PRATHYUSHA

ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REGULATION 2021

I YEAR - I SEMESTER

GE3151 - PROBLEM SOLVING AND PYTHON PROGRAMMING

GE3151 PROBLEM SOLVING AND PYTHON PROGRAMMING

COURSE OBJECTIVES:

- To understand the basics of algorithmic problem solving.
- To learn to solve problems using Python conditionals and loops.
- To define Python functions and use function calls to solve problems.
- To use Python data structures – lists, tuples, dictionaries to represent complex data.
- To do input/output with files in Python.

UNIT I COMPUTATIONAL THINKING AND PROBLEM SOLVING

Fundamentals of Computing – Identification of Computational Problems -Algorithms, building blocks of algorithms (statements, state, control flow, functions), notation (pseudo code, flow chart, programming language), algorithmic problem solving, simple strategies for developing algorithms (iteration, recursion). Illustrative problems: find minimum in a list, insert a card in a list of sorted cards, guess an integer number in a range, Towers of Hanoi.

UNIT II DATA TYPES, EXPRESSIONS, STATEMENTS

Python interpreter and interactive mode, debugging; values and types: int, float, boolean, string, and list; variables, expressions, statements, tuple assignment, precedence of operators, comments; Illustrative programs: exchange the values of two variables, circulate the values of n variables, distance between two points.

UNIT III CONTROL FLOW, FUNCTIONS, STRINGS

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: state, while, for, break, continue, pass; Fruitful functions: return values,
parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Lists as arrays. Illustrative programs: square root, gcd, exponentiation, sum an array of numbers, linear search, binary search

UNIT IV LISTS, TUPLES, DICTIONARIES

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters; Tuples: tuple assignment, tuple as return value; Dictionaries: operations and methods; advanced list processing – list comprehension; Illustrative programs: simple sorting, histogram, Students marks statement, Retail bill preparation.

UNIT V FILES, MODULES, PACKAGES

Files and exception: text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages; Illustrative programs: word count, copy file, Voter's age validation, Marks range validation (0-100).

TOTAL : 45 PERIODS

COURSE OUTCOMES:

Upon completion of the course, students will be able to
CO1: Develop algorithmic solutions to simple computational problems.
CO2: Develop and execute simple Python programs.
CO3: Write simple Python programs using conditionals and looping for solving problems.
CO4: Decompose a Python program into functions.
CO5: Represent compound data using Python lists, tuples, dictionaries etc.
CO6: Read and write data from/to files in Python programs.

TEXT BOOKS: GE3151 Syllabus PROBLEM SOLVING AND PYTHON PROGRAMMING

1. Allen B. Downey, “Think Python: How to Think like a Computer Scientist”, 2nd Edition, O’Reilly Publishers, 2016.
2. Karl Beecher, “Computational Thinking: A Beginner's Guide to Problem Solving and programming”, 1st Edition, BCS Learning & Development Limited, 2017.

REFERENCES: GE3151 Syllabus PSPP

1. Paul Deitel and Harvey Deitel, “Python for Programmers”, Pearson Education, 1st Edition, 2021.
2. G Venkatesh and Madhavan Mukund, “Computational Thinking: A Primer for Programmers and Data Scientists”, 1st Edition, Notion Press, 2021.
3. John V Guttag, "Introduction to Computation and Programming Using Python: With Applications to Computational Modeling and Understanding Data‘‘, Third Edition, MIT Press 2021
4. Eric Matthes, “Python Crash Course, A Hands – on Project Based Introduction to Programming”, 2nd Edition, No Starch Press, 2019.
5. https://www.python.org/
6. Martin C. Brown, “Python: The Complete Reference”, 4th Edition, Mc-Graw Hill, 2018.

Unit I

ALGORITHMIC PROBLEM SOLVING

Algorithms, building blocks of algorithms (statements, state, control flow, functions), notation (pseudo code, flow chart, programming language), algorithmic problem solving, simple strategies for developing algorithms (iteration, recursion). Illustrative problems: find minimum in a list, insert a card in a list of sorted cards, guess an integer number in a range, Towers of Hanoi.

1.1 ALGORITHMS

What is algorithm?

An algorithm is a finite number of clearly described, unambiguous —"doable" steps that can be systematically followed to produce a desired result for given input in a finite amount of time . Algorithms are the initial stage of problem solving. Algorithms can be simply stated as a sequence of actions or computation methods to be done for solving a particular problem. An algorithm should eventually terminate and used to solve general problems and not specific problems.

Al-Khwarizmi

The word —algorithm is derived from the ninth-century Arab mathematician, Al-Khwarizmi. He worked on —written processes to achieve some goal. Computer

algorithms are central to computer science. They provide step-by-step methods of computation that computers can carry out. High speed computers can follow a given set of instructions for their computation which are programs. Programs are built from algorithms. However, the computation that a given computer performs is only as good as the underlying algorithm used.

Features

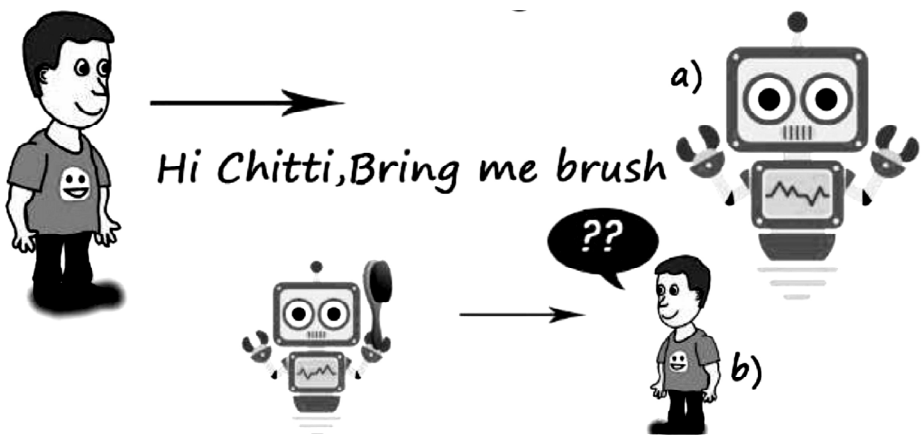
- An algorithm is a collection of **well-defined, unambiguous and effectively computable instructions**, if execute it will return the proper output.
- **Well-defined-** The instructions given in an algorithm should be simple and defined well.
- **Unambiguous-** The instructions should be clear,there should not be ambiguity .
- **Effectively computable-** The instructions should be written step by step ,which helps computer to understand the control flow.

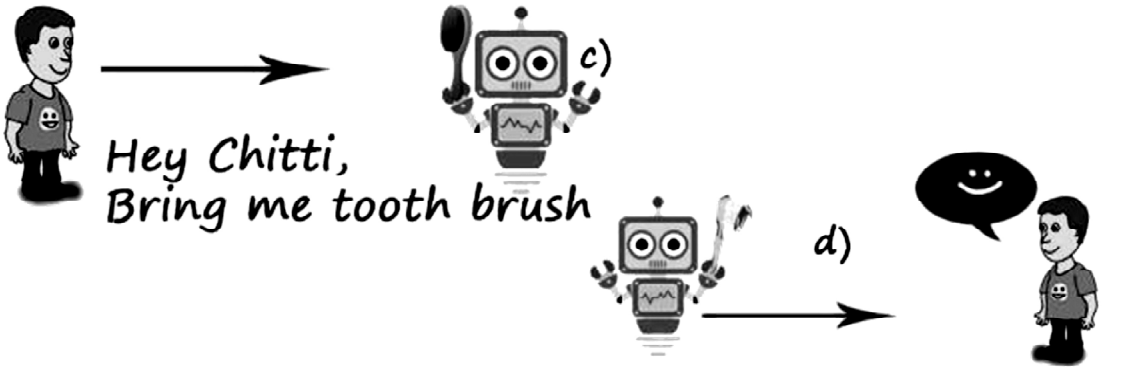
We often use algorithm in our day-to-day life, but when and how?

- Our cooking receipe
- Our daily routine as a student
- When we buy something
- When we go outing
- Our class routine

How it helps?

To understand the usage of algorithm, look over the following conversation.





Lets discuss about the conversation,tom wants brush his teeth, so he asks chitti to bring brush,what happens chitti returns cleaning brush.

Why this was happened ?

Because the statement given by tom was **not** well defined and it is **ambiguous** statement so chitti get confused and bring some brush to Tom.This is what happen if the user gives ambiguity statement to the computer.Therefore an algorithm should be **simple and well defined**.

How an algorithm should be?

It should be in simple English, what a programmer wants to say. It has a **start, a middle and an end**. Probably an algorithm should have,

Start

1. In the middle it should have set of tasks that computer wants to do and it should be in simple English and clear.
2. To avoid ambiguous should give no for each step.

Stop

Lets look over the simple example,

The following algorithm helps the computer to validate user's email address.

Start

Create a variable to get the user's email address clear the variable, incase its not empty.

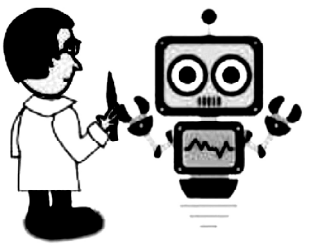
Ask the user for an email address.

Store the response in the variable.

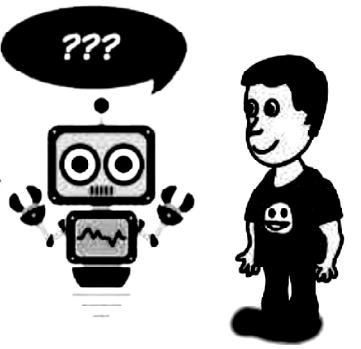
Check the stored response to see if it is a valid email address Not valid? Go back

Stop

Lets see how it works?



Dr·Paul feed the above algorithm to chitti and gonna test how it will work·



Chitti asked Tom to enter email address·Tom entered ,but chitti gets confused and donot know which process to be done next??

Why this Happened?

This was happened because the instructions given in an algorithm does not have numbering for each step.So Chitti gets confused which step have to do. To avoid this ambiguity ,we should number each step while writing an algorithm.

So let's rewrite the algorithm....

Step1: Start

Step2: Create a variable to get the user's email address

Step3: Clear the variable, incase its not empty.

Step4: Ask the user for an email address.

Step5: Store the response in the variable.

Step6: Check the stored response to see if it is a valid email address

Step7: Not valid? Go back

Step8: Stop

Suggested link to refer :

Link 1	https://www.youtube.com/watch?v=AVScy7YsKM0
Link 2	https://www.youtube.com/watch?v=CvSOaYi89B4
Link 3	https://www.youtube.com/watch?v=Da5TOXCwLSg

1.2 BUILDING BLOCKS OF ALGORITHMS

Building blocks are necessary to decide how we want to manipulate units of work. The basis of every algorithm is steps or blocks of operations.

The building blocks are:

- Statements
- State
- Control flow
- Functions

Statement is a single action in a computer.

In a computer statements might include some of the following actions

- input data-information given to the program
- process data-perform operation on a given input
- output data-processed result

State:

Transition from one process to another process under specified condition with in a time is called state**.**

Control flow:

The process of executing the individual statements in a given order is called control flow.The control can be executed in three ways

1. sequence
2. selection
3. iteration

It has been proven that any algorithm can be constructed from just three basic building blocks. These three building blocks are **Sequence, Selection, and Iteration (Repetition)**.

Sequence

This describes a sequence of actions that a program carries out one after another, unconditionally.Execute a list of statements **in order**. Consider an example,

Example 1.1 Algorithm for Baking Bread

Step1: Add flour.
Step 2: Add salt.
Step 3: Add yeast.
Step 4: Mix.
Step 5: Add water.
Step 6: Knead.
Step 7: Let rise.
Step 8: Bake.

Bread has been baked successfully.

Example 1.2 Algorithm for Addition of two numbers:

Step1: Start
Step 2: Get two numbers as input and store it in to a and b
Step 3: Set c = a+b
Step 4: Print c
Step 5: Stop.

Example 1.3 Algorithm to prepare Green Tea

Step 1: Fill the kettle with water
Step 2: Boil the water in kettle
Step 3: Put the green tea leaves in the pot
Step 4: Pour the boiling water in the pot
Step 5: Steep the tea leaves for 2 – 3 minutes
Step 6: Stop

Example 1.4 Algorithm to multiply two numbers

Step 1: Input first number as A
Step 2: Input second number as B
Step 3: Set Mul = A*B
Step 4: Print Mul
Step 5: End

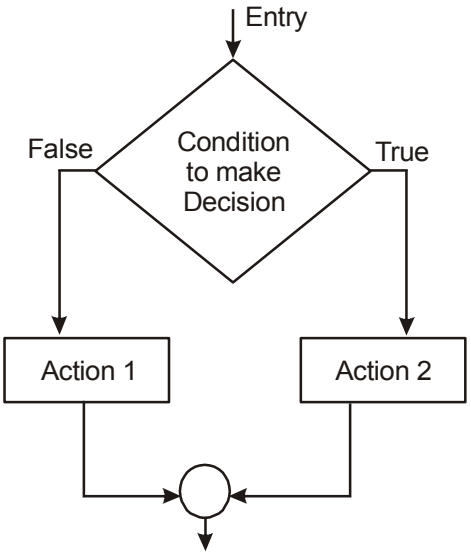
Example 1.5 Algorithm for interchanging of two numbers

Step 1 : Start
Start 2 : READ num1, num2
Start 3 : temp = num1
Start 4 : num1 = num2
Start 5 : num2 = temp
Start 6 : PRINT num1, num2
Start 7 : Stop

Selection

Control flow statements are able to make decisions based on the conditions.

Selection is the program construct that allows a program to choose between different actions. Choose at most one action from several alternative conditions.



Algorithm for path chooser

Step 1: Check for the destination located from current position.
Step 2:If it is located in right then choose right way
Step 3:If it is located in left then choose left way.
Step 4:Else comeback and search for new way.

Path has been chosen successfully.

The decision statements are:

- if
- if/else
- switch

The general form of the if construct can be:

IF condition then process

Although a program might seem like a linear path—one statement following another— conditional statements act like intersections, allowing you to change directions on the basis of a given condition.

A condition flow can also be stated in the following manner:

IFcondition
then process 1
ELSE
process 2

This form is known as the if – else construct. Here, if the condition is true then process 1 is

executed, else process 2 is executed.

Example 1.6 Algorithm for printing Grade

Marks	Grade
Above 75	O
60-75	A
50-60	B
40-50	C
Less than 40	D

Step 1 : Enter the marks obtained as M
Step 2 : IF M > 75
Print “O”
Step 3 : IF M>=60 and M<75
Print “A”
Step 4 : IF M>50 and M<60
Print “B”

```
Step 5 :   IF M>=40 and M<50
                Print "C"
           ELSE
                Print "D"
           [END of IF]
Step 6 : End
```

Example 1.7 Algorithm to find the equality of two numbers

```
Step 1 :   Start
Step 2 :   Input first number as A
Step 3 :   Input second number as B
Step 4 :   IF A==B
                Print "Equal"
           ELSE
                Print " Not Equal"
Strep 5 :  Stop
```

Example 1.8 Algorithm to find the largest of three numbers

```
Step 1:  Start
Step 2:  Declare variables a,b and c.
Step 3:  Read variables a,b and c.
Step 4:  If a>b
         If a>c
              Display a is the largest number.
         Else
              Display c is the largest number.
         Else
              If b>c
                   Display b is the largest number.
              Else
                   Display c is the greatest number.
Step 5: Stop
```

Example 1.9 Algorithm to find biggest among 2 nos:

Step1: Start
Step 2: Get two numbers as input and store it in to a and b
Step 3: If a is greater than b then
Step 4: Print a is big
Step 5: else
Step 6: Print b is big
Step 7: Stop

Repetition

While Statement:

The WHILE construct is used to specify a loop with a test at the top. The beginning and ending of the loop are indicated by two keywords WHILE and ENDWHILE.

The general form is:

WHILE condition
Sequence
END WHILE

FOR loop:

This loop is a specialized construct for iterating a specific number of times, often called a "counting" loop. Two keywords, FOR and ENDFOR are used.

The general form is:

FOR iteration bounds
Sequence
END FOR

Repetition (loop) may be defined as a smaller program the can be executed several times in a main program. Repeat a block of statements while a condition is true.

Example 1.10 Algorithm for Washing Dishes

Step1: Stack dishes by sink.
Step 2: Fill sink with hot soapy water.
Step 3: While moreDishes

Step 4: Get dish from counter,Wash dish

Step 5: Put dish in drain rack.

Step 6: End While

Step 7: Wipe off counter.

Step 8: Rinse out sink.

Example 1.11 Algorithm to calculate factorial no:

Step1: Start

Step 2: Read the number num.

Step 3: Initialize i is equal to 1 and fact is equal to 1

Step 4: Repeat step4 through 6 until I is equal to num

Step 5: fact = fact * i

Step 6: i = i+1

Step 7: Print fact

Step 8: Stop

Example 1.12 Algorithm to find the factorial of a number

Step 1. Read the value of n.

Step 2. i = 1 , F =1

Step 3. if (i > n) go to 7

Step 4. F = F * i

Step 5. i = i + 1

Step 6. go to 3

Step 7. Display the value of S

Step 8. Stop

Example 1.13 Algorithm to print numbers from 1 to 10

Step 1: Set i=1, n=10

Step 2: Repeat steps 3 and 4 while i<=n

Step 3: Print i

Step 4: Set i=i+1

[End of loop]

Step 5: End

Recursion

Recursion is a technique of solving a problem by breaking it down into smaller and smaller sub problems until you get to a small enough problem that it can be easily solved. Usually, recursion involves a function calling itself until a specified a specified condition is met.

Example 1.14 Algorithm for factorial using recursion

```
Step 1 : Start
Step 2 : Input number as n
Step 3 : Call factorial(n)
Step 4 : End
```

```
Factorial(n)
Step 1 : Set f=1
Step 2:  IF n==1 then return 1
         ELSE
       Set f=n*factorial(n-1)
Step 3 : print f
```

Suggested Link to refer:

https://www.youtube.com/watch?v=wOnjfIXCVpU

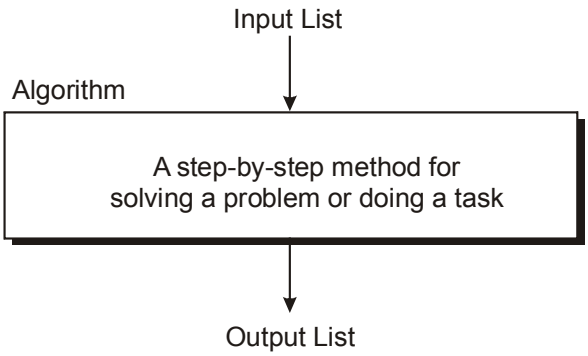
Functions:

Functions allow us to conceive of our program as a bunch of sub-steps. When any program seems too hard, just break the overall program into sub-steps! They allow us to reuse code instead of rewriting it. Every programming language lets you create blocks of code that, when called, perform tasks. All programming functions have input and output. The function contains instructions used to create the output from its input. The general form of a function definition has return type, parameter list, function name and function body.

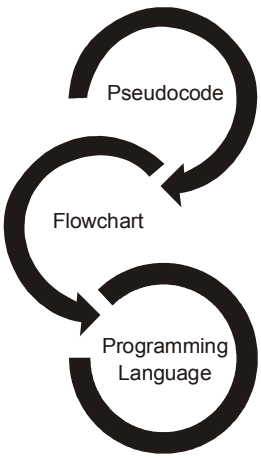
```
def function_name( parameter list ):
    body of the function
return [expression]
```

1.3 ALGORITHM NOTATIONS (EXPRESSING ALGORITHMS)

As we know that, an algorithm is a sequence of finite instructions, often used for **calculation and data processing**.



Algorithms can be expressed in many kinds of notation, including



1.3.1 Pseudocode

Pseudo code consists of short, readable and formally styled English languages used for explain an algorithm.

- It does not include details like variable declaration, subroutines.
- It is easier to understand for the programmer or non programmer to understand the general working of the program, because it is not based on any programming language.
- It gives us the sketch of the program before actual coding.
- It is not a machine readable
- Pseudo code can't be compiled and executed.

- There is no standard syntax for pseudo code.

1.3.1.1 Guidelines for writing pseudo code

- Write one statement per line
- Capitalize initial keyword
- Indent to hierarchy
- End multiline structure
- Keep statements language independent

1.3.1.2 Common keywords used in pseudo code

The following gives common keywords used in pseudo codes.

1. This keyword used to represent a comment.
2. **BEGIN,END:** Begin is the first statement and end is the last statement.
3. **INPUT, GET, READ**: The keyword is used to inputting data.
4. **COMPUTE, CALCULATE**: used for calculation of the result of the given expression.
5. **ADD, SUBTRACT, INITIALIZE** used for addition, subtraction and initialization**.**
6. **OUTPUT, PRINT, DISPLAY**: It is used to display the output of the program.
7. **IF, ELSE, ENDIF**: used to make decision.
8. **WHILE, ENDWHILE**: used for iterative statements.
9. **FOR, ENDFOR**: Another iterative incremented/decremented tested automatically.

1.3.1.3 How to write a pseudo code

Start by writing down the purpose of the process.

Write initial steps of pseudo code that set the stage for functions.

Write functional pseudo code.

Add comments, if necessary.

Read over the finished project for logic errors .

Review the pseudo code.

Advantages:

- Pseudo is independent of any language; it can be used by most programmers.
- It is easy to translate pseudo code into a programming language.
- It can be easily modified as compared to flowchart.
- Converting a pseudo code to programming language is very easy as compared with converting a flowchart to programming language.

Disadvantages:

- It does not provide visual representation of the program's logic.
- There are no accepted standards for writing pseudo codes.
- It cannot be compiled nor executed.
- For a beginner, It is more difficult to follow the logic or write pseudo code as compared to flowchart.

Syntax for if else:	**Example: Greates of two numbers**

IF (condition)THEN
statement
....
ELSE
statement
....
ENDIF
	BEGIN
	READ a,b
	IF (a>b) THEN
	DISPLAY a is greater
	ELSE
	DISPLAY b is greater
	END IF
	END

Syntax for For:	**Example: Print n natural numbers**

FOR (start-value to end-value)DO
statement
....
	BEGIN
	GET n
	INITIALIZE i=1
	FOR (i<=n) DO
	PRINT i
	i=i+1
	ENDFOR
	END

Syntax for While:	**Example: Print n natural numbers**

WHILE (condition) DO
statement
...
ENDWHILE
	BEGIN
	GET n
	INITIALIZE i=1
	WHILE (i<=n) DO
	PRINT i
	i=i+1
	ENDWHILE
	END

pseudo code is made up of the following logic structure,

- Sequential logic
- Selection logic
- Iteration logic

Sequence Logic

- It is used to perform instructions in a sequence,that is **one after another**
- Thus,for sequence logic ,pseudocode instructions are written in an order in which they are to be performed.
- The logic flow of pseudocode is from **top to bottom.**

Example 1.15 Pseudo code to add two numbers:

```
START
READ a,b
COMPUTE c by adding a &b
PRINT c
STOP
```

Selection Logic

- It is used for making decisions and for selecting the proper path out of two or more alternative paths in program logic.
- It is also known as decision logic.
- Selection logic is depicted as either an **IF..THEN or an IF…THEN..ELSE Structure**.

Example 1.16 Pseudocode to Find Biggest of two numbers:

```
START
READ a and b
IF a>b THEN
PRINT “A is big”
ELSE
PRINT “B is big”
ENDIF
STOP
```

Repetition Logic

- It is used to produce loops when one or more instructions may be executed several times depending on some conditions.
- It uses structures called **DO_WHILE,FOR and REPEAT__UNTIL**

Example 1.17 Pseudocode to print first 10 natural numbers

```
START
INITIALIZE a?0
WHILE a<10
PRINT a
ENDWHILE
STOP
```

Suggested Link to refer :

https://www.youtube.com/watch?v=4jLO0vXPktU

1.3.2 Flowchart

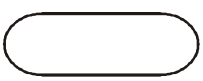
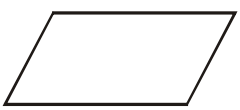
A flowchart is a visual representation of the sequence of steps and decision needed to perform a process.

Flow chart is defined as graphical representation of the logic for problem solving.

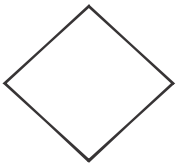
The purpose of flowchart is making the logic of the program clear in a visual representation.

Flowchart symbols

Here are some of the common flowchart symbols.

Name	Symbol	Use in flowchart
Oval		Denotes the beginning or end of a program.
Flow line		Denotes the direction of logic flow in a program.
Parallelogram		Denotes either an input operation (e.g., INPUT) or an output operation (e.g. PRINT)

Rectangle  Denotes a proces sto be carried out (e.g., an addition)

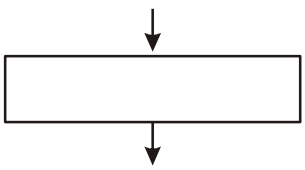
Diamond  Denotes a decision (or branch) to be made. The program should continue along one of two routes (e.g, IF.THEN/ELSE)

Suggested Links : (Flowchart Symbols)

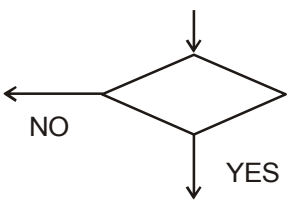
https://www.youtube.com/watch?v=kxZJv56BxU8

Rules for drawing a flowchart

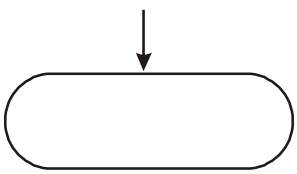
1. The flowchart should be clear, neat and easy to follow.
2. The flowchart must have a logical start and finish.
3. Only one flow line should come out from a process symbol.



4. Only one flow line should enter a decision symbol. However, two or three flow lines may leave the decision symbol.



5. Only one flow line is used with a terminal symbol.



6. Within standard symbols, write briefly and precisely.
7. Intersection of flow lines should be avoided.

Advantages of flowchart:

1. **Communication: -** Flowcharts are better way of communicating the logic of a system to all concerned.
2. **Effective analysis: -** With the help of flowchart, problem can be analyzed in more effective way.
3. **Proper documentation: -** Program flowcharts serve as a good program documentation, which is needed for various purposes.
4. **Efficient Coding: -** The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
5. **Proper Debugging: -** The flowchart helps in debugging process.
6. **Efficient Program Maintenance: -** The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

Disadvantages of flow chart:

1. **Complex logic: -** Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. **Alterations and Modifications: -** If alterations are required the flowchart may require re-drawing completely.
3. **Reproduction: -** As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.
4. **Cost: F**or large application the time and cost of flowchart drawing becomes costly.

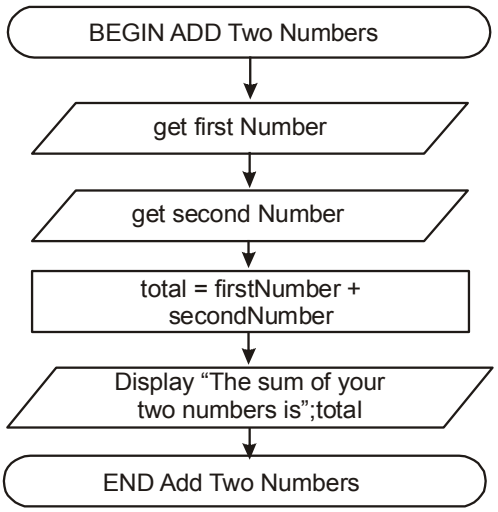
Flowchart is made up of the following logic structure,

- **Sequential logic**
- **Selection logic**
- **Iteration logic**

Sequence Logic

In a computer program or an algorithm, sequence involves simple steps which are to be executed one after the other. The steps are executed in the same order in which they are written.

Below is an example set of instructions to add two numbers and display the answer.



Selection Logic

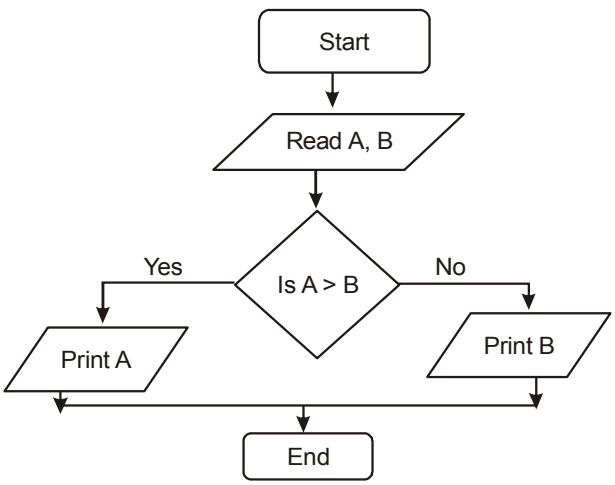
Selection is used in a computer program or algorithm to determine which particular step or set of steps is to be executed. This is also referred to as a 'decision'.

A selection statement can be used to choose a specific path dependent on a condition.

