UNIT - I

Computational Thinking and Problem Solving

Fundamentals of computing

computer

The term computer is derived from the word compute. The word compute means "to calculate"

It is an electronic machine that accepts data from user, processes the data by performing calculations and operations on it and generates the desired output results.

characteristics of computer

- Speed
- Accuracy
- Diligence
- storage capacity
- Versatility

Generation of computers

The computer has evolved from large sized simple calculating machine to a smaller but much more powerful machine. The 5 generations of computer are identified in terms of
- technology used by them
- computing characteristics
- physical appearance
- their applications

First Generation (1940-1956) : Using Vacuum Tubes

Hardware Technology

Vacuum tubes are used for circuitry and magnetic drums for memory. Punched cards were used for input & outputs was displayed as printouts

Software Technology

- The instructions were in machine language and this language uses 0's and 1's for coding of the instructions.
- The first generation computer could solve one problem at a time.

computing characteristics

The computation time was in milliseconds.

physical Appearance

These computers were enormous in size and required a large room for installation

Application

They were used for scientific applications as they were the fastest computing device of their time

Example

- UNIVAC
- ENIAC
- EDVAC

Disadvantage

- Generated a lot of heat and were expensive to operate
- Machines were prone to frequent malfunctioning

Second Generation (1956-1963) using Transistors

Hardware Technology

- Transistors allowed computers to become smaller, faster, cheaper, energy efficient and reliable.
- This generation computers used magnetic core technology for primary memory and used magnetic tapes and disks for secondary storage.
- They used the concept of stored program

Software Technology

- The instruction were written using assembly language. This language uses mnemonics like ADD for Addition and SUB for subtraction for coding of the instructions.
- Easier to write instructions in Assembly language.

computing characteristics

The computation time was in microseconds

physical Appearance

Transistors are smaller in size compared to vaccum tubes. Thus the size of the computer was also reduced.

Application

The cost of commercial production of these computers was very high though less than the first generation computers

Example

PDP-8, IBM 1401 and CDC 1604.

Disadvantage

- Generated lot of heat but less than 1st generation computers
- Less maintenance required

Third Generation (1964-1971): Using Integrated Circuits (IC)

Hardware Technology

- Uses IC chips. An IC chip contains transistors, wiring and other components on a single silicon chip.
- The Keyboard and monitor were used to interact with 3rd generation computer.

Software Technology

- Keyboard and monitor were interfaced through OS
- OS allowed different applications to run at the same time

Computing characteristics

The computation time was in nanoseconds

Physical Appearance

The size of these computers were small compared to 2nd generation computers.

Application

computers become accessible to mass audience.

Example

IBM 370, PDP 11

Advantages

- This generation computers used less power and generated less heat.
- The cost of the computer was less

Fourth Generation (1971 to present): Using Microprocessors

Hardware Technology

- They use Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) technology.
- Thousands of Transistors are integrated on a small silicon chip using LSI technology.
- Hundreds of thousands of transistors are integrated on a small chip using VLSI technology.
- Semiconductor memory replaced magnetic core memory resulting in fast random access to memory.

Software Technology

-OS like MS-DOS and MS-Windows were developed. This generation of computers supported GUI

- GUI is user friendly interface that allows user to interact with computers via menus and icons

- High level programming languages are used for writing of programs

Computing characteristics

The computation time is in picoseconds.

Physical Appearance

Smaller than computers of previous generations

Application

- They became widely available for commercial purposes

- Personal computers became available to the home user.

Example

Intel 4004 chip was the first microprocessor.

Advantages

- The fourth generation computers are portable and more reliable

- They generate much less heat and require less maintenance.

Fifth Generation (present and next) Artificial Intelligence

5th generation computers use super large scale Integrated (SLSI) chips that are able to store millions of components on a single chip. These computers have large memory requirements. This generation of computers uses parallel processing that allows several instructions to be executed in parallel.

The intel dual core microprocessors uses parallel processing.

AI includes area like Expert System (ES) Natural language Processing (NLP) Speech recognition, Voice recognition, Robotics etc.

Classification of computers

Digital computers

- This computers are used for data processing and problem solving purpose using programs

- A digital computer operate by counting digits in the binary form.

Analog computers

Analog computers are generally used in industrial process controls and to measure physical quantities such as pressure

temperature etc.

This computer does not operate on binary digits to compute. It works on continuous electrical signal and output is displayed continuously.

Hybrid computers

This computers are the combination of digital and analog computers. A hybrid computer uses the best features of digital and analog computers.

The digital computers are classified into 4 categories based on the size and type

- Micro computers
- Mini computers
- Mainframe computers
- Super computers

Micro computer

- Small, low cost and single user digital computer.
- Consists of CPU, I/p unit, O/p unit, storage unit and the software
- They are powerful and easy to operate.
- Example: IBM PC based on Pentium Microprocessor and Apple Macintosh

Mini computer

- Mini computers are digital computers generally used in multi user systems.
- They have high processing speed and high storage capacity.
- Mini computers can support 4-200 users simultaneously.

Eg: PDP11, IBM

Mainframe computers

Mainframe computers are multiuser, multi programming and high performance computers. They operate at a very high speed have large storage capacity and can handle the workload of many users.

The user accesses the mainframe computer via a terminal that may be dumb terminal an intelligent terminal or a PC.

A dumb terminal cannot store data or do processing of its own. It has the input and output device only.

Super computers

- Those computers are the fastest and expensive machines. They have high processing speed compared to other computers.
- The speed of the computer is measured in FLOPS. This computers can perform trillions of calculation per second.
- Eg: IBM Roadrunner, IBM Blue gene.

Basic organization of a computers

The computer system hardware comprises of 3 main components

- Input / output unit
- Central Processing Unit
- Memory unit

Input / output (I/O) unit

- The I/O unit consists of the I/p unit and the O/p unit.
- The user interacts with the computer via the I/O unit.
- The i/p unit accepts data from user and the o/p unit provide processed data to the user.
- The i/p unit accepts data from user and converts into a form that is understandable by the computer
- The o/p unit provides the o/p in the form that is understandable by the user.
- Some of the commonly used O/p devices are monitor & Printer

Central Processing Unit

The CPU or the processor is also often called the brain of the computer

The CPU controls, coordinates and supervises the operations of the computer.

It is responsible for processing of the I/p data.

CPU consists of control unit and Arithmetic and Logical unit.

In Addition CPU also has a set of registers which are temporary storage areas for holding data and instructions.

- ALU performs all the arithmetic and logical operations on the i/p data.
- CU controls the overall operations of the computer.
- CPU uses the registers to store the data instructions during processing

Arithmetic and logical unit

- ALU consists of two units - arithmetic unit and logic unit.
- The arithmetic unit performs arithmetic operations such as addition, subtraction, multiplication and division
- The logic unit performs logical operations such as less than greater than, less than or equal to, greater than or equal to, equal to and not equal to.

Registers

- Registers are high speed storage areas within the CPU but have the least storage capacity.
- Registers store data, instructions, addresses and intermediate results of processing.
- Some of the important registers in CPU are as follows
  - Accumulator
  - Instruction Register
  - Program counter
  - Memory Address Register
  - Memory buffer Register
  - Data Register
- The size of register also called word size
- The size of a register may be 8, 16, 32 or 64 bits

control unit

The control unit of a computer does not do any actual processing of data. It organizes the processing of data and instructions.

- It acts as a supervisor and controls and coordinates the activity of the other units of computer.
- CU coordinates the i/p and o/p
- It also instructs the ALU to perform the arithmetic and logic operations.
- CU also holds the CPU's instruction set, which is a list of all operations that the CPU can perform.

Memory unit

The memory unit consists of cache memory and primary memory.

In addition to the main memory there is another kind of storage device known as the secondary memory.

Secondary memory is non-volatile and is used for permanent storage of data and programs

Cache memory

- It is a very high speed memory placed in between RAM and CPU.
- Cache memory increases the speed of processing
- cache memory is a storage buffer that stores the data that is used more often
- Cache memory is very expensive so it is smaller in size

Primary memory

- It is the main memory of computer
- It is used to store the data and instructions during execution of the instructions
- Primary memory is of 2 kinds - Random Access Memory and Read only Memory.
- RAM is volatile. The information gets erased when the computer is turned off.
- ROM is non-volatile memory but is a read only memory ROM comes programmed by the manufacturer.

Secondary Memory

- The secondary memory stores data and instructions permanently.
- It provides back up storage for data and instruction
- Hard disk drive, floppy drive and optical disk drives are some examples of storage devices
- Secondary memory has high storage capacity than the primary memory.

Applications of computer

Computers play a role in every field of life. They are used in homes, business, educational institutions, research

organizations, medical field, government offices, environment etc

Identification of computational Problems

In theoretical computer science a computational problem is a problem that a computer might be able to solve or a question that a computer may be able to answer

A computational problem can be viewed as a set of instances or cases together with a possibly empty set of solutions for every instance/case

Computational problems are one of the main objects of study in theoretical computer science.

