications programs:** Integrity of data means that data stored in the database is both accurate and consistent. Centralized control ensures that adequate checks are incorporated in a database while entering data so that DBMS provide data integrity. DBMS provide sufficient validations to make sure that data fall within a specified range and are of the correct format.
 - **Improved data control:** In many organizations, where typically each application has its own private files. This makes the operational data widely dispersed and difficult to control. Central database provide the easy maintenance of the database Access to users through the use of host and query languages
 - **Improved data security:** The data is protected in a central system. Data is more secured against unauthorized access control with authentication and encoding mechanism. Different checks and rights can be applied for the access of information from the database
 - Reduced data entry, storage, and retrieval costs
 - Facilitated development of new applications program

Limitations of Database:

- **Substantial hardware and software start-up costs:** Extensive conversion costs is involved in moving from a file-based system to a database system
- **Database systems are complex,** difficult, and time-consuming to design.
- Damage to database affects virtually all applications programs
- Initial training required for all programmers and users.
- Cost: The cost of required hardware, DB development, and DB maintenance is high.
- Complexity: Due to its complexity, the user should understand it well enough to use it efficiently and effectively.

DBMS Architecture

DBMS is designed on the basis of architecture. This design can be of various forms centralized, client-server systems, parallel systems, distributed and hierarchical. The architecture of a DBMS can be seen as either a single tier or multi-tier. An n-tier

architecture divides the whole system into related but independent n modules, which can be independently modified, altered, changed, or replaced. Generally, three-tier architecture is followed, which consist of three layers:

1. Presentation layer (laptop, PC, Tablet, Mobile, etc.)
2. Application layer (server)
3. Database Server

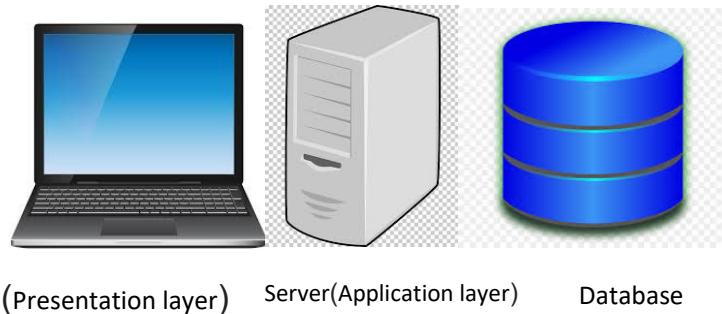


Fig 16.1(b)Three tier architecture of DBMS

Multiple-tier database architecture is highly modifiable, as almost all its components are independent and can be changed independently.

Data models

Once the architecture of DBMS is designed, then the next phase is modelling. The data model represents the logical structure of the database. It decides how the data will be stored, connected to each other, processed and stored. Many data models were proposed to store the data like network model, hierarchical model and relational model where each new model was invented with improved features. The first model was based on **Flat File**, where data is stored in a single table in a file, which could be a plain text file or binary file. It is suitable only for a small amount of data. Records follow a uniform format, and there are no structures for indexing or recognizing relationships between records. Later on, the concept of relation is introduced where data is stored in the form of multiple tables and tables are linked using a common field. It is suitable for handling medium to a large amount of data. This is the most promising model ever implemented for DBMS solutions. So, let us see in detail about the structure and features of the relational model.

Entity-Relationship Model

An **Entity-relationship model (ER model)** describes the logical structure of a database with the help of a diagram, which is known as an **Entity Relationship Diagram (ER Diagram)**. It is based on the notion of real-world entities and relationships among them. While formulating real-world scenario into the database model, the ER Model creates an entity set, relationship set, general attributes and constraints. ER Model is best used for the conceptual design of a database. ER Model is based on – Entities and their attributes. Relationships among entities. These concepts are explained below.

- **Entity:** An entity is a real-world object which can be easily identifiable like in an employee database, employee, department, products can be considered as entities.

- **Attributes:** Entities are represented by means of having properties called **attributes**. Every attribute is defined by its set of values called domain. For example, in an employee database, an employee is considered as an entity. An employee has various attributes like empname, age, department, salary etc.
- **Relationship:** describes the logical association among entities. These relationships are mapped with entities in various ways, and the number of association between two entities defines mapping cardinalities. Mapping cardinalities are one to one, one to many, many to many and many to one. For example, a doctor can have many patients in a database which shows one to many relationships.

These E-R diagrams are represented by special symbols which are listed below:

Rectangle: Represents Entity sets.

Ellipses: Attributes

Diamonds: Relationship Set

Lines: They link attributes to Entity Sets and Entity sets to Relationship Set

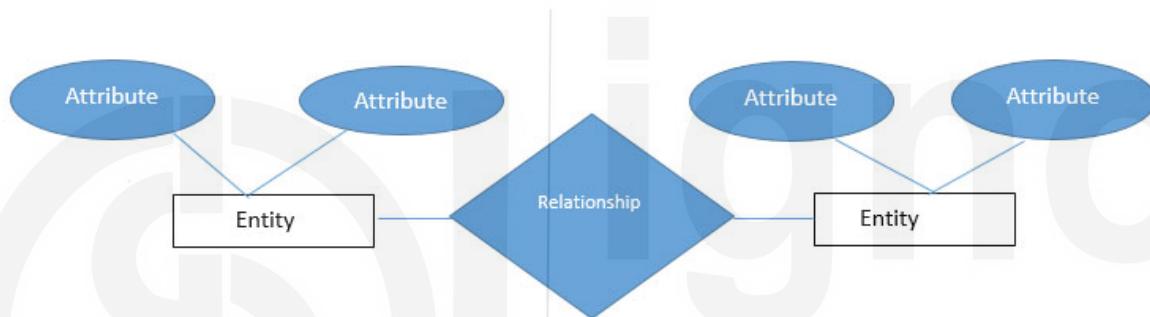


Fig 16.1(c) E-R diagram

Now let us take an example of an organization where they want to design an entity relationship between various departments and their employees. So, In the following E-R diagram we have taken two entities Employee and Department and showing their relationship. The relationship between Employee and Department is many to one as a department can have many employees however an employee cannot work in multiple departments at the same time. Employee entity has attributes such as Emp_Id, Emp_Name & Emp_Addr and Department entity has attributes such as Dept_ID & Dept_Name.