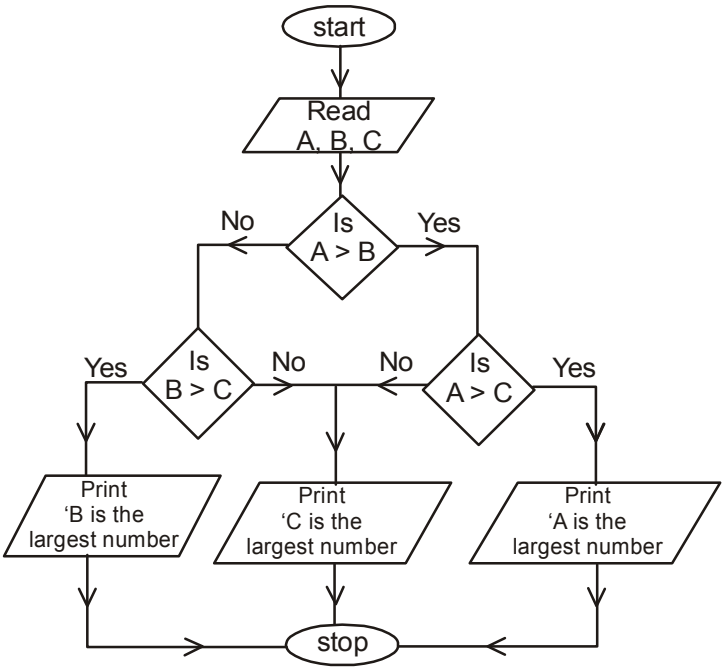
There are two types of selection:

- **binary selection (two possible pathways)**
- **multi-way selection (many possible pathways)**

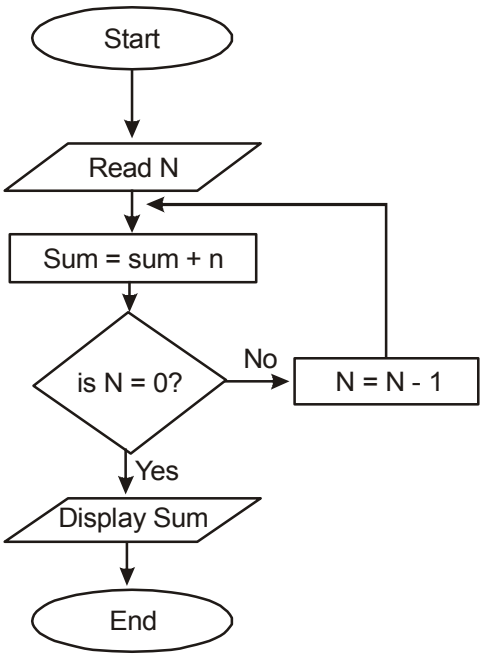
Following is the example flowchart to find biggest among two numbers



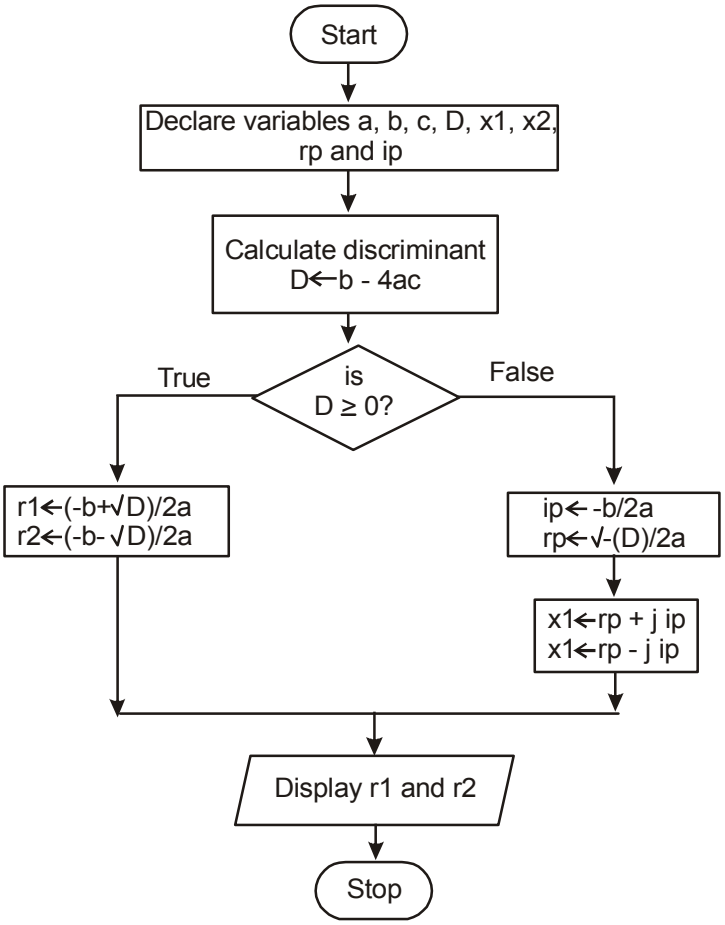
Example 1.18 Flow chart for biggest of three numbers



Example 1.19 Flow chart for sum of N Natural numbers



Example 1.20 Roots of a quadratic equation ax2+bx+c=0



Repetition Logic

Repetition allows for a portion of an algorithm or computer program to be executed any number of times dependent on some condition being met.

An occurrence of repetition is usually known as a **loop**.

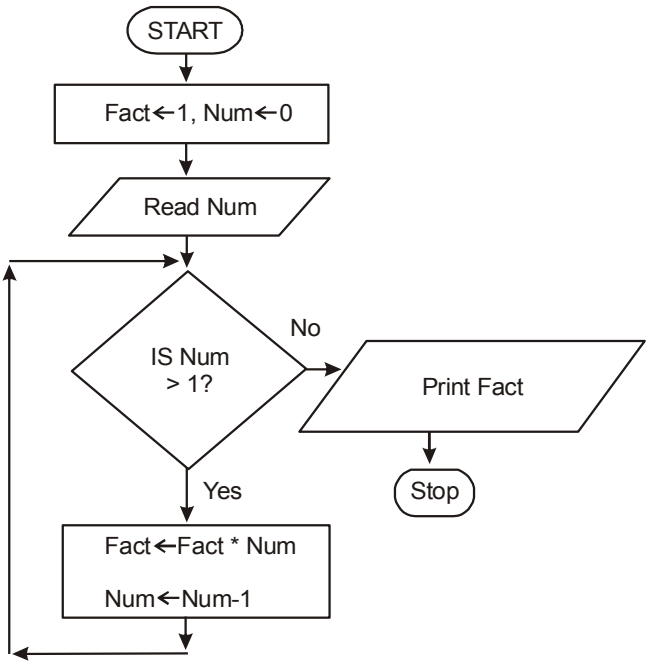
The termination condition can be checked or tested at the **beginning** or **end** of the loop, and is known as a **pre-test** or **post-test**, respectively.

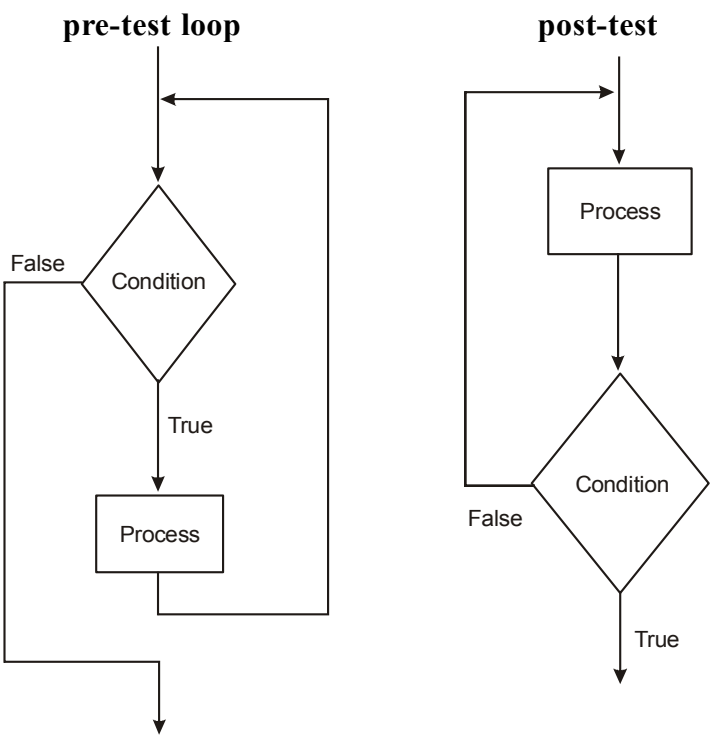
Iteration & Recursion

Iteration and recursion are key Computer Science techniques used in creating algorithms and developing software.

In simple terms, an iterative function is one that loops to repeat some part of the code, and a recursive function is one that calls itself again to repeat the code.

Example 1.21 Flowchart to find factorial of given no





1.3.3 Representation of Algorithm using Programming Language

- Algorithms describe the solution to a problem in terms of the data needed to represent the problem instance and the set of steps necessary to produce the intended result.
- Programming languages must provide a notational way to represent both the process and the data.
- To this end, languages provide control constructs and data types.

Programming is the process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer.

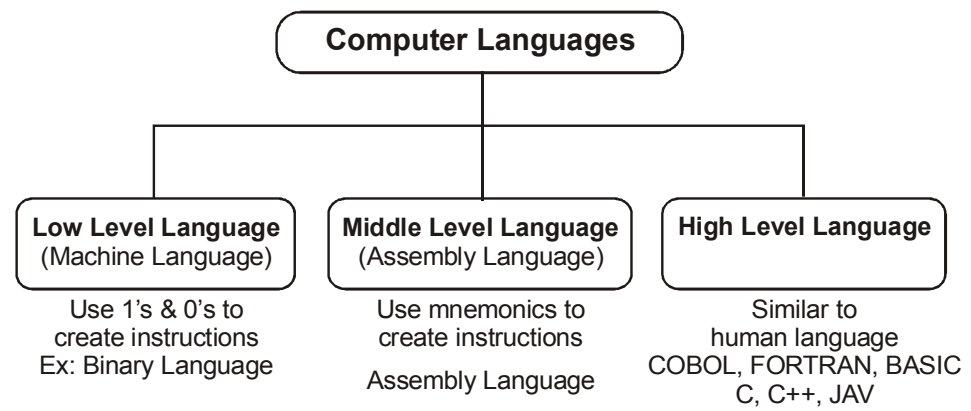
Although many programming languages and many different types of computers exist, the important first step is the need to have the solution.

Without an algorithm there can be no program.

- Control constructs allow algorithmic steps to be represented in a convenient yet unambiguous way.
- At a minimum, algorithms require constructs that perform sequential processing, selection for decision-making, and iteration for repetitive control.
- As long as the language provides these basic statements, it can be used for algorithm representation.

Simply we can say programming as like below

Programming is implementing the already solved problem (algorithm) in a specific computer language where syntax and other relevant parameters are different, based on different programming languages.



Low level Language(Machine level Language)

A *low-level language* is a programming *language* that deals with a computer's hardware components and constraints.

In simple we can say that ,low level language can only be understand by computer processor and components.

Middle level Language(Intermediate Language)

Medium-level language serves as the bridge between the raw hardware and programming layer of a computer system.

Medium-level language is also known as intermediate programming language and pseudo language.

C intermediate language and Java byte code are some examples of medium-level language.

High level Language (Human understandable Language)

A high-level language is any programming language that enables development of a program in a much more user-friendly programming context.

High-level languages are designed to be used by the human operator or the programmer.

They are referred to as "closer to humans." In other words, their programming style and context is easier to learn and implement than low-level languages

BASIC, C/C++ and Java are popular examples of high-level languages.

1.4 ALGORITHMIC PROBLEM SOLVING

"Algorithmic-problem solving"; this means solving problems that require the formulation of an algorithm for their solution.

The formulation of algorithms has always been an important element of problem-solving.

Why we need to go for algorithm to solve problem?

- A computer is a tool that can be used to implement a plan for solving a problem.
- A computer program is a set of instructions for a computer. These instructions describe the steps that the computer must follow to implement a plan.

- An algorithm is a plan for solving a problem.
- A person must design an algorithm.
- A person must translate an algorithm into a computer program.

An algorithmic Development Process

Every problem solution starts with a plan. That plan is called an algorithm.

An algorithm is a plan for solving a problem.

There are many ways to write an algorithm.

- Some are very informal.
- some are quite formal.
- mathematical in nature.
- some are quite graphical.

Once we have an algorithm, we can translate it into a computer program in some programming language. Our algorithm development process consists of five major steps.

Step 1: Obtain a description of the problem.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

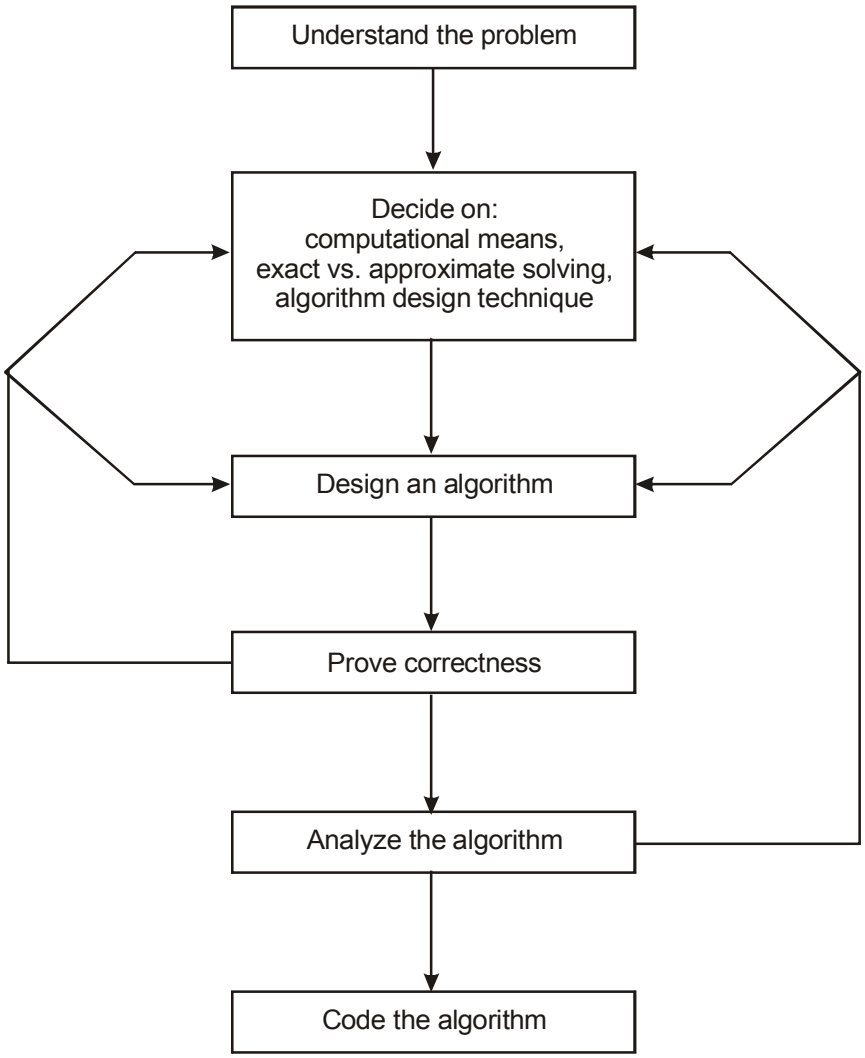
Step 5: Review the algorithm.

1. Understanding the Problem

- It is the process of finding the input of the problem that the algorithm solves.
- It is very important to specify exactly the set of inputs the algorithm needs to handle.
- A correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

Ascertaining the Capabilities of the Computational Device

- If the instructions are executed one after another, it is called sequential algorithm.
- If the instructions are executed concurrently, it is called parallel algorithm.



Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly or solving it approximately.
- Based on this, the algorithms are classified as exact *algorithm* and *approximation algorithm.*

2. Deciding a data structure:

- Data structure plays a vital role in designing and analysis the algorithms.
- Some of the algorithm design techniques also depend on the structuring data specifying a problem's instance
- Algorithm+ Data structure=programs.

Algorithm Design Techniques

- An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of proble ms from different areas of computing.
- Learning these techniques is of utmost importance for the following reasons.
- First, they provide guidance for designing algorithms for new problems,
- Second, algorithms are the cornerstone of computer science

Methods of Specifying an Algorithm

- ***Pseudocode*** is a mixture of a natural language and programming language-like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.
- In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.
- **Programming language** can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language. We can look at such a program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm's implementation.

3. Proving an Algorithm's Correctness

- Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

- A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

4. Analysing an Algorithm

1. ***Efficiency***.

Time efficiency, indicating how fast the algorithm runs, *Space efficiency*, indicating how much extra memory it uses.

2. ***Simplicity***.

- An algorithm should be precisely defined and investigated with mathematical expressions.
- Simpler algorithms are easier to understand and easier to program.
- Simple algorithms usually contain fewer bugs.

5. Coding an Algorithm

- Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity.
- A working program provides an additional opportunity in allowing an empirical analysis of the underlying algorithm. Such an analysis is based on timing the program on several inputs and then analysing the results obtained.

1.5 SIMPLE STRATEGIES FOR DEVELOPING ALGORITHMS

1. Iterations
2. Recursions

Iterations:

A sequence of statements is executed until a specified condition is true is called iterations.

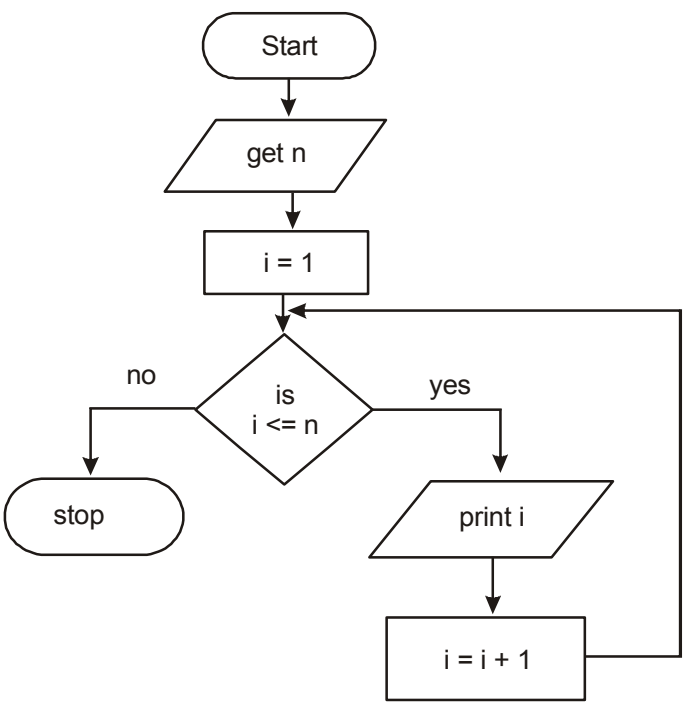
1. for loop
2. While loop

Syntax for For:	Example: Print n natural numbers

FOR (start-value to end-value)DO
statement
....
ENDFOR
	BEGIN
	GET n
	INITIALIZE i=1
	FOR (i<=n) DO
	PRINT i
	i=i+1
	ENDFOR
	END

Syntax for While:	**Example: Print n natural numbers**

WHILE (condition) DO
statement
...
ENDWHILE
	BEGIN
	GET n
	INITIALIZE i=1
	WHILE (i<=n) DO
	PRINT i
	i=i+1
	ENDWHILE
	END



Recursions:

- A function that calls itself is known as recursion.
- Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

Example 1.22 Algorithm for factorial of n numbers using recursion:

Main function:

Step1: Start

Step2: Get n

Step3: call factorial(n)

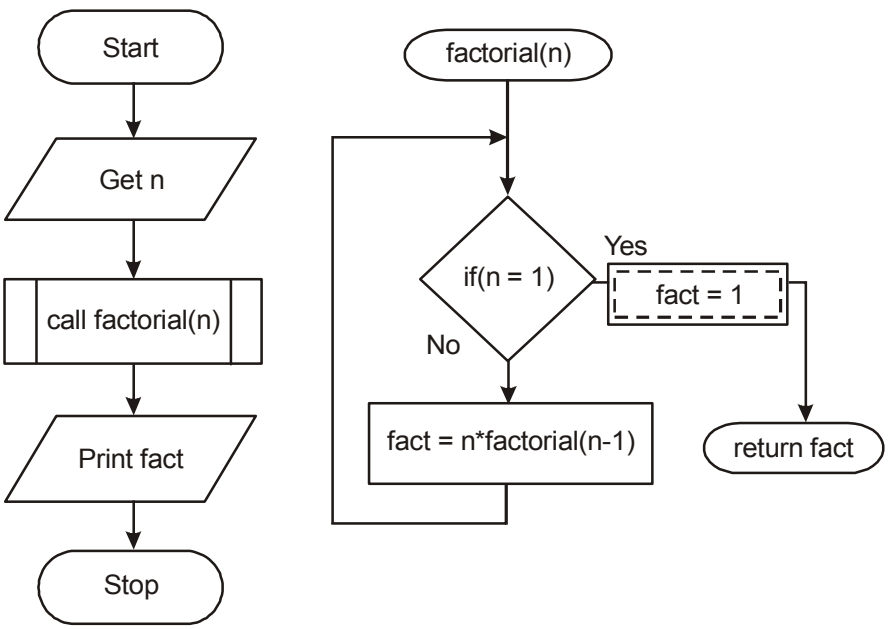
Step4: print fact

Step5: Stop

Sub function factorial(n):

Step1: if(n==1) then fact=1 return fact

Step2: else fact=n*factorial(n-1) and return fact



Example 1.23 Pseudo code for factorial using recursion:

```
Main function:
BEGIN
GET n
CALL factorial(n)
PRINT fact
BIN
Sub function factorial(n):
IF(n==1) THEN
fact=1
RETURN fact
ELSE
RETURN fact=n*factorial(n-1)
```

1.6 EXAMPLE ALGORITHMS (ILLUSTRATIVE PROBLEMS)

1.6.1 Find minimum in a list

Problem: Given a list of positive numbers, return the smallest number on the list.

Inputs: A list L of positive numbers. This list must contain at least one number. (Asking for the smallest number in a list of no numbers is not a meaningful question.)

Outputs: A number n, which will be the smallest number of the list.

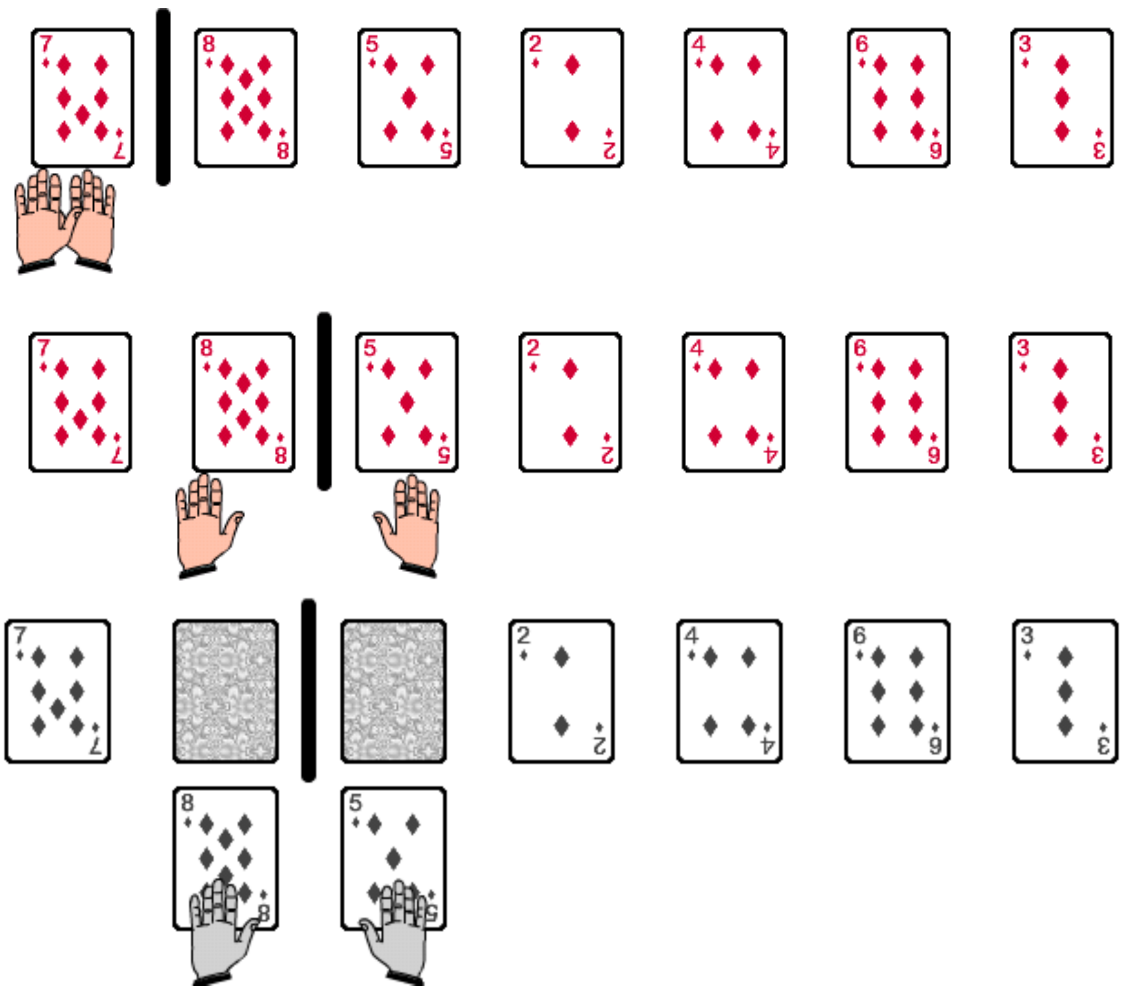
Algorithm:

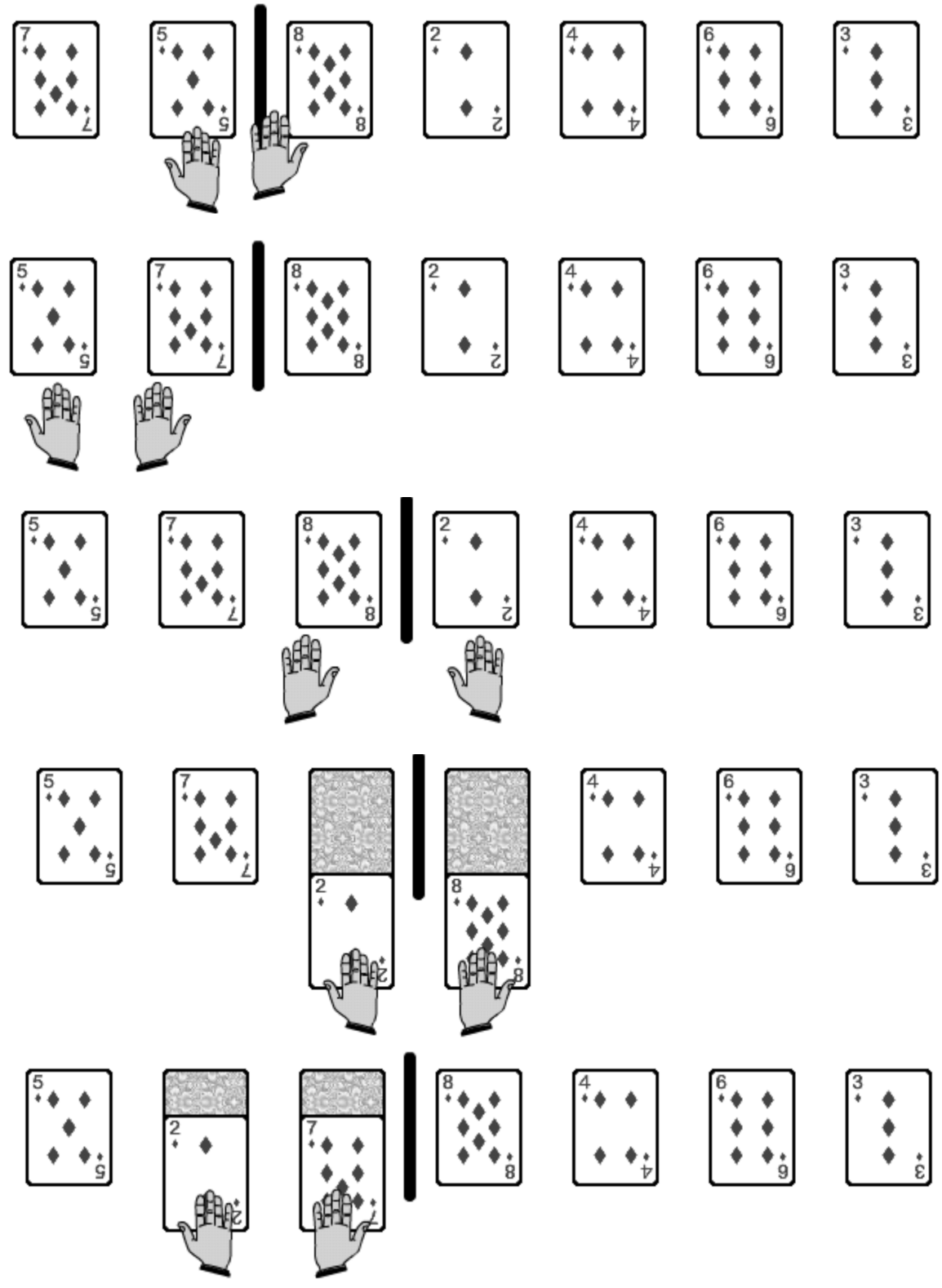
1. Start
2. Get positive numbers from user and add it in to the List L
3. Set min to L[0].
4. For each number x in the list L, compare it to min. If x is smaller, set min to x.
5. min is now set to the minimum number in the list.
6. Stop