Types

Decision problem

A decision problem is a computational problem where the answer for every instance is either yes or no. An example of a decision problem is primality testing. "Given a positive integer n, determine if n is prime"

Search problem

In a search problem the answers can be arbitrary strings

A search problem is represented as a relation consisting of all the instance-solution pairs called a search relation For example, factoring can be represented as the relation

$R = \{(4,2), (6,2), (6,3), (8,2), (9,3), (10,2), (10,5) \cdots\}$

which consist of all pairs of numbers (n, p) where p is a nontrivial prime factor of n.

Counting problem

A counting problem asks for the number of solutions to a given search problem.

A counting problem can be represented by a function f from $\{0,1\}^*$ to the non negative integers.

optimization problem

An optimization problem asks for finding a "best possible" solution among the set of all possible solutions to a search problem.

Algorithms

It is defined as a sequence of instructions that describe a method for solving a problem. In other words it is a step by step procedure for solving a problem.

properties of Algorithm

- should be written in simple English
- Each and every instruction should be precise and unambiguous.
- Instructions in an algorithm should not be repeated infinitely.
- Algorithm should conclude after a finite number of steps.
- should have an end point
- Derived results should be obtained only after the algorithm terminates.

Qualities of a good algorithm

The following are the primary factors that are often used to judge the quality of the algorithms.

- Time
- memory
- Accuracy

Algorithm 1: Add two numbers entered by the user

Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1, num2 and
Step 4: Add num1 and num2 and assign the results to sum
sum ← num1 + num2
Step 5: Display sum
Step 6: Stop

Algorithm 2: Find the largest number among 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
Steps: stop

Algorithm 3 Find sum and average of 3 numbers

Step 1: Start
Step 2: Declare the variables a and b
Step 3: Read variables a, b and c
Step 4: Add a, b and c and assign the result to sum
sum ← a+b+c
Step 5: Divide sum by 3 and assign the result to avg.
Avg ← sum/3
Step 6: Display sum and avg
Step 7: Stop

Algorithm 4: Find the smallest of two numbers

Step 1: Stad
Step 2: Declare the variables a and b
Step 3: Read the values a and b
Step 4: if a < b then a is smallest
Else b is smallest
Step 5: Stop.

Building blocks of Algorithms (statement, state, control flow, function)

Algorithms can be constructed from basic building block namely, sequence, selection and iteration

statement

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 i/p
- output data processed result

State

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

Control flow

The process of executing the individual statements in a given order is called control flow. The control can be executed in 3 ways

- Sequence
- Selection
- Iteration

Sequence

All the instructions are executed one after another is called Sequence execution

Example

Add two numbers
Step1: Start
Step2: get a, b
Step3: calculate c=a+b
Step4: Display c
Step5: Stop

Selection

A selection statement causes the program control to be transferred to a specific part of the program based upon the condition

If the conditional test is true one part of the program will be executed, otherwise it will execute the other part of the program.

Ex: write an algorithm to check whether he is eligible to vote?

Step1: Start
Step2: Get age
Step3: if age >= 18 print "Eligible to vote"
Step4: else print "Not eligible to vote"
Step 6: Stop

Iteration

In some programs certain set of statements are executed again and again based upon conditional test i.e, executed more than one time. This type of execution is called looping or iteration

Example write an algorithm to print all natual numbers up to n

step 1: start
step 2: get n value
step 3: initialize i = 1
step 4: if (i <= n) go to step 5 else go to step 7
step 5: Print i value and increment i value by 1
step 6: go to step 4
step 7: stop

Functions

Function is a sub program which consists of block of code (set of instructions) that performs a particular task.

Benefits of using function

- Reduction in line of code
- code reuse
- better readability
- Information hiding
- Easy to debug & test
- Improved maintainability

Ex: Algorithm for addition of two numbers using function

main function ()
step1: start
step2: call the function add()
step3: stop
Sub function add()
step1: function start
step2: Get a, b values
step3 add c = a + b
step4: print c
step5 Return

Notation (Pseudo code, flow chart, programming language)

Pseudocode

Pseudocode 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 not a machine readable
- Pseudo code can't be compiled and executed.
- There is no standard syntax for pseudo code

Guidelines for writing pseudo code

- write one statement per line
- Capitalize initial Keyword
- Indent to hierarchy
- End multiline structure
- Keep statement language independent

common Keywords used in pseudocode

- //: This keyword used in pseudocodes
- BEGIN, END Begin is the first statement and end is the last statement
- INPUT, GIET, READ: The Keyword is used to inputting data.
- COMPUTE, CALCULATE: used for calculation of the result of the given expression.
- ADD, SUBTRACT, INITIALIZE used for addition, subtraction and initialization
- OUTPUT, PRINT, DISPLAY: It is used to display the output of the program
- IF, ELSE, ENDIF: used to make decision
- WHILE, ENDWHILE: used for iterative statements
- FOR, ENDFOR: Another iterative incremented/decremented tested automatically.

Syntax for if else

```
IF (condition) THEN
Statement
...
ELSE
Statement
...
ENDIF
```

Syntax for FOR

```
FOR (start value to end-value) DO
statement
...
ENDFOR
```

Syntax for while

```
WHILE (condition) DO
Statement
...
ENDWHILE
```

Advantages

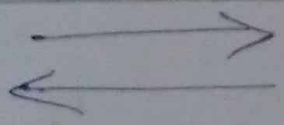
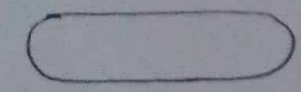
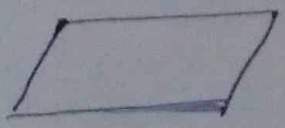
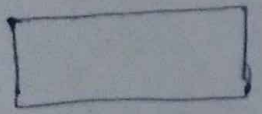
- Pseudo is independent of any language; it can be used by most programmers
- It is easy to translate pseudo code into a programming language.
- It can be easily modified as compared to flowchart

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.

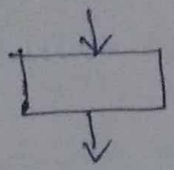
Flow chart

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.

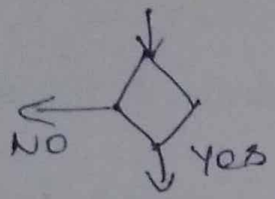
Symbol	Symbol Name	Description
	Flow lines	used to connect symbols
	Terminal	used to start, pause or halt in the program logic.
	Input/output	Represents the information entering or leaving the system
	Processing	Represents arithmetic & logical instructions
	Decision	Represents a decision to be made
	Connector	Used to join different flow lines
	Sub function	used to call function

Rules for drawing a flowchart

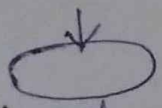
- The flow chart should be clear neat and easy to follow
- The flowchart must have a logical
- only one flow line should come out from a process symbol



- only one flow line should enter a decision symbol. However two or three flow lines may leave the decision symbol.



- only one flow line is used with a terminal symbol



- within standard symbols write briefly and precisely
- Intersection of flow lines should be avoided.

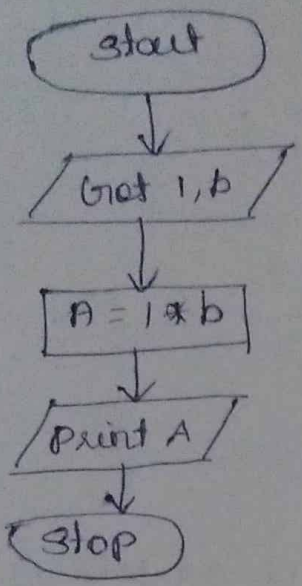
Advantages of flowchart

- communication
- Effective analysis
- Proper documentation
- Efficient coding
- Proper Debugging
- Efficient program Maintanance