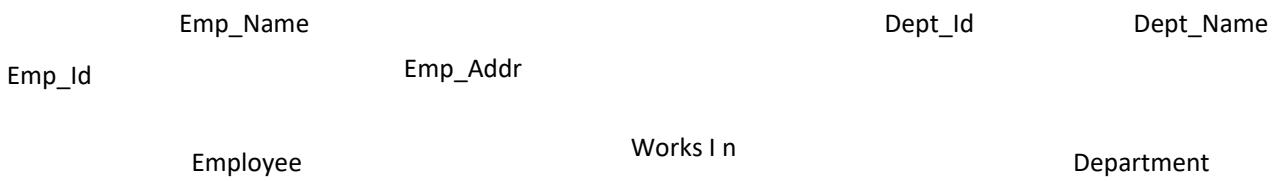


Fig 16.1(d) Sample E-R Diagram showing employee and department relationship.

Structure of Database

A database system is a computer-based system to record and maintain information. The information concerned can be anything of significance to the organization for whose use it is intended. The contents of a database can hold a variety of different things. To make database design more straight-forward, databases contents are divided up into two concepts:

- Schema
- Data

The Schema is the structure of data, whereas the Data are the "facts". Schema can be complex to understand to begin with, but really indicates the rules which the Data must obey. Let us consider a real-world scenario where we want to store facts about employees for an organization. Such facts may include Emp_name, Emp_address, Date of Birth, and salary. In a database, all the information on all employees would be held in a single storage "container", called a *table*. This table is a tabular object like a spreadsheet page, with different employees as the rows, and the facts (e.g. their names) as columns. Let's call this table EMP, and it could look something like:

Table EMP

Emp_name	Emp_address	Date of Birth	Salary
SmritiKumari	B-15 ,ParkAvenue Delhi	1/3/1986	71000
Samishtha Dutta	D- 45 ,Sector 37,Delhi	7/9/1989	93000
Rahul Sachdeva	C- 6 ,sector 45 Noida	3/2/1982	78000

Fig 16.1(e) Schema of Employee Database

From this information, the *Schema* would define that EMP has four components, "Emp_name", "Emp_address", "Date of Birth", "Salary". As database designers, we can call the columns what we like, a meaningful name helps. In addition to the name, we want to try and make sure that people don't accidentally store a name in the DOB column, or some other silly error. We can say things like:

- Emp_name is a string, holding a minimum of 12 characters.
- Emp_address is a string, holding a minimum of 12 characters
- Date of Birth is a date. The company can put validation for age must be greater than 18 and less than 60 years.
- A salary is a number. It must be greater than zero.

Such rules can be enforced by a database. The three schema architecture separates the user application and physical database. **The internal level schema or Physical Schema or internal Schema:** determines the physical storage structure of the database. **The conceptual Schema** is the high-level description of the whole database. It hides the detail of the physical storage and focuses on the data types, relationships user operations and constraints. **The external view or is the user view** that describes the view of different user groups.

An eminent scientist E.F.Codd proposed a database model, based on the relational structure of data. In this model data is organized in the form of tables, which is a collection of records and each record in a table consists of fields. A relational database allows the definition of data structures, storage and retrieval operations and integrity constraints. It is proved to be the most effective way of storing data.

Database Design in Relational Model

- Table:** A table is a set of data elements that is organized using a model of vertical columns and horizontal rows. Each row is identified by a unique key index or the key field.
- Columns/Field/Attributes:** A column is a set of data values of a particularly simple type, one for each row of the table. For eg. Emp_Code, Emp_Name, Emp_Address etc.
- ROWS/ RECORDS /TUPLES:** A row represents a single, data items in a table. Each row in a table represents a set of related data, and every row in the table has the same structure.
- DATA TYPES:** -Data types are used to identify the type of data we are going to store in the database.

In the relational model, every tuple must have a unique identification or key based on the data. In this figure, an employee code(Emp_code) is the key that uniquely identifies each tuple in the relation and declares as a primary key. Often, keys are used to join data from two or more relations based on matching identification.

Primary Key: A primary key is a unique value that identifies a row in a table. These keys are also indexed in the database, making it faster for the database to search a record. All values in the primary key must be unique and NOT NULL, and there can be only one primary key for a table.

Foreign Key: The foreign key identifies a column or set of columns in one (referencing) table that refers to a column or set of columns in another (referenced) table.

TABLE EMPLOYEE

COLUMNS OR FIELD OR ATTRIBUTES				
Emp_Code	Emp_Name	Emp_Address	Sal	DOB
101	Madhuri	112,Sector-45 ,Noida	85000	4.12.1982
102	Ankush	82,Sector-75 ,Noida	86000	14.11.1983
103	Samay	172,Sector-35 ,Noida	78000	2.10.1981
104	Tripti	52,Sector-85 ,Noida	91000	8.11.1980

Fig 16.1(f) Components of Relational Database

The relational model indicates that each row in a table is unique. If you allow duplicate rows in a table, then there's no way to uniquely address a given row via programming. This creates all sorts of ambiguities and problems that are best avoided. You guarantee uniqueness for a table by designating a primary key—a column that contains unique values for a table. Each table can have only one primary key, even though several columns or combination of columns may contain unique values.

All columns (or combination of columns) in a table with unique values are referred to as candidate keys, from which the primary key must be drawn. All other candidate key columns are referred to as alternate keys. Keys can be simple or composite. A simple key is a key made up of one column, whereas a composite key is made up of two or more columns.

The relational model also includes concepts of foreign keys, which are primary keys in one relation, that are kept as non-primary key in another relation, to allow for the joining of data.

Now let us see how to choose the primary key in the table. Consider the Product table given below:

Relating to the employee Table presented in the last section, Now define a second table, order as shown in the figure provided below:

PRODUCT TABLE		
Product_ID	EmpCode	Order_Date
P101	101	5/1/2020
P201	102	10/4/2020
P301	103	15/5/2020
P401	104	20/6/2020

Fig 16.1(g) Demo table of Foreign Key

Product_ID is the PrimaryKey in above table Product. Emp_code is considered a FOREIGN KEY in ProductTable, since it can be used to refer to given Product in the Product table.