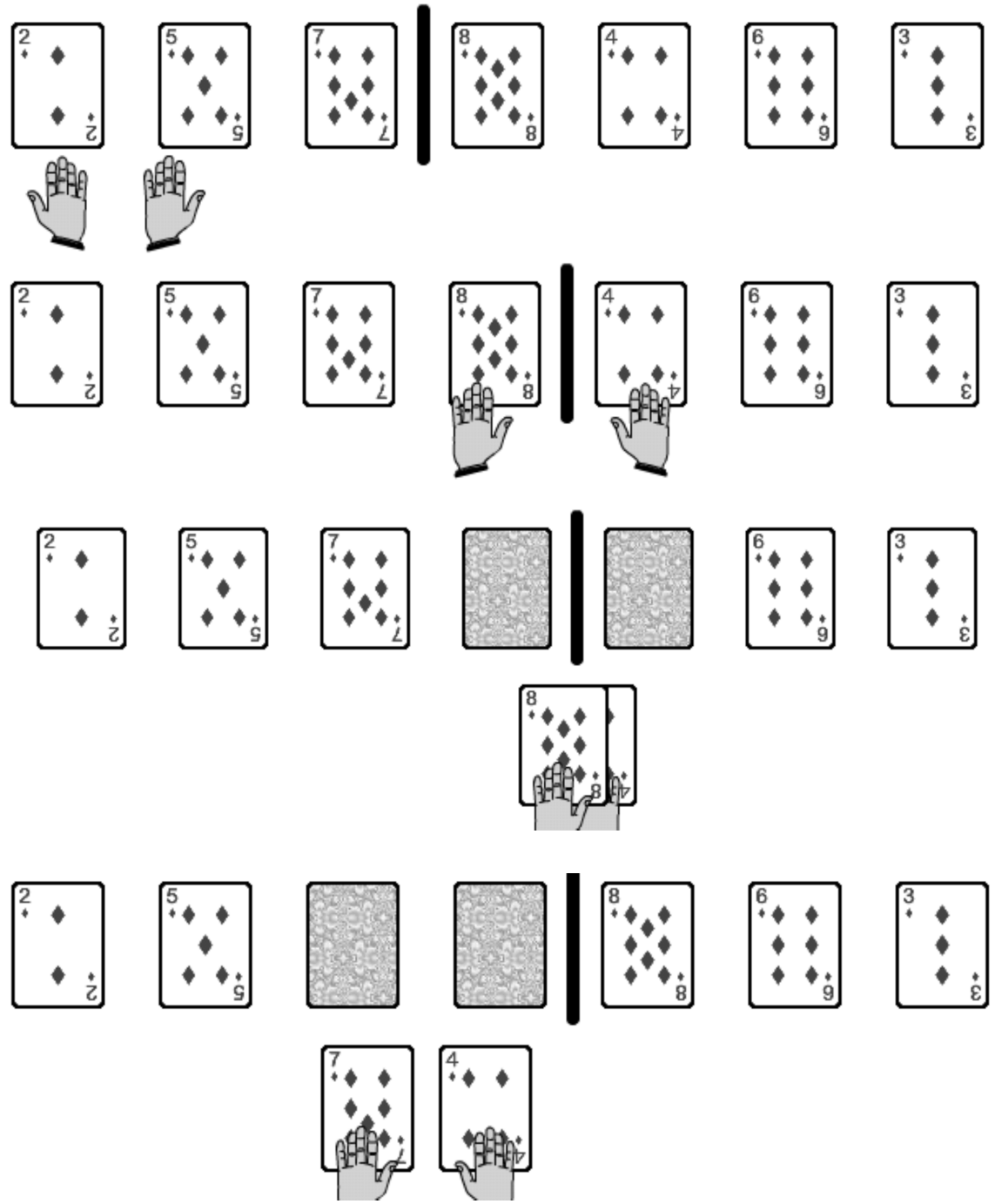
1.6.2 Insert a card in a list of sorted cards

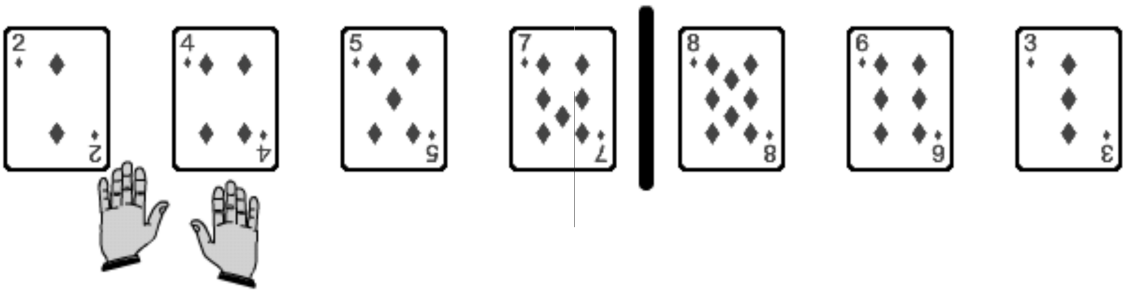
1. Get a hand of unsorted cards
2. Set a marker for the sorted section after the first card of the hand
3. Repeat steps 4 through 6 until the unsorted section is empty
4. Select the first unsorted card
5. Swap this card to the left until it arrives at the correct sorted position.
6. Advance the marker to the right one card
7. Stop

1.6.2 Insert a card in a list of sorted cards

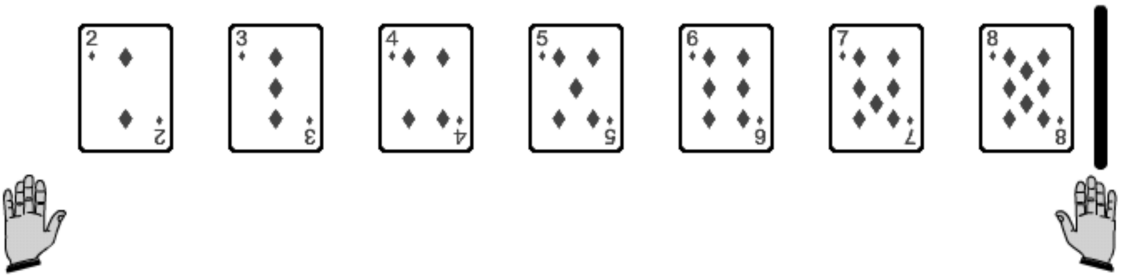








The sorting recursively been done until the cards are fully sorted. The final sorted cards are



Algorithm:

1. Start
2. Ask for value to insert
3. Find the correct position to insert, If position cannot be found ,then insert at the end.
4. Move each element from the backup to one position, until you get position to insert.
5. Insert a value into the required position
6. Increase array counter
7. Stop

1.6.3 Guess an integer number in a range

Let's play a little game to give you an idea of how different algorithms for the same problem can have wildly different efficiencies.

The computer is going to randomly select an integer from 1 to 20. We have to guess the number by making guesses until you find the number that the computer chose.

Algorithm:

1. Start
2. Generate a random number from 1 to 20 and store it into the variable number.
3. Ask the user to guess number between 1 and 20 and store it into guess for six chances.
4. Check if guess is equal to number
5. If guess is greater than number print ,the number you guessed is greater
6. If guess is lesser then print,the number you guessed is lesser.
7. If guess is equal to number then,print you guessed is right.
8. If guess is not equal and chance is greater than six print you fail and stop the execution
9. Stop

The following table shows the sample scenario of the range 1-10 that asks a series of questions, reducing the problem size by about half each time.

Number	First guess	Second guess	Third guess
1	Is it 6? Too high	Is it 3? Too high	Is it 1? You win
2	Is it 5? Too high	Is it 2? You win	
3	Is it 5? Too high	Is it 2? Too high	Is it 3? You win
4	Is it 5? Too high	Is it 2? Too low	Is it 3? Too low, so it must be 4.
5	Is it 5? You Win		
6	Is it 5? Too	Is it 8? Too High	Is it 6? You Win
7	Is it 5? Too low	Is it 8? Too low	Is it 6? Too low, so it must be 7
8	Is it 5? Too low	Is it 8? You win	

9	Is it 5? Too low	Is it 8? Too low	Is it 9? You win
10	Is it 5? Too low	Is it 8? Too low	Is it 9? Too low, so it must be 10

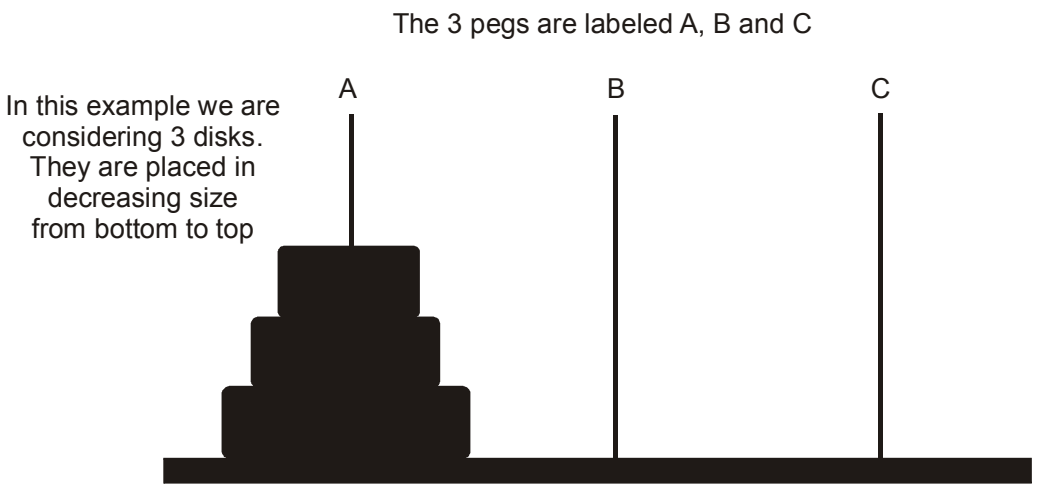
1.6.4 Tower of Hanoi

Tower of Hanoi is a very famous game.

In this game there are 3 rods(pegs) and N number of disks placed one over another in decreasing size.

The objective of the game is to move the disks one by one from the first rod to last rod.

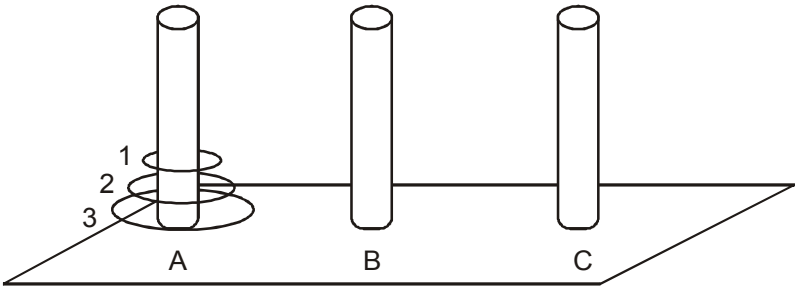
And there is two one condition; only one disk can be moved at a time and we cannot place a bigger disk on top of a smaller disk.



Our objective is to move the disks from peg A to peg C in such a way that they are in the same order: RED then GREEN then BLUE from top to bottom as they are in peg A.

Problem Analysis:

We will first attempt to solve this problem for three disks to gain some insight into the problem, and then develop a general solution for any number of disks. Thus, we will solve the simple problem of moving three disks from peg A to peg C as shown in Figure



Towers of Hanoi Problem for Three Disks

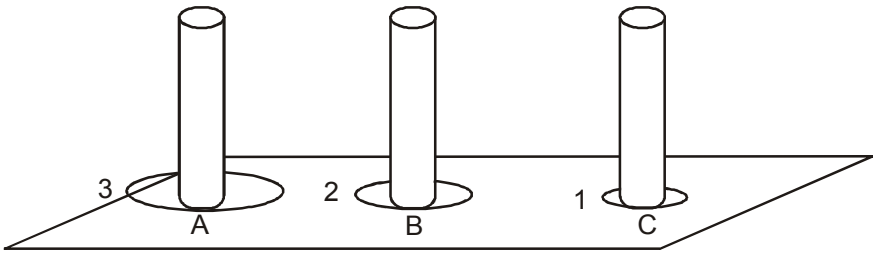
First Move: Move 1

Remove the smallest disk and place it on either peg B or peg C.

Move 2:

If we place the smallest disk on peg B, place the next smallest disk on peg C.

If we place the smallest disk on peg C, place the next smallest disk on peg B.

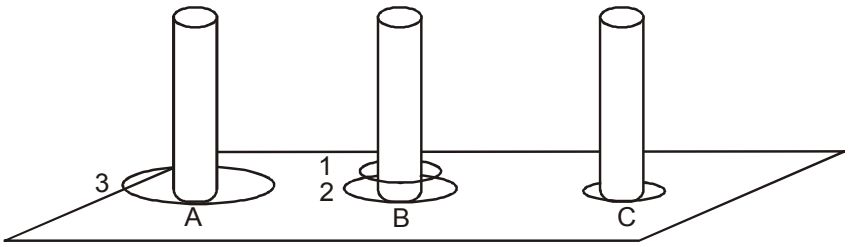


After Move 1 and Move 2

Move 3:

We can place the disk currently on peg B back on peg A, but that will be undoing what we just did in the last step. So in order to make progress, we move the smallest disk on peg C somewhere.

We can move it on either peg A or peg B, since in each case it would be placed on a larger disk. Let's assume that we move the smallest disk currently on peg C on top of the second smallest disk on peg B. This move results are shown in the figure.

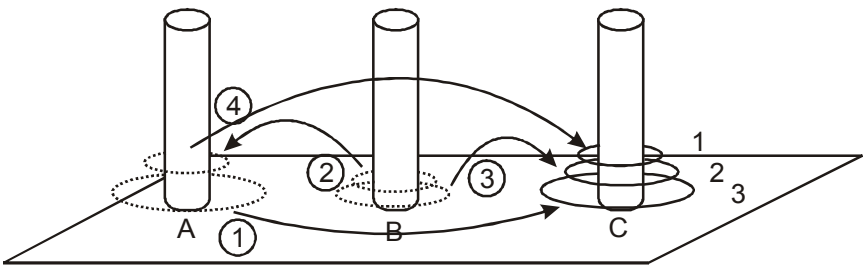


After Move 3

Move 4:

We can move the largest disk currently on peg A to peg C.

Then we can move the smallest disk from peg B to peg A, then move the second smallest disk from peg B to peg C, and finally move the smallest disk from peg A to peg C, thereby solving the problem.



After Move 4

Recursive solution for tower of Hanoi problem:

The fundamental step of a recursive solution for the Towers of Hanoi problem is given below:

Step 1: View the stack as two stacks, one on top of the other. Call the top stack Stack1, and the bottom stack Stack2.

Step 2: Recursively move Stack1 from peg A to peg B.

Step 3: Recursively move (the exposed) Stack2 from peg A to peg C.

Step 4: Recursively move Stack1 from peg B to peg C.

Repeat moving a stack of disks from one peg to another. It does not matter that the stacks being moved are different, or are being moved between different pegs. Each of the sub problems can be recursively solved in a similar way, and thus we can assume that they can be solved without explicitly specifying how. How to break down a stack of disks into two separate (sub) stacks?

1. Start
2. Move disk1 from pegA to pegC
3. Move disk2 from pegA to pegB
4. Move disk3 from pegC to pegB

5. Move disk1 from pegA to pegC
6. Move disk1 from pegB to pegA
7. Move disk2 from pegB to pegC
8. Move disk1 from pegA to peg C

Stop

Suggested Link to Refer :

Reference Link 1: https://www.youtube.com/watch?v=5_6nsViVM00

Link 2 : https://www.youtube.com/watch?v=fffbT41IuB4

Assignments :

Try the following with Algorithm, flowchart and pseudo code.

1. Find the area of triangle
2. Area and circumference of a circle
3. Calculating simple interest
4. Calculating engineering cutoff
5. Greatest of two numbers
6. Greatest of three numbers
7. Check leap year or not
8. Check the number is odd or even
9. Check the number is positive or negative
10. Print all the prime numbers upto N
11. Print square and cube of a number
12. Print sum of N numbers
13. Find the factorial of a number
14. Convert Temperature from Fahrenheit (°F) to Celsius (°C)

PART A QUESTION AND ANSWERS

1. Define Algorithm.

The word —algorithm is derived from the ninth-century Arab mathematician, Al-Khwarizmi. An algorithm is a finite number of clearly described, unambiguous —doable steps.

It has to be followed to get the desired output.

It can be simply stated as a sequence of actions to be carried out to solve a particular problem.

2. Write an algorithm for addition of two numbers.

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum. sum !num1+num2

Step 5: Display sum

Step 6: Stop

3. What are termed as the basic building blocks of an algorithm?

The basis of every algorithm is steps or blocks of operations. The building blocks are:

- Sequencing – Instructions State
- Control Flow and Functions

4. Give the syntax of conditional statements.

The conditional statements act like intersections, allowing us to change directions on the basis of a given condition.

The decision statements are:

- if
- if/else
- switch

If-else statement :

If condition

then process 1

else

process 2

5. What is mean by repetition or loop?

Looping / Repetition/ Iteration means executing the same set of statements again and again until a condition gets satisfied.

The looping statements in python are for, while and do while.

6. Give the syntax of for and while statements.

WHILE condition Sequence

END WHILE

FOR iteration bounds Sequence

END FOR

7. Define function.

Function allows us to frame the programs into sub process. It has a sub sets in the program.

def function_name(parameter list):

body of the function return [expression]

8. Define recursion.

Recursion involves a function calling itself until a specified a specified condition is met.

9. List the keywords for pseudo code. Looping and selection

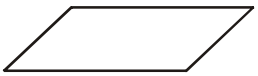
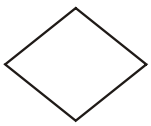
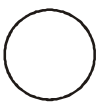
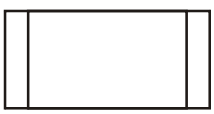
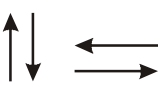
Keywords :

Do While...EndDo;
Do Until...Enddo;
Case...EndCase;
If...Endif;
Generate, Compute, Process, etc.
Displaying : print, display, input, output, edit, test

10. What is mean by pseudo code?

It is an informal high level way of algorithm. It focus on the concept of solving a problem. It contains English phrases.

11. What are the symbols used in flowchart?

Symbol	Name	Function
	Process	Indicates any type of internal operation inside the Processor or Memory
	Input/Output	Used for any Input / Output (I/O) operation. Indicates that the computer is to obtain data or output results
	Decision	Used to ask a question that can be answered in a binary format (Yes/ No, True/False)
	Connector	Allows the flowchart to be drawn without intersecting lines or without a reverse flow
	Predefined Process	Used to invoke a subroutine or an Interrupt program
	Terminal	Indicates the starting or ending of the program, process, or interrupt program
	Flow Lines	Shows direction of flow

12. Define flowchart.

A flowchart is a graphical or symbolic representation of a process. It is basically used to design and document virtually complex processes

13. What are the advantages of flowchart? Advantages of Using Flowcharts:

- Communication: Flowcharts are better way of communicating the logic
- Effective analysis: problem can be analyzed in more effective way
- Proper documentation: Program flowcharts serve as a good program documentation
- Efficient Coding: The flowcharts act as a guide or blueprint
- Proper Debugging: The flowchart helps in debugging process.
- Efficient Program Maintenance: The maintenance of operating program becomes easy

14. What are the disadvantages of flowcharts? Disadvantages of using Flowcharts:

- Complex logic
- Alterations and Modifications
- Reproduction

15. Mention the difference between algorithm and pseudo code.

Algorithm	Pseudo code
An algorithm gives a solution to a particular problem as a well defined set of steps.	Pseudo code is one of the methods that could be used to represent an algorithm.
Algorithms can be written in natural language	Pseudo code is written in a format that is closely related to programming language

PART – B QUESTIONS

1. Explain in detail about the Algorithmic problem solving
2. Detail on the building blocks of algorithm
3. Analyze the following problems:
   a. Find minimum in a list
   b. Insert a card in a list of sorted cards
   c. Guess a number in a range
4. Discuss about Tower of Hanoi Problem
5. Write an algorithm, pseudo code and flowchart for finding the minimum number in a list.
6. Write an algorithm, pseudo code and flow chart to find the factorial of a given number.
7. Write an algorithm, pseudo code and flow chart for Fibonacci series.
8. Write an algorithm, pseudo code and flow chart to find whether a number is prime or not.

Unit II

DATA, EXPRESSIONS, STATEMENTS

Python interpreter and interactive mode; values and types: int, float, booleans, strings, and lists; variables, expressions, statements, tuple assignment, precedence of operators, comments; modules and functions, function definition and use, flow of execution, parameters and arguments; Illustrative programs: exchange the values of two variables, circulate the values of n variables, test for leap year.

2.1 INTRODUCTION

What is a program?

A computer program is a collection of instructions that perform a speciûc task when executed by a computer." - Process of writing a program is called programming. Computer programs are written to solve speciûc problems.

What is logic?

Steps involved in solving a problem is known as Logic.There can be multiple logic to solve the same problem. Algorithm is a widely used form of representing Logic.

No. of teachers available= Total no. of teachers– No. of teachers busy in other classes in 4th lecture – No. of teachers on leaveTop of Form.

Python programming language

- Python is an example of a high level language; other high-level languages you might have heard of are C++, PHP, Pascal, C#, and Java.
- There are also low-level languages, sometimes referred to as machine languages or assembly languages.
- Computers can only execute programs written in low-level languages.
- Thus programs written in the high level language have to be translated into something more suitable before they can run.

- It is easier to program high level languages.
- These high level languages are executed in different computers.

Python features

Python is a High Level, Interpreted, Interactive and Object Oriented Programming Language

- Beginners Language
- Extensive Standard Library
- Cross Platform Compatibility
- Interactive Mode
- Portable and Extendable
- Databases and GUI Programming
- Scalable and Dynamic Semantics
- Automatic Garbage Collection

High level languages – Only humans can understand

Interpreter / Compiler – Translate the high level language to machine language and vice ersa **Low level language** – Only machine can understand (0s and 1s)

The engine that translates and runs Python is called the Python Interpreter:

There are two ways to use it:

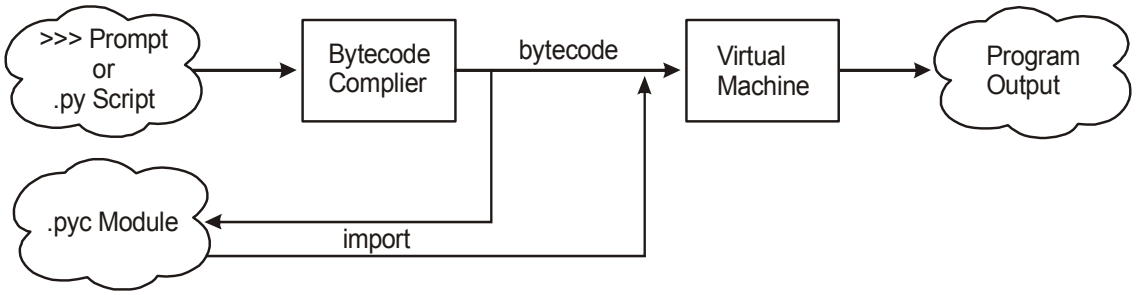
1. Immediate mode or interactive mode and
2. Script mode.

Python interpreter and interactive mode

An interpreter is a computer program which executes other programs. Python interpreter carries out instructions in your program. This interpreter can be used interactively to test out instructions on the fly.

Python interpreters

Python interpreters are available for many operating systems, allowing Python code to run on a wide variety of systems.



Two Parts

1. **Python byte code compiler**

   The byte code compiler accepts human readable python expressions and statements as input and produces machine readable python code as output.

2. **A virtual Machine which executes python byte code.**

   The virtual machine accepts Python byte code as input. And executes the virtual machine instructions represented by the byte code.

2.2 PYTHON MODES

1. Interactive mode

Interactive Mode, as the name suggests, allows us to interact with OS. Here, when we type Python statement, interpreter displays the result(s) immediately. That means, when we type Python expression / statement / command after the prompt (>>>), the Python immediately responses with the output of it.

The three right arrows are the *expression prompt*. The prompt is telling you that the Python system is waiting for you to type an expression. The window in which the output is displayed and input is gathered is termed the *console*.

Let's see what will happen when we type print "WELCOME TO PYTHON PROGRAMMING" after the prompt.

```
>>>print "WELCOME TO PYTHON PROGRAMMING"
WELCOME TO PYTHON PROGRAMMING
```

Example:

```
>>> print 5+10
    15
```

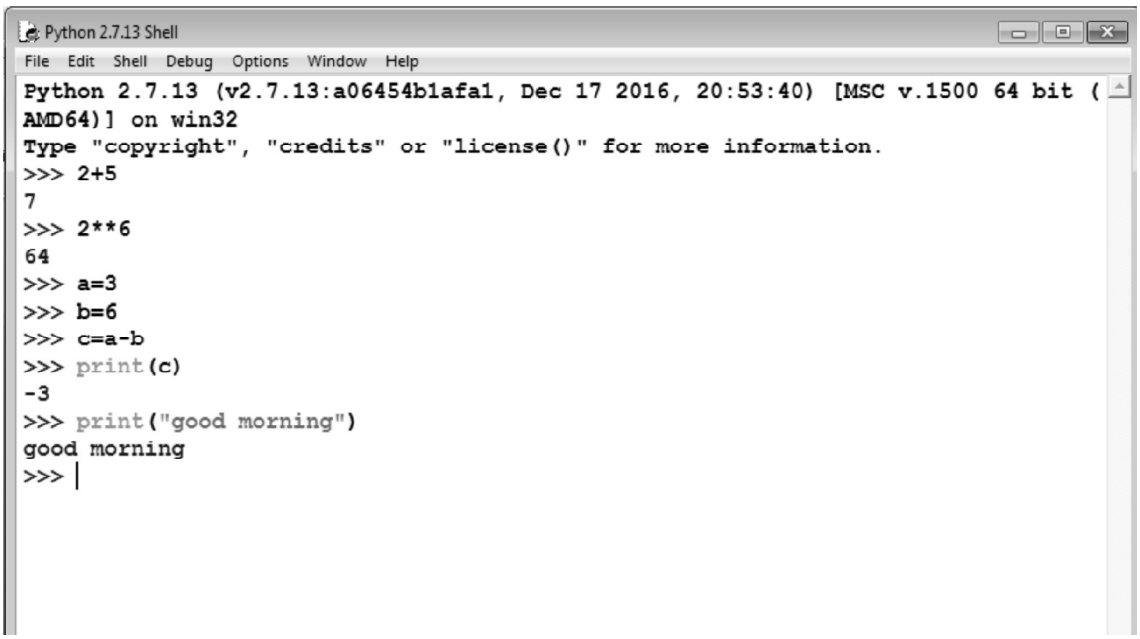
```
>>> x=10
>>>y=20
>>> print x*y
    200
```

Almost all Python expressions are typed on a single line. An exception to this rule occurs when expressions are typed in parenthesis. Each opening parenthesis must be matched to a closing parenthesis. If the closing parenthesis is not found, a *continuation prompt* is given. Normally this continuation prompt looks like an ellipsis, that is, three dots.

```
(2 +...
```

The continuation prompt is telling you that there is more to the expression you need to type. Fill in the rest of the expression, hit return, and the expression will be evaluated as before:

```
>>> (2+
……..3)
>>>5
```



2. Script Mode

In script mode, we type Python program in a file and then use interpreter to execute the content of the file. Working in interactive mode is convenient for beginners and for testing small pieces of code, as one can test them immediately.

But for coding of more than few lines, we should always save our code so that it can be modified and reused.

Python, in interactive mode, is good enough to learn, experiment or explore, but its only drawback is that we cannot save the statements and have to retype all the statements once again to re-run them.

Example: Input any two numbers and to find Quotient and Remainder.

Code:

```
a = input ("Enter first number")
b = input ("Enter second number")
print "Quotient", a/b
print "Remainder", a%b
        Enter first number10
                Enter second number3
                Quotient 3
```

2.3 VALUES AND DATA TYPES

2.3.1 VALUES

A **value** is one of the basic things a program. There are different values integers, float and strings. The numbers with a decimal point belong to a type called float. The values written in quotes will be considered as string, even it's an integer. If type of value is not known it can be interpreted as

Example:

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

Every object has:

- An Identity,
- A type, and
- A value.

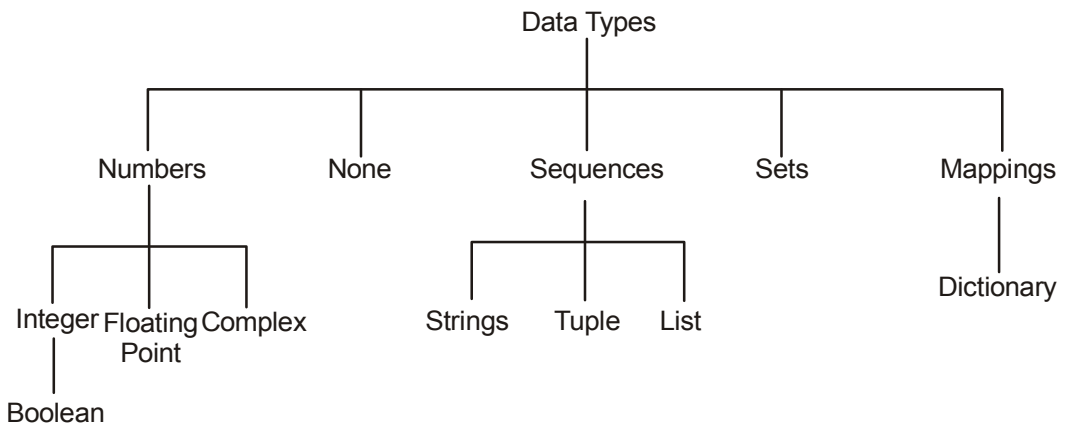
2.3.2 Identifier

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules

1. 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 myClass, var_1 and print_this_to_screen, all are valid example.
2. An identifier cannot start with a digit.
3. Keywords cannot be used as identifiers.
4. We cannot use special symbols like !, @, #, $, % etc. in our identifier.
5. Identifier can be of any length.

2.3.3 Data Type



It is a set of values, and the allowable operations on those values. It can be one of the following:

Category	Data Type	Example
Numeric	int	675
	complex	2 + 5j
	float	642.43
Textual	String(will be discussed in detail later)	"John"
Logical	Boolean	True, False

2.3.3.1 Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them.

For example

var1 = 1

var2 = 10

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement.

For example

del var

del var_a, var_b

Python supports four different numerical types

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Examples

Here are some examples of numbers

int	Long	float	complex
10	51924361L	0.0	3.14j
100	–0x19323L	15.20	45.j
–786	0122L	–21.9	9.322e–36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
–0490	535633629843L	–90.	–.6545+0J
–0x260	–052318172735L	–32.54e100	3e+26J
0x69	–4721885298529L	70.2–E12	4.53e–7j

Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

The built in numeric types supports the following operations:

x + y	sum of x and y
x – y	difference of x and y
x * y	product of x and y
x / y	quotent of x and y
x//y	(floored) quotient of x and y
x%y	remainder of x/y
–x	x negated
+	x unchanged
abs(x)	absolute value of magnitude of x
int(x)	x converted to integer
long(x)	x converted to long integer

float(x)	x converted to floating point
complex(re.im)	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
c.conjugate()	conjugate of the complex number c. (Identity on real numbers)
divmod(x, y)	the pair (x//y, x%y)
pow(x, y)	x to the power y
x**y	x to the power y

2.3.3.2 Boolean

A Boolean value is either true or false. It is named after the British mathematician, George Boole, who first formulated Boolean algebra.