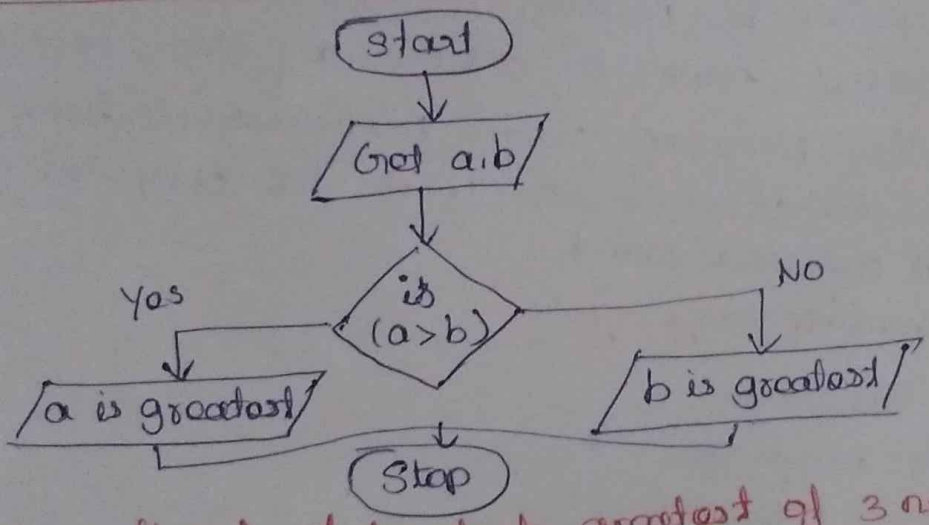
Disadvantages

- complex logic
- Alternations and Modifications
- Reproduction
- cost

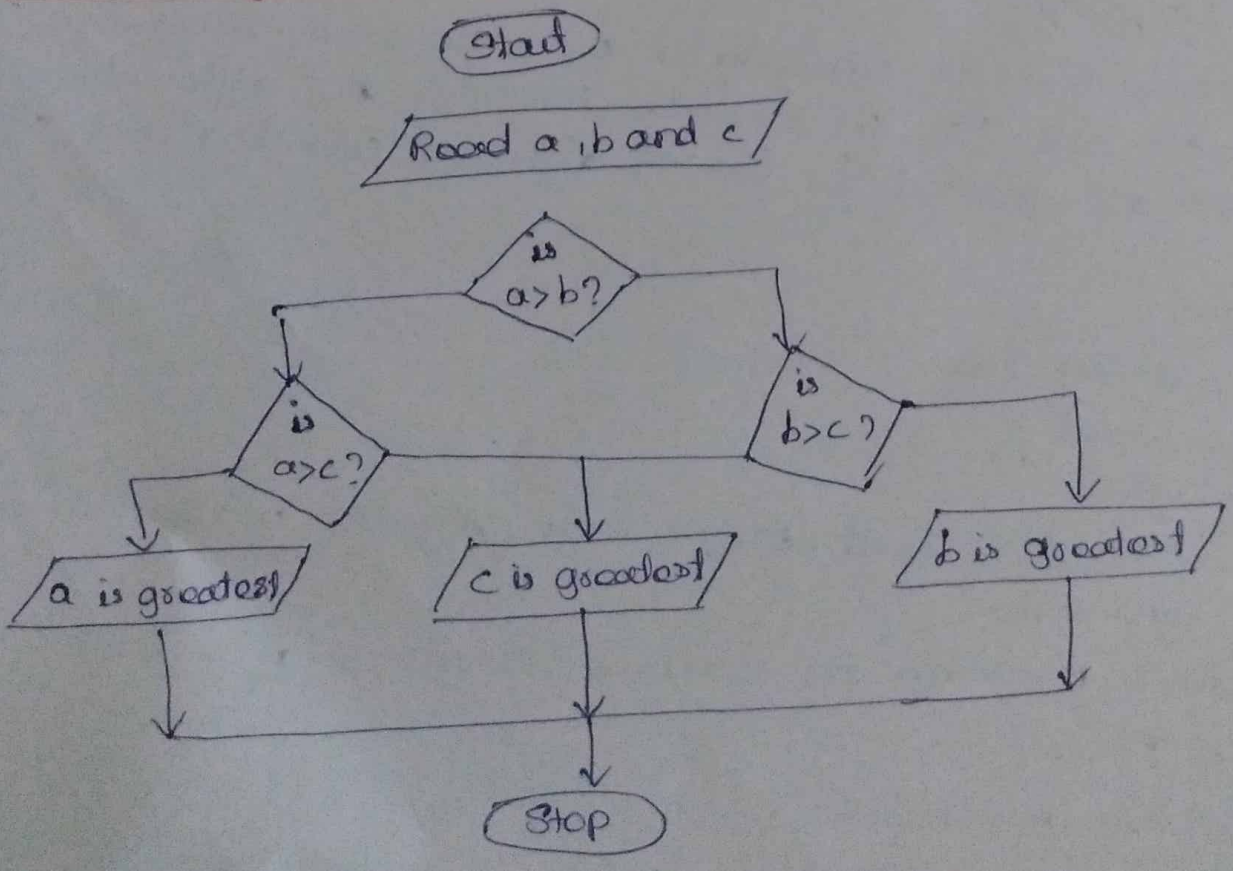
Draw a flowchart to find area of a rectangle



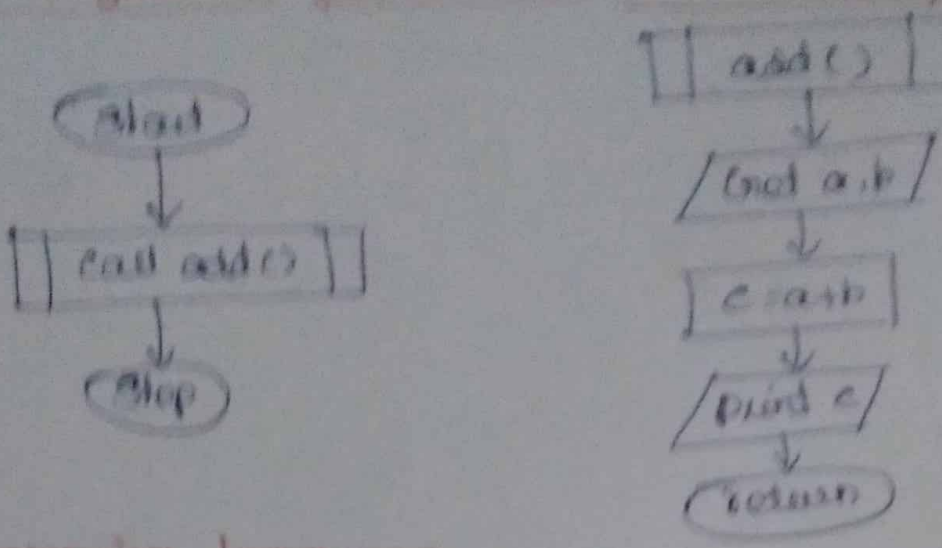
Draw a flowchart to check greatest of two numbers



Draw a flowchart to check greatest of 3 numbers



Draw a flowchart to add two numbers using function



Programming Language

- A programming language is a set of symbols & rules for instructing a computer to perform specific tasks
- The programmers have to follow all the specified rules before writing program using programming language.
- The user has to communicate with the computer using language which it can understand

Types of programming language

- Machine language
- Assembly language
- High level language

Machine language

The computer can understand only machine language which uses 0's and 1's. In machine language the different instructions are formed by taking different combinations of 0's and 1's.

Advantages

- Translation free
- High speed

Disadvantage

- It is hard to find errors in a program written in the machine language.
- Writing program in machine language is a time consuming process.
- Machine dependent

Assembly language

To overcome the issues in programming language and make the programming process easier an assembly language is developed which is logically equivalent to machine language but it is easier for people to read write and understand.

Ex. ADD a,b

Assembler

Assembler is the program which translates assembly language instructions in to a machine language.

Advantage

- Easy to understand and use
- It is easy to locate and corrected errors

Disadvantage

- Machine dependent
- Hard to learn
- less efficient

High level language

High level language contains English words and symbols the specified rules are to be followed while writing program in high level language.

Translating high level language to machine language

The programs that translate high level language into machine language are called interpreter or compiler.

Compiler

A compiler is a program which translates the source code written in a high level language in to object code which is in machine language program. Compiler reads the whole program written in high level language and translates it to machine language. If any error is found it display error message on the screen.

Interpreter

Interpreter translates the high level language program in line by line manner. The interpreter translates a high level language statement in a source program to a machine code and executes it immediately before translating the next statement.

Advantages

- Readability
- Machine independent
- Easy debugging

Disadvantages

The translation process increases the execution time of the program. Program in high level language require more memory and take more execution time to execute.

They are divided into following categories

- Interpreted programming languages
- Functional programming languages
- Compiled programming language
- Procedural programming language
- Scripting programming language
- Markup programming language
- Concurrent programming language
- object oriented programming language

Algorithmic Problem Solving

- understanding the problem
- Ascertain the capabilities of the computational device
- Exact / approximate solution
- Decide on the appropriate data structure
- Algorithm design techniques
- Methods of specifying an algorithm
- Proving an algorithms correctness
- Analysing an algorithm

Simple Strategies for developing Algorithms (Iteration, Recursion)

Iteration

A loop is one or more instructions that the computer performs repeatedly.