COMMANDS TO DEFINE STRUCTURE AND MANIPULATE THE DATA

In a database, we can define the structure of the data and manipulate the data using some commands. The most common data manipulation language is SQL. SQL commands are instructions used to communicate with the database to perform a specific task that works with data. SQL commands can be used not only for searching the database but also to perform various other functions like, for example, you can create tables, add data to tables, or modify data, drop the table, set permissions for users. SQL commands are grouped into four major categories depending on their functionality:

- **Data Definition Language (DDL)** - These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.
- **Data Manipulation Language (DML)** - These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.
- **Transaction Control Language (TCL)** - These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.
- **Data Control Language (DCL)** - These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.
- Some of the commonly used datatypes in SQL AND MYSQL are listed below:

Data type	Description
CHAR(n)	Character string, fixed length n ,with a maximum size of 2000 characters . Values of this data type must be enclosed in single quotes “ .the storage size of the char value is equal to the maximum size for this column i.e. nColumns , that will not be used for arithmetic operations usually are assigned data types of char.
CHARACTER ,VARYING(n) or VARCHAR(n) ,VARCHAR2(n)	Variable length character string , can store upto 4000 characters . varchar is a variable-length data type, the storage size of the varchar value is the actual length of the data entered, not the maximum size for this column i.e .n
INT	A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647.
DATE	This data type allows to store valid date type data from January 1, 4712 BC to December 31, 4712 AD with standard oracle data format DD-MM-YY.
TIME	Stores the time in a HH:MM:SS format.
TIMESTAMP	A timestamp between midnight, January 1 st , 1970 and sometime in 2037. This looks like the previous DATETIME format, only without the hyphens between numbers; 3:30 in the afternoon on December 30 th , 1973 would be stored as 19731230153000 (YYYYMMDDHHMMSS).

SQL COMMANDS

DDL COMMANDS[CREATE, ALTER,DROP]

- **CREATE**

It is the mostly used, Data definition command in SQL . It is used to create table index or view. This command describes the layout of the table. The create statement specifies the name of the table , names and types of each column of the table . Each table must have at least one column. The general syntax for creating table is shown below:

Syntax for CREATE TABLE is:

```
CREATE TABLE <table name>
    (<attribute name><data type>[<size>]<column constraint>,
     <attribute name><data type>[<size>]<column constraint>,
     .....)
```

where

<**table name** > : It is the name, of the table to be created

<**attribute name** > : It is the name of the column heading or the field in a table.

<**data Type**> : It defines the type of values that can be stored in the field or the column of the table .

Example:

```
CREATE TABLE Student
```

```
(      roll_no number(5),
       name   char(20),
       birth_data date );
```

- **ALTER**

Alter is a DDL command in SQL which is used to perform the following operations in table.

1. Add column to the existing table
4. To modify the column of the existing table.

- **Adding column(s) to a table**

To add a column to an existing table, the ALTER TABLE syntax is:

```
ALTER           TABLE          table_name
(ADD column1_name column-definition);
```

Example :

```
ALTER TABLE PRODUCT ADD (SALESvarchar2 (50));
```

This will add a column called *sales* to the *product* table.

- **Modifying column(s) in a table**

To modify a column in an existing table, the ALTER TABLE syntax is:

```
ALTER           TABLE          table_name
    MODIFY (column1_name column_type ,
               [column2_name column_type,]
               .....);
```

Example:

```
ALTER           TABLE          PRODUCT
MODIFY Prod_name varchar2(100) not null;
```

This will modify the column called *prod_name* to be a data type of varchar2(100) and force the column to not allow null values.

- **Drop column(s) in a table**

To drop a column in an existing table, the ALTER TABLE syntax is:

ALTER TABLE	table_name	Data Access Using Python
DROP COLUMN column_name;		

Example:

```
ALTER TABLE supplier DROP COLUMN Prod_name;
```

This will drop the column called *prod_name* from the table called *Product*.

DML COMMANDS –[INSERT,UPDATE,DELETE]

- **INSERT Statement** is used to add new rows of data to a tablet. The value of each field of the record is inserted in the table.

Syntax :

```
INSERT INTO <table _name> [(col1, col2, col3,...colN)]
VALUES (value1, value2, value3,...valueN);
where col1, col2,...colN -- the names of the columns in the table into which you want to insert data .
```

Example: If you want to insert a row to the employee table, the query would be like,

```
INSERT INTO employee (emp_id, emp_name, emp_dept, age, salary )
```

```
VALUES (105, 'Vaishnavi', 'Medical', 27, 33000);
```

NOTE: When adding a row, only the characters or date values should be enclosed with single quotes.

- **SQL SELECT Statement**

The most commonly used SQL command is **SELECT statement**. The SQL **SELECT statement**, the only data retrieval in SQL , used to query or retrieve data from a table in the database.

Syntax of SQL SELECT Statement:

SELECT	[distinct]	column_list	FROM	table-name
[WHERE				Clause]
[GROUP		BY		clause]
[HAVING				clause]
[ORDER BY clause];				

- **table-name** is the name of the table from which the information is retrieved.
 - **column_list** includes one or more columns from which data is retrieved.
 - The code within the brackets are optional.
- Example : Write the query in SQL to perform the following in the given Table Student.

R.No	Name	Course	Age	Stream	Sports	Marks	Grade
11	Rajat	BCA	15	Science	Cricket	78	B
12	Anuja	MCA	16	Science	Football	80	B
13	Munil	BCA	15	Science	Cricket	78	B
14	Sonia	MCA	16	Humanities	Badminton	95	A
15	Adya	MCA	15	Commerce	Chess	96	A

- a. Display the name of all students from the table Student .
- b. Display the name of students who are in course 'MCA'
- c. Display the name of students who play 'Chess'
- d. Display the name of students according to the marks obtained in descending order.

Solution

- ```

a. SELECT * FROM student;

SELECT ALL FROM student;

b. SELECT * FROM student where course='MCA';
c. SELECT * FROM student where sports='Chess';
d. SELECT * FROM student ORDERBY Marks Desc;

```

Note: SQL is case insensitive

- **UPDATE**

Update statement is used to modify the existing data of the tables .

|                          |                       |                           |
|--------------------------|-----------------------|---------------------------|
| <b>UPDATE</b>            |                       | <i>table_name</i>         |
| <b>SET</b>               | <i>column1=value,</i> |                           |
| <b>WHERE</b> <condition> |                       | <i>column2=value2,...</i> |

**Remember :** The WHERE clause in the UPDATE syntax specifies which record or records that should be updated. If you remove the WHERE clause, all records will be updated by default.

Example :Modify the marks of Anuja from 80 to 87 which was entered wrongly.

```

UPDATE student
SET marks=87
WHERE name='Anuja' and marks='80';

```

- **DELETE Statement**

The DELETE statement allows you to delete a single record or multiple records or all records from the table.