In Python, the two Boolean values are True and False (the capitalization must be exactly as shown), and the Python type is bool.

```
>>> type(True)
    <class 'bool'>
>>> type(true)
    Traceback (most recent call last):
    File "<interactive input>", line 1, in <module>
    NameError: name 'true' is not defined
```

Example:

```
>>> 5 == (3 + 2) # Is five equal 5 to the result of 3 + 2?
    True
>>> 5 == 6
    False
>>> j = "hel"
>>> j + "lo" == "hello"
    True
```

2.3.4 String

A String in Python consists of a series or sequence of characters - letters, numbers, and special characters.

Strings are marked by quotes:

- single quotes (‘ ‘)

  Eg, ‘This a string in single quotes’

- double quotes (“ “)

  Eg, “‘This a string in double quotes’”

- triple quotes(“”” “”””)

  Eg, This is a paragraph. It is made up of multiple lines and sentences.”””””

- Individual character in a string is accessed using a subscript (index).
- Characters can be accessed using indexing and slicing operations

Example

```
str = ‘Hello World!’
print str    # Prints complete string
print str[0]        # Prints first character of the string
print str[2:5]      # Prints characters starting from 3rd to 5th
print str[2:]       # Prints string starting from 3rd character
print str * 2       # Prints string two times
print str + “TEST” # Prints concatenated string
```

Output

```
Hello World!
H
lo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Strings in Python can be enclosed in either single quotes (‘) or double quotes (“), or three of each (‘ or “””)

```
>>>type(‘This is a string.‘) <class ‘str‘>
>>> type(“And so is this.”)
    <class ‘str‘>
>>>type(“””and this.”””)
    <class ‘str‘>
>>>type(“‘and even this...“‘)
    <class ‘str‘>
```

Double quoted strings can contain single quotes inside them.

(“Alice‘s Cup”)

Single quoted strings can contain double quotes inside them

(‘Alice said ,”Hello”’)

Strings enclosed with three occurrences of either quote symbol are called triple quoted strings. They can contain either single or double quotes:

```
>>>print(“““Oh no”, she exclaimed, “Ben‘s bike is broken!”“‘) “Oh no”, she exclaimed, “Ben‘s bike is broken!”
  >>>Triple quoted strings can even span multiple lines:
>>>message = “””This message will
    ... span several
    ... lines.”””
>>>print(message)
```

This message will span several lines.

2.3.5 Lists

List is also a sequence of values of any type. Values in the list are called elements

/items. These are mutable and indexed/ordered. List is enclosed in square brackets ([]).

Example

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print list  # Prints complete list
print list[0]          # Prints first element of the list
print list[1:3]        # Prints elements starting from 2nd till 3rd
print list[2:]         # Prints elements starting from 3rd element
print tinylist * 2     # Prints list two times
print list + tinylist  # Prints concatenated lists
```

Output

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

2.3.6 Variable

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

The assignment statement creates new variables and gives them values:

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

This example makes three assignments. The first assigns the string "What's up, Doc?" to a new variable named message. The second gives the integer 17 to n, and the third gives the floating-point number 3.14159 to pi.

The print statement also works with variables.

```
>>> print message What's up, Doc?
>>> print n
      17
>>> print pi
      3.14159
```

The type of a variable is the type of the value it refers to.

```
>>> type(message) <type 'str'>
>>> type(n)
      <type 'int'>
>>> type(pi)
    <type 'float'>
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example "

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example "

```
a,b,c = 1,2,"john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

2.3.7 Keywords

Keywords define the language's rules and structure, and they cannot be used as variable names. Python has thirty-one keywords:

and	**del**	**from**	**not**	**while**
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

2.3.8 Expressions And Statements

Expressions

An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter evaluates it and displays the result:

```
>>> 1 + 1
    2
```

Example : Finding area and perimeter

```
length = 5
breadth = 2
area = length * breadth
print('Area is', area)
print('Perimeter is', 2 * (length + breadth))
```

Output:

```
Area is 10
Perimeter is 14
```

Statements

A statement is an instruction that the Python interpreter can execute. We have seen two kinds of statements: print and assignment. When you type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result. A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

Example

```
print 1
  x = 2
print x
It produces the following output
          1
          2
```

2.3.9 Tuple Assignment

Tuples are a sequence of values of any type and are indexed by integers. They are immutable. Tuples are enclosed in ().

Example

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
print tuple                # Prints complete list
print tuple[0]             # Prints first element of the list
print tuple[1:3]           # Prints elements starting from 2nd till 3rd
print tuple[2:]            # Prints elements starting from 3rd element
print tinytuple * 2        # Prints list two times
print tuple + tinytuple    # Prints concatenated lists
```

Output

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)

(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tuple[2] = 1000          # Invalid syntax with tuple
list[2] = 1000           # Valid syntax with list
```

Python has a very powerful tuple assignment feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

```
(name, surname, b_year, movie, m_year, profession, b_place) = julia
```

This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.

One way to think of tuple assignment is as tuple packing/unpacking.

In tuple packing, the values on the left are 'packed' together in a tuple:

```
>>> b = ("Bob", 19, "CS") # tuple packing
```

In tuple unpacking, the values in a tuple on the right are 'unpacked' into the variables/names on the right:

```
>>> b = ("Bob", 19, "CS")
>>> (name, age, studies) = b # tuple unpacking
>>> name
```

```
    'Bob'
>>>age
    19
>>>studies
    'CS'
```

Swapping in tuple assignment:

```
(a, b) = (b, a)
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

Naturally, the number of variables on the left and the number of values on the right have to be the same:

```
>>> (a, b, c, d) = (1, 2, 3)
```

Value Error: need more than 3 values to unpack

2.3.10 Precedence of Operators

Operators are the constructs which can manipulate the value of operands.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym PEMDAS is a useful way to remember the order of operations:

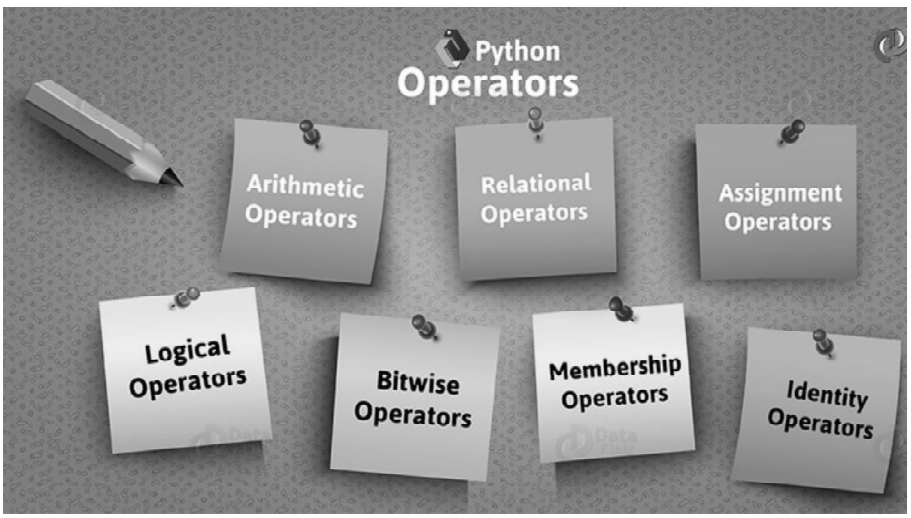
1. **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3–1) is 4, and (1+1)**(5–2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even though it doesn't change the result.

2. **E**xponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.

3. **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So 2*3–1 yields 5 rather than 4, and 2/3-1 is -1, not 1 (remember that in integer division, 2/3=0).

4. Operators with the same precedence are evaluated from left to right. So in the expression minute*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had been evaluated from right to left, the result would have been 59*1, which is 59, which is wrong.

2.3.11 Operators



Types of Operators:

– Python language supports the following types of operators

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic operators:

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Arithmetic operators in Python

Operator	Meaning	Example
+	Add two operands or unary plus	x+y
–	Subtract right operand from the left or unary minus	x-y
*	Multiply two operands	x*y
/	Divide left operand by the right one (always results into float)	x/y
%	Modulus - remainder of the division of left operand by the right	x%y
//		x//y
**	Exponent - left operand raised to the power of right	x**y(x to the power y)

Examples

```
a=10
b=5
print("a+b=",a+b)
print("a-b=",a-b)
print("a*b=",a*b)
print("a/b=",a/b)
print("a%b=",a%b)
print("a//b=",a//b)
print("a**b=",a**b)
```

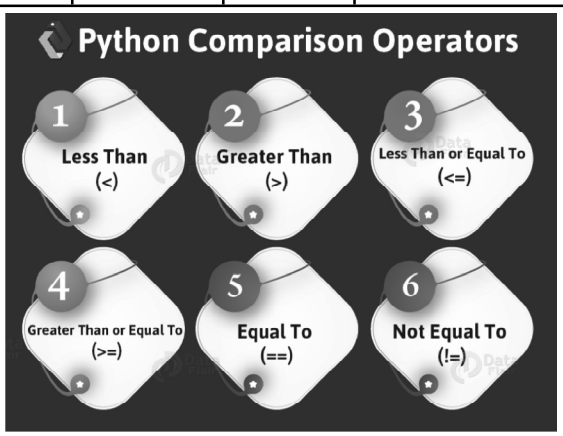
Output

```
a+b= 15
a-b= 5
a*b= 50
a/b= 2.0
a%b= 0
a//b= 2
a**b= 100000
```

Comparison (Relational) Operators:

- Comparison operators are used to compare values.
- It either returns True or False according to the condition. **Assume, a=10 and b=5.**

Operator	Symbol	Form	Result
greater than	>	a > b	True if a is greater than b; else False
less than	<	a < b	True if a is less than b; else False
greater than or equal to	>=	a>=b	True if a is greater than or equal to b; false False
less than or equal to	<=	a <=b	True if a is less than or equal to b; else False
equal to	==	a==b	True if a is equal to b; else False
not equal to	!=	a!=b	True if a is not equal to b; else False



Example

```
a=10
b=5
print("a>b=>",a>b)
print("a>b=>",a<b)
print("a==b=>",a==b)
print("a!=b=>",a!=b)
print("a>=b=>",a<=b)
print("a>=b=>",a>=b)
```

Output:

```
a>b=> True
a>b=> False
a==b=> False
a!=b=> True
a>=b=> False
a>=b=> True
```

Assignment Operators

Assignment Operator combines the effect of arithmetic and assignment operator

Operator	Name	Example	Equivalent
+=	Addition assignment	1 + = 8	i = i + 8
–=	Subtraction assignment	i –=8	i = i – 8
*=	Multiplication assignment	1 * = 8	1 = 1 * 8
/=	Float division assignment	i / = 8	i = i / 8
//=	Integer division assignment	i //=8	i = i // 8
%=	Remainder assignment	i %=8	i \ i %8
**=	Exponent assignment	i **=8	i = i ** 8

Example

```
a = 21
b = 10
c = 0
c = a + b
print(“Line 1 - Value of c is “, c)
c += a
print(“Line 2 - Value of c is “, c)
c *= a
print(“Line 3 - Value of c is “, c)
c /= a
print(“Line 4 - Value of c is “, c)
c = 2
c %= a
print(“Line 5 - Value of c is “, c)
c **= a
print(“Line 6 - Value of c is “, c)
c //= a
print(“Line 7 - Value of c is “, c)
```

Output

```
Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52.0
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864
```

Logical Operators

Symbol	Description
or	If any one of the operand is true, then the condition becomes true.
and	If both the operands are true, then the condition becomes true.
not	Reverses the state of operand/condition.

Example

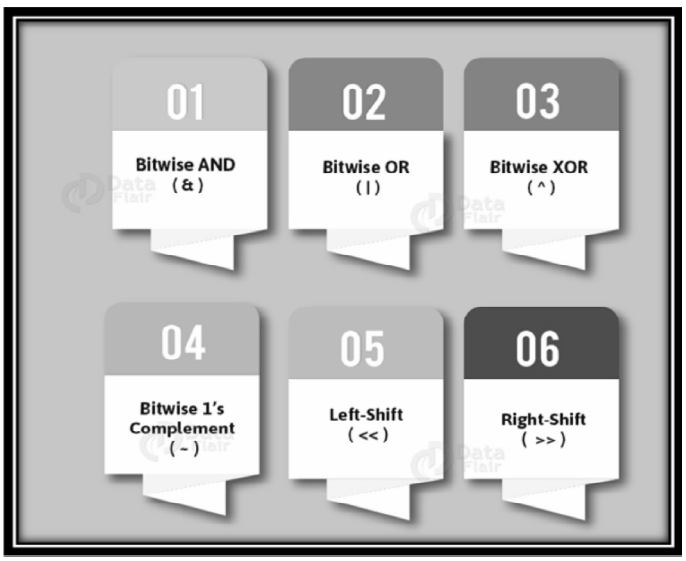
```
a = True
b = False
print('a and b is',a and b)
print('a or b is',a or b)
print('not a is',not a)
```

Output

```
x and y is False
x or y is True
not x is False
```

Bitwise Operators:

A **bitwise operation** operates on one or more **bit** patterns at the level of individual bits.



Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Example:

```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

c = a & b;        # 12 = 0000 1100
print "Line 1 - Value of c is ", c

c = a | b;        # 61 = 0011 1101
print "Line 2 - Value of c is ", c

c = a ^ b;        # 49 = 0011 0001
print "Line 3 - Value of c is ", c

c = ~a;           # -61 = 1100 0011
print "Line 4 - Value of c is ", c
```

```
c = a << 2;       # 240 = 1111 0000
print "Line 5 - Value of c is ", c

c = a >> 2;       # 15 = 0000 1111
print "Line 6 - Value of c is ", c
```

Output:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Membership Operators:

These operators test whether a value is a member of a sequence. The sequence may be a list, a string, or a tuple. We have two membership python operators- 'in' and 'not in'.

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is a member of sequence y.

Example:

a. in

This checks if a value is a member of a sequence. In our example, we see that the string 'fox' does not belong to the list pets. But the string 'cat' belongs to it, so it returns True. Also, the string 'me' is a substring to the string 'disappointment'. Therefore, it returns true.

```
>>> pets=['dog','cat','ferret']
```

```
>>> 'fox' in pets
False
>>> 'cat' in pets
True
>>> 'me' in 'disappointment'
True
```

b. not in

Unlike 'in', 'not in' checks if a value is not a member of a sequence.

```
>>> 'pot' not in 'disappointment'
    True
```

Identity Operator

Let us proceed towards identity Python Operator.

These operators test if the two operands share an identity. We have two identity operators- 'is' and 'is not'.

a. is

If two operands have the same identity, it returns True. Otherwise, it returns False. Here, 2 is not the same as 20, so it returns False. Also, '2' and "2" are the same. The difference in quotes does not make them different. So, it returns True.

```
>>> 2 is 20
    False
>>> '2' is "2"
    True
```

b. is not

2 is a number, and '2' is a string. So, it returns a True to that.

```
>>> 2 is not '2'
    True
```

Comments

As programs get bigger and more complicated, they get more difficult to read. It is a good idea to add notes to your programs to explain in natural language what the program is doing.

A comment in a computer program is text that is intended only for the human reader — it is completely ignored by the interpreter.

There are two types of comments in python:

Single line comments

Multi line comments

Single Line Comments

In Python, the # token starts a comment..

Example

```
print("Not a comment")
#print("Am a comment")
```

Result

Not a comment

Multiple Line Comments

Multiple line comments are slightly different. Simply use 3 single quotes before and after the part you want commented.

Example

```
''
print("We are in a comment")
print ("We are still in a comment")
''
print("We are out of the comment")
```

Result

We are out of the comment

Suggested links to refer

What can you do with Python?

https://www.youtube.com/watch?v=hxGB7LU4i1I (Duration: 3:56).

Python Variables and Data Types

https://www.youtube.com/watch?v=657yt4gYjRo (Duration: 16:45)

https://www.youtube.com/watch?v=jfk6ZHdRyAQ

Food for Thought :What are the diûerent variables and data types used in the video?

http://www.tutorialspoint.com/python/python_variable_types

OPERATORS

http://www.tutorialspoint.com/python/python_basic_operators.htm

Assignment Operators:

https://www.youtube.com/watch?v=_gFrIwXHfL0(Duration: 7:06)

Arithmetic Operators:

https://www.youtube.com/watch?v=lqtj8XM0leA(Duration: 10:40)

Relational Operators:

https://www.youtube.com/watch?v=Kte_yYE153M(Duration: 13:05)

Logical Operators

https://www.youtube.com/watch?v=Kte_yYE153M(Duration: 8:54)

What can you do with Python?

https://www.youtube.com/watch?v=hxGB7LU4i1I (Duration: 3:56).

2.4 MODULES AND FUNCTION

2.4.1 Modules

- Python has a way to put related code in a ûle and use that ûle in other Python ûles. It is called a module. It allows logical organization of code. It can be used to deûne variables, functions and classes.
- Grouping related code into module makes it easier to understand and use.

- Module is nothing but a Python ûle with '.py' extension.
- Module must be imported before using it's functions in any other module/ python file.
- Modules should have short, all-lowercase names. Underscores can be used in the module name to improve readability.

Syntax to import a module:

import <<modulename>>

Example

1. Math functions:

Python has a math module that provides mathematical functions.

```
>>> import math
```

This statement creates a **module object** named math. If you print the module object, you get some information about it:

```
>>> print math
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module.

To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

```
import math
print("The value of pi is", math.pi)
```

Eg:

```
>>> math.sqrt(2)
    2.0 0.707106781187
```

There are **four ways to import a module** in our program, they are

Import: It is simplest and most common way to use modules in our code **Example :** import math x = math pi print("The value of pi is", x) Output: The value of pi is 3.141592653589793	**from import:** It is used to get a specific function in the code instead of complete file. **Example :** from math import pi x = pi print("The value of pi is", x) Output: The value of pi is 3.141592653589793
import with renaming: We can import a module by renaming the module as our wish. **Example :** import math as m x = m.pi print("The value of pi is", x) Output: The value of pi is 3.141592653589793	**import all:** We can import all names(definitions) gotm s mofulr udinh* **Example :** from math import* x = pi print("The value of pi is", x) Output: The value of pi is 3.141592653589793

2. Calendar functions :

Python has a cal module that provides calendar functions.

```
>>> import cal
```

Example:

```
import cal
x=cal.add(5,4)
print(x)
```

3. Random functions :

Python offers `random` module that can generate random numbers.

These are pseudo-random number as the sequence of number generated depends on the seed.

1. **Randint**

Example : Randint accepts two parameters: a lowest and a highest number.

```
import random
print( random.randint(0, 5))
```

2. **Random**

Example : If you want a larger number, you can multiply it.

```
import random
print(random.random() * 100)
```

3. **Choice**

Example: Generate a random value from the sequence sequence.

```
import random
print(random.choice( ['red', 'black', 'green'] ))z
```

4. **Shuffle**

Example :The shuffle function, shuffles the elements in list in place, so they are in a random order.

```
from random import shuffle
x = [[i] for i in range(10)]
shuffle(x)
print(x)
```

5. **Randrange**

Example :Generate a randomly selected element from range(start, stop, step)

```
import random
for i in range(3):
print random.randrange(0, 101, 5)
```

2.4.2 Function

In Python, function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Need of function

Provide better modularity and high degree of reusability.

Python supports:

1. Built-in functions e.g. print()
2. User-deûned functions

i) Built in functions

Built in functions are the functions that are already created and stored in python.

These built in functions are always available for usage and accessed by a programmer. It cannot be modified. Some examples are below :

abs ( )	divmod ( )	input ( )
all ( )	enumerate ( )	int ( )
any ( )	eval ( )	ininstance ( )
basestring ( )	execfile ( )	issubclass ( )
bin ( )	file ( )	iter ( )
bool ( )	filter ( )	len ( )
bytearray ( )	float ( )	list ( )
callable ( )	format ( )	locals ( )
chr ( )	frozenset ( )	long ( )
classmethod ( )	getattr ( )	map ( )
cmp ( )	globals ( )	max ( )
compile ( )	hasattr ( )	memoryview ( )
complex ( )	hash ( )	min ( )
delattr ( )	help ( )	next ( )
dict ( )	hex ( )	object ( )
dir ( )	id ( )	oct ( )

ii) **User Defined Functions:**

- User defined functions are the functions that programmers create for their requirement and use.
- These functions can then be **combined to form module** which can be used in other programs by importing them.

Advantages of user defined functions:

- Programmers working on large project can divide the workload by making different functions.
- If repeated code occurs in a program, function can be used to include those codes and execute when needed by calling that function.

Defining a function

- Function block start with a keyword def` followed by

  function_name, paranthesis (**( )**) and colon (**:**).
- Arguments are placed inside the parenthesis
- Function block can have optional statement/comment for documentation as its first line.
- Every line inside code is indented return expression (statement) exits the function by returning an expression to the caller function return statement with no expression is same as return None.

Syntax

```
def NAME( LIST OF PARAMETERS ):
    STATEMENTS
return [expression]
```

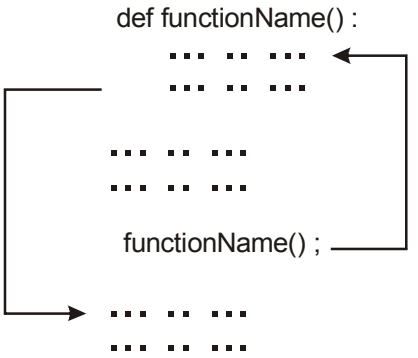
The function definition have the following parts:

A **header**, which begins with a keyword and ends with a colon.

A **body** consisting of one or more Python statements, each indented the same amount (4 spacesis the Python standard) from the header.

Calling a function

Function calls contain the name of the function being executed followed by a list of values, called arguments, which are assigned to the parameters in the function definition.



Example

```
# Function definition is here
    def printme( str ):
        "This prints a passed string into this function"
        print (str)
        return;
# Now you can call printme function
        printme("I'm first call to user defined function!")
        printme("Again second call to the same function")
//Scope of a variable
        def my_func():
            x = 10
            print("Value inside function:",x)
            x = 20
        my_func()
        print("Value outside function:",x)
```

Output

```
Value inside function: 10
Value outside function: 20
```

Flow of Execution:

- The order in which statements are executed is called the **flow of execution**
- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Example

```
def f1():
    print("Moe")
def f2():
    f4()
    print("Meeny")
def f3():
    f2()
    print("Miny")
    f1()
def f4():
    print("Eeny")
    f3()
```

Output

```
Eeny
Meeny
Miny
Moe
```

2.4.3 Parameters And Arguments

Parameters:

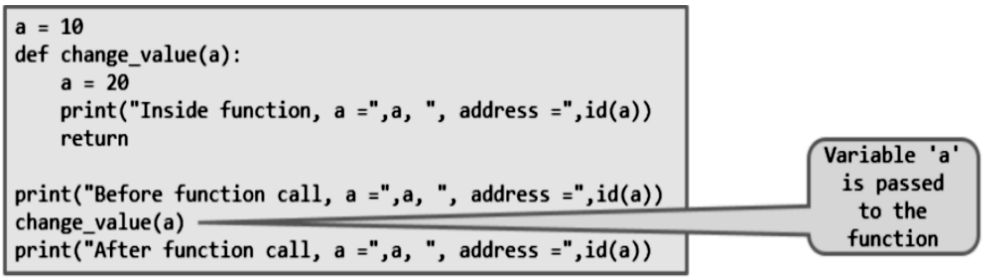
- Parameters are the value(s) provided in the parenthesis when we write function header.
- These are the values required by function to work.
- If there is more than one value required, all of them will be listed in parameter list separated by comma.
- Example: def my_add(a,b):

Arguments :

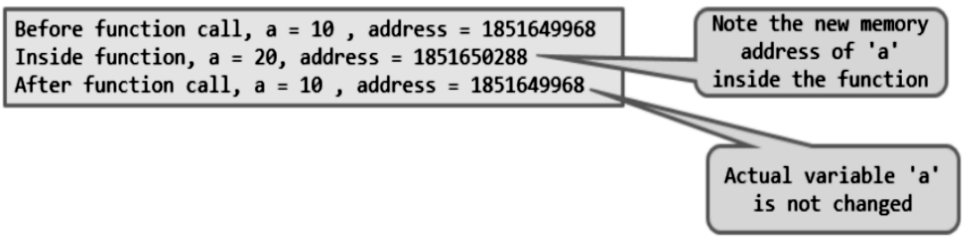
- Arguments are the value(s) provided in function call/invoke statement.
- List of arguments should be supplied in same way as parameters are listed.
- Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.
- Example: my_add(x,y)

Pass by value

In pass-by-value, the function receives a copy of the argument objects passed to it by the caller, stored in a new location in memory.

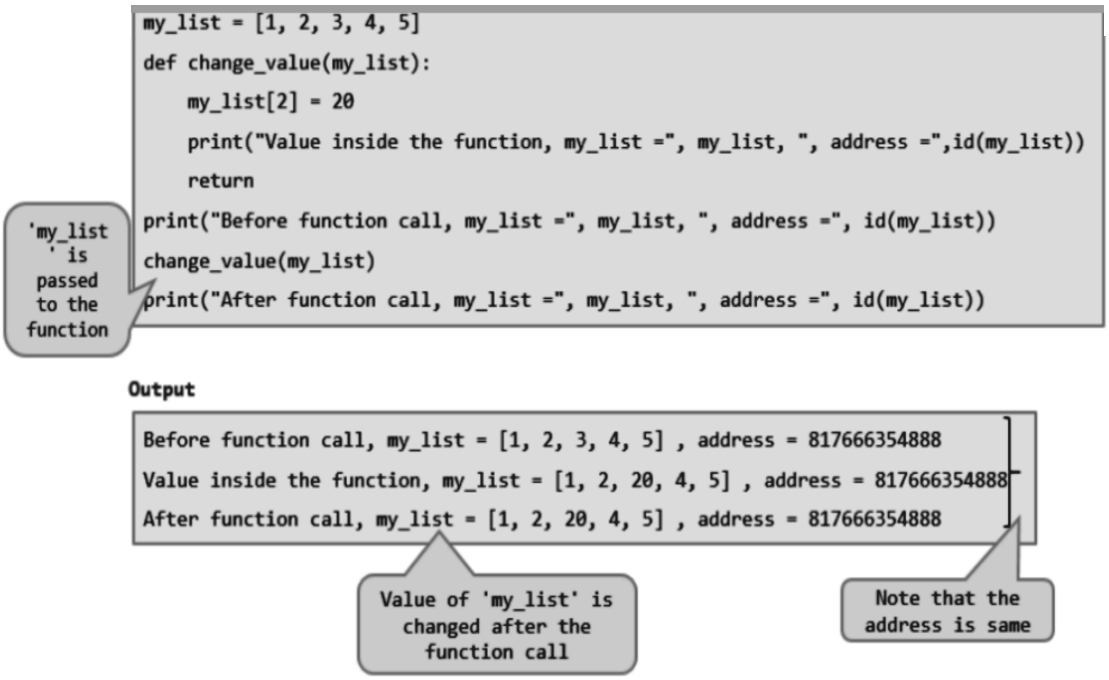


Output



Pass By reference

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:



2.4.4 Function Arguments

You can call a function by using the following types of formal arguments:

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

Required arguments

The number of arguments in the function call should match exactly with the function definition.

Example :

```
def my_details( name, age ):
    print("Name: ", name)
    print("Age ", age)
    return
my_details("george",56)
Output:
    Name: george
    Age 56
```

Keyword Arguments:

Python interpreter is able to use the keywords provided to match the values with parameters even though if they are arranged in out of order.

Example:

```
def my_details( name, age ):
    print("Name: ", name)
    print("Age ", age)
    return
my_details(age=56,name="george")
```

Output:

```
    Name: george
    Age 56
```

Default Arguments:

Assumes a default value if a value is not provided in the function call for that argument.

Example :

```
def my_details( name, age=40 ):
    print("Name: ", name)
    print("Age ", age)
    return
my_details(name="george")
```

Output:

Name: george

Age 40

Variable length Arguments:

If we want to specify more arguments than specified while defining the function, variable length arguments are used. It is denoted by * symbol before parameter.

Example :

```
def my_details(*name ):
    print(*name)
my_details("rajan","rahul","micheal",ärjun")
```

Output:

rajan rahul micheal ärjun

2.4.5 The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

Syntax

lambda [arg1 [,arg2,.....argn]]:expression

Example

```
# Function definition is here
    sum = lambda arg1, arg2: arg1 + arg2;
# Now you can call sum as a function
    print "Value of total : ", sum( 10, 20 )
    print "Value of total : ", sum( 20, 20 )
#When the above code is executed, it produces the following result "
    Value of total : 30
    Value of total : 40
```

2.4.6 Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable. The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python “

Diûerent types of variables

Local variables

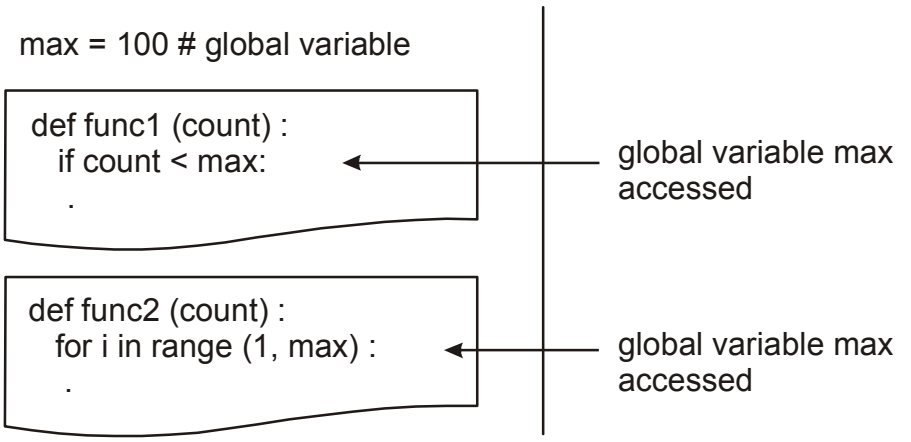
- Variables deûned inside the function have local scope
- Can be accessed only inside the function in which it is deûned

Example

```
def func1 ( ) :
    n = 10
    print (‘n in func1 = ’, n)
def func2 ( ) :
    n = 20
    print (‘n in func2 before call to func1 =’, n)
    func1 ( )
    print (‘n in func2 after call to func1 =’, n)
>>> func2 ( )
n in func2 before clal to func1 = 20
n in func1 = 10
n in func2 after clal to func1 = 20
```

Global variables

- Variables deûned outside the function have global scope.
- Variables can be accessed throughout the program by all other functions as wellGlobal/Local variables.