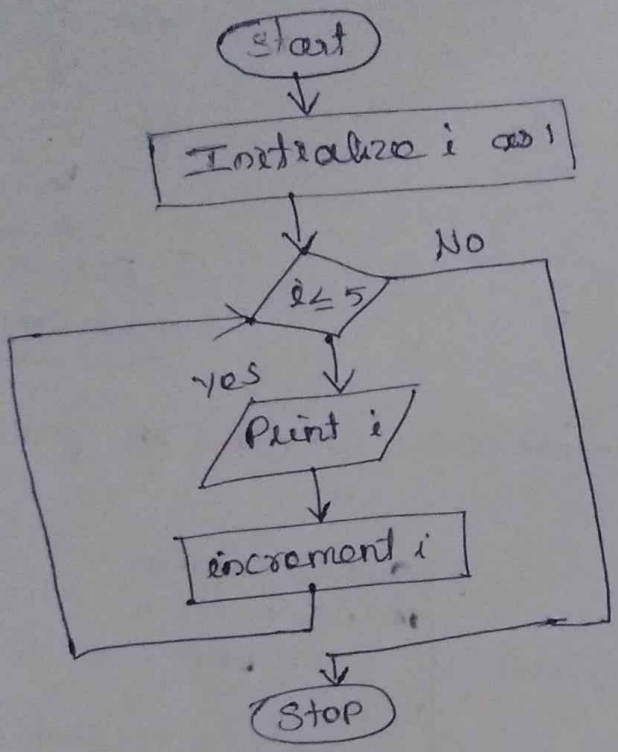
Algorithm

Step1: Start
Step2: Initialize the value of i as 1
Step3: Print i and increment the value of i
Step4: Repeat the step 3 until the value of i <= 5
Step5: Stop.

Pseudocode: write a Pseudocode to print 1 to 5

```
INITIALIZE i to 1
WHILE i<=5
    PRINT i
    i = i+1
END WHILE
```

Flowchart



Recursion

A function that calls itself is known as recursive function and the phenomenon is called as recursion

Algorithm: write an algorithm to find the factorial of a given num

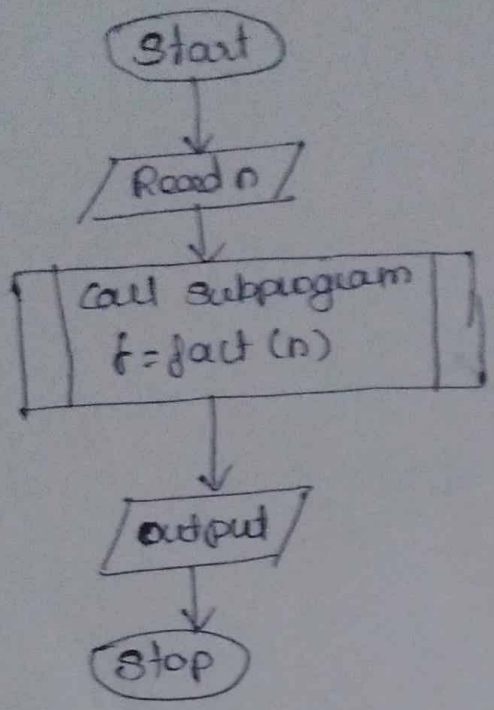
main program

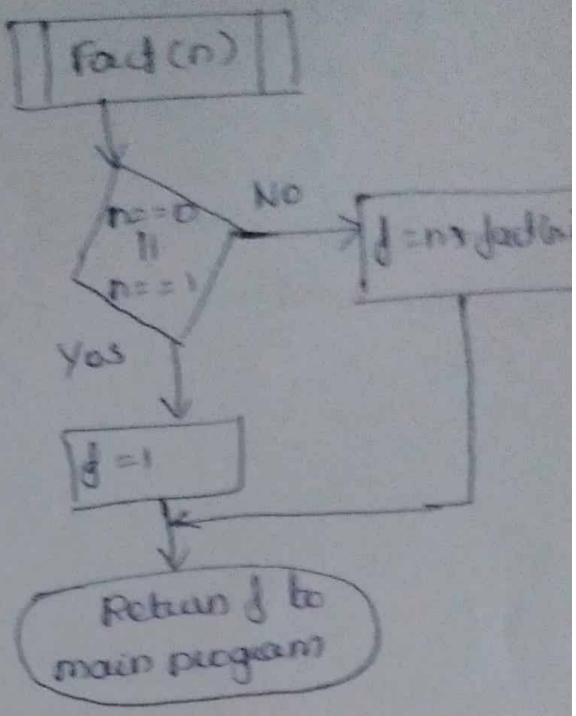
Step1: Start
Step2: read n
Step3: call the sub program as f = fact(n)
Step4: Print f value
Step5: Stop

Sub program

Step 1: if n==0 or n==1 return 1 to the main program
otherwise goto step 2
Step 2: return n * fact(n-1) to main program

Flowchart: Draw a flowchart to find the factorial of a given no





Pseudocode: write a pseudocode to find the factorial of a given numbers

```
main Program
READ n
CALL the sub program as f=fact(n)
PRINT f value
```

```
sub program
IF n==0 OR n==1 THEN
    RETURN 1 to the main program
ELSE
    RETURN n*fact(n-1) to main
                        Program
END IF
```

Illustrative Problems

Find minimum in a list

Algorithm

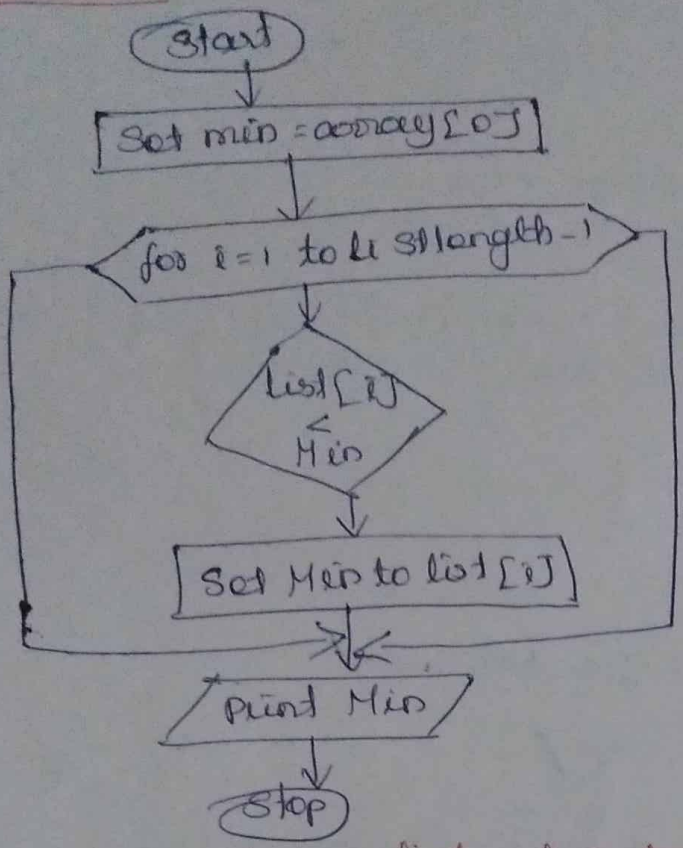
Step 1: Start
Step 2: Read the list of numbers
Step 3: Set Min to list[0]
Step 4: For each number x in the list L compare it to min
    Step 4a: if x is larger assign min to x
Step 5: Print min
Step 6: Stop

Pseudocode

```
READ the list of numbers
SET min to list[0]
FOR i=1 to list length-1
  IF list[i]<min THEN
    SET Min to list[i]
  END IF
```