Syntax:

|                                 |             |              |
|---------------------------------|-------------|--------------|
| <b>DELETE</b>                   | <b>FROM</b> | <b>table</b> |
| <b>WHERE &lt;condition&gt;;</b> |             |              |

**Example :**

**DELETE FROM Student WHERE course = 'MCA';**

This would delete all records from the Student table where the course opted by students is MCA

- **DCL (Data Control Language)**

DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.

Examples of DCL commands:

- GRANT-gives user's access privileges to database.
- REVOKE-withdraw user's access privileges given by using the GRANT command.

- **TCL (transaction Control Language)**

TCL commands deals with the transaction within the database.

Examples of TCL commands:

- COMMIT-commits a Transaction.
- ROLLBACK-rollbacks a transaction in case of any error occurs.
- SAVEPOINT-sets a savepoint within a transaction.
- SET TRANSACTION-specify characteristics for the transaction.

**CHECK YOUR PROGRESS:**

1. Define the term database. List its four important features.
2. What do you understand by DDL and DML?
3. How will you explain the term entity and attributes in the table?
4. What is Primary Key?

---

## 16.3 CREATING DATABASE

---

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces follow this standard. Python Database API supports a wide range of database servers such as MySQL, PostgreSQL, Microsoft SQL Server 2000, Informix, Interbase, Oracle, Sybase. As a data scientist, you have the freedom to choose the appropriate server for our project. To do so, we need to download a separate DB API module for each database you need to access. For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following –

- Importing the API module.
- Acquiring a connection with the database.

- Issuing SQL statements and stored procedures.
- Closing the connection

ANACONDA  
JUPYTER /SPYDER  
NOTEBOOK

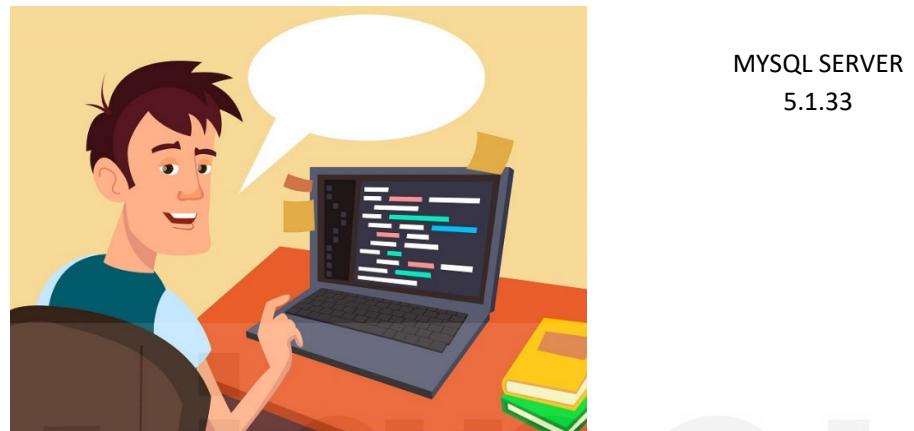


Fig: shows all applications required to install for setting up database connectivity in python with mysql.

#### **Steps for setting python In Anaconda for Database connectivity**

1. Install **Anaconda** — <https://www.anaconda.com/distribution/>
2. Select **python version 3.7 (preferably)**
3. MySQL Server (any version: we have used 5.1.33)
4. Mysql.connector or MySQLdb

NOTE: Anaconda is a python and R distribution that aims to provide everything you need:

- core python language,
- python packages,
- IDE/editor — Jupyter and Spyder
- Package manager — Conda, (for managing your packages)

Once it is done, we are ready to have database connectivity in python.

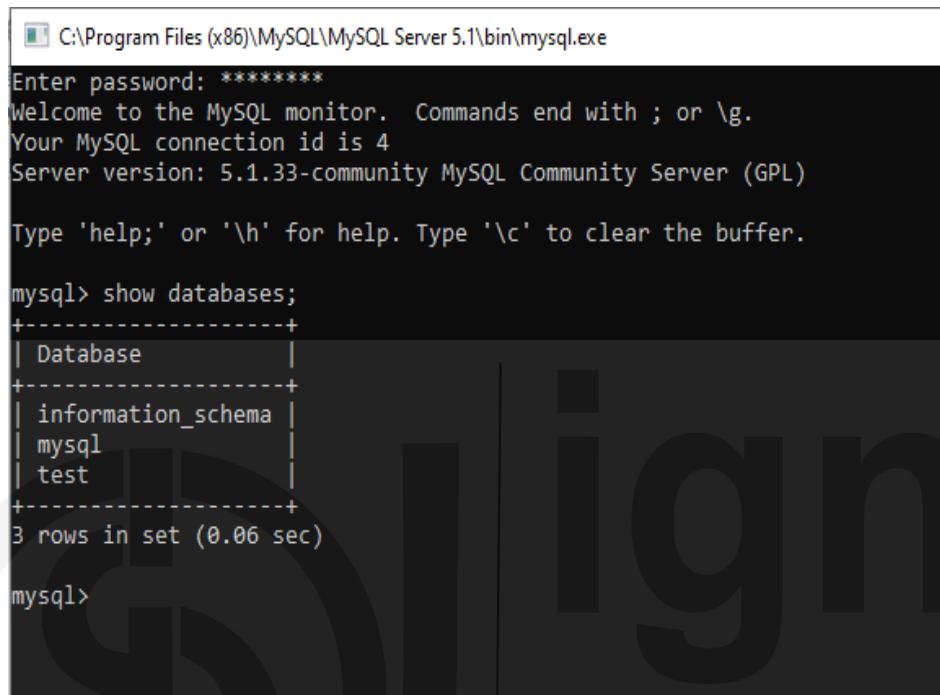
#### **Steps to install MySql Setup**

1. Download and install **MySql Community server** —  
<https://downloads.mysql.com/archives/community/>
2. During the installation setup, you will be prompted for a "root" password in the server configuration step.
3. Launch workbench, at the home page, setup a new connection profile with the configuration (Connection method: Standard (TCP/IP), Hostname:

127.0.0.1,Port:3306,Username: root, Password: *yourpassword*) and test your connection.

4. Double click on your local instance and it should bring you the schemas view where you can see all your databases and tables.