Variable max is defi ned outside func1 and func2 and therefore “global” to each

Suggested Link to Refer:

Built-in Functions in Python

https://www.youtube.com/watch?v=GTpl5yq3bvk (Duration: 10:40)

https://docs.python.org/3/library/functions.html

Defining Functions

https://www.youtube.com/watch?v=TkBLZk_hV5Y(Duration: 10:27)

Pass-by-Value vs. Pass-by-Reference

https://www.youtube.com/watch?v=hhCs-UllGfw (Duration: 2:52)

Types of function arguments

https://www.youtube.com/watch?v=RDzm2oHSAug (Duration: 11:38)

http://www.tutorialspoint.com/python/python_functions.htm

ILLUSTRATIVE EXAMPLES

1. Python Program to swap or exchange the values of two variables

Method 1

```
a = 10
b = 20
print("before swapping\na=", a, " b=", b)
temp = a
a = b
b = temp
print("\nafter swapping\na=", a, " b=", b)
```

Method 2

```
a = 30
b = 20
print("\nBefore swap a = %d and b = %d" %(a, b))
a, b = b, a
print("\nAfter swaping a = %d and b = %d" %(a, b))
```

2. Circulate the values of n Variables

```
l=[1,2,3,4,5]
print(l[::-1])
```

3. Python Program to test the year is leap or not

```
year=int(input("Enter year to be checked:"))
if(year%4==0 and year%100!=0 or year%400==0):
    print("The year is a leap year!)
else:
    print("The year isn't a leap year!)
```

ASSIGNMENT QUESTIONS

1. Program variables have data types such as: *Integer*; *Float* and *String*. After the execution of the following snippet of code what are the data type of the three variables *var_one*, *var_two* and *var_three*?

   var_one = 57

   var_two = 9.81

   var_three = 'What have the Romans ever done for us?'

2. The Data type of a variable defines the way in which the variable can be processed. With this in mind what will happen when the following snippet of code is executed.

   var_one = 'What have the Romans ever done for us?'

   var_two = 'He is not the messiah he is a very naughty boy!'

   var_three = var_one * var_two

3. Write a Python program that accepts an integer (n) and computes the value of n+nn+nnn

4. Write a program that calculates and prints the value according to the given formula:

   Q = Square root of [(2 * C * D)/H]

   Following are the fixed values of C and H:

   C is 50. H is 30.

   D is the variable whose values should be input to your program in a comma-separated sequence.

   Example

   Let us assume the following comma separated input sequence is given to the program:

   100,150,180

   The output of the program should be:

   18, 22, 24

5. A circular swimming pool is x metres in diameter. What volume of water does it contain if the pool is the same depth at all points?

PROGRAM EXERCISES

1. Write a Python program to compute Greatest Common Divisor of two numbers using function.
2. Write a Python program to swap two numbers using function.
3. Code a python program to accept two numbers m and n, find the quotient, remainder and print the result.
4. Write a python program to merge a two list.
5. Write a python program to swap variables without using third variable.
6. Write a python program to add two numbers
7. Write a python program to find the area of a triangle
8. Write a python program to convert Celsius to Farenheit.
9. Write a python program to concatenate two strings
10. Write a python program to solve a quadratic equation

PART – A QUESTION AND ANSWERS

1. What is mean by high level and low level programming language?

High level language is a computer programming language that resembles natural language or mathematical notation. It is human readable form. Eg: C, Java, Python.

Low level language can be understood only by computers. (0,1)

2. What is mean by a python interpreter?

The engine that translates and runs Python is called the Python Interpreter. It is available for many OS. Two parts of the interpreter are: (i) Python byte code compiler (ii) A virtual machine

3. What are the two modes of python?

Interactive mode

Interactive Mode, as the name suggests, allows us to interact with OS. Here, when we type Python statement, interpreter displays the result(s) immediately. It acts as a calculator.

```
>>>print "WELCOME TO PYTHON PROGRAMMING"
WELCOME TO PYTHON PROGRAMMING
```

Script Mode

In script mode, we type Python program in a file and then use interpreter to execute the content of the file.

4. Define continuation prompt.

The continuation prompt will notify you that there is more to the expression you need to

type.

```
>>> (2 +
... 3) # continuation
5
```

5. What are application and system programs?

Application programs are developed for particular application. System programs keep the hardware and software running together smoothly.

6. What is mean by values?

A **value** is one of the basic things a program works with, like a letter or a number.

These values are classified into different classes, or data types:

1,2 – Integers

'Hello World'- String

7. Define identifiers.

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

8. What are the rules for declaring the identifiers?

1. 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 (_).
2. An identifier cannot start with a digit.
3. Keywords cannot be used as identifiers.

4. We cannot use special symbols like !, @, #, $, % etc. in our identifier.
5. Identifier can be of any length.

9. Define data type.

The data type specifies the type of data which can be stored.

10. What is Boolean?

A Boolean value is either true or false. In Python, the two Boolean values are True and False and the Python type is bool

11. What is the use of type() function?

The type function is used to identify the type of the variable.

```
>>> type("Hello, World!")
<class 'str'>
>>> type(17)
<class 'int'>
```

12. Define String.

It is an ordered sequence of letters/characters. They are enclosed in single quotes (' ') or

double quotes (‘’ “).

```
str = 'Hello World!'
>>> type('Hello World!')
<class 'str'>
```

13. Define Lists.

List is also a sequence of values of any type. Values in the list are called elements / items.

These are mutable and indexed/ordered. List is enclosed in square brackets ([]).

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
```

14. Define variable.

A variable is a name that refers to a value.

The assignment statement creates new variables and gives them values.

>>> message = “Who is this?”

>>> n = 17

15. What is mean by multiple assignment?

Python allows you to assign a single value to several variables simultaneously.

For example “ **a = b = c = 1**

16. Define keywords.

Keywords define the language‘s rules and structure, and they cannot be used as variable names. Python has thirty-one keywords. Eg: global, or, with, assert**,** else, if, pass, yield

17. State the definition of statement.

A statement is an instruction that the Python interpreter can execute.

Eg: x=2

18. How will you assign a tuple?

Tuples are a sequence of values of any type and are indexed by integers. They are immutable. Tuples are enclosed in (). Eg: tuple = (‘abcd’, 786 , 2.23, ‘john’, 70.2)

19. State the precedence of operators.

The operator precedence rule in python is used to execute the expression contains more than one operator.

The precedence rule is:

P EMDAS –

1. Parenthesis
2. Exponentiation
3. Multiplication, Division (Same precedence) – If both comes, evaluate from left to right
4. Addition , Subtraction (Same precedence) – If both comes, evaluate from left to right

20. What are comments?

A comment in a computer program is text that is intended only for the human reader

Comments are used for documenting our program.

It is useful in large programs.

It is not printed and ignored by the interpreter.

Types of comments:

Single line comments – Starts with # symbol.

Eg: #print("Am a comment")

Multi line comments – Starts and ends with three single quotes.

Eg:

‘’’

print("We are in a comment")

print ("We are still in a comment")

‘’’

21. What is mean by modules?

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for e.g.: example.py, is called a module.

We can use modules by :

import module1[, module2[,... moduleN]

22. What is mean by functions? Mention the syntax of functions

Function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks.

Defining a function

Function block start with a keyword "def" followed by function_name, paranthesis (**( )**) and colon (**:**). Arguments are placed inside the parenthesis

Every line inside code is indented return expression (statement) exits the function by returning an expression to the caller function

def NAME(LIST OF PARAMETERS):

STATEMENTS

return [expression]

23. How will you call a function?

A function call is used to call the function which substitutes the entire definition of the function.

Eg:

```
def mul():
a=4
b=8
c=a*b
print(c)
mul() # function call
```

24. Define parameters.

Function calls contain the name of the function being executed followed by a list of values, called arguments, which are assigned to the parameters in the function definition.

Eg: def mul(a,b)

25. Define call by value and call by reference.

There are two ways to pass value or data to function, call by value and reference.

Original value is not modified in call by value but it is modified in call by reference.

26. What is mean by local and global variables?

Variables that are defined inside a function body are called as local scope variables.

Variables that are defined outside a function body are called as global scope variables.

Eg:

```
total = 0; # This is global variable.
def sum( arg1, arg2 ):
total = arg1 + arg2; # Here total is local variable.
```

27. Give the different types of function argument.

The four types of function arguments are:

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

PART – B QUESTIONS

1. Discuss about the various operators in python.
2. Discuss about operator precedence in python with eg.
3. Explain in detail about functions and parameters.
4. Explain about the various arguments with eg.
5. What is the use of comments and modules. Explain
6. Write a program to
   a. exchange the values of variables
   b. circulate the value of n variables
   c. find distance between two points

Unit III

CONTROL FLOW, FUNCTIONS

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: state, while, for, break, continue, pass; Fruitful functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Lists as arrays. Illustrative programs: square root, gcd, exponentiation, sum an array of numbers, linear search, binary search.

3.1 BOOLEAN VALUES AND OPERATOR

A Boolean expression (or logical expression) evaluates to one of two states true or false. Python provides the boolean type that can be either set to **False** or **True**. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

Example 3.1:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True and False are special values that belong to the type bool; they are not strings:

Example 3.2:

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

Capitalization should be exact. Ignoring the proper use of upper or lower case, will result in error.

Example 3.3:

```
>>> type(true)
Traceback (most recent call last):
File "<interactive input>, line 1 in <module>,
NameError: name 'true' is not defined
```

3.1.1 Relational Operators

The relational operators (Comparison Operators) which return the Boolean values are:

x != y	x is not equal to y
x==y	x is equal to y
x > y	x is greater than y
x < y	x is less than y
x >= y	x is greater than or equal to y
x <= y	x is less than or equal to y

Note: A common error is using the mathematical symbol = in the expressions and it is an assignment operator. There are no operators like => or =<.

3.1.2 Logical Operators

There are three **logical operators**: and, or, and not. The semantics (meaning)of these operators is similar to their meaning in English.

Example3.4:

x>0	True, if x is greater than zero.
x<5	True, if x is less than five.
n%2==0 or n%3==0	True, if either (any one) of the condition is True
	True, if both the conditions are True

Example Program3.5:

Program 1:

```
n=88
if n%2 ==0 or n%3==0:
    print("yes")
```

Output:

yes

Program 2:

```
n=6
if n%2==0 or n%3==0:
    print("yes")
```

Output:

yes

The not operator negates a boolean expression, so **not** (x > y) is True if x > y is False, that is, if x is less than or equal to y.

The expression on the left of the **or** operator is evaluated first: if the result is True, Python does not evaluate the expression on the right — this is called short-circuit evaluation.

Similarly, for the **—and** operator, if the expression on the left yields False, Python does not evaluate the expression on the right. So there are no unnecessary evaluations.

Python is not very strict. Any nonzero number is interpreted as True:

```
>>> 42 and True
True
```

Recommended online video tutorial link: Boolean values and operators

https://www.youtube.com/watch?v=xgzdb2hTdL0

3.1.3 Control Statement

A control statement is a statement that determines the control flow of a set of instructions. There are three fundamental forms of control that programming languages provide—

1. Sequential structure (if) – Instructions are executed in an order that they are written
2. Selection Structure/Branching/Decision Making – Instructions are being executed selectively based on conditions
   - **if** statement
   - **If..else** statement
   - **If..elif..else** statement
3. Repetition structure/Looping/Iterative – Instructions are repeatedly executed
   - **while**
   - **For**

Unconditional Structure

- Break
- Continue
- Pass

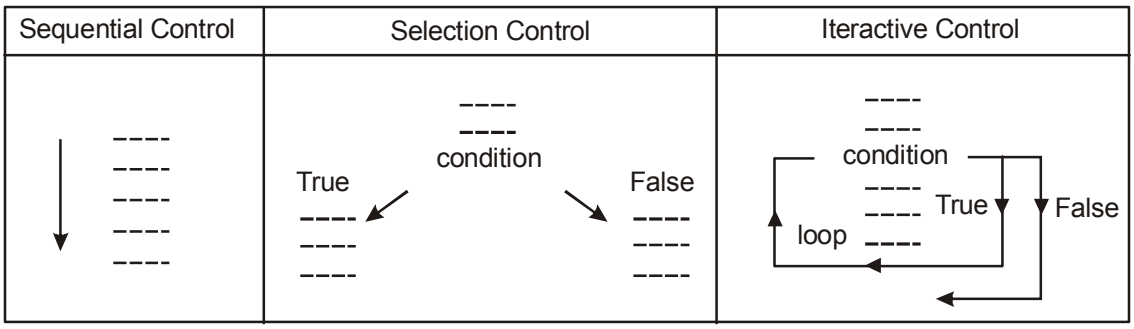


Fig 3.1 Flow of Control Statements

3.1.3.1 IF Statement

Conditional statement checks conditions and change the behavior of the program accordingly.

The simplest form is the if statement:

```
>>>if x > 0:
        print('x is positive')
```

The boolean expression after —if is called the condition. If it is true, the indented statement runs. If not, nothing happens.

The syntax of 'if' statement:

```
if <test_expression>:
    <body>
```

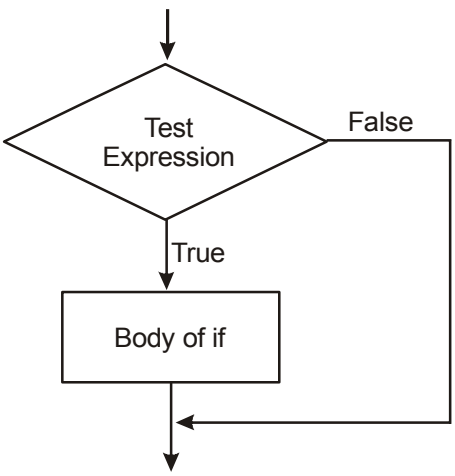


Fig: 3.2 – Operation of IF statement

Compound statements

Syntax for Compound statements

```
Header
Body of if
if condition:
    Statements
```

Statements like this are called compound statements. At least one statement must be there inside the statement and there is no limit for the number of statements. Occasionally, it is useful to have a body with no statements. In that case, we can use the pass statement, which does nothing.

Example:

```
if x < 0:
    pass
if grade>=70:
    print ( First class‘)
```

3.1.3.2 if-else Statement (Alternative)

The **if..else** statement evaluates test expression and will execute body of **if** only when test condition is **True**. If the condition is **False**, body of **else** is executed. Indentation is used to separate the blocks.

The syntax of ‘if..else’ statement:

```
if test condition:
    Body of if
else:
Body of else
```

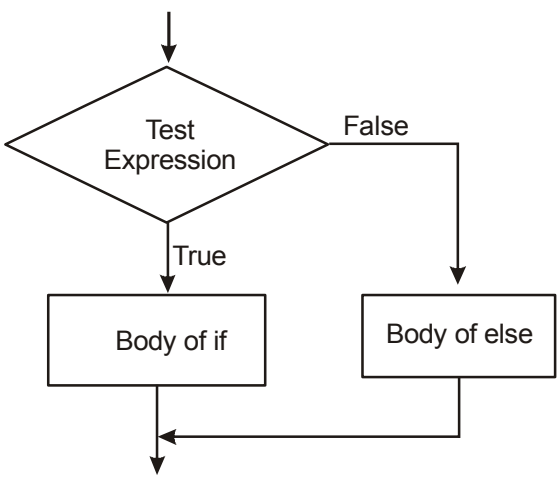


Fig 3.3 – Operation of IF..ELSE Statement

Example:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called branches, because they are branches in the flow of execution.

3.1.3.3 Chained Conditionals (IF-ELIF-ELSE Statement)

- Sometimes there are more than two possibilities and we need more than two branches.
- The elif is short for else if. It allows us to check for multiple expressions.
- If the condition for if is False, it checks the condition of the next elif block and so on.
- If all the conditions are False, body of else is executed.
- Only one block among the several if...elif...else blocks is executed according to the condition.
- The if block can have only one else block. But it can have multiple elif blocks.

The syntax of 'if..elif..else' statement:

```
if condition:
    Body of if
elif condition:
    Body of elif
else:
    Body of else
```

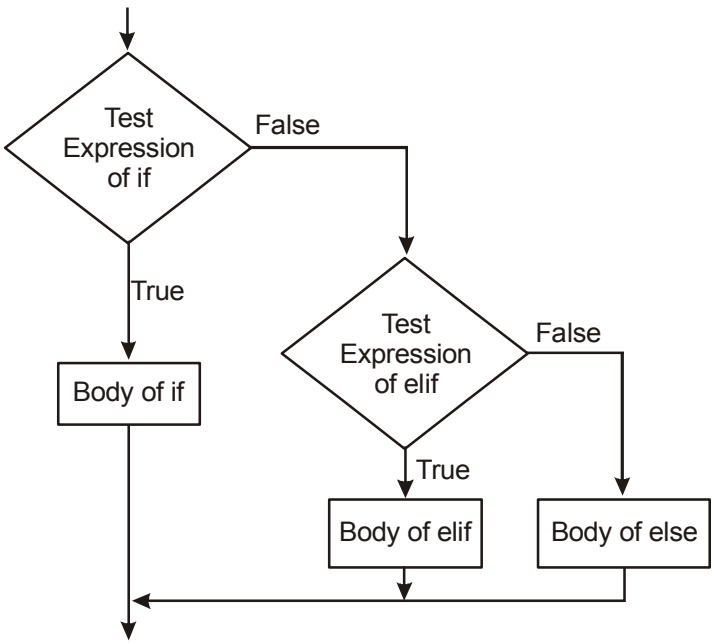


Fig 3.4 – Operation of if..elif..else statement

Example 1:

```
if x < y:
        print('x is less than y')
elif x > y:
        print('x is greater than y')
else:
        print('x and y are equal')
```

elif is an abbreviation of —else if . Again, exactly one branch will run. There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.

Example 2:

```
if choice == 'a':
        draw_a()
elif choice == 'b':
        draw_b()
elif choice == 'c':
        draw_c()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends.

3.2 ITERATION

Iteration is repeating a set of instructions and controlling their execution for a particular number of times. Iteration statements are also called as **loops.**

3.2.1 State

A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

The first time we display x, its value is 5; the second time, its value is 7. Python uses the equal sign (=) for assignment. First, equality is a symmetric relationship and assignment is not.

For example, in mathematics, if a = 7 then 7 = a. But in Python, the statement a = 7 is legal and 7 = a is not. Also, in mathematics, a proposition of equality is either true or false for all time. If a =b now, then a will always equal b. In Python, an assignment statement can make two variables equal.

```
>>>     a = 5
>>>     b = a # a and b are now equal
```

```
>>>  a = 3 # a and b are no longer equal
>>>  b
5
```

The third line changes the value of —a but does not change the value of —b , so they are no longer equal. A common kind of reassignment is an **update**, where the new value of the variable depends on the old.

```
>>> x = x + 1
```

This means —get the current value of x, add one, and then update x with the new value. If we try to update a variable that doesn‘t exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
>>> x = x + 1
```

NameError: name ‘x’ is not defined, Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>>x = 0
>>>x = x + 1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

3.2.2 While loop

- Repeats a statement or block of statements while a given condition is **TRUE.**
- Tests the condition before executing the body of loop.

Or

A **while statement** is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression (condition).

Syntax of While Loop:

```
while condition:
    statements
```

Example:

```
while n > 0:
    print(n)
    n = n – 1
```

The above while statement prints the value of n a number of times until n is greater than zero.

The flow of execution for a while statement:

1. Determine whether the condition is true or false.
2. If false, exit the while statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1

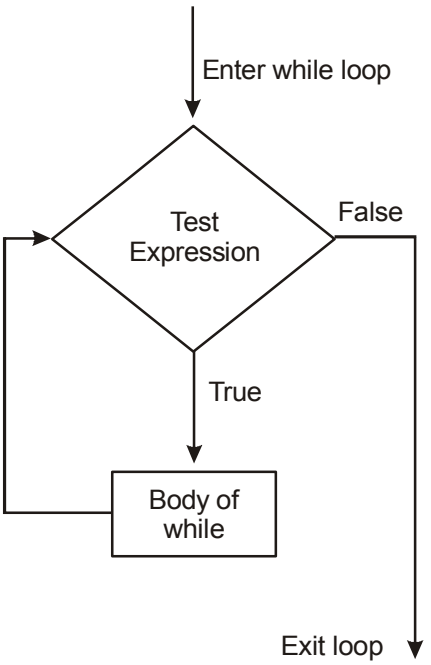


Fig 3.5 Operation of While Loop

The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. In the case of countdown, we can prove that the loop terminates: if n is zero or negative, the loop never runs. Otherwise, n gets smaller each time through the loop, so eventually we have to get to 0. For some other loops, it is not so easy to tell.

Example: Python while Loop

```
# Program to add natural numbers up to sum = 1+2+3+...+n
# To take input from the user
n = 10
# initialize sum and counter
sum = 0
 i = 1
 while i <= n:
        sum = sum + i
        i = i+1                          # update counter
print("The sum is", sum)                 # print the sum
```

Output:

The sum is 55

In the above program, the test condition will be True as long as our counter variable i is less than or equal to n. We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop). Finally the result is displayed.

While loop with else

We can have an optional else block with while loop as well. The else part is executed if the condition in the while loop evaluates to False. The while loop can be terminated with a break statement. In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

Example

```
 # Program to illustrate  the use of else statement with the while loop
counter = 0
while counter < 3:
        print("Inside loop")
        counter = counter + 1
 else:
        print("Inside else")
```

Output

Inside loop
Inside loop
Inside loop
Inside else

Here, we use a counter variable to print the string Inside loop three times. On the forth iteration, the condition in while becomes False. Hence, the else part is executed.

3.2.3 For Loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal. The for loops are used to construct definite loops.

Syntax of FOR LOOP:

for val in sequence:

    Body of for

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

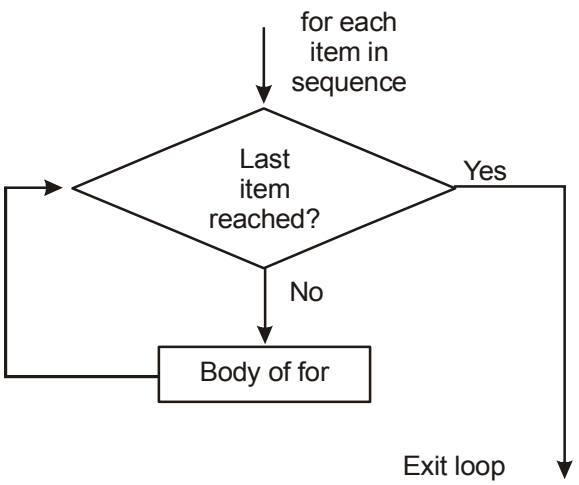


Fig: operation of for loop

Fig 3.6. Operation of FOR LOOP

Example 1:

```
for k in [4, 2 ,3, 1]:
    print (k)
```

Output:

```
4
2
3
1
```

Example 2:

```
for k in ['Apple', 'Banana', 'Pear']:
    print(k)
```

Output:

```
Apple
Banana
Pear
```

The range() function:

- We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).
- We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.
- This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.
- To force this function to output all the items, we can use the function list().

Example:

```
sum = 0
for k in range(1, 11):
    sum = sum + k
```

The values in the generated sequence include the starting value, up to *but not including* the ending value. For example, range(1, 11) generates the sequence [1, 2, 3, 4, 5, 6, 7, 8,9, 10].

The range function is convenient when long sequences of integers are needed. Actually, range does not create a sequence of integers. It creates a *generator function* able to produce each next item of the sequence when needed.

```
for i in range(4):
    print('Hello!')
```

for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts. break statement can be used to stop a for loop. In such case, the else part is ignored. Hence, a for loop's else part runs if no break occurs.

Example:

```
digits = [0, 1, 5]
for i in digits:
        print(i)
 else:
        print("No items left.")
```

Output:

```
0
1
5
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints No items left.

3.2.4 Break

Break statement is used to break the loop. For example, suppose you want to take input from the user until they type done.

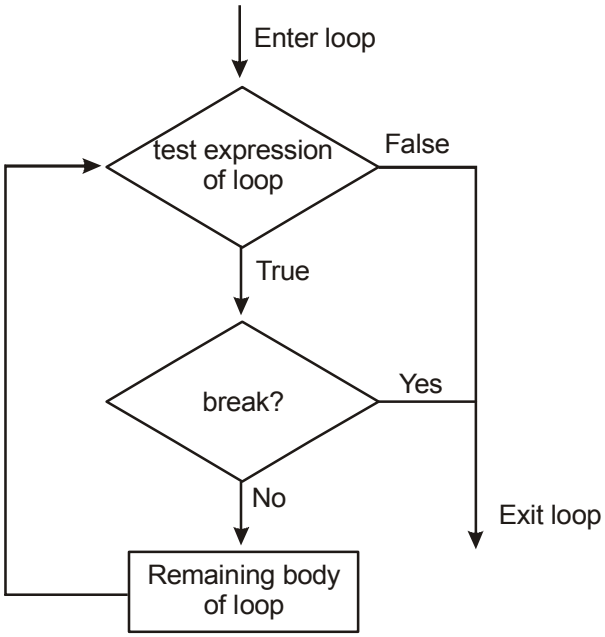


Fig 3.7 Flowchart of Break

You could write:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
    print('Done!')
```

The loop condition is True, which is always true, so the loop runs until it hits the break statement. Each time through, it prompts the user with an angle bracket. If the user types done, the break statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop.

Here's a sample run:

```
> not done
 not done
> done
Done!
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively (—stop when this happens) rather than negatively (—keep going until that happens).

3.2.5 Continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Example: Python continue

 # Program to show the use of continue statement inside loops

```
for val in “string”:
    if val == “i”:
        continue
    print(val)
print(“The end”)
```

Output

```
s
t
r
n
g
The end
```

This program is same as the above example except the break statement has been replaced with continue. We continue with the loop, if the string is “i”, not executing the rest of the block. Hence, we see in our output that all the letters except “i” gets printed.

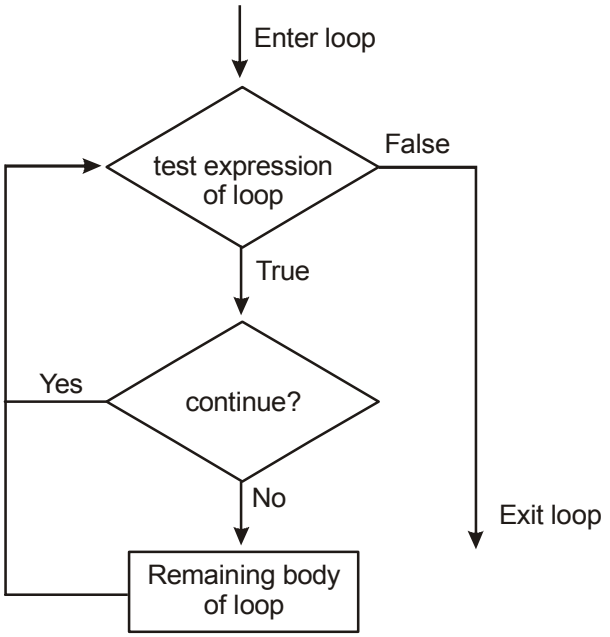


Fig 3.8 Flowchart of Continue

3.2.6 Pass

It is used when a statement is required syntactically but you do not want any command or code to execute. The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet.

We generally use it as a placeholder.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

Example:

```
for letter in 'Python':
    if letter == 'h':
        pass
        print 'This is pass block'
    print 'Current Letter :', letter
print "Good bye!"
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Difference between various iterations *Pass Continue Break*

Pass

- Statement simply means 'do nothing'

  When the python interpreter encounters the pass statement, it simply continues with its execution

Continue

- Continue with the loop

  resume execution at the top of the loop or goes to next iteration

Break

- Breaks the loop

  When a break statement is encountered, it terminates the block and gets the control out of the loop

While

- Indefinite Loops

  The exit condition will be evaluated again, and execution resumes from the top.

For

- Definite Loop

  The item being iterated over will move to its next element.

3.3 FRUITFUL FUNCTIONS

3.3.1 Return Value

The functions with return values are called as fruitful functions. The built-in functions we have used, such as abs, pow, and max, have produced results. Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.

```
biggest = max(3, 7, 2, 5)
x = abs(3 - 11) + 10
```

But so far, none of the functions we have written has returned a value.

The first example is area, which returns the area of a circle with the given radius:

```
def area(radius):
temp = 3.14159 * radius**2
return temp
```

We have seen the return statement before, but in a fruitful function the return statement includes a **return value**. This statement means: Return immediately from this function and use the following expression as a return value. The expression provided can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):
    return 3.14159 * radius**2
```

On the other hand, **temporary variables** like temp often make debugging easier. Sometimes it is useful to have multiple return statements, one in each branch of a conditional. We have already seen the built-in abs, now we see how to write our own:

```
def absolute_value(x):
    if x < 0:
            return -x
    else:
            return x
```

Since these return statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any

subsequent statements. Another way to write the above function is to leave out the else and just follow the if condition by the second return statement.

```
def absolute_value(x):
    if x < 0:
        return -x
    return x
```

Think about this version and convince yourself it works the same as the first one. Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. The following version of absolute value fails to do this:

```
def absolute_value(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

This version is not correct because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. In this case, the return value is a special value called **None**:

```
>>> print
absolute_value(0)
None
```

None is the unique value of a type called the NoneType:

```
>>> type(None)
```

All Python functions return None whenever they do not return another value. The scope of an identifier is the region of program code in which the identifier can be accessed, or used.

3.3.2 Scope

There are three important scopes in Python:

- Local scope refers to identifiers declared within a function. These identifiers are kept in the namespace that belongs to the function, and each function has its own namespace.
- Global scope refers to all the identifiers declared within the current module, or file.
- Built-in scope refers to all the identifiers built into Python—those like range and min that can be used without having to import anything, and are (almost) always available.

Python (like most other computer languages) uses precedence rules: the same name could occur in more than one of these scopes, but the innermost, or local scope, will always take precedence over the global scope, and the global scope always gets used in preference to the built-in scope.

Let‘s start with a simple example:

```
def range(n):
    return 123*n
    print(range(10))
```

Using the scope lookup rules determines this: our own range function, not the built-in one, is called, because our function range is in the global namespace, which takes precedence over the built in names. So although names likes range and min are built-in, they can be ―hidden from your use if you choose to define your own variables or functions that reuse those names.