ENDFOR
PRINT Min

Flowchart



Insert a card in a list of sorted cards

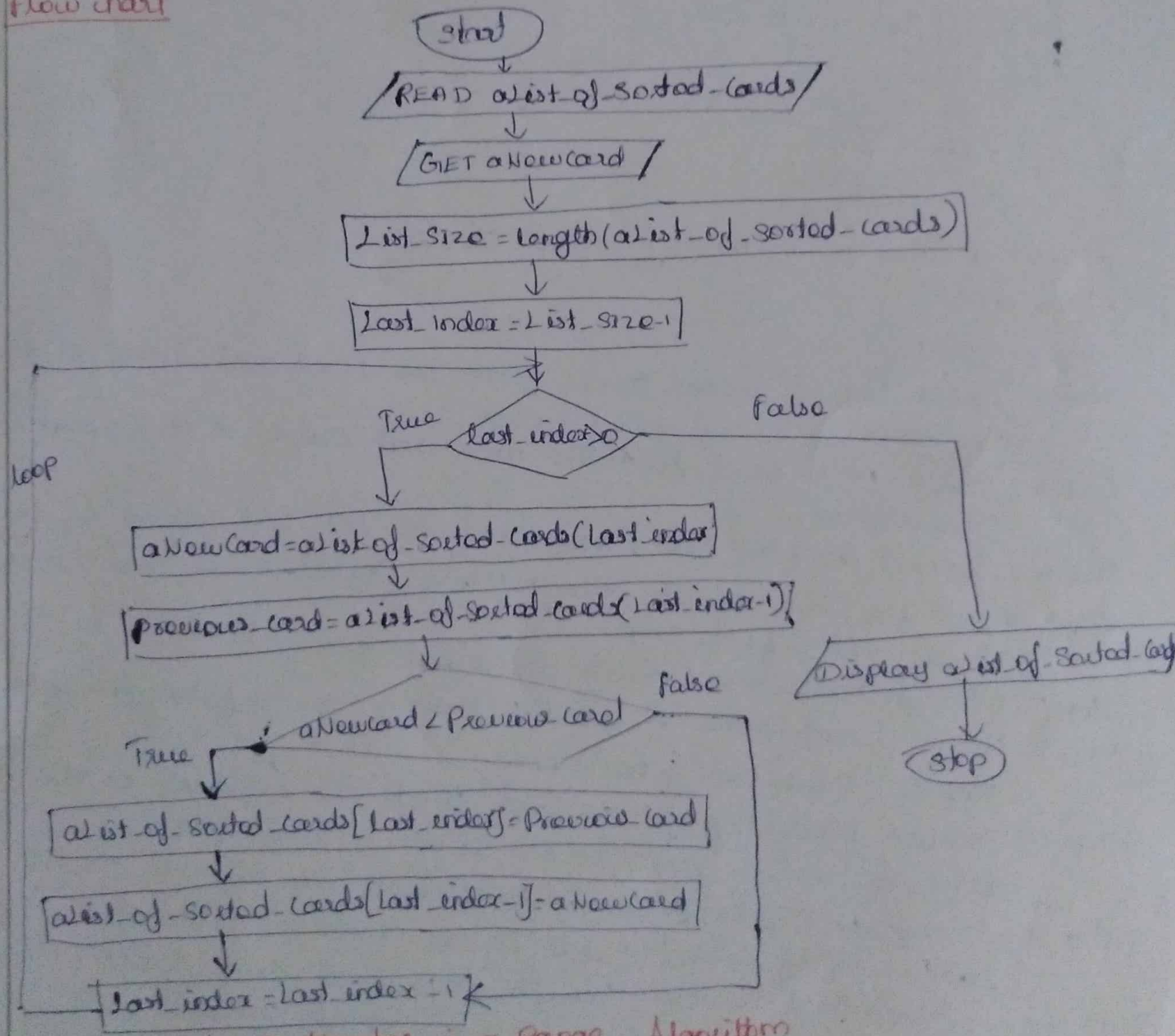
Algorithm

step1 : Start
Step2 : Get a list of sorted cards from the user and store it in variable a list_of_sorted_cards
Step3 : Get a new card from the user and store it in variable aNewCard
Step4 : Append the new card, aNewCard, to the list, aList_of_sorted_cards, and store it
Step5 : Find the size of the list, aList_of_sorted_cards, and store it variable. List_size
step6 : Assign Last_index with value list size - 1
step7 : Check if Last Index is greater than zero. If false the goto step 8 otherwise goto step12
Step8 : Assign variable aNewCard with the element in the list, aList_of_sorted_cards. in the position Last_index
Step9 : Assign variable previous_card with the element in the list, aList_of_Sorted_cards in the position last index-1
step10 : If aNewCard is less than Previous_card then Swap both those values in the list aList_of_Sorted cards
step11 : Display the list, aList_of_sorted_cards
step13 : Stop

Pseudocode

```
GET a_list_of_sorted_cards
GET a_NewCard
APPEND a_NewCard to a_list_of_sorted_cards)
COMPUTE List_Size = length(a_list_of_sorted_cards)
ASSIGN Last_Index = List_Size-1
WHILE Last_Index >=0
    a_NewCard = a_list_of_sorted_cards[Last_Index]
    Previous_card = a_list_of_sorted_cards[Last_Index-1]
    IF a_NewCard < Previous_card
        a_list_of_sorted_cards[Last_Index] = Previous_card
        a_list_of_sorted_cards[Last_Index-1] = a_NewCard
    Last_Index = Last_Index - 1
END WHILE
DISPLAY a_list_of_sorted_cards
```

Flow chart



Guess an integer Number in a Range Algorithm

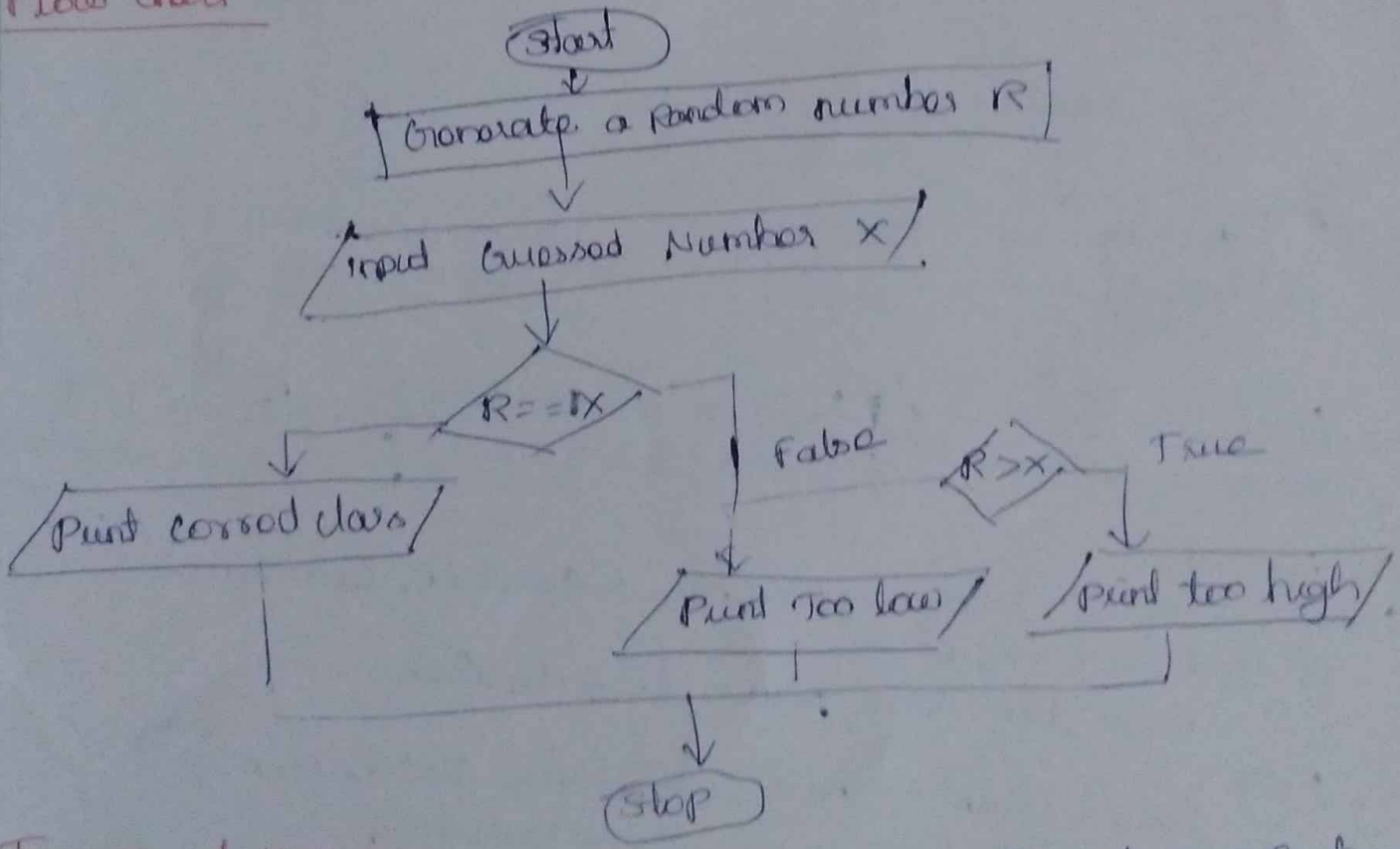
step1: start
step2: Generate a random number in between 1 and 100
step3: Get the guessed number from the user.

step4 : If the random number is greater than the guessed number print guessing is ~~too~~ high and goto step 7 corrod
step5 : Otherwise if the random number is greater than the guessed number print guessing is too high and goto step 7
step6 : otherwise print guessing is too high and goto 7
step 7 : stop.

Pseudocode

GENERATE a random number
READ number from player.
IF random number is equal to number THEN
    PRINT correct
ELSE IF random number is greater than the number THEN
    PRINT too high
ELSE
    PRINT too low
ENDIF