The following screenshot gives a glimpse of the screen you may get:



C:\Program Files (x86)\MySQL\MySQL Server 5.1\bin\mysql.exe  
Enter password: \*\*\*\*\*  
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 4  
Server version: 5.1.33-community MySQL Community Server (GPL)  
  
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.  
  
mysql> show databases;  
+-----+  
| Database |  
+-----+  
| information\_schema |  
| mysql |  
| test |  
+-----+  
3 rows in set (0.06 sec)  
  
mysql>

### Mysql.connector

It is an interface for connecting to a MySQL database server from python. It implements the Python Database API v2.0 and is built on top of the MySQL C API. It consists of four methods which are used to establish connection. The following methods are listed below:

- **Steps to install mysql.connector in Anaconda :**

To install mysql.connector navigate to the anaconda prompt and type the following command

1. Using terminal and **conda** to download

```
conda install -c anaconda mysql-connector-python
```

Another way to install mysql.connector is on windows command prompt for using Python d

1. Download Mysql API, exe file and install it.(click here to download)

2. Install mysql-Python Connector (Open command prompt and execute command)

```
>pip install mysql-connector
```

3. Now connect Mysql server using python

4. Write python statement in python shell import mysql.connector

If no error message is shown means mysql connector is properly installed. Mysqlconnector has four important methods which is used to establish and retrieve records from the database.

1. **connect()** : This method is used for creating a connection to our database it have four arguments:
  - a. Host Name
  - b. Database User Name
  - c. Password
  - d. Database Name
2. **cursor()** : This method creates a cursor object that is capable for executing sql query on database.
3. **execute ()**: This method is used for executing SQL query on database. It takes a sql query (as string) as an argument.
4. **fetchone()** : This method retrieves the next row of a query result set and returns a single sequence, or none if no more rows are available.
5. **close ()**: This method close the database connection.

➤ In this chapter, we will be using Anaconda Jupyter notebook for demonstration of all programs

### Connect to MySql

1. Launch Anaconda-Navigator to bring you a list of applications available to install. The application we are interested in is **Jupyter Notebook** which is a web-based python IDE. Install and launch it.
2. In the notebook, create a new python file. In the first cell, write the following code to test the mysql connection.

```
importmysql.connector

mydb = mysql.connector.connect(
 host="localhost",
 user="root",
 passwd="password",charset='utf8'
)

print(mydb)
```

Optional required only when this error comes:  
**ProgrammingError:**  
1115 (42000): Unknown character set: 'utf8mb4'

3. If successful, you should get an object returned with its memory address  
`<mysql.connector.connection_cext.CMySQLConnection object at 0x10b4e1320>`

**Cursor object:** The MySQLCursor class instantiates objects that can execute operations such as SQL statements. Cursor objects interact with the MySQL server using a MySQLConnection object.

#### How to create a cursor object and use it import mysql.connector

Example

Following is the example of connecting with MySQL and display the version of the database.

```
importmysql.connector

Open database connection

mydb = mysql.connector.connect (
 host="localhost",
 user="root",
 passwd="admin123",charset='utf8'
)

prepare a cursor object using cursor() method
mycursor= mydb.cursor() → creates a cursor object that is capable for
 executing sql query on database.

execute SQL query using execute() method.
mycursor.execute("SELECT VERSION()")

Fetch a single row using fetchone() method.
data= mycursor.fetchone() ↗ retrieves the next row of a query result set

print ("Database version :",data)

disconnect from server
mydb.close() ↗ close the database connection.
```

While running this script, it is producing the following result in my machine. (It could change in your system)

Database version : ('5.1.33-community',)

In the above code we are creating a cursor to execute the sql query to print database version next line executes the sql query show databases and store result in mycursor as collection ,whose values are being fetched in data. On execution of above program cursor will execute the query and print version of database shown.

#### Creating Database

1. Let's create a database called *TEST\_DB IN Python Anaconda* .

```
importmysql.connector

mydb = mysql.connector.connect(
 host="localhost",
 user="root",
 passwd="admin123",charset='utf8'
)
mycursor = mydb.cursor()
mycursor.execute("CREATE DATABASE TEST_DB")
```

Next, we will try to connect to this new database.

```
importmysql.connector

mydb = mysql.connector.connect(
 host="localhost",
 user="root",
 passwd="admin123",charset='utf8',database= TEST_DB
)
```

#### CHECK YOUR PROGRESS

1. How will you create a database in mysql .Write the command to create a database named as Start\_db.
2. Name the four arguments required to set connection with mysql.connector.connect.

---

## 16.4 QUERYING DATABASE

---

In this section, you will learn to execute SQL queries at run time of the Anaconda environment. To perform SQL queries you need to create a table in the database and then operations like insert, select, update and delete are shown with examples.

- **Creating Table**

Once a database connection is established, we are ready to create tables or records into the database tables using the **execute** method of the created cursor.

#### Example

To create table EMPLOYEE in database TEST\_DB with following data FIRST\_NAME, LAST\_NAME, AGE, SEX, INCOME.

```

import mysql.connector
mydb = mysql.connector.connect(
 host="localhost",
 user="root",
 passwd="admin123",charset='utf8',database="TEST_DB"
)
prepare a cursor object using cursor() method
cursor = mydb.cursor()
Drop table if it already exist using execute() method.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
Create table as per requirement
sql = """CREATE TABLE EMPLOYEE (
 FIRST_NAME CHAR(20) NOT NULL,
 LAST_NAME CHAR(20),
 AGE INT,
 SEX CHAR(1),
 INCOME FLOAT)"""
cursor.execute(sql)
#Get database table
cursor.execute("SHOW TABLES")
for table in cursor:
 print(table)
disconnect from server
mydb.close()

('employee',)

```

- **How to change table structure/(add,edit,remove column of a table) at run time**

To modify the structure of the table, we just have to use alter table query.

Below program will add a column address in the EMPLOYEE table.

```

import
mysql.connectormydb=mysql.connector.connect(host="localhost",user="root",
passwd ="root",database="school")
mycursor=mydb.cursor()
mycursor.execute("alter table student add (address varchar(2))")
mycursor.execute("desc employee")
for x in mycursor:
 print(x)

```

Above program will add a column marks in the table student and will display the structure of the table

- **INSERT Operation**

Insert is used to feed the data in the table created. It is required when you want to create your records into a database table.

## Example

The following example, executes SQL *INSERT* statement to create a record into EMPLOYEE table

```
import mysql.connector

mydb = mysql.connector.connect(
 host="localhost",
 user="root",
 passwd="admin123",charset='utf8',database="TEST_DB"
)

prepare a cursor object using cursor() method
cursor = mydb.cursor()
Prepare SQL query to INSERT a record into the database.

sql = "INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME) VALUES('CHINMAY', 'SWAMI', 25, 'M', 50000)"

cursor.execute(sql)
mydb.commit()
print(cursor.rowcount, "Record Inserted")
```

1 Record Inserted

- **UPDATE OPERATION**

UPDATE Operation on any database means to update one or more records, which are already available in the database. Here is an example to show where we run the query to updates all the records having SEX as 'M'. Here, we increase the AGE of all the males by one year.