```
n = 10
2 m = 3
def f(n):
    m = 7
    return 2*n+m
print(f(5), n, m)
```

This prints 17 10 3. The reason is that the two variables m and n in lines 1 and 2 are outside the function in the global namespace. Inside the function, new variables called n and m are created just for the duration of the execution of f. These are created

in the local namespace of function f. Within the body of f, the scope lookup rules determine that we use the local variables m and n. By contrast, after we've returned from f, the n and m arguments to the print function refer to the original variables on lines 1 and 2, and these have not been changed in any way by executing function f. Notice too that the def puts name f into the global namespace here. So it can be called on line 7. What is the scope of the variable n on line 1? Its scope—the region in which it is visible—is lines 1, 2, 6, 7. It is hidden from view in lines 3, 4, 5 because of the local variable n.

3.3.3 Composition

You can call one function from within another. This ability is called composition.

Example:

Write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle. Assume that the center point is stored in the variables xc and yc, and the perimeter point is in xp and yp. The first step is to find the radius of the circle, which is the distance between the two points.

```
radius = distance(xc, yc, xp, yp)
```

The second step is to find the area of a circle with that radius and return it. Again we will useone of our earlier functions:

```
result = area(radius)
return result
```

#Wrapping that up in a function, we get:

```
def area2(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

We called this function area2 to distinguish it from the area function defined earlier. The temporary variables radius and result are useful for development, debugging, and single-stepping through the code to inspect what is happening, but once the program is working, we can make it more concise by composing the function calls:

```
def area2(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

3.3.4 Recursion

Recursion is the process of calling the function that is currently executing. It is legal for one function to call another; it is also legal for a function to call itself. An example of recursive function to find the factorial of an integer.

Example:

0! = 1

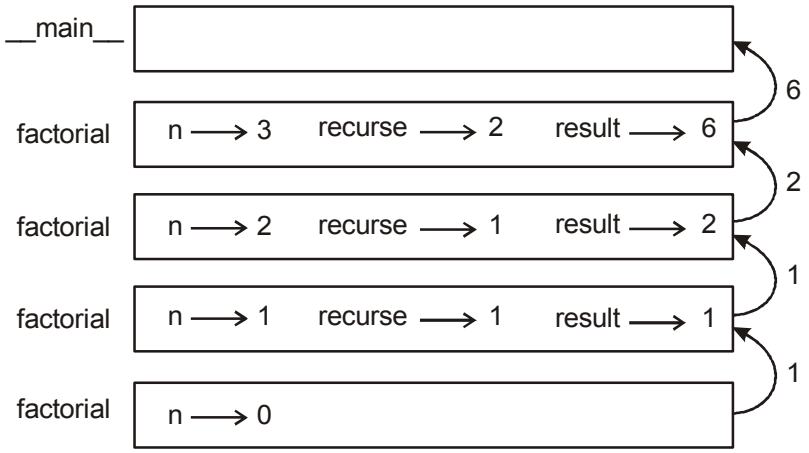
n! = n(n - 1)!

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of n - 1.

So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, 3! equals 3 times 2 times 1 times 1, which is 6.

The flow of execution for this program is similar to the flow of countdown. If we call factorial with the value 3:

- Since 3 is not 0, we take the second branch and calculate the factorial of n-1...
- Since 2 is not 0, we take the second branch and calculate the factorial of n-1...
- Since 1 is not 0, we take the second branch and calculate the factorial of n-1...
- Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.
- The return value, 1, is multiplied by n, which is 1, and the result is returned.
- The return value, 1, is multiplied by n, which is 2, and the result is returned.
- The return value (2) is multiplied by n, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.
- The return values are shown being passed back up the stack. In each frame, the return value is the value of result, which is the product of n and recurse. In the last frame, the local variables recurse and result do not exist, because the branch that creates them does not run.



Example:

```
# An example of a recursive function to find the factorial of a number
def factorial(x):
"""This is a recursive functionto find the factorial of an integer"""
if x == 1:
    return 1
else:
    return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

Output:

```
The factorial of 3 is 6
```

```
# An example of to find the factorial of a number without using Recursion function
  n=int(input("Enter number:"))
  fact=1
  while(n>0):
      fact=fact*n
      n=n-1
  print("Factorial of the number is: ")
  print(fact)
```

The Advantages of recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

The Disadvantages of recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

3.4 STRINGS

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
letter = fruit[1]
```

The second statement selects character number 1 from fruit and assigns it to letter. The expression in brackets is called an index. The index indicates which character in the sequence you want (hence the name).But you might not get what you expect:

```
>>> letter
'a'
```

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> letter
'b'
```

So b is the 0th letter (—zero-eth) of 'banana', a is the 1th letter (—one-eth), and n is the 2th letter (—two-eth). As an index you can use an expression that contains variables and operators:

```
>>>  i = 1
>>>  fruit[i]
 'a'
```

But the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
```

TypeError: string indices must be integers

3.4.1 String slices

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

The operator [n:m] returns the part of the string from the —n-eth character to the —m-eth character, including the first but excluding the last. This behavior is counter intuitive, butit might help to imagine the indices pointing between the characters. If you omit the first index (before the colon), the slice starts at the beginning of the string.

If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

3.4.2 Immutability

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.

For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
```

TypeError: 'str' object does not support item assignment

The —object in this case is the string and the —item is the character you tried to assign. The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

Length

The len function, when applied to a string, returns the number of characters in a string:

```
>>>  fruit = "banana"
>>>  len(fruit)
6
```

The index starts counting form zero. In the above string, the index value in from 0 to 5.

To get the last character, we have to subtract 1 from the length of fruit:

```
size = len(fruit)

last = fruit[size-1]
```

Alternatively, we can use negative indices, which count backward from the end of the string.

The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

3.4.3 String methods

Strings provide methods that perform a variety of useful operations. A method is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method upper takes a string and returns a new string with all uppercase letters.

Upper method:

Instead of the function syntax upper(word), it uses the method syntax word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no arguments. A method call is called an **invocation**; in this case, we would say that we are invoking upper on word.

Find Method:

The find method can find character and sub strings in a string.

```
>>>  word = 'banana'
>>>  index = word.find('a')
>>>  index
1
>>>  word.find('na')
2
```

By default, find starts at the beginning of the string, but it can take a second argument, the index where it should start:

```
>>> word.find('na', 3)
4
```

This is an example of an **optional argument**; find can also take a third argument, the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because b does not appear in the index range from 1 to 2, not including 2.

Split Method:

One of the most useful methods on strings is the split method: it splits a single multi-word string into a list of individual words, removing all the whitespace between them. The input string can be read as a single string, and will be split.

```
>>> ss = "Well I never did said Alice"
>>> wds = ss.split()
>>> wds
['Well', 'I', 'never', 'did', 'said', 'Alice']
```

String format method

The easiest and most powerful way to format a string in Python 3 is to use the format method.

Program:

```
s1 = "His name is {0}!".format("Arthur")
print(s1)
name = "Alice"
age = 10
s2 = "I am {1} and I am {0} years old.".format(age, name)
```

Output:

His name is Arthur!

I am Alice and I am 10 years old.

The template string contains place holders, ... {0} ... {1} ... {2} ... etc. The format method substitutes its arguments into the place holders. The numbers in the place holders are indexes that determine which argument gets substituted. Each of the replacement fields can also contain a format specification — it is always introduced by the : symbol . This modifies how the substitutions are made into the template, and can control things like: whether the field is aligned to the left <, center ^, or right > the width allocated to the field within the result string (a number like 10) the type of conversion if the type conversion is a float, you can also specify how many decimal places are wanted.

print("Pi to three decimal places is {0:.3f}".format(3.1415926))

3.4.4 Looping and counting

The following program counts the number of times the letter a appears in a string:

Example:

```
word = 'banana'
count = 0
for letter in word:
if letter == 'a':
count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a **counter**. The variable count is initialized to 0 and then incremented each time an **a** is found. When the loop exits, count contains the result—the total number of a's.

3.4.5 String module

The **Python Standard Library** is a collection of *built-in modules* , each providing specific functionality beyond what is included in the —core part of Python.

This string module contains a number of functions to process standard Python strings.

Example: Using the string module

```
import string
text = "Monty Python's Flying Circus"
print "upper", "=>", string.upper(text)
print "lower", "=>", string.lower(text)
print "split", "=>", string.split(text)
print "join", "=>", string.join(string.split(text), "+")
print "replace", "=>", string.replace(text, "Python", "Java")
print "find", "=>", string.find(text, "Python"), string.find(text, "Java")
print "count", "=>", string.count(text, "n")
upper => MONTY PYTHON'S FLYING CIRCUS
 lower => monty python's flying circus
split => ['Monty', "Python's", 'Flying', 'Circus']
join => Monty+Python's+Flying+Circus replace => Monty Java's Flying Circus
find => 6 -1
count => 3
```

String Example Program:

1. Write a python program to count the number of vowels in a string

Program:

```
string=raw_input("Enter string:")
vowels=0
for i in string:
    if(i=='a' or i=='e' or i=='i' or i=='o' or i=='u' or i=='A' or i=='E' or i=='I' or i=='O' or i=='U'):
        vowels=vowels+1
print("Number of vowels are:")
print(vowels)
```

Output 1:

```
Enter string:Hello world
```

Number of vowels are:

3

Output 2:

Enter string:WELCOME

Number of vowels are:

3

2. Write a Python program to count the occurrences of each word in a given sentence

Program:

```
def word_count(str):
    counts = dict()
    words = str.split()
    for word in words:
        if word in counts:
            counts[word] += 1
        else:
            counts[word] = 1
    return counts
print( word_count('the quick brown fox jumps over the lazy dog.'))
```

Output:

{'the': 2, 'jumps': 1, 'brown': 1, 'lazy': 1, 'fox': 1, 'over': 1, 'quick': 1, 'dog.': 1}

3.5 LISTS AS ARRAYS

- Arrays and lists are both used in Python to store data.
- NumPy is used to create an array in python.
- It is a module which contains array.
- They are used to store any type of data and can be indexed.
- The functions which can be performed in list and array are distinct.

Example 1:

```
x = array([3, 6, 9, 12])
x/3.0x
print(x)
```

Output:

```
array([1, 2, 3, 4])
```

Example 2:

```
y = [3, 6, 9, 12]
y/3.0
print(y)
```

The program will result in the error.Arithmetic calculations – can be done in array.

Storing data efficiently – Array

3.6 ILLUSTRATIVE PROGRAMS

1. Compute the GCD of two numbers

Program:

```
def gcd(m,n):
    if m<n:
        (m,n)=(n,m)
while (m%n)==0:
    return (m)
else:
return(gcd(n,m%n))
a=150
b=50
g=gcd(a,b)
print(g)
```

Output:

```
50
```

2. Compute the exponent of a number.

Program:

```
def exp(m,n):
    return m**n
    i=input("Enter the base:")
    j=input("Enter the pow:")
    d=exp(i,j)
    print d
```

Output:

```
Enter the base: 40
Enter the pow: 2
1600
```

3. Linear Search in Python Programming

Program:

```
list = [4, 2, 8, 9, 3, 7]
    x= int(input("Enter number to search: )) found = False
    for i in range(len(list)):
        if(list[i] == x):
                found = True
                print("%d found at %dth position"%(x,i))
                break
    if(found == False):
                print("%d is not in list"%x)
```

Output:

```
Enter number to search: 2
```

4. Binary Search using Python Programming

Program:

```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False
    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
                found = True
        else:
                if item < alist[midpoint]:
                        last = midpoint-1
                else:
                first = midpoint+1
        return found
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

Output:

```
False
True
```

5. Sum an array of N numbers

Program

```
x=[1,2,3,4,5]
sum=0
for s in range(0,len(x)):
    sum=sum+x[s]
    print(sum)
```

Output:

```
15
```

Suggested Links to Refer

1. https://www.youtube.com/watch?v=-nJvt_7l-sk – if..else
2. https://www.youtube.com/watch?v=1QfPTboc1Pw – elif and nested if
3. https://www.youtube.com/watch?v=UZRD8rmOC7M – if
4. https://www.youtube.com/watch?v=9LgyKiq_hU0 - For loop
5. https://www.youtube.com/watch?v=D0Nb2Fs3Q8c – While loop
6. https://www.youtube.com/watch?v=aUXi4L8eUmw – Break, Continue, Pass

NPTEL – You Tube Link:

1. https://www.youtube.com/watch?v=9MmC_uGjBsM&feature=youtu.be – GCD
2. https://www.youtube.com/watch?v=I7zOYKF4AHc&feature=youtu.be – String
3. https://www.youtube.com/watch?v=oe6iF3yzMo8&feature=youtu.be – Control Flow

ASSIGNMENT:

Write python program for the following:

1. Display all even numbers between 50 and 80 (both inclusive) using "for" loop.
2. Add natural numbers up to n where n is taken as an input from user. Print the sum.
3. Prompt the user to enter a number. Print whether the number is prime or not.
4. Print Fibonacci series till nth term where n is taken as an input from user.

   Hint – Fibonacci series is a series of numbers in which each number is the sum of the two preceding numbers. Series start from 1 and goes like : 1, 1, 2, 3, 5, 8, 13 ….
5. Find the first N prime Numbers.
6. Program that reads a positive integer and then prints out all the positive divisors of that integer.

   Hint – The positive divisor of positive integer 36 are 36,18,12,9,6,4,3,2 and 1.

7. Program that reads a character and prints out whether or not it is a vowel or consonant.
8. Program to print the first 'n' numbers divisible by 7

PART A QUESTION AND ANSWERS

1. What are the two Boolean values in python

A Boolean value is either True or False.

Python type – bool.

2. What is an operator?

Operator is used to perform operations between operands.

Eg: c+d

+ - operator

C,d – operands

3. Specify the three fundamental forms of control statement

The three forms of control statements are:

- Sequential control
- Iterative control
- Selective control

4. Write the difference between if and if else statement?

If statement – Its used to check a condition and execute the statements.

```
if(c==5):
print("five")
```

If else is —alternative execution , in which there are two possibilities and the condition determines which one runs.

```
if(c==5):
print("five")
else:
print("not five")
```

5. What is iteration?

Iteration is repeating a set of instructions and controlling their execution for a particular number of times. Iteration statements are also called as loops.

6. Specify the use of range function in for loop.

The range function is generator and able to produce next item of the sequence when needed.

It is used in a for loop.

Eg: for val in values: print(val)

7. Differentiate break and continue statement with example

The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

Break statement is used to break the loop.

Eg:

```
for number in range(1,5):
if(number==3):
continue
print(number)
```

Output:

```
1
2
4
for number in range(1,5):
if(number==3):
break
print(number)
```

Output:

```
1
2
3
```

8. What is the use of pass statement in python?

It is used when a statement is required syntactically but you do not want any command or code to execute. The **pass** statement is a *null* operation; nothing happens when it executes.

Eg:

```
for letter in 'Python':
if letter == 'h':
pass
```

9. Describe about dead code.

Code that ppears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.

```
def absolute_value(x):
if x < 0:
return -x
return x
```

10. What refers to local and global scope?

Local scope refers to identifiers declared within a function. These identifiers are kept in the namespace that belongs to the function, and each function has its own namespace.

Global scope refers to all the identifiers declared within the current module, or file.

Built-in scope refers to all the identifiers built into Python

11. Which function is called composition?

A function calls one function with another is called composition.

Eg:

```
def val():
result = area(radius)
return result
```

12. What is a recursion? How it differs from loop?

The function which calls itself again and again is called recursion.

Eg:

```
def factorial(n):
if n == 0:
return 1
else:
recurse = factorial(n-1)
result = n * recurse
return result
```

13. What is a srting? Mention that how do we represent a string in python?

A string is a sequence, which means it is an ordered collection of other values.

Strings can be represented in python using single quotes, double quotes or triple double quotes.

Eg: >>> fruit = ‘orange’

14. How does indexing differ from slice operation?

Indexing is assigning the address of every character in string.

Eg: s=”hello”

S[0]=h, s[1]=e, s[2]=l, s[3]=l,s[4]=o

A segment of a string is called a slice. Selecting a slice is similar to selecting a character.

S=”hello”

S[2:]=”llo”

15. What is string immutability?

We cant exchange the characters of the string. This is called as immutable.

>>> greeting = ‘Hello, world!’

>>> greeting[0] = ‘J’

TypeError: 'str' object does not support item assignment

16. List out the methods in string type.

The methods are:

Upper, find, split, string format, etc..

17. Write a python program for exponent of a number

```
def exp(m,n):
return m**n
i=input("Enter the base:")
j=input("Enter the pow:")
d=exp(i,j)
print d
```

18. Define fruitful functions.

Functions which returns a value is called as fruitful function.

```
def sum(c,d):
e=c+d
return (e)
print(sum(2,3))
```

19. Define python array.

Arrays and lists are both used in Python to store data.

NumPy is used to create an array in python.

It is a module which contains array.

They are used to store any type of data and can be indexed.

PART B QUESTIONS

1. Discuss about the iteration statements(state, while, for, break, continue, pass)
2. Discuss about the conditional statements(if, if-else, if-elif)
3. Explain about the function composition and recursion with eg.
4. Discuss about the fruitful functions with eg.
5. Explain about the string operations with programs.
6. Write a program to
   a. Find the square root of a number
   b. Calculate gcd of two numbers
   c. Find the exponent of a number
   d. Find the sum of array of numbers
7. Write a program to
   a. Perform binary search
   b. Linear search

Unit IV

LISTS, TUPLES, DICTIONARIES

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters; Tuples: tuple assignment, tuple as return value; Dictionaries: operations and methods; advanced list processing - list comprehension; Illustrative programs: selection sort, insertion sort, merge sort, histogram.

4.1 LISTS

List is an ordered sequence of items. Values in the list are called elements / items. It can be written as a list of comma-separated items (values) between square brackets [ ]. Items in the lists can be of different data types.

Example:

ps = [10, 20, 30, 40]

qs = [“spam”, “bungee”, “swallow”]

The first example is a list of four integers. The second is a list of three strings. The elements of a list don t have to be the same type. The following list contains a string, a float, an integer, and another list: zs = [“hello”, 2.0, 5, [10, 20]]

A list within another list is said to be nested. Finally, a list with no elements is called an empty list, and is denoted [].

A list of Integers : [1951,1952,1953,1958,1957]

A list of Strings : [„orange , blue , yellow]

An empty list : []

4.1.1 List Operations:

Operations	Examples	Description

create a list	>>> a=[2,3,4,5,6,7,8,9,10]
	>>> print(a)
	[2, 3, 4, 5, 6, 7, 8, 9, 10]
		in this way we can create alist at compile time

Indexing	>>> print(a[0])
	2
	>>> print(a[8])
	10
	>>> print(a[-1])
	10
		Accessing the item in the position 0
		Accessing the item in the position 8
		Accessing a last elementusing negative indexing.

Slicing	>>> print(a[0:3])
	[2, 3, 4]
	>>> print(a[0:])
	[2, 3, 4, 5, 6, 7, 8, 9, 10]
		Printing a part of the list.

Concatenation	>>>b=[20,30] >>> print(a+b) [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30]	Adding and printing theitems of two lists.q
Repetition	>>> print(b*3) [20, 30, 20, 30, 20, 30]	Create a multiple copies ofthe same list.

Updating	>>> print(a[2])
	4>>> a[2]=100
	>>> print(a)
	[2, 3, 100, 5, 6, 7, 8, 9, 10]
		Updating the list using index value.

Membership	>>> a=[2,3,4,5,6,7,8,9,10]
	>>> 5 in a
	True
	>>> 100 in a
	False
	>>> 2 not in a
	False
		Returns True if element ispresent in list. Otherwisereturns false.

Comparison	>>> a=[2,3,4,5,6,7,8,9,10]
	>>>b=[2,3,4]
	>>> a==b
	False
	>>> a!=b
	True
		Returns True if all elementsin both elements are same.Otherwise returns false

4.1.2 List Slicing

The slice operation uses the square brackets. This is termed indexing, or slicing. An index or slice returns a portion of a larger value. In a string this can be used to produce a substring. Index values start at zero, and extend upwards to the number of characters in the string minus one. When a single argument is given it is one character out of the string. When two integers are written, separated by a colon, it is termed a slice. The second value is an ending position. A portion of the string starting at the given position up to but not including the ending position is produced.

Syntax:

Listname[start:stop]

Listname[start:stop:steps]

Example :

Slices	Example	Description

a[0:3]	>>> a=[9,8,7,6,5,4]
	>>> a[0:3]
	[9, 8, 7]
		Printing a part of a list from 0 to 2.

a[:4]	>>> a[:4] [9, 8, 7, 6]	Default start value is 0. soprints from 0 to 3
a[1:]	>>> a[1:] [8, 7, 6, 5, 4]	default stop value will ben-1. so prints from 1 to 5
a[:]	>>> a[:] [9, 8, 7, 6, 5, 4]	Prints the entire list.
a[2:2]	>>> a[2:2] [ ]	print an empty slice
a[0:6:2]	>>> a[0:6:2] [9, 7, 5]	Slicing list values with stepsize 2.
a[::-1]	>>> a[::-1] [4, 5, 6, 7, 8, 9]	Returns reverse of given listvalues

4.1.3 List Methods

Methods used in lists are used to manipulate the data quickly. These methods work only on lists. They do not work on the other sequence types that are not mutable, that is, the values they contain cannot be changed, added, or deleted.

Syntax:

List_name.method_name(element/index/list)

syntax	example	description

a.append(element)	>>> a=[1,2,3,4,5]
	>>> a.append(6)
	>>> print(a)
	[1, 2, 3, 4, 5, 6]
		Add an element to the end of the list

a.insert(index,element)	>>> a.insert(0,0)
	>>> print(a)
	[0, 1, 2, 3, 4, 5, 6]
		Insert an item at the defined index

a.extend(b)	>>> b=[7,8,9]
	>>> a.extend(b)
	>>> print(a)
	[0, 1, 2, 3, 4, 5, 6, 7, 8,9]
		Add all elements of a list to the another list

a.index(element)	>>> a.index(8) 8	Returns the index of the first matched item

a.sort()	>>> a.sort()
	>>> print(a)
	[0, 1, 2, 3, 4, 5, 6, 7, 8]
		Sort items in a list in ascending order

a.reverse()	>>> a.reverse()
	>>> print(a)
	[8, 7, 6, 5, 4, 3, 2, 1, 0]
		Reverse the order of items in the list

a.pop()	>>> a.pop() 0	Removes and returns an element at the last element
a.pop(index)	>>> a.pop(0) 8	Remove the particular element and return it.

a.remove(element)	>>> a.remove(1)
	>>> print(a)
	[7, 6, 5, 4, 3, 2]
		Removes an item from the list

a.count(element)	>>> a.count(6) 1	Returns the count of number of items passed as an argument

a.copy()	>>> b=a.copy()
	>>> print(b)
	[7, 6, 5, 4, 3, 2]
		Returns a shallow copy of the list

len(list)	>>> len(a) 6	Return the length of the length
min(list)	>>> min(a) 2	Return the minimum element in a list
max(list)	>>> max(a) 7	Return the maximum element in a list.

a.clear()	>>> a.clear()
	>>> print(a)
	[ ]
		Removes all items from the list.

del(a)	>>> del(a)
	>>> print(a)
	Error: name 'a' is not defined
		Delete the entire list.

append():

Example 1:

```
animal = ['cat', 'dog', 'rabbit']
animal.append('goat')
print('Updated animal list: ', animal)
```

Output:

```
Updated animal list: ['cat', 'dog', 'rabbit', 'goat']
```

Example 2:

```
animal = [‘cat’, ‘dog’, ‘rabbit’]
wild_animal = [‘tiger’, ‘fox’]
animal.append(wild_animal)
print(‘Updated animal list: ‘, animal)
```

Output:

```
Updated animal list: [‘cat’, ‘dog’, ‘rabbit’, [‘tiger’, ‘fox’]]
```

insert():

Example:

```
vowel = [‘a’, ‘e’, ‘i’, ‘u’]
vowel.insert(3, ‘o’)
print(‘Updated List: ‘, vowel)
```

Output:

```
Updated List: [‘a’, ‘e’, ‘i’, ‘u’, ‘o’]
```

extend():

Example:

```
language = [‘French’, ‘English’, ‘German’]
language1 = [‘Spanish’, ‘Portuguese’]
language.extend(language1)
print(‘Language List: ‘, language)
```

Output:

```
Language List: [‘French’, ‘English’, ‘German’, ‘Spanish’, ‘Portuguese’]
```

Index():

Example 1:

```
vowels = [‘a’, ‘e’, ‘i’, ‘o’, ‘i’, ‘u’]
index = vowels.index(‘e’)
print(‘The index of e:’, index)
index = vowels.index(‘i’)
print(‘The index of i:’, index)
```

Output:

The index of e: 1

The index of e: 2

Example 2:

```
vowels = [‘a’, ‘e’, ‘i’, ‘o’, ‘u’]
index = vowels.index(‘p’)
print(‘The index of p:’, index)
```

Output:

ValueError: ‘p’ is not in list

remove():

Example:

```
animal = [‘cat’, ‘dog’, ‘rabbit’]
animal.remove(‘rabbit’)
print(‘Updated animal list: ‘, animal)
```

Output:

Updated animal list: [‘cat’, ‘dog’]

clear():

Example:

```
list = [{1, 2}, (‘a’), [‘1.1’, ‘2.2’]]
list.clear()
print(‘List:’, list)
Output: List: []
```

Sort():

Example 1:

```
vowels = [‘e’, ‘a’, ‘u’, ‘o’, ‘i’]
vowels.sort()
print(‘Sorted list:’, vowels)
```

Output:

Sorted list: [‘a’, ‘e’, ‘i’, ‘o’, ‘u’]

Example 2:

```
vowels = [‘e’, ‘a’, ‘u’, ‘o’, ‘i’]
vowels.sort(reverse=True)
print(‘Sorted list (in Descending):’, vowels)
```

Output:

Updated List: [‘a’, ‘e’, ‘i’, ‘u’, ‘o’]

reverse():

Example:

```
os = [‘Windows’, ‘macOS’, ‘Linux’]
print(‘Original List:’, os)
os.reverse()
print(‘Updated List:’, os)
```

Output:

Original List: [‘Windows’, ‘macOS’, ‘Linux’]
Updated List: [‘Linux’, ‘macOS’, ‘Windows’]

Example:

```
os = [‘Windows’, ‘macOS’, ‘Linux’]
print(‘Original List:’, os)
reversed_list = os[::-1]
print(‘Updated List:’, reversed_list)
```

Output:

Original List: [‘Windows’, ‘macOS’, ‘Linux’]
Updated List: [‘Linux’, ‘macOS’, ‘Windows’]

Pop():

Example:

```
language = [‘Python’, ‘Java’, ‘C++’, ‘French’, ‘C’]
```

```
return_value = language.pop(3)
print('Return Value: ', return_value)
print('Updated List: ', language)
```

Output:

```
Return Value: French
Updated List: ['Python', 'Java', 'C++', 'C']
```

Example:

```
language = ['Python', 'Java', 'C++', 'Ruby', 'C']
print('When index is not passed:')
print('Return Value: ', language.pop())
print('Updated List: ', language)
print('\nWhen -1 is passed:')
print('Return Value: ', language.pop(-1))
print('Updated List: ', language)
print('\nWhen -3 is passed:')
print('Return Value: ', language.pop(-3))
print('Updated List: ', language)
```

Output:

```
When index is not passed:
Return Value: C
Updated List: ['Python', 'Java', 'C++', 'Ruby']
When -1 is passed:
Return Value: Ruby
Updated List: ['Python', 'Java', 'C++']
When -3 is passed:
Return Value: Python
Updated List: ['Java', 'C++']
```

4.1.4 List loop

Python's for statement provides a convenient means of iterating over lists.

For loop:

A for statement is an iterative control statement that iterates once for each element in a specified sequence of elements. Thus, for loops are used to construct definite loops.

Syntax:

for val in sequence:

Example1:

```
a=[10,20,30,40,50]
for i in a:
print(i)
```

Output :

```
1
2
3
4
5
```

Example 2 :

```
a=[10,20,30,40,50]
for i in range(0,len(a),1):
print(i)
```

Output :

```
0
1
2
3
4
```

Example 3 :

```
a=[10,20,30,40,50]
for i in range(0,len(a),1):
print(a[i])
```

Output ;

```
10
20
30
40
50
```

While loop:

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Syntax:

```
while (condition):
    body of while
```

Example : Sum of Elements in a List

```
a=[1,2,3,4,5]
i=0
sum=0
while i<len(a):
    sum=sum+a[i]
    i=i+1
print(sum)
```

4.1.5 List Mutability

Lists are mutable. When the bracket operator appears on the left side of an assignment, it Identifies the element of the list that will be assigned.

Example:

Changing Single Element

```
>>> a=[1,2,3,4,5]
>>> a[0]=100
>>> print(a)
[100, 2, 3, 4, 5]
```

Changing multiple element

```
>>> a=[1,2,3,4,5]
>>> a[0:3]=[100,100,100]
>>> print(a)
[100, 100, 100, 4, 5]
```

The elements from a list can also be removed by assigning the empty list to them.

```
>>> a=[1,2,3,4,5]
>>> a[0:3]=[ ]
>>> print(a)
[4, 5]
```

The elements can be inserted into a list by squeezing them into an empty slice at the desired location.

```
>>> a=[1,2,3,4,5]
>>> a[0:0]=[20,30,45]
>>> print(a)
[20,30,45,1, 2, 3, 4, 5]
```

4.1.6 List Aliasing

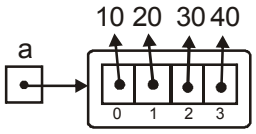
Creating a copy of a list is called aliasing. When you create a copy both list will be having same memory location. Changes in one list will affect another list. Aliasing refers to having different names for same list values.