Flow chart



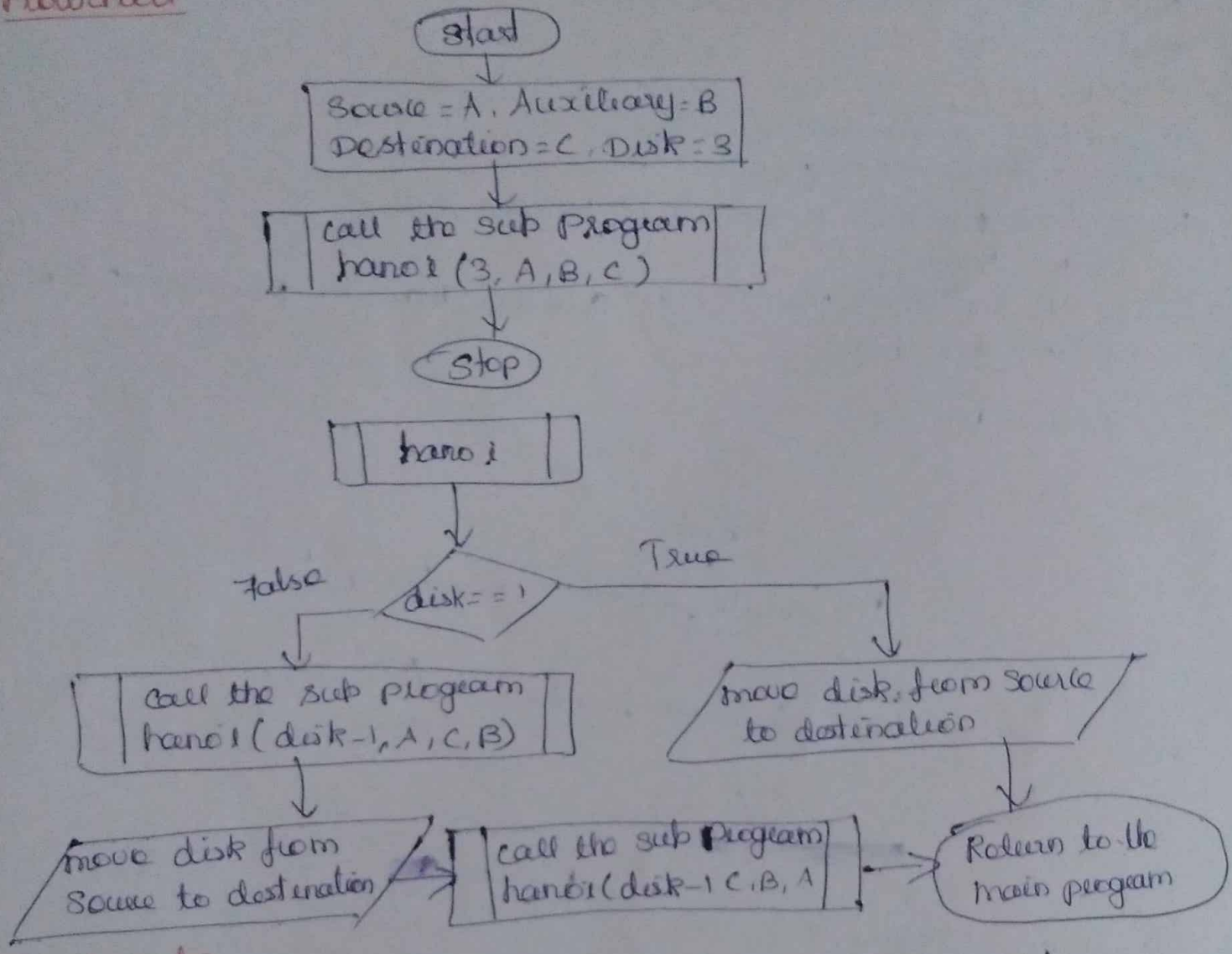
Towers of Hanoi

Towers of Hanoi is a mathematical puzzle where we have 3 rods and n disks. The objective of the puzzle is to move the entire stack to another rat, obeying the following simple rules
- only one disk can be moved at a time
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack ie, a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk

Algorithm

step1: Move n-1 disks from source to auxiliary
step2: Move n^{th} disk from source to destination
step3: Move n-1 disks from auxiliary to destination

Flowchart



Pseudocode

```
SUBPROCEDURE hanoi (disk, source, destination, auxiliary)
   IF disk == 1 THEN
      move disk from source to destination
   ELSE
      hanoi (disk-1, source, auxiliary, destination)
      move disk from source to destination
      hanoi (disk-1, auxiliary, destination, source)
   END IF
END PROCEDURE
CALL hanoi WITH 3, A, B, C.
```

Example

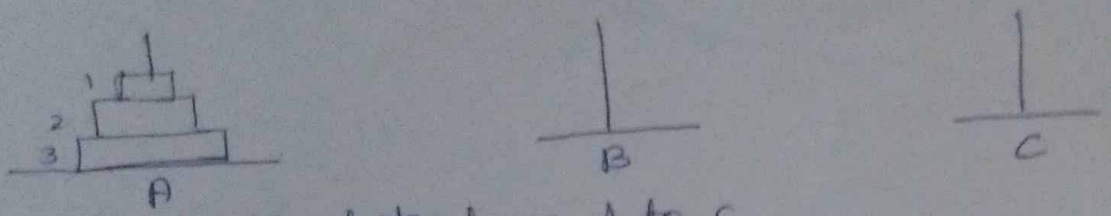
Solution for Tower of Hanoi Problem with 3 Disks

Three Rods – A, B, C
A – Source B – Auxiliary C – Destination

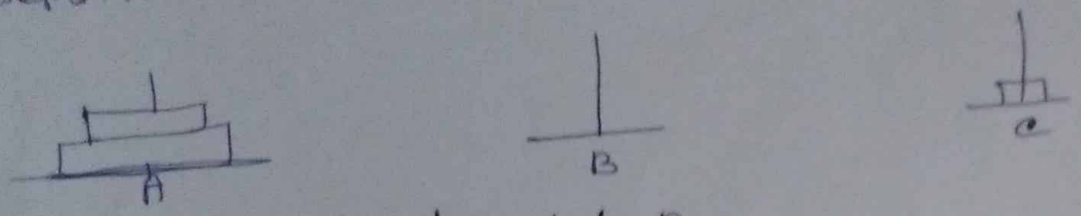
Objective

To move the disk from source to destination i.e., from A to C

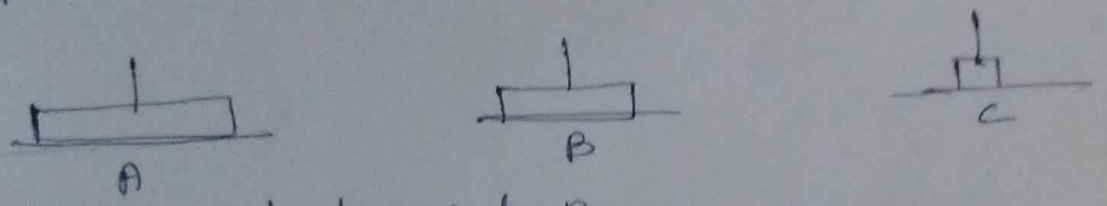
Step1: Initially 3 disk are placed in A



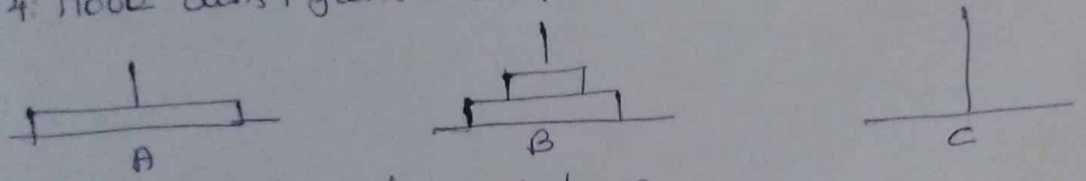
Step2: Move the disk1 from A to C



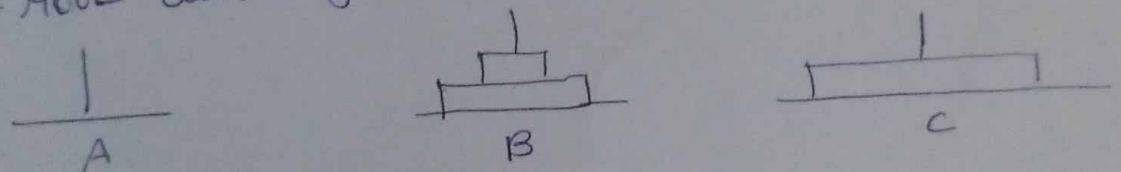
Step3: Move disk2 from A to B



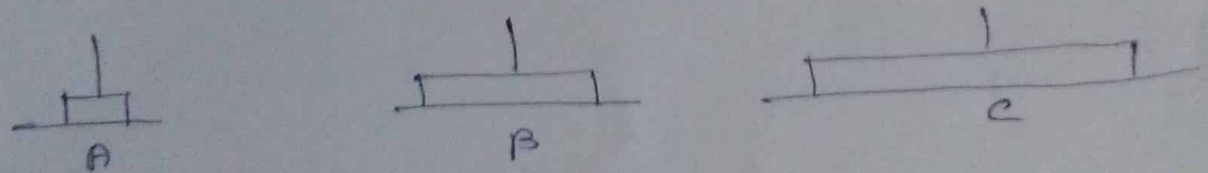
Step4: Move disk1 from C to B



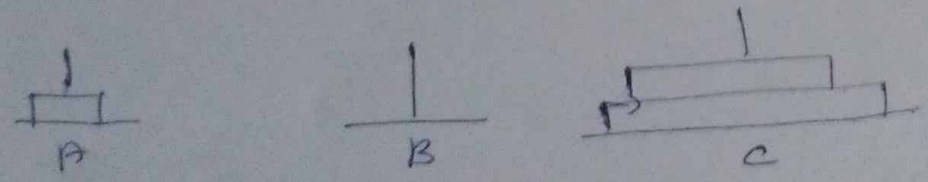
Step5: Move disk3 from A to C



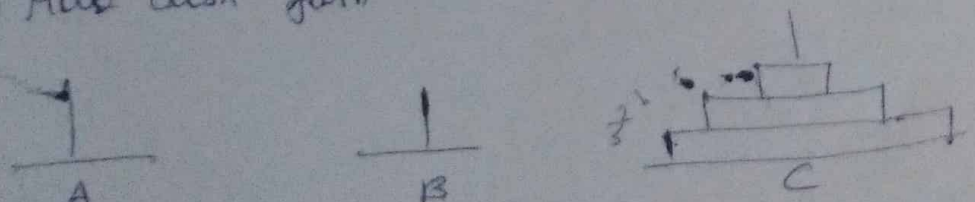
Step6: Move disk1 from B to A



Step7: Move disk2 from B to C



Step8: Move disk from



UNIT-II

Data, 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 point

Python interpreter and interactive mode

Python interpreter

Interpreter executes the instructions directly without previous compiling a program into machine-language instructions Python is a interpreted language because it is executed by the interpreter.