## Example

```
import mysql.connector
mydb=mysql.connector.connect(host="localhost",user="root",
 passwd ="admin123",database="TEST_DB",charset='utf8')
mycursor=mydb.cursor()

Prepare SQL query to UPDATE required records
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = ('M')"
try:
 # Execute the SQL command
 mycursor.execute(sql)
 # Commit your changes in the database
 mydb.commit()
except:
 # Rollback in case there is any error
 mydb.rollback()

disconnect from server
mydb.close()
```

The above code will update the age of all the male employees in the table EMPLOYEE.

- **DELETE Operation**

DELETE operation is required when you want to delete some records from your database. Following example show the procedure to delete all the records from EMPLOYEE where AGE is more than 20.

```

import mysql.connector
mydb=mysql.connector.connect(host="localhost",user="root",
 passwd ="admin123",database="TEST_DB",charset='utf8')
mycursor=mydb.cursor()

Prepare SQL query to UPDATE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > 20"
try:
 # Execute the SQL command
 mycursor.execute(sql)
 # Commit your changes in the database
 mydb.commit()
except:
 # Rollback in case there is any error
 mydb.rollback()

disconnect from server
mydb.close()

```

The above code will delete all the records from the table EMPLOYEE where age is greater than 20.

### **Steps to perform SQL queries in the database using Python**

Step1. Import mysql.connector

Step 2. Create a variable and assign it to mysql.connector.connect().

Step3. Provide arguments host,user,passwd,database, charset to mysql.connector.connect().

Step4. Call cursor() and assign it to variable.

Step5. Prepare SQL query (Insert or Update or delete)

Step 5. Disconnect from the server using close() function.

Please refer to the above examples with their screenshots which will help in better understanding.

### **CHECK YOUR PROGRESS:**

1. Write the syntax of the following commands in SQL : insert, update, delete

---

## **16.5 USING SQL TO GET MORE OUT OF DATABASE**

---

### **READ Operation**

READ operation on any database means to fetch some useful information from the database.

Once our database connection is established, you are ready to make a query into this database. You can use either **fetchone()** method to fetch a single record or **fetchall()** method to fetch multiple values from a database table.

- **fetchall()** – It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.

- **fetchone()** – It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **rowcount** – This is a read-only attribute and returns the number of rows that were affected by an execute() method.
- **fetchall()**

The method fetches all (or all remaining) rows of a query result set and returns a list of tuples. If no more rows are available, it returns an empty list. The following procedure queries all the records from the EMPLOYEE table

```
import mysql.connector

mydb = mysql.connector.connect(
 host="localhost",
 user="root",
 passwd="admin123",charset='utf8',database="TEST_DB"
)

prepare a cursor object using cursor() method
cursor = mydb.cursor()
Prepare SQL query to SELECT ALL records| from the database.

sql = "SELECT * FROM EMPLOYEE"

cursor.execute(sql)
record=cursor.fetchall()
for x in record:

 print(x)

('CHINMAY', 'SWAMI', 25, 'M', 50000.0)
```

- **fetchone()**

To fetch one record from the table fetchone is used in the same manner as fetchall() is shown in above code. This method retrieves the next row of a query result set and returns a single sequence, or None if no more rows are available. By default, the returned tuple consists of data returned by the MySQL server, converted to Python objects.

```
Fetch a single row using fetchone() method.

record = cursor.fetchone()
```

- **rowcount()**

Rows affected by the query. We can get a number of rows affected by the query by using rowcount. We will use one SELECT query here.

```
import mysql.connector
mydb=mysql.connector.connect(host="localhost",user="root",passwd="admin123",database="TEST_DB")
mycursor=mydb.cursor()
mycursor = mydb.cursor(buffered=True)
mycursor.execute("select * from employee")
noofrows=mycursor.rowcount
print("No of rows in employee table are",noofrows)
```

In the above code buffered=True. We have used mycursor as buffered cursor which fetches rows and buffers them after getting output from MySQL database. It is used as an iterator, but there is no point in using buffered cursor for the single record as in such case if we don't use a buffered cursor, then we will get -1 as output from rowcount.

- **Manage Database Transaction**

Database transaction represents a single unit of work. Any operation which modifies the state of the MySQL database is a transaction. Python MySQL Connector provides the following method to manage database transactions.

- **Commit** – MySQLConnection.commit() method sends a COMMIT statement to the MySQL server, committing the current transaction.
- **Rollback** – MySQLConnection.rollback() revert the changes made by the current transaction.
- **autoCommit** – MySQLConnection.auto-commit value can be assigned as True or False to enable or disable the auto-commit feature of MySQL. By default, its value is False.

#### Steps to perform commit, rollback and auto-commit.

Step 1. Import the MySQL connector python module

Step 2. After a successful MySQL connection, set auto-commit to false, i.e., we need to commit

the transaction only when both the transactions complete successfully.

Step 3. Call the cursor() function .

Step4. Execute the queries using a cursor. Execute method.

Step 5 .After successful execution of the queries, commit your changes to the database using

a conn.commit() .In case of an exception or failure of one of the queries, you can revert your changes using a conn.rollback()

We placed all our code in the "try-except" block to catch the database exceptions that may occur during the process.

The following code shows the role of commit, rollback and auto-commit while performing SQL queries in python.

```

try:
 conn = mysql.connector.connect(host='localhost', database='TEST_DB', user='root', passwd='admin123')
 conn.autocommit = False
 cursor = conn.cursor()
 sql_update_query = """Update EMPLOYEE set INCOME = 95000 where FIRST_NAME = raghav"""
 cursor.execute(sql_update_query)
 print("Record Updated successfully ")
 #Commit your changes
 conn.commit()
except mysql.connector.Error as error :
 print("Failed to update record to database rollback: {}".format(error))
 #reverting changes because of exception
 conn.rollback()
finally:
 #Closing database connection.
 if(conn.is_connected()):
 cursor.close()
 conn.close()
 print("connection is closed")

```

In the above code if update query is successfully executed then commit() method will be executed otherwise, an exception error part will be executed. Rollback issued to revert of update query if happened due to error. Finally, we are closing cursor as well as connection. Here the purpose of conn. Auto-commit = false is to activate the role of rollback else rollback will not work.