Example :

```
a= [10,20,30,40]
b=a
```

```
print (b)
a is b
b[2]=35
print(a)
print(b)
```

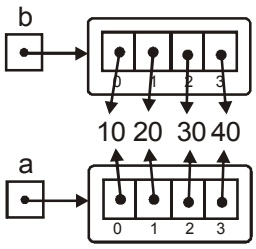
a = [10, 20, 30]



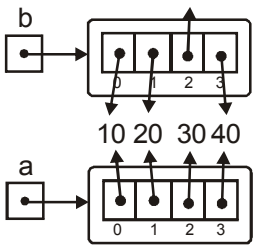
Output :

```
[10,20,30,40]
True
[10,20,30,40]
[10,20,35,40]
```

b = [10, 20, 30]



b [2] = 35



In this a single list object is created and modified using the subscript operator. When the first element of the list named "a" is replaced, the first element of the list named "b" is also replaced. This type of change is what is known as a side effect. This happens because after the assignment b=a, the variables a and b refer to the exact same list object.

They are aliases for the same object. This phenomenon is known as aliasing. To prevent aliasing, a new object can be created and the contents of the original can be copied which is called cloning.

4.1.7 List Cloning

To avoid the disadvantages of copying we are using cloning. Creating a copy of a same list of elements with two different memory locations is called cloning. Changes in one list will not affect locations of another list.

Cloning is a process of making a copy of the list without modifying the original list.

1. Slicing
2. list()method
3. copy() method

Cloning using Slicing

```
>>>a=[1,2,3,4,5]
>>>b=a[:]
>>>print(b)
[1,2,3,4,5]
>>>a is b
False
```

Cloning using List( ) method

```
>>>a=[1,2,3,4,5]
>>>b=list
>>>print(b)
[1,2,3,4,5]
>>>a is b
false
>>>a[0]=100
>>>print(a)
>>>a=[100,2,3,4,5]
>>>print(b)
>>>b=[1,2,3,4,5]
```

Cloning using copy() method

```
a=[1,2,3,4,5]
>>>b=a.copy()
>>>   print(b) [1, 2, 3, 4, 5]
>>>   a is b
False
```

4.1.8 List Parameters

In python, arguments are passed by reference. If any changes are done in the parameter which refers within the function, then the changes also reflects back in the calling function. When a list to a function is passed, the function gets a reference to the list.

Passing a list as an argument actually passes a reference to the list, not a copy of the list. Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing.

Example1 :

```
def remove(a):
  a.remove(1)
a=[1,2,3,4,5]
remove(a)
print(a)
```

Output:

```
[2,3,4,5]
```

Example 2:

```
def inside(a):
        for i in range(0,len(a),1):
              a[i]=a[i]+10
        print("inside",a)
a=[1,2,3,4,5]
inside(a)
print("outside",a)
```

Output

```
inside [11, 12, 13, 14, 15]
outside [11, 12, 13, 14, 15]
```

Example 3

```
def insert(a):
        a.insert(0,30)
```

```
a=[1,2,3,4,5]
insert(a)
print(a)
```

Output

```
[30, 1, 2, 3, 4, 5]
```

Suggested links to Refer :

List :

https://www.tutorialspoint.com/videotutorials/video_course_view.php?course=python_online_training&chapter=python_basic_list_operation

List Using for Loop:

https://www.youtube.com/watch?v=YT6ldZuBW4Y

Python List functions:

https://www.youtube.com/watch?v=96Wr1OO-4d8

Python Slicing:

https://www.youtube.com/watch?v=iD6a0G8MnjA

4.2 TUPLES

A tuple is an immutable linear data structure. Thus, in contrast to lists, once a tuple is defined, it cannot be altered. Otherwise, tuples and lists are essentially the same. To distinguish tuples from lists, tuples are denoted by parentheses instead of square brackets.

Benefit of Tuple:

- Tuples are faster than lists.
- If the user wants to protect the data from accidental changes, tuple can be used.
- Tuples can be used as keys in dictionaries, while lists can't.

An **empty tuple** is represented by a set of empty parentheses, ().The elements of tuples are accessed the same as lists, with square brackets, Any attempt to alter a tuple is invalid. Thus, delete, update, insert, and append operations are not defined on tuples.

4.2.1 Operations on Tuples

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Membership
6. Comparison

Creating a tuple

```
>>>a=(20,40,60,”apple”,”ball”)
```

Indexing

```
>>>print(a[0])
    20
>>> a[2]
    60
```

Slicing

```
>>>print(a[1:3])
(40,60)
```

Concatenation

```
>>> b=(2,4)
>>>print(a+b)
>>>(20,40,60,”apple”,”ball”,2,4)
```

Repetition

```
>>>print(b*2)
>>>(2,4,2,4)
```

Membership

```
>>> a=(2,3,4,5,6,7,8,9,10)
>>> 5 in a
True
```

```
>>> 100 in a
False
>>> 2 not in a
False
```

Comparison

```
>>> a=(2,3,4,5,6,7,8,9,10)
>>>b=(2,3,4)
>>> a==b
    False
>>> a!=b
    True
```

4.2.2 Tuple methods

Tuple is immutable so changes cannot be done on the elements of a tuple once it is assigned.

a.index(tuple)

```
>>> a=(1,2,3,4,5)
>>> a.index(5)
    4
```

a.count(tuple)

```
>>>a=(1,2,3,4,5)
>>> a.count(3)
    1
```

len(tuple)

```
>>> len(a)
5
```

min(tuple)

```
>>> min(a)
1
```

max(tuple)

```
>>> max(a)
5
```

del(tuple)

```
>>> del(a)
```

4.2.3 Tuple Assignment

Tuple assignment allows variables on the left of an assignment operator and values of tuple on the right of the assignment operator.

Multiple assignment works by creating a tuple of expressions from the right hand side, and a tuple of targets from the left, and then matching each expression to a target.Because multiple assignments use tuples to work, it is often termed tuple assignment.

Uses of Tuple assignment:

It is often useful to swap the values of two variables.

Example:

Swapping using temporary variable:

```
a=20
b=50
temp = a
a = b
b = temp
print("value after swapping is",a,b)
```

Swapping using tuple assignment:

```
a=20
b=50
(a,b)=(b,a)
print("value after swapping is",a,b)
```

Multiple assignments:

Multiple values can be assigned to multiple variables using tuple assignment.

```
>>>(a,b,c)=(1,2,3)
>>>print(a)
  1
>>>print(b)
  2
>>>print(c)
  3
```

4.2.4 Tuple as return value

A Tuple is a comma separated sequence of items. It is created with or without ( ). A function can return one value. if you want to return more than one value from a function. We can use tuple as return value.

Example :

Example1:	**Output:**

def div(a,b):
  r=a%b
  q=a//b
  return(r,q)
a=eval(input("enter a value:"))
b=eval(input("enter b value:"))
r,q=div(a,b)
print("reminder:",r)
print("quotient:",q)
	enter a value:4
	enter b value:3
	reminder: 1
	quotient: 1

Example2:	**Output:**

def min_max(a):
  small=min(a)
  big=max(a)
  return(small,big)
a=[1,2,3,4,6]
small,big=min_max(a)
print("smallest:",small)
print("biggest:",big)
	smallest: 1
	biggest: 6

4.2.5 Tuple as argument

The parameter name that begins with * gathers argument into a tuple.

Example:

```
def printall(*args):
print(args)
printall(2,3,'a')
```

Output :

(2, 3, 'a')

Suggested Links to Refer

Tuples : https://www.youtube.com/watch?v=R8mOaSIHT8U
Tuple as Paramerter : https://www.youtube.com/watch?v=sgnP62EXUtA
Tuple assignment : https://www.youtube.com/watch?v=AhSW1sEOzWY

4.3 DICTIONARIES

A dictionary organizes information by **association**, not position.

Example: when you use a dictionary to look up the definition of "mammal," you don t start at page 1; instead, you turn directly to the words beginning with "M." Phone books, address books, encyclopedias, and other reference sources also organize information by association. In Python, a **dictionary** associates a set of **keys** with data values. A Python dictionary is written as a sequence of key/value pairs separated by commas. These pairs are sometimes called **entries**. The entire sequence of entries is enclosed in curly braces ({ and }). A colon (**:**) separates a key and its value.

Note : Elements in Dictionaries are accessed via keys and not by their position.

Here are some example dictionaries:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

Output:

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

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error. We can even create an empty dictionary—that is, a dictionary that contains no entries. We would create an empty dictionary in a program that builds a dictionary from scratch.

4.3.1 Dictionary Operations

Operations on dictionary:

1. Accessing an element
2. Update
3. Add element
4. Membership

Creating a dictionary

```
>>> a={1:"one",2:"two"}
>>> print(a)
    {1: 'one', 2: 'two'}
```

Accessing an element

```
>>> a[1]
    'one'
>>> a[0]
    KeyError: 0
```

Update

```
>>> a[1]="ONE"
>>> print(a)
    {1: 'ONE', 2: 'two'}
```

Add element

```
    >>> a[3]="three"
>>> print(a)
{1: 'ONE', 2: 'two', 3: 'three'}
```

Membership

```
a={1: 'ONE', 2: 'two', 3: 'three'}
>>> 1 in a
        True
>>> 3 not in a
        False
```

4.3.2 Dictionary Methods

Method	Example	Description

a.copy()	a={l: 'ONE'. 2: 'two'. 3: 'three'}
	>>> b=a.copy()
	>>> print(b)
	{1: 'ONE', 2: 'two'. 3: 'three'}
		It returns copy of the dictionary, here copy of dictionary 'a' get stored in to dictionary 'b'

a.items()	>>> a.items() diet_items([(l, 'ONE'). (2. 'two'). (3. 'three')])	Return a new view of the dictionary's items. It displays a list of dictionary's (key, value) tuple pairs.
a.keys()	>> a.keys() diet keys([1, 2,])	It displays list of keys in a dictionary
a.values()	>>> a.values() diet_valuesf (['ONE'. 'two', 'three'])	It displays list of values in dictionary

a.pop(key)	>>> a.pop(3)
	'three'
	>>> print(a)
	{1: 'ONE'. 2: 'two'}
		Remove the element with key and return its value from the dictionary.

setdefault (key.value)	>>> a.setdefault(3,"three")
	'three'
	>>> print(a)
	{1: 'ONE'. 2: 'two', 3: 'three'}
	>>> a.setdefault(2)
	'two'
		If key is in the dictionary, return its value. If key is not present, insert key with a value of dictionary and return dictionary.

a.update (dictionary)	>>> b={4:"four"}
	>>> a.update(b)
	>>> print(a)
	{1: 'ONE', 2: 'two', 3: 'three', 4: 'four'}
		It will add the dictionary with the existing dictionary

fromkeys()	>>> key={"apple"."ball"}
	>>> value="for kids"
	>>> d=dict.fromkeys(key,value)
	>>> print(d)
	{'apple': 'for kids', 'ball': 'for kids'}
		It creates a dictionary from key and values.

len(a)	a={l: 'ONE', 2: 'two', 3: 'three'} >>>lena(a) 3	It returns the length of the list
clear()	a={1: 'ONE', 2: 'two'. 3: 'three'} >>>a.clear() >>>print(a) >>>{}	Remove all elements form the dictionary.
del(a)	a={l: 'ONE', 2: 'two'. 3: 'three'} >>> del(a)	It will delete the entire dictionary.

4.4 ADVANCED LIST PROCESSING

1. List Comprehension:

List comprehensions provide a concise way to apply operations on a list. It creates a new list in which each element is the result of applying a given operation in a list. It consists of brackets containing an expression followed by a "for" clause, then a list. The list comprehension always returns a result list.

Syntax:

list=[expression for item in list if conditional]

Example 1:

```
>>>L=[x**2 for x in range(0,5)]
>>>print(L)
```

Output :

[0, 1, 4, 9, 16]

Example 2 :

```
>>>[x for x in range(1,10) if x%2==0]
```

Output :

```
[2, 4, 6, 8]
```

Example 3 :

```
>>>[x+3 for x in [1,2,3]]
```

Output :

```
[4, 5, 6]
```

Example 4 :

```
>>> [x*x for x in range(5)]
```

Output :

```
[0, 1, 4, 9, 16]
```

2. Nested list:

List inside another list is called nested list.

Example:

```
>>> a=[56,34,5,[34,57]]
>>> a[0]
56
>>> a[3]
[34, 57]
>>> a[3][0]
34
>>> a[3][1]
57
```

Reference Links :

List :

NPTEL : https://www.youtube.com/watch?time_continue=4&v=0y5HOotxpys
https://www.youtube.com/watch?v=ffMgNo17Ork

Youtube : https://www.youtube.com/watch?v=ohCDWZgNIU0

Tuples

NPTEL : https://www.youtube.com/watch?time_continue=6&v=lR8DWx2fcbQ
Youtube : https://www.youtube.com/watch?v=vQItTnigtrg

Courseera : https://www.coursera.org/lecture/python-for-applied-data-science/list-and-tuples-bUWEy

Dictionaries:

NPTEL : https://www.youtube.com/watch?time_continue=1&v=lR8DWx2fcbQ
Youtube : https://www.youtube.com/watch?v=V8CQkDTt7eA

ILLUSTRATIVE PROGRAMS

1. Program for matrix addition

```
X = [[12,7,3],
   [4 ,5,6],
   [7 ,8,9]]

Y = [[5,8,1],
   [6,7,3],
   [4,5,9]]

result = [[0,0,0],
        [0,0,0],
        [0,0,0]]

# iterate through rows
for i in range(len(X)):
```

```
    # iterate through columns
    for j in range(len(X[0])):
        result[i][j] = X[i][j] + Y[i][j]

for r in result:
    print(r)
```

2. Program for matrix subtraction

```
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

Y = [[5,8,1],
    [6,7,3],
    [4,5,9]]

result = [[0,0,0],
          [0,0,0],
          [0,0,0]]
# iterate through rows
for i in range(len(X)):
    # iterate through columns
    for j in range(len(X[0])):
        result[i][j] = X[i][j] - Y[i][j]

for r in result:
    print(r)
```

3. Program for matrix Multiplication

```
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]
```

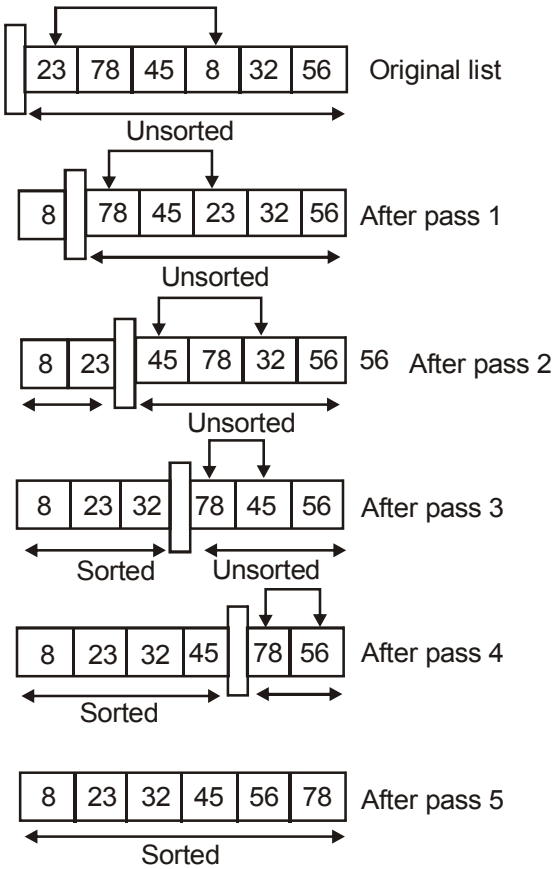
```
# 3x4 matrix
Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]
# result is 3x4
result = [[0,0,0,0],
          [0,0,0,0],
          [0,0,0,0]]
# iterate through rows of X
for i in range(len(X)):
   # iterate through columns of Y
   for j in range(len(Y[0])):
       # iterate through rows of Y
       for k in range(len(Y)):
           result[i][j] += X[i][k] * Y[k][j]
for r in result:
   print(r)
```

4. Program for Selection Sort

Selection sort is one of the simplest sorting algorithms. It is similar to the hand picking where we take the smallest element and put it in the first position and the second smallest at the second position and so on.

We first check for smallest element in the list and swap it with the first element of the list. Again, we check for the smallest number in a sublist, excluding the first element of the list as it is where it should be (at the first position) and put it in the second position of the list. We continue repeating this process until the list gets sorted.

Working of selection sort:



Let's code this in Python

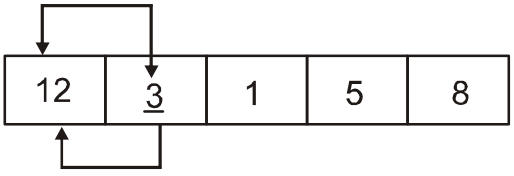
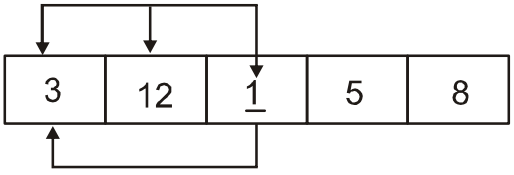
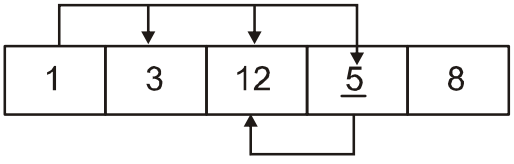
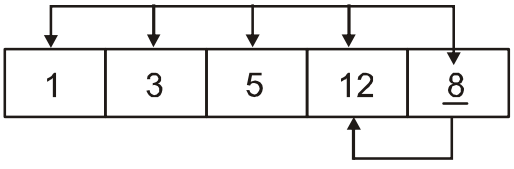
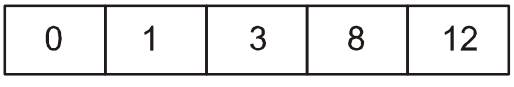
```
a = [23,78,45,8,32,56]
i = 0
while i<len(a):
    #smallest element in the sublist
    smallest = min(a[i:])
    #index of smallest element
    index_of_smallest = a.index(smallest)
    #swapping
    a[i],a[index_of_smallest] = a[index_of_smallest],a[i]
    i=i+1
print (a)
```

Output

[8,23,32,45,56,78]

5. Program for insertion sort

Insertion sort is similar to arranging the documents of a bunch of students in order of their ascending roll number. Starting from the second element, we compare it with the first element and swap it if it is not in order. Similarly, we take the third element in the next iteration and place it at the right place in the sublist of the first and second elements (as the sublist containing the first and second elements is already sorted). We repeat this step with the fourth element of the list in the next iteration and place it at the right position in the sublist containing the first, second and the third elements. We repeat this process until our list gets sorted.

Step 1		Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2		Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3		Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4		Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
		Sorted Array in Ascending Order

Let us code this in python

```
a = [12,3,1,5,8]
#iterating over a
for i in a:
    j = a.index(i)
    #i is not the first element
    while j>0:
        #not in order
        if a[j-1] > a[j]:
            #swap
            a[j-1],a[j] = a[j],a[j-1]
        else:
            #in order
            break
        j = j-1
print (a)
```

Output :

[1,3,5,8,12]

Explanation of the code

for i in a – We are iterating over the list 'a'

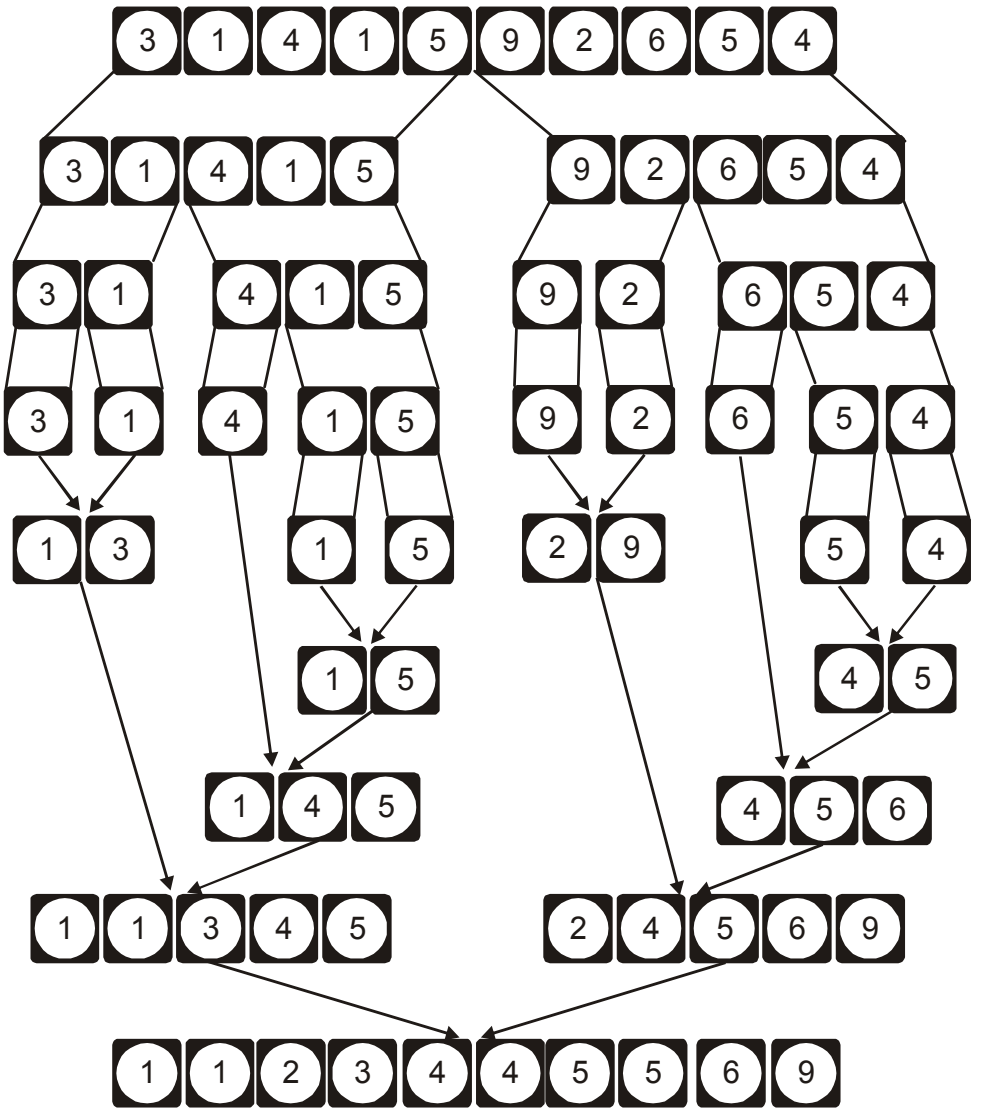
while j>0 – Index of the current element is positive. This means that the element at the index of 'j' is not the first element and we need to compare the values.

if a[j-1] > a[j] – These two elements are not in order. We need to swap them. else – The elements are in order, we can break the while loop.

a[j-1],a[j] = a[j],a[j-1] – Swapping.

6. Program for merge sort

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.



Python Code

```
def mergeSort(nlist):
    print("Splitting ",nlist)
    if len(nlist)>1:
        mid = len(nlist)//2
```

```
        lefthalf = nlist[:mid]
        righthalf = nlist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)
        i=j=k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                nlist[k]=lefthalf[i]
                i=i+1
            else:
                nlist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            nlist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            nlist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",nlist)
nlist = [14,46,43,27,57,41,45,21,70]
mergeSort(nlist)
print(nlist)
```

Output:

```
Splitting  [14, 46, 43, 27, 57, 41, 45, 21, 70]
Splitting  [14, 46, 43, 27]
Splitting  [14, 46]
Splitting  [14]
Merging  [14]
```

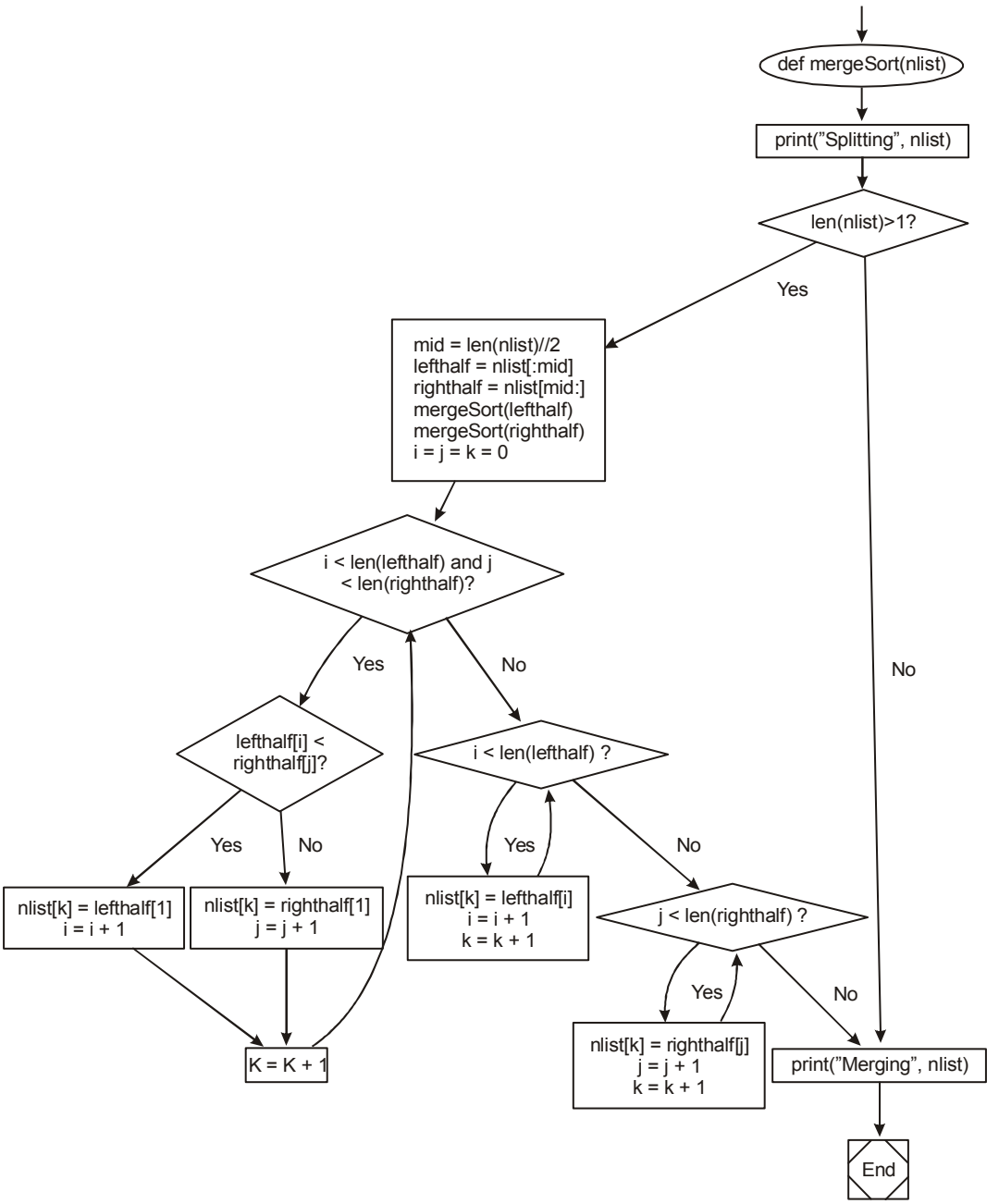
Splitting [46]

———

Merging [14, 21, 27, 41, 43, 45, 46, 57, 70]

[14, 21, 27, 41, 43, 45, 46, 57, 70]

Flowchart



7. Python program for printing histogram

Write a Python program to create a histogram from a given list of integers.

```
def histogram( items ):
    for n in items:
        output = ''
        times = n
        while( times > 0 ):
            output += '*'
            times = times - 1
        print(output)
histogram([2, 3, 6, 5])
```

Output:

```
**
***
******
*****
```

8. Python program to print the calendar of the month

```
import calendar
y=int(input("enter year:"))
m=int(input("enter month:"))
print(calendar.month(y,m))
```