Invoking the interpreter

On windows machines, the python installation is usually placed in C:\Python32. It can be changed while running the installer.

The command quit() can be used to exit the interpreter

The interpreter when called with standard input it reads and executes commands interactively when called with standard input it reads and executes a script from that file.

Some Python modules are also useful as scripts

when a script file is used to run the script it enters the interactive mode.

Modes of python Interpretor

- Shell mode or Interactive mode
- Program mode or Script mode

Shell mode or interactive mode

When commands are read the interpreter is said to be in interactive mode

The interpreter prints a welcome message stating its version numbers and a copyright notice before printing the first prompt

```
$ Python 3.2
Python 3.2 (Py3K, Sep 12 2007, 10:21:02)
```

[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on linux2
Type "help", "copyright", "credits" or "license" for more information
>>>

In shell mode, Python expressions are typed into the Python shell and the interpreter immediately shows the result

```
>>> 2+3
5
```

In this mode it prompts for the next command with the Primary prompt, usually 3 greater than signs (>>>) >>> is also called as Python prompt or chevrons.

```
>>> a = 1
>>>
```

For continuation lines, it prompts with the secondary prompt by default 3 dots (...)

Example

```
>>> if (a>0):
... Print ("positive Number")
...
positive number
```

Values and types

Python sets the variable type based on the value that is assigned to it. Python will change the variable type if the variable value is set to another value

Python has 5 standard datatypes. They are

- Number
- String
- List
- Tuple
- Dictionary

Number

Numbers are created by numeric literals. Numeric objects are immutable which means when an object is created its value cannot be changed. Python will automatically convert a number from one type to another value if it needs

Python has 3 distinct numeric types

- Integers
  - Boolean
- Floating point numbers
- Complex numbers

String

A String type object is a sequence (left-to-right order) of characters. Strings start and end with single or double quotes. Python strings are immutable ie, once a string is generated characters within the string cannot be modified

```
>>> type ('python language')
<class 'str'>
```

List

A list is a container which holds comma, separated values b/w square brackets where all the items or elements need not to have the same type. A list without any element is called an empty list. Python lists are mutable

```
>>> list1 = ['computer', 2018, 8.25, 'Python']
>>> type (list1)
<class 'list'>
```

Tuple

Tuple is an ordered sequence of items same as list The only difference is that tuples are immutable Tuples once created cannot be modified

```
>>> a = (1, 'python')
>>> typ(a)
<class 'tuple'>
```

Dictionary

It is an unordered collection of items with Key: value pairs.

Variables

It is a location in memory used to store some data They are given unique names to differentiate b/w different memory location

Keywords cannot be used as variable names

Expressions & statements

It is a combination of values, variables & operators A value itself can also be considered as an expression and a variable can also be considered as an expression

```
>>> 42
42
>>> n = 17
17
>>> n + 25
42
```

A statement is a unit of code that has an effect like creating a variable or displaying a value.

Tuple Assignment

It is an ordered sequence of items same as list The only difference is that tuples are immutable - Tuples once created cannot be modified.

Tuple Assignment is often useful to swap the values of 2 variables Tuple assignment is more elegant

```
>>>a, b = b, a
```

Example

```
# Python program to swap 2 numbers
a = 5
b = 10
Print ("Before Swapping")
Print ('a =', a, "b =", b)
# swap the values
a, b = b, a
Print ("After swapping")
Print ("a =", a, "b =", b)
```

Operators in Python

The types of operators in Python are

- Arithmetic
- Comparison
- Logical
- Bitwise
- Assignment
- Special

Arithmetic

They are used to perform mathematical operations like addition, subtraction, multiplication etc

operator	Meaning	Example
+	Add 2 operand or unary plus	x + y
-	subtract right operand from the left or unary minus	x - y
*	multiply 2 operands	x * y
/	divide left operand by right one	x / y
%	modulus - remainder of the division of left operand by the right	x % y
**	Exponent - left operand raised to the power of right	x ** y

Example

```
x = 15
y = 4
Print (x+y)
Print (x/y)
Print (x%y)
Print (x**y
```

o/p

```
19
3.75
3
50625
```

comparison operators

They are used to compare values

operator	Meaning	Example
>	Greater than	x > y
<	Less than	x < y
==	Equal to	x == y
!=	Not equal to	x != y
>=	Greater than or equal to	x >= y
<=	less than or equal to	x <= y

logical operators

They are the and, or, not operators

operator	Meaning	Example
and	True if both operands are true	x and y
or	True if either of the opr is true	x or y
not	True if operand is false	not x

Bitwise operator

It's act on operands as if they were string of binary digits
It operates bit by bit

operator	Meaning	Example
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x >> 2
<<	Bitwise left shift	x << 2

Assignment operator

They are used in Python to assign values to variables

operator	Example
=	x = 5
+=	x += 5
-=	x -= 5

Special operators

Python language offers some special type of operators like the identity operator or the membership operator.

Identity operators

is and is not are the identity operators. They are used to check if 2 values are located on the same part of the memory.

Example

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
print(x1 is not y1)
print(x2 is y2)
```

O/P

```
False
True
```

Membership operators

in and not in are the membership operator. They are used to test whether a value or variable is found in a sequence

Example

```
x = 'Hello world'
y = {1:'a', 2:'b'}
print('H' in x)
print('a' not in y)
```

O/P

```
True
False
```

Precedence of operators

When an expression contains more than one operator the order of evaluation depends on the order of operations or precedence of operators.

Comments

As programs get bigger & more complicated, they are difficult to understand. So notes can be added to the programs in natural language for better understanding. These notes are called comments & they start with the # symbol.

Example

```
#compute the percentage of the hour
percentage = (minute * 100)/60
```

Modules

A module is a collection of functions that are grouped together in a single file.

Functions in a module are usually related to each other.

There are 2 types of modules

Built-in Modules

User Defined Modules

Importing Modules

Before using a function in the module, import the module using import keyword.

Syntax

import modulename

Example

import math

Once module is imported built in help function can be used to see the contents of that module.

Syntax

help(modulename)

Example

```
>>> help(math)
```

Help on built in module math :-

NAME
math
FILE
(built-in)
DESCRIPTION
It provides access to the mathematical functions
FUNCTIONS
sqrt(x)
Return the square root of x
pow(x, y)
Return x**y (x to the power of y)

By combining the module's name & function's name using a dot, the function in the module can be used.

```
>>> math.sqrt(9)
3.0
```

In built-in math module floor function rounds a number down

In user-defined module house floor function calculates a price for a given area

```
>>> import math
>>> import house
>>> floor (22.7)
```

The first floor function rounds a number down which is specified in math module. The second floor function calculates a price given an area which is specified in house module

The value can also be assigned to variables which is imported from modules.

```
>>> import math
>>> math.pi = 3 # default value is 3.14 Here 3 is assigned to pi
>>> radius = 5
>>> print ("circumference is", 2 * math.pi * radius)
circumference is 30
```

Combining module's name with the names of the function can also be used.

```
>>> from math import sqrt
>>> sqrt (9)
3.0
```

Unimport / Reimport a module

once a module has been imported it stays in memory until the program ends.

There are ways to "unimport" a module or to reimport a module that has changed while the program is running

Defining your own modules

Python allows us to define our own modules.

The name of the module is the same as the name of the file but without the .py extension

Example

```
# Python program to create own modules
# File Name: area.py
import math
def distance(xc, yc, xp, yp):
    dx = xp - xc
    dy = yp - yc
```

```
    r = math.sqrt(dx**2 + dy**2)
    return r
def circle(radius):
    return (math.pi * radius * radius)
```

After creating this file, the file can be imported like any other module.