#### CHECK YOUR PROGRESS:

1. What is the purpose of rollback and commit in SQL.
2. Fill in the blank provided :

```

importmysql.connector
mydb = mysql._____ .connect(
 host="localhost",
 _____ ="myusername",
 passwd="mypassword"
)
mycursor = mydb._____()
mycursor._____ ("CREATE DATABASE mydatabase")

```

## 16.6 CSV FILES IN PYTHON

As we have studied various operations on databases, the next thing which comes in our way to implement real dataset for machine learning purpose using python. For the same, we should be aware about all the basic types of database connectivity among which the most common type for connecting real dataset is CSV (Comma Separated Values).

CSV files are ordinarily made by programs that handle a lot of information. It is just like a text file in a human-readable format which is used to store tabular data in a spreadsheet or database. They are a helpful method to send out information from spreadsheets and data sets just as import or use it in different projects. For instance, you may trade the aftereffects of an information mining project to a CSV document and afterwards import that into a spreadsheet to dissect the information, create charts for an introduction, or set up a report for distribution. The separator character of CSV files is called a delimiter. Default delimiter is comma (,) others are tab (\t), (: ), (;) etc.

CSV documents are exceptionally simple to work for database connectivity. Any language that underpins text record info and string control (like python) can work with CSV documents straightforwardly.

Some of the features of CSV are highlighted below, which makes it exceptionally useful in the analysis of the large dataset.

- Simple and easy to use.
- Can store a large amount of data.
- He was preferred to import and export format for data handling.
- Each line of the file is a record.
- Each record consist of fields separated by commas(delimiter)
- They are used for storing tabular data in a spreadsheet or database.

Let us see how to import CSV file in python.In this unit, we will be using pandas module in python to import the CSVfile.*Pandas* is a powerful Python package that can be used to perform statistical analysis. In this chapter, you'll also see how to use Pandas to calculate stats from an imported CSV file.

### CASE STUDY OF PIMA INDIAN DATASET

The dataset used here isPima Indian diabetes data for learning purpose.Pima Indian diabetes dataset describes the medical records for Pima Indiansand whether or not each patient will have an onset of diabetes within given years.

Fields description follow:

preg = number of times pregnant

plas = Plasma glucose concentration a 2 hours in an oral glucose tolerance test

pres = Diastolic blood pressure (mm Hg)

skin = Triceps skin fold thickness (mm)

test = 2-Hour serum insulin (mu U/ml)

mass = Body mass index (weight in kg/(height in m)<sup>2</sup>)

pedi = Diabetes pedigree function

age = Age (years)

class = Class variable (1:tested positive for diabetes, 0: tested negative for diabetes)

(This dataset can be downloaded from

<https://www.kaggle.com/kumargh/pimaindiansdiabetescsv?select=pima-indians-diabetes.csv>)

It consists of above mentioned eight features, and one class variableandis very commonly used in the research of diabetes.Given below are the steps and the code to import CSV file in python.

Steps to import a CSV file into Python Using Pandas.

Step1. Import python module pandas.

Step2. Capture the file path where CSV file is stored (don't forget to include filename and file extension ).

Step3. Read the CSV file using `read_csv`

Step4. Print the desired file.

Step5 .Run the code to generate the output.

```
import pandas as pd

df = pd.read_csv ('F:\pimaindiansdiabetescsv\pima-indians-diabetes.csv')

print (df)
```

Note: #read the csv file (put 'r' before the path string to address any special characters in the path, such as '\'). Don't forget to put the file name at the end of the path + ".csv".

This path is taken for reference; it will change as per the location of the file.

#### Output

```
6 148 72 35 0 33.6 0.627 50 1
0 1 85 66 29 0 26.6 0.351 31 0
1 8 183 64 0 0 23.3 0.672 32 1
2 1 89 66 23 94 28.1 0.167 21 0
3 0 137 40 35 168 43.1 2.288 33 1
4 5 116 74 0 0 25.6 0.201 30 0
...
762 10 101 76 48 180 32.9 0.171 63 0
763 2 122 70 27 0 36.8 0.340 27 0
764 5 121 72 23 112 26.2 0.245 30 0
765 1 126 60 0 0 30.1 0.349 47 1
766 1 93 70 31 0 30.4 0.315 23 0
```

[767 rows x 9 columns]

The above output consists of 769 rows and ninecolumns.TheCSV file is just like excel file which consists of data in tabular form arranged as rows and columns.But the columns are without heading.In the next example, column headings are provided using

```
Colnames=['col1 ',' col2','col3','col4 ']
```

Any userdefined  
name

These are the field names or  
column names

- **Calculate stats using Pandas from an Imported CSV File.**

Finally you will learn to calculate the following statistics using the Pandas package:

- Mean
- Total sum

- Maximum
- Minimum
- Count
- Median
- Standard deviation
- Variance

Simple functions for each of the above mentioned stats is available in pandas package which can be easily applied in the following manner.

```
import pandas as pd
colnames=['pre','pgc','dbp','tsf','2hs','bmi','dpf','age','class']
df=pd.read_csv('F:\pimaindiansdiabetescsv\pima-indiansdiabetes.csv',names=colnames)
mean1 = df['age'].mean()
sum1 = df['age'].sum()
max1 = df['age'].max()
min1 = df['age'].min()
count1 = df['age'].count()
median1 = df['age'].median()
std1 = df['age'].std()
var1 = df['age'].var()

print block 1
print ('Mean Age: ' + str(mean1))
print ('Sum of Age: ' + str(sum1))
print ('Max Age: ' + str(max1))
print ('Min Age: ' + str(min1))
print ('Count of Age: ' + str(count1))
print ('Median Age: ' + str(median1))
print ('Std of Age: ' + str(std1))
print ('Var of Age: ' + str(var1))
```

Output:

```
Mean Age: 33.240885416666664
Sum of Age: 25529
Max Age: 81
Min Age: 21
Count of Age: 768
Median Age: 29.0
Std of Age: 11.76023154067868
Var of Age: 138.30304589037365
```

Here is a table that summarizes the operations performed in the code:

| Variable | Syntax             | Description                                 |
|----------|--------------------|---------------------------------------------|
| mean1    | df['age'].mean()   | Average of all values under the age column. |
| sum1     | df['age'].sum()    | Sum of all values under the age column.     |
| max1     | df['age'].max      | Maximum of all values under the age column. |
| min1     | df['age'].min()    | Minimum of all values under the age column. |
| count1   | df['age'].count()  | Count of all values under the age column.   |
| median1  | df['age'].median() | Median of all values under the age column.  |

|      |                              |                                                     |
|------|------------------------------|-----------------------------------------------------|
| std1 | <code>df['age'].std()</code> | Standard deviation all values under the age column. |
| var1 | <code>df['age'].var()</code> | Variance of all values under the age column.        |