Output:

```
enter year: 2018
enter month: 8
    August 2018
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Suggested Link to Refer :

Selection Sort : https://www.youtube.com/watch?v=mI3KgJy_d7Y
Insertion Sort : https://www.youtube.com/watch?v=Nkw6Jg_Gi4w
Quick Sort(NPTEL) : https://youtu.be/zjqzrcljMlI
Merge Sort (NPTEL) : https://youtu.be/V7fvTmhqokM

Reference:

http://interactivepython.org/courselib/static/pythonds/SortSearch/TheMergeSort.html

PRACTICE PROBLEMS

1. Write a python Program to calculate the average of numbers in a list
2. Write a python Program to find the maximum and minimum number in a list
3. Write a python Program to list even and odd numbers of a list
4. Write a Python program
   i. To add new elements to the end of the list
   ii. To reverse elements in the list
   iii. To display same list elements multiple times.
   iv. To concatenate two list.
   v. To sort the elements in the list in ascending order.
5. Write a Python program for cloning the list and aliasing the list.
6. Write a Python program: “tuple1 = (10,50,20,40,30)”
   i. To display the elements 10 and 50 from tuple1
   ii. To display length of a tuple1.
   iii. To find the minimum element from tuple1.
   iv. To add all elements in the tuple1.
   v. To display same tuple1 multiple times

7. Write a Python program
   i. To create a dictionary
   ii. To adding an element to dictionary
   iii. To display length of the dictionary.
   iv. To updating element in dictionary.
   v. To remove all elements from the dictionary.
8. Write a python program to accept 'n' names, sort names in alphabetic order and print the result.
9. Write a python program to store 'n' names in a list and sort the list using selection sort.
10. Write a python program to merge two lists.
11. Write a python program to remove duplicates from a list.
12. Write a python program to find the 1st and 2nd largest element in the list.

ASSIGNMENT QUESTIONS

1. With a given integral number n, write a program to generate a dictionary that contains (i, i*i) such that is an integral number between 1 and n (both included). and then the program should print the dictionary.

   Suppose the following input is supplied to the program:
   8
   Then, the output should be:
   {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64}

   Hints:

   In case of input data being supplied to the question, it should be assumed to be a console input.

   Consider use dict()

2. Define a function which can print a dictionary where the keys are numbers between 1 and 3 (both included) and the values are square of keys.

Hints:

Use dict[key]=value pattern to put entry into a dictionary.

Use ** operator to get power of a number.

3. Define a function which can print a dictionary where the keys are numbers between 1 and 20 (both included) and the values are square of keys.

Hints:

Use dict[key]=value pattern to put entry into a dictionary.

Use ** operator to get power of a number.

Use range() for loops.

4. Define a function which can generate and print a list where the values are square of numbers between 1 and 20 (both included).

Hints:

Use ** operator to get power of a number.

Use range() for loops.

Use list.append() to add values into a list.

5. With a given tuple (1,2,3,4,5,6,7,8,9,10), write a program to print the first half values in one line and the last half values in one line.

Hints: Use [n1:n2] notation to get a slice from a tuple.

6. By using list comprehension, please write a program to print the list after removing the 0th, 2nd, 4th,6th numbers in [12,24,35,70,88,120,155].

Hints:

Use list comprehension to delete a bunch of element from a list.

Use enumerate() to get (index, value) tuple.

7. With a given list [12,24,35,24,88,120,155,88,120,155], write a program to print this list after removing all duplicate values with original order reserved.

Hints:

Use set() to store a number of values without duplicate.

8. Define the variables x and y as lists of numbers, and z as a tuple.

x=[1, 2, 3, 4, 5]
y=[11, 12, 13, 14, 15]
z=(21, 22, 23, 24, 25)

(a) What is the value of 3*x?
(b) What is the value of x+y?
(c) What is the value of x-y?
(d) What is the value of x[1]?
(e) What is the value of x[0]?
(f) What is the value of x[-1]?
(g) What is the value of x[:]?
(h) What is the value of x[2:4]?
(i) What is the value of x[1:4:2]?
(j) What is the value of x[:2]?
(k) What is the value of x[::2]?
(l) What is the result of the following two expressions?

x[3]=8
print x

(m) What is the result of the above pair of expressions if the list x were replaced with the tuple z?

9. An assignment statement containing the expression a[m:n] on the left side and a list on the right side can modify list a. Complete the following table by supplying the ***m*** and ***n*** values in the slice assignment statement needed to produce the indicated list from the given original list.

Original List	**Target List**	**Slice indices**	
		m	**n**
[2, 4, 6, 8 , 10]	[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]		
[2, 4, 6, 8 , 10]	[–10, –8, –6, –4, –2, 0, 2, 4, 6, 8, 10]		
[2, 4, 6, 8 , 10]	[2, 3, 4, 5, 6, 7, 8, 10]		
[2, 4, 6, 8 , 10]	[2, 4, 6, ‘a’, ‘b’, ‘c’, 8, 10]		
[2, 4, 6, 8 , 10]	[2, 4, 6, 8, 10]		
[2, 4, 6, 8 , 10]	[]		
[2, 4, 6, 8 , 10]	[10, 8, 6, 4, 2]		
[2, 4, 6, 8 , 10]	[2, 4, 6]		
[2, 4, 6, 8 , 10]	[6, 8, 10]		
[2, 4, 6, 8 , 10]	[2, 10]		
[2, 4, 6, 8 , 10]	[4, 6, 8]		

PART A QUESTION AND ANSWERS

1. Define python list.

The list is written as a sequence of data values separated by commas. There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]): ps = [10, 20, 30, 40]

2. Write the different ways to create the list.

List1=[4,5,6].
List2=[]
List2[0]=4
List2[1]=5
List2[2]=6

3. Give an example for nest list

A list within the another list is called as nested list.

Eg: zs = [“hello”, 2.0, 5, [10, 20]]

4. Write the syntax for accessing the list elements.

The way to access list elements is index operator.

Eg: List1=[4,5,6]
List1[0]=4
List1[1]=5
List1[2]=6

5. List out the methods used in list.

Append, Extend, Insert, Index, sort, reverse, pop, remove, clear

6. Write a python program for reversing a list.

vowels = [‘e’, ‘a’, ‘u’, ‘o’, ‘i’] vowels.sort(reverse=True)
print(‘Sorted list (in Descending):’, vowels)

7. Write a python program to add a single element to the end of the list.

vowels = [‘e’, ‘a’, ‘u’, ‘i’]
vowels.append(‘o’)

8. Write a python program to print the list elements using for loop.

num=[10,22,33,44,50,66]

for k in num: print (k)

9. Is list is mutable? Justify your answer.

List elements can be changed once after initializing the list. This is called as mutability.

>>> numbers = [42, 123]

>>> numbers[1] = 5

>>> numbers[42, 5]

10. Write difference between list aliasing and cloning.

An object with more than one reference has more than one name, the object is **aliased**. If the aliased object is mutable, changes made with one alias affect the other. Clone changes are not being affected the other copy of list.

11. How tuple is differ from list?

Tuples are created by using ( ).

Tuples are immutable.

List is created by using [ ].

List is mutable.

12. Explain how tuples are used as return values

A function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values.

>>> t = divmod(7, 3)

>>> t (2,1)

13. What is a dictionary? Give an example.

Python dictionary is written as a sequence of key/value pairs separated by commas. These pairs are sometimes called **entries**. The entire sequence of entries is enclosed in curly braces ({ and }). A colon (**:**) separates a key and its value.

Eg: dict = {'Name': 'Zara', 'Age': 7}

14. List out the methods used in dictionary.

Get, fromkeys, clear, copy, items, pop,popitem, setdefault, update, etc. How do we done the membership test in dictionary?

```
>>>C={1:1,2:2}
>>>2  in C True
```

15. Write a python program for iterating the dictionary elements.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49}
for i in squares: print(squares[i])
```

16. What is called entries in dictionary?

Python dictionary is written as a sequence of key/value pairs separated by commas. These pairs are sometimes called entries.

17. List out the built in functions of dictionary.

The functions in dictionary : len( ), any( ), all( ), comp( ), sorted( )

PART B QUESTIONS

1. Discuss about tuple in detail.
2. Define list. Mention the list operations with programs.
3. Explain about advanced list processing and list comprehension.
4. Discuss about Dictionaries, its operations and methods.
5. Code programs :
   a. Insertion sort
   b. Selection sort
   c. Merge sort
   d. Histogram

Unit V

FILES, MODULES, PACKAGES

Files and exception: text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages; illustrative programs: word count, copy file.

5.1 FILES

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

5.1.1 Opening a file

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")                    # open file in current directory
>>> f = open("C:/Python33/README.txt") # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

The open function gets two arguments

1. File Name → Name of the file
2. Mode → indicates that files is opened to write

5.1.2 Python File Modes

Mode	Description
'r'	Open a file for reading. (default)
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and w

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')          # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode
```

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt",mode = 'r')
```

5.1.3 Closing a File

When we are done with operations to the file, we need to properly close it.

Closing a file will free up the resources that were tied with the file and is done using the close() method.

```
f= open("test.txt",'r')          # perform file operations
f.close()
```

5.1.4 Reading and writing

Reading from a file

A txt file is generally stored in the secondary media like hard disk drive, CD.

We know how to open and close a file object. But what are the actual commands for reading? There are three ways to read from a file.

- read([n])
- readline([n])
- readlines()

Note: that n is the number of bytes to be read.

read( ) method

```
my_file=open("D:\\new_dir\\multiplelines.txt","r")
my_file.read()
```

The read() method just outputs the entire file if number of bytes are not given in the argument. If you execute my_file.read(3), you will get back the first three characters of the file

```
my_file=open("D:\\new_dir\\multiplelines.txt","r")
my_file.read(3)
```

readline (n)

Outputs at most n bytes of a single line of a file. It does not read more than one line.

```
my_file.close ()
my_file=open ("D:\\new_dir\\multiplelines.txt","r")
#Use print to print the line else will remain in buffer and replaced by next statement
print(my_file.readline())
# outputs first two characters of next line
print(my_file.readline(2))
```

readlines( )

This method maintains a list of each line in the file

```
my_file=open("D:\\new_dir\\multiplelines.txt","r")
my_file.readlines()
```

5.1.5 Writing to a file

Writing methods also come in a pair: **.write ()** and **.writelines ().** Like the corresponding reading methods, .write() handles a single string, while .writelines() handles a list of strings.

Below, .write() writes a single string each time to the designated output file:

```
>>> fout = open('hello.txt', 'w')
>>> fout.write('Hello, world!\n')              # .write(str)
>>> fout.write('My name is Homer.\n')
>>> fout.write("What a beautiful day we're having.\n")
>>> fout.close( )
```

This time, we have tobuy, a list of strings, which .writelines() writes out at once:

```
>>> tobuy = ['milk\n', 'butter\n', 'coffee beans\n', 'arugula\n']
>>> fout = open('grocerylist.txt', 'w')
>>> fout.writelines(tobuy)                    # .writelines(list)
>>> fout.close()
```

Note that all strings in the examples have the line break '\n' at the end. Without it, all strings will be printed out on the same line

Example for writing a file:

```
file = open("testfile.txt", "w")
file.write("This is a test")
file.write("To add more lines.")
file.close()
```

Examples

The following function copies a file, reading and writing up to fifty characters at a time.

```
def copyFile(oldFile, newFile):
    f1 = open(oldFile, "r")
    f2 = open(newFile, "w")
while True:
    text = f1.read(50)
    if text == "":
        break
    f2.write(text)
    f1.close()
    f2.close()
    return
```

Simple Examples

To open a text file, use:

```
fh = open("hello.txt", "r")
```

To read a text file, use:

```
fh = open("hello.txt","r")
print fh.read()
```

To read one line at a time, use:

```
fh = open("hello".txt", "r")
print fh.readline()
```

To read a list of lines use:

```
fh = open("hello.txt.", "r")
print fh.readlines()
```

To write to a file, use:

```
fh = open("hello.txt","w")
write("Hello World")
fh.close()
```

To write to a file, use:

```
fh = open("hello.txt", "w")
lines_of_text = ["a line of text", "another line of text", "a third line"]
fh.writelines(lines_of_text)
fh.close()
```

To append to file, use:

```
fh = open("Hello.txt", "a")
write("Hello World again")
fh.close
```

To close a file, use

```
fh = open("hello.txt", "r")
print fh.read()
fh.close()
```

Suggested Links to refer:

Link: https://www.youtube.com/watch?v=efpFDaXOG6Y

Link: https://www.youtube.com/watch?v=dkLTmpldS-w

Link: https://www.youtube.com/watch?v=9Lu6597k2mg

Link: https://www.youtube.com/watch?v=sNVpwjGdfiE

5.1.6 Format operator

String objects have one unique built-in operation:

The % operator(modulo).this is also known as the stringformatting or interpolation operator.

The format operator % are replaced with zero or more elements of values. The built-in format() method returns a formatted representation of the given value controlled by the format specifier.

The syntax of format() is: format(value[, format_spec])

The format() method takes two parameters:

- **value** - value that needs to be formatted
- **format_spec** - The specification on how the value should be formatted.

Example:

```
#   d, f and b are type
#   integer
print(format(453, "d"))
#   float arguments print(format(978.2229911, "f"))
#   binary format
print(format(2, "b"))
```

This will returns the following output

```
453
978.222991
10
```

Basic formatting

Simple positional formatting is probably the most common use-case. Use it if the order of your arguments is not likely to change and you only have very few elements you want to concatenate.

Old '%S %S' % ('one', 'two')

New '{ } { }', format ('one', 'two')

Output

o	n	e		t	w	o

Old '%d %d' % (1, 2)

New '{ } { }', format (1, 2)

Output 1 2

An alternative is to use the format operator, %. When applied to integers, % is the modulus operator. A format sequence can appear anywhere in the string,so we can embed a value in a sentence:

print('In %d years I have spotted %g %s.' % (3, 0.1, 'camels'))

This will results

In 3 years I have spotted 0.1 camels.

The number of elements in the tuple has to match the number of format sequences in the string.

Also, the types of the elements have to match the format sequences:

Example

Suppose if our code is

%d %d %d' % (1, 2)

Then this will generate an error like Type Error: not enough arguments for format string

5.1.7 Command line arguments

What are command line arguments in python?

In the **command line**, we can start a program with additional arguments. These **arguments** are passed **into the program**.

Python programs can start with command line arguments.

For example: $python **program.py image.bmp**

Where **program.py** and **image.bmp** is are arguments.

How to use command line arguments in python?

We can use modules to get arguments.

Sys argv

You can get access to the command line parameters using the sys module. len(sys.argv) contains the number of arguments. To print all of the arguments simply execute str(sys.argv)

```
#!/usr/bin/python
import sys
print('Arguments:', len(sys.argv))
print('List:', str(sys.argv))
```

Example:

```
$ python3 example.py image.bmp color
Arguments: 3
List: ['example.py', 'image.bmp', 'color']
```

Storing command line arguments

You can store the arguments given at the start of the program in variables.For example, an image loader program may start like this:

```
#!/usr/bin/python
#!/usr/bin/python
import sys
print('Arguments:', len(sys.argv))
print('List:', str(sys.argv))
if sys.argv < 2:
   print('To few arguments, please specify a filename')
filename = sys.argv[1]
print('Filename:', filename)
```

Another example:

```
('Arguments:', 2)
('List:', "['example.py', 'world.png']")
('Filename:', 'world.png')
```

Suggested Links to refer

Open, Read, Display a text file :
Link : https://www.youtube.com/watch?v=dkLTmpldS-w

Create a Text File
Link : https://www.youtube.com/watch?v=DRZdfd5_rdg

Format Method
Link : https://www.youtube.com/watch?v=mmJPx6YsOMI

Command Line arguments
Link : https://www.youtube.com/watch?v=d3uv23jvp4w
Linkhttps://www.tutorialspoint.com/python_online_training/python_from_command_line.asp

5.2 ERRORS AND EXCEPTIONS

In Python, there are two kinds of errors: syntax errors and exceptions.

Syntax Errors

This is an error, Raised when there is an error in Python syntax.

Example:

```
if a<5
File "<interactive input>", line 1
    if a < 5
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little „arrow pointing at the earliest point in the line where the error was detected.

Exceptions

An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash. The types in the example are ZeroDivisionError, NameError and TypeError.

Example

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1,
in <module> ZeroDivisionError:
division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str
implicitly
```

Why use Exceptions?

Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.

Raising an Exception

You can raise an exception in your own program by using the raise exception statement. Raising an exception breaks current code execution and returns the exception back until it is handled.

5.2.1 List of Standard Exceptions

Exception Name	Description
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input()function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.

UnboundLocalError	Raised when trying to access a local variable in a function or methodbut no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
IOError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit()function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

5.2.2 Handling Exceptions

It is possible to write programs that handle selected exceptions. Exception is an event, which occurs during the execution of program and disrupts the normal flow of programs instructions. When a python script raises an exception, it must either handle

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

Different ways of exception handling in python are:

→ try… except

→try… finally

try…except

a single try statement can have multiple except statements.

Useful when we jave a try block that may throw different types of exceptions. Code in else- block executes if the code in the try:block does not raise an exception

Syntax:

try:

You do your operations here;

.....

except ExceptionI:

If there is ExceptionI, then execute this block.

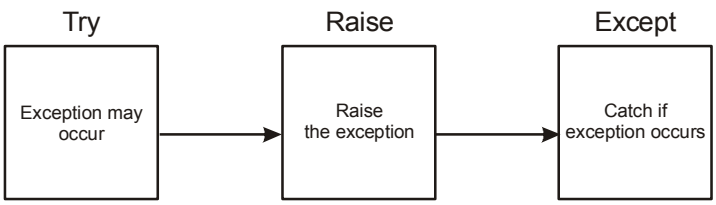
except ExceptionII:

If there is ExceptionII, then execute this block.

.....

else:

If there is no exception then execute this block.



Checklist To Handle An Exception In Python.

Here is a checklist for using the Python try statement effectively. A single try statement can have multiple except statements depending on the requirement. In this case, a try block contains statements that can throw different types of exceptions. We can also add a generic except clause which can handle all possible types of exceptions. We can even include an else clause after the except clause. The code in the else block will execute if the code in the try block doesn t raise any exception

Example:

```
while True:
    try:
        n = raw_input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print "Great, you successfully entered an integer!"
```

Output

```
Please enter a number: 23.5
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'
```

It's a loop, which breaks only, if a valid integer has been given.

The example script works like this:

The while loop is entered. The code within the try clause will be executed statement by statement. If no exception occurs during the execution, the execution will reach the break statement and the while loop will be left. If an exception occurs, i.e. in the casting of n, the rest of the try block will be skipped and the except clause will be executed. The raised error, in our case a Value Error, has to match one of the names after except. In our example only one, i.e. "Value Error:". After having printed the text of the print statement, the execution does another loop. It starts with a new raw input ().

Example:

```
num1=10
num2=2
list1=[10,20,30]
try:
        num3=num1/num2
        print(num3)
        print(list1[4])
except ZeroDivisionError:
        print("Division By Zero")
except IndexError:
        print("Index out of range")
except:
            print("Not Specific Error")
        print(num1)
        print(num2)
```

The above code prints the following output

```
5.0
Index out of range
10
2
```

try ..finally

finally block is a place to put anny code that must execute irrespective of try block raised an exception or not. except block can be used with finally block

syntax:

```
try:
        you do your operations here;
        ……..……
        Due to any exceptions this may be skipped.
except:
        ……..
```

```
finally:
    this would always be executed
```

Example:

```
num=4
try:
    res=num/0
    print(res)
except:
    print("Zero division error")
finally:
    print("inside finally")
print("Out of try except")
```

The above code will prints the following output

```
Zero division error

inside finally
Out of try except
```

Suggested Links to refer

Link 1 : https://www.youtube.com/watch?v=NIWwJbo-9_8

Link 2 : https://www.youtube.com/watch?v=hrR0WrQMhSs

5.3 MODULES

A Python module is simply a Python source file, which can expose classes, functions and global variables. When imported from another Python source file, the file name is treated as a namespace. A module is a file containing Python definitions and statements intended for use in other Python programs. There are many Python modules that come with Python as part of the standard library.

Random numbers

Uses of Random numbers, To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin,

- To shuffle a deck of playing cards randomly,

- To allow/make an enemy spaceship appear at a random location and start shooting at the player,
- To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam,
- For encrypting banking sessions on the Internet Python provides a module random that helps with tasks like this.

Example

```
import random
#   Create a black box object that generates
random numbers rng = random.Random()
dice_throw = rng.randrange(1,7) # Return an int, one of
1,2,3,4,5,6 delay_in_seconds = rng.random() * 5.0
```

Repeatability and Testing

Random number generators are based on a deterministic algorithm — repeatable and predictable. So they re called pseudo-random generators — they are not genuinely random. They start with a seed value. Each time you ask for another random number, you ll get one based on the current seed attribute, and the state of the seed (which is one of the attributes of the generator) will be updated.

The time module

This module provides a number of functions to deal with dates and the time within a day. It s a thin layer on top of the C runtime library. A given date and time can either be represented as a floating point value (the number of seconds since a reference date, usually January 1st, 1970), or as a time tuple. The time module has a function called clock that is recommended for this purpose. Whenever clock is called, it returns a floating point number representing how many seconds have elapsed since your program started running

Example :

```
import time;  # This is required to include time module.
ticks = time.time()
print “Number of ticks since 12:00am, January 1, 1970:”, ticks
```

Output:

Number of ticks since 12:00am, January 1, 1970: 7186862.73399

Getting current time

```
import time;
localtime = time.asctime( time.localtime(time.time()) )
print "Local current time :", localtime
```

Output:

('Local current time :', 'Mon Jul 09 21:32:29 2018')

The math module

The math module contains the kinds of mathematical functions you d typically find on your calculator (sin, cos, sqrt, asin, log, log10) and some mathematical constants like pi and e:

Example:

```
import math
print(math.pi)
print(math.e)
print(math.sqrt(4.0))
print(math.radians(90))
print(math.sin(math.radians(90)))
```

this would print the following results

```
3.141592653589793
2.718281828459045
2.0
1.5707963267948966
1.0
```

Creating your own modules

Python allows you to create our own modules is to save our script as a file with a .py extension.

```
def remove_at(pos, seq):
return seq[:pos] + seq[pos+1:]
```

We can now use our module, both in scripts we write, or in the interactive Python interpreter.

To do so, we must first import the module.

import seqtools

```
s = "A string!"
seqtools.remove_at(4, s)
 A sting!
```

Namespaces

A namespace is a collection of identifiers that belong to a module, or to a function, (and as we will see soon, in classes too). Generally, we like a namespace to hold "related" things, e.g. all the math functions, or all the typical things we d do with random numbers. Each module has its own namespace, so we can use the same identifier name in multiple modules without causing an identification problem.

 # Module1.py

```
question = "What is the meaning of Life, the Universe, and Everything?"
answer = 42
```

 # Module2.py

```
question = "What is your quest?"
answer = "To seek the holy grail."
```

We can now import both modules and access question and answer in each:

```
import module1
import module2
print(module1.question)
print(module2.question)
print(module1.answer)
print(module2.answer)
```

Will output the following:

```
What is the meaning of Life, the Universe, and Everything?
What is your quest?
```

42

To seek the holy grail.

Scope and lookup rules

The scope of an identifier is the region of program code in which the identifier can be accessed, or used.

There are three important scopes in Python:

Local scope refers to identifiers declared within a function. These identifiers are kept in the namespace that belongs to the function, and each function has its own namespace.

Global scope refers to all the identifiers declared within the current module, or file. Built-in scope refers to all the identifiers built into Python — those like range and min that can be used without having to import anything, and are (almost) always available.

Example

```
def range(n):
        return 123*n
print(range(10))
this would print
1230
n = 10
m = 3
def f(n):
m = 7
return 2*n+m
print(f(5), n, m)
the above code will prints 17 10 3
```

Attributes and the dot operator

Variables defined inside a module are called attributes of the module. We ve seen that objects have attributes too: for example, most objects have a __doc__ attribute, some functions have a __annotations__ attribute. Attributes are accessed using the dot operator (.). The question attribute of module1 and module2 is accessed using module1.question and module2.question.

Modules contain functions as well as attributes, and the dot operator is used to access them in the same way. seqtools.remove_at refers to the remove_at function in the seqtools module. When we use a dotted name, we often refer to it as a fully qualified name, because we re saying exactly which question attribute we mean.

Three import statement variants

Here are three different ways to import names into the current namespace, and to use them:

1. import math x = sqrt(10)
2. from math import

   cos, sin, sqrt x =

   sqrt(10)

3. from math import *

   x = sqrt(10)

Suggested Links to refer

Link 1 : https://www.youtube.com/watch?v=UK97NoQK23k

Link2 : https://www.youtube.com/watch?v=fPrzjXiXpnc

5.4 PYTHON PACKAGES

A Python package refers to a directory of Python module(s). This feature comes in handy for organizing modules of one type at one place.

A python package is normally installed in:

/usr/lib/python/site-packages # for linux

C:/Python27/Lib/site-packages/ # for windows

To use the package in a script, you will have to first initialize the package using:

mypackage/__init__.pymypackage/mymodule.py

You can then import the package

```
import mypackage.mymodule
```

or

```
from mypackage.mymodule import myclass
```

In addition to creating ones own packages, Python is home a large and growing collection of packages (from individual programmers) which is available from the Python Package Index.

Package installation

There are two standard methods for installing a package.

pip install

The pip install script is available within our scientific Python installation and is very easy to use (when it works). During the installation process you already saw many examples of pip install in action. Features include:

If supplied with a package name then it will query the PyPI site to find out about that package.

Assuming the package is there then pip install will automatically download and install the package.

Will accept a local tar file (assuming it contains an installable Python package) or a URL pointing to a tar file.

Can install in the user package area via pip install <package or URL> —user

python setup.py install

Some packages may fail to install via pip install. Most often there will be some obvious (or not) error message about compilation or missing dependency. In this case the likely next step is to download the installation tar file and untar it. Go into the package directory and look for files like:

```
INSTALL
README
setup.py
setup.cfg
```

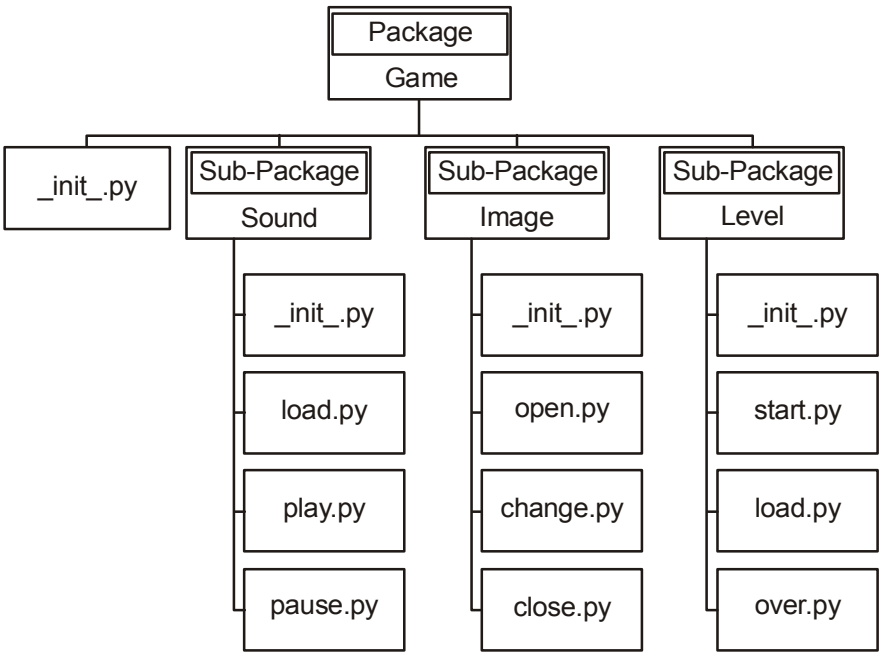
If there is an INSTALL or README file then hopefully you will find useful installation instructions. Most well-behaved python packages do the installation via a standard setup.py script. This is used as follows:

python setup.py —help # get options

python setup.py install # install in the python area (root / admin req'd)

python setup.py install —user # install to user's package area

Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



Suggested Links to refer

Link 1: https://www.youtube.com/watch?v=qh3mJ1elP8I

Link2 : https://www.youtube.com/watch?v=qmsTqQbcBNM

SAMPLE PROGRAMS

1. Program to count the no of words in a given sentence

```
while True:
    print("Enter 'x' for exit.")
    string = input("Enter any string: ")
    if string == 'x':
        break
    else:
        word_length = len(string.split())
        print("Number of words =",word_length,"\n")
```

Output:

Enter 'x' for exit.
Enter any string: Prathyusha ENgineering College
Number of words = 3

2. To find the most frequent appearance of words in the text.

```
from string import punctuation
from operator import itemgetter
N = 10
words = {}
words_gen = (word.strip(punctuation).lower()
for line in open("license.txt")
for word in line.split())
for word in words_gen:
    words[word] = words.get(word, 0) + 1

top_words = sorted(words.iteritems(), key=itemgetter(1), reverse=True)[:N]
for word, frequency in top_words:
if frequency>100:
    print ("%s: %d" % (word, frequency))
```

Output:

```
the: 338
the: 338
:255
of: 220
 or: 174
 and: 151
 in: 132
 this: 120
to: 105
software: 103
```

3. **To count word frequency in a given text**

```
test_string=input("Enter string:")
l=[]
l=test_string.split()
wordfreq=[l.count(p) for p in l]
print(dict(zip(l,wordfreq)))
```

Output:

```
Enter string: All is Well All is Well
{'All': 2, 'is': 2, 'Well': 2}
```

4. **program to copy file from one place to another**

```
from shutil import copyfile
from sys import exit
source = input("Enter source file with full path: ")
target = input("Enter target file with full path: ")
# adding exception handling
try:
    copyfile(source, target)
except IOError as e:
    print("Unable to copy file. %s" % e)
    exit(1)
except:
```

```
        print("Unexpected error:", sys.exc_info())
        exit(1)

print("\nFile copy done!\n")

while True:
    print("Do you like to print the file ? (y/n): ")
    check = input()
    if check == 'n':
            break
    elif check == 'y':
            file = open(target, "r")
            print("\nHere follows the file content:\n")
            print(file.read())
            file.close()
            print()
            break
    else:
            continue
```

ASSIGNMENT QUESTIONS

1. Write a Python program to append text to a file and display the text.
2. Write a Python program to read a file line by line and store it into a list.
3. Write a python program to find the longest words in a file.
4. Write a Python program to assess if a file is closed or not.

PART A QUESTION AND ANSWERS

1. Define a file

Files are collection of data. It is stored in computer memory and can be taken any time we require it. Each file is identified by a unique name. In general a text **file** is a file that contains printable characters and whitespace, organized into lines separated by newline characters.

Eg: file.txt

2. List the two methods used for installing python pacakage.

There are two standard methods for installing a package.

pip install

The pip install script is available within our scientific Python installation.

python setup.py install

3. What python package refers to?

A Python package refers to a directory of Python module.

To import a package :

```
import mypackage.mymodule
```

4. How do we import the packages in python? Give and example.

We can then import the package by:

```
import mypackage.mymodule
or
from mypackage.mymodule import myclass
```

5. What is the use of dot(.) operator?

Variables defined inside a module are called attributes of the module. Attributes are accessed using the dot operator (.)

6. Define namespaces in python.

A namespace is a collection of identifiers that belong to a module, or to a function.

Each module has its own name space.

7. Write a program to generate a numbers using random module.

```
import random
rng = random.Random()
dice_throw = rng.randrange(1,7)
```

8. What is called exception?

Exception is an event , which occurs during the execution of program and distrupts the normal flow of programs instructions.

9. What are the two ways to handle the exceptions.

The two ways to handle exception is:

try.... catch

try...finally

10. What is syntax error ? Give an example.

This is an error, Raised when there is an error in Python syntax.

Example:

```
if a<5
File "<interactive input>", line 1
if a < 5
^
```

SyntaxError: invalid syntax

11. Describe about the command line arguments.

The sys.argv is a list in Python, which contains the command-line arguments passed to the script. With the len(sys.argv) function you can count the number of arguments.

12. Discover the format operator available in files.

A format sequence can appear anywhere in the string,so we can embed a value in a sentence: print('In %d years I have spotted %g %s.' % (3, 0.1, 'camels'))

```
'{} {}'.format(1,2)
```

1 2

13. Write a program to write a data in a file.

Eg:

```
f = open("pec.dat","w")
f.write("Welcome to PEC")
f.close()
```

Eg:

```
Text=f.read()
Print text
```

14. What are modules?

A Python module is simply a Python source file, which can expose classes, functions and global variables.

Wg: import math

PART B QUESTIONS

1. Explain about files with file operations.
2. Discuss on errors, exceptions and exception handling.
3. Discuss about modules and how will we create own modules.
4. Explain about packages and command line arguments.