We have learnt how to calculate simple stats using *Pandas and import any dataset for machine learning purpose.*

## 16.7 SUMMARY

- A database is an organized, logical collection of data. It is designed to facilitate the access by one or more applications programs for easy access and analysis and to minimize data redundancy.
- DBMS is a software system that enables you to store, modify, and extract information from a database. It has been designed systematically so that it can be used by multiple applications and users
- A relational database is a collection of related tables
- An entity is a person place or things or event.
- Tables in the database are entities.
- An attribute is a property of entity. Attributes are columns in the table.
- A relationship is the association between tables in the database.
- Each columns of the table correspond to an attribute of the relation and is named
- Fields : Columns in the table are called fields or attributes.
- Rows of the relation is referred as tuples to the relation . A tuple or row contains all the data of a single instance of the table such as a employee number 201.
- Records : Rows in a table often are called records . Rows are also known as tuples.
- The values for an attribute or a column are drawn from a set of values known as domain.
- Primary key : An attribute or a set of attributes which can uniquely identify each record (tuple) of a relation (table). All values in the primary key must be unique and not Null, and there can be only one primary key for a table.
- Foreign Key : An attribute which is a regular attribute in one table but a primary key in another table.
- Mysql.connector is an interface for connecting to a MySQL database server from python
- **connect()** is a method used for creating a connection to our database.
- Connect has four arguments:hostname,username,password,databasename .
- **fetchall()** – It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.

- **fetchone()** – It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **rowcount** – This is a read-only attribute and returns the number of rows that were affected by an execute() method.
- Commit, rollback and autocommit are database transactions commands.

## 16.8 SOLUTIONS TO CHECK YOUR PROGRESS

### Section 16.2 Solutions

1. Database is a collection of interrelated, organized form of persistent data. It has properties of integrity and redundancy to serve numerous applications. It is storehouse of all important information, not only for large business enterprises rather for anyone who is handling data even at micro level. Its features are
  - Real world entity.
  - Reduced data redundancy.
  - Data consistency.
  - Sharing of Data.
2. DDL and DML (Refer to section 16.2 SQL Commands)
3. Entity and attributes (Refer to section 16.2 Entity-Relationship Model)
4. Primary key: A primary key is a unique value that identifies a row in a table. These keys are also indexed in the database, making it faster for the database to search a record. All values in the primary key must be unique and NOT NULL, and there can be only one primary key for a table.

### Section 16.3 Solutions

1. Create a database called *Start\_db* in Python Anaconda.  

```
importmysql.connector
```

```
mydb = mysql.connector.connect(
 host="localhost",
 user="root",
 passwd="admin123",charset='utf8'
)
mycursor = mydb.cursor()
mycursor.execute("CREATE DATABASE Start_db")
```

**Next, we will try to connect to this new database.**

```
importmysql.connector
mydb = mysql.connector.connect(
 host="localhost",
 user="root",
 passwd="admin123",charset='utf8',database= Start_db)
```

2. Refer to question 1 second part (host,user,passwd,database)

### Section 16.4 Solutions

#### 1. Syntax of Insert :

INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN)

VALUES (value1, value2, value3,...valueN);

**Syntax of Update:**

UPDATE table\_name  
SET column1 = value1, column2 = value2...., columnN = valueN  
WHERE [condition];

**Syntax of Delete:**

DELETE FROM table\_name WHERE [condition];

### Section 16.5 Solutions

1. Rollback and commit(Refer to section 16.5 Manage Database transaction)

```
2. importmysql.connector
 mydb = mysql.connector.connect(
 host="localhost",
 user="myusername",
 passwd="mypassword"
)
 mycursor = mydb.cursor()
 mycursor.execute("CREATE DATABASE mydatabase")
```

#### MULTIPLE CHOICE QUESTIONS

1. Which SQL keyword is used to retrieve a maximum value?
  - a. MAX
  - b. MIN
  - c. LARGE
  - d. GREATER
2. Which SQL keyword is used to specify conditional search?
  - a. SEARCH
  - b. WHERE
  - c. FIND
  - d. FROM
3. A relation /table is a
  - a. Collection of fields
  - b. collection of records
  - c. collection of data
  - d. collections of tables
4. \_\_\_\_\_ is a collection of interrelated data and a set of program to access those data .
  - a. table
  - b. record
  - c. Database
  - d. Field
5. A row in a table is called as \_\_\_\_\_
  - a. table
  - b. tuple

- c. Database  
d. Field
6. Each table in a relational Database must have \_\_\_\_\_.  
a. primary key  
b. candidate key  
c. Foreign key  
d. composite key
7. Attribute combinations that can serve as a primary key in a table are  
a. primary key  
b. candidate key  
c. Alternate key  
d. composite key
8. Duplication of data is known as \_\_\_\_\_  
a. Data redundancy  
b. Data consistency  
c. data management  
d. data schema
9. \_\_\_\_\_ is the total number of rows/tuples in the table .  
a. degree  
b. cardinality  
c. records  
d. domain
10. \_\_\_\_\_ is the total number of columns/fields in the table  
a. degree  
b. cardinality  
c. records  
d. domain
11. A collection of fields that contains data about a single entity is called  
a. table  
b. record  
c. database  
d. Field
12. A set of related characters is called as  
a. table  
b. record  
c. database  
d. Field
13. \_\_\_\_\_ is the independence of application programs from the details of the data representation and data storage.  
a. Data redundancy  
b. Data consistency  
c. data independence  
d. data schema
14. The \_\_\_\_\_ is used for creating a connection to the database in python.  
a. connect()  
b. cursor()

- c. execute()
- d. close()

15. Which of the following module is provided by python to do several operations on the CSV files?

- a. py
- b. xls
- c. csv
- d. os

|     |     |   |     |    |     |   |     |    |     |   |
|-----|-----|---|-----|----|-----|---|-----|----|-----|---|
| Ans | 1-  | a | 2-  | b  | 3-  | b | 4-  | c  | 5-  | b |
|     | 6-  | a | 7-  | b  | 8-  | a | 9-  | b  | 10- | a |
|     | 11- | b | 12- | d. | 13- | c | 14- | a. | 15- | c |