```
>>> import area
>>> area.circle(distance(2,3,3,4))
6.28
```

Functions

A function is a block of organized reusable code that is used to perform a single related action.

Functions provide better modularity for application and a high degree of code reusing

There are 2 types of functions

- Built-in functions
- User defined functions

Function Definition

In Python before using the functions in the program, the function must be defined.

Syntax

```
def name of function (list of formal parameters):
        body of function
```

Example

```
def max(x,y):
    if x > y:
        return x
    else:
        return y
```

name of function - max

list of formal parameters - x, y

Function use

A function call is an expression that has a value which is the value returned by the invoked function

When the function is used the formal parameters are bound to the actual parameters of the function invocation

Example The function invocation

```
max(3,4)
```

binds x to 3 and y to 4

The execution of a return statement terminates the execution of the function

Example

```
#python program to add 2 numbers using function
def add(x,y):
    return x+y
a=int(input("enter a"))
b=int(input("enter b"))
c=add(a,b)
print("Addition:",c)
```

x,y - formal parameters a,b - actual parameters

Flow of execution

It refers to the order with which the statements in the program run.

Execution always begins at the first ~~stat~~ statement of the program. Statements are run one at a time in order from top to bottom.

Function call is like a route in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statement there, & then comes back to pick up where it left off.

parameters & Arguments

Inside the function the arguments are assigned to variables called parameters. The function that takes an argument is as follows

```
def Print_twice(bruce):
    Print(bruce)
    Print(bruce)
```

This function assigns the argument to a parameter named bruce. when the function ~~that~~ that is called, it prints the value of the parameter twice.

This function works with any value ~~that~~ that can be printed

```
>>> Print_twice('Spam')
Spam
Spam
>>> Print_twice(42)
42
42
```

Illustrative Programs

Exchange the values of two variables

```
x = int(input("enter x value:"))
y = int(input("enter y value:"))
print("Before swapping")
print("x=",x,"/=",y)
# create a temporary variable temp
# swap 2 values from 2 different variables
temp=x
x=y
y=temp
print("After swapping")
print("x=",x,"y=",y)
```

output

```
enter x value
10
enter y value
20
Before swapping
x=10  y=20
After swapping
 x=20  y=10
```

circulate the values of n variables

```
a=[1,2,3]
print("entered list:",a)
j=len(a)-1
while j>0:
    temp=a[j]
    a[j]=a[j-1]
    a[j-1]=temp
    j=j-1
print("circulated list:",a)
```

output

```
entered list: [1,2,3]
circulated list: [3,1,2]
```

Distance b/w 2 points

formula: Distance $= \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$

Program using function

```
import math
def calculate_distance(x1,y1,x2,y2):
    dist = math.sqrt((x2-x1)**2+(y2-y1)**2)
    print(dist)
calculate_distance(2,4,6,8)
```

output

```
5.656854249492381
```

Program using list

```
import math
p1=[4,0]
p2=[6,6]
distance = math.sqrt(((p1[0]-p2[0])**2)+((p1[1]-p2[1])**2))
print(distance)
```

output

```
6.324555320334
```

UNIT - III

control flow, function

conditionals

Boolean Values and operators

Boolean values

A boolean expression is an expression that is either true or false. True & false are special values that belong to the type bool. they are not strings

Ex

```
>>> type (true)
<class 'bool'>
>>> type (false)
<class 'bool'>
```

The == (equal to) operator is one of the relational operators. The operator == compares two operands and returns True if they are equal otherwise it returns false.

Ex

```
>>> 5 == 5
true
>>> 5 == 6
False
```

Operators

comparison operators (Relational operators)

comparison operators are used to compare values. It either returns true or false according to the condition.

operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

Example

```
x = 10
y = 12
Print (x > y)
Print (x < y)
Print (x != y)
```

Logical operators

Logical operators are and, or, not operators. In python, any non zero number is interpreted as true.

operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example

```
x = true
y = false
Print (x and y)
Print (x or y)
Print (not x)
```

o/p

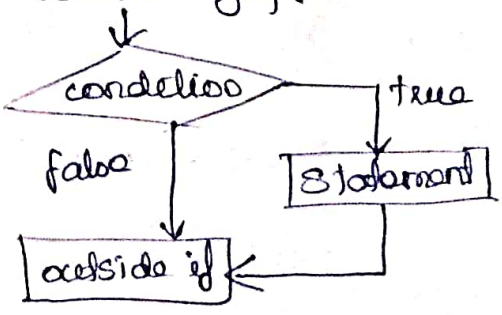
```
False
true
False
```

conditional (if)

conditional statements give us the ability to check conditions and change the behavior of the program accordingly.

Syntax

```
if booleanexpression:
    statements
```



Example

```
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.

if statements have the same structure as function definitions a header followed by an indented body. statements like this are called compound statements

Ex

```
if x<0:
    pass
```

Program

```
x = int (input ("enter a number"))
if (x>0)
    Print ("positive Number")
```

o/p

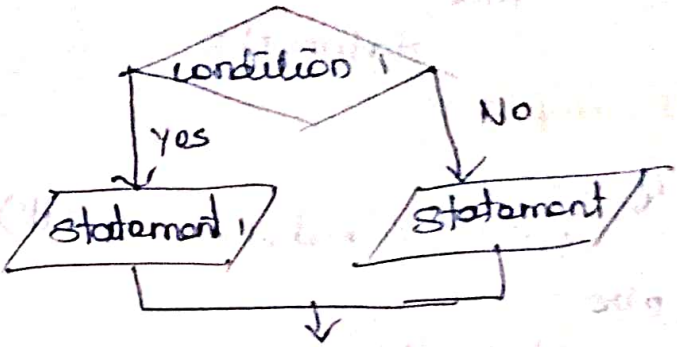
Enter a number
3
positive Number

Alternative (if else)

A second form of the if statement is "alternative execution" in which there are two possibilities and the condition determines which on runs.

Syntax

```
if booleanexpression:
        statement 1
else
        statement 2
```



Example

```
if x % 2 == 0
        Print ("x is even")
else:
        Print ("x is odd")
```

chained conditional (if-elif-else)

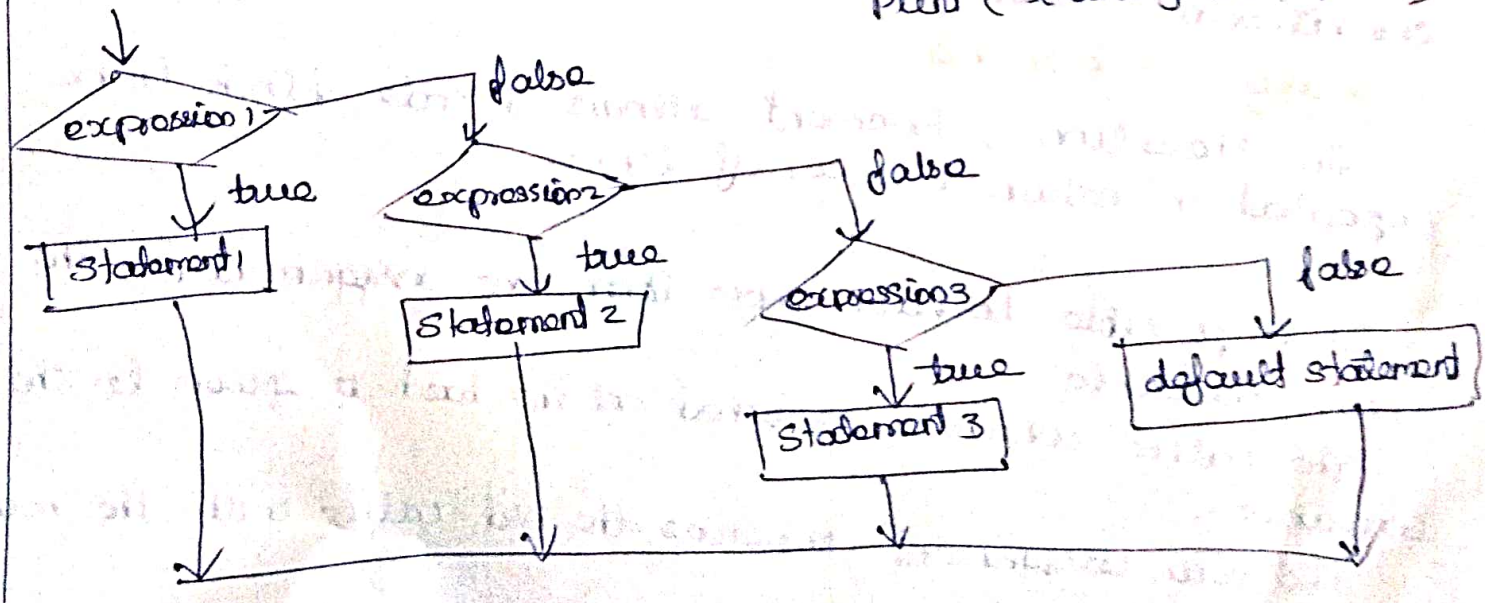
If there are more than 2 possibilities, more than 2 branches is needed, so chained conditional is used.

Syntax

```
if booleanexpression 1:
        statement 1
elif booleanexpression 2:
        statement 2
else:
        statement 3
```

Example

```
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")
```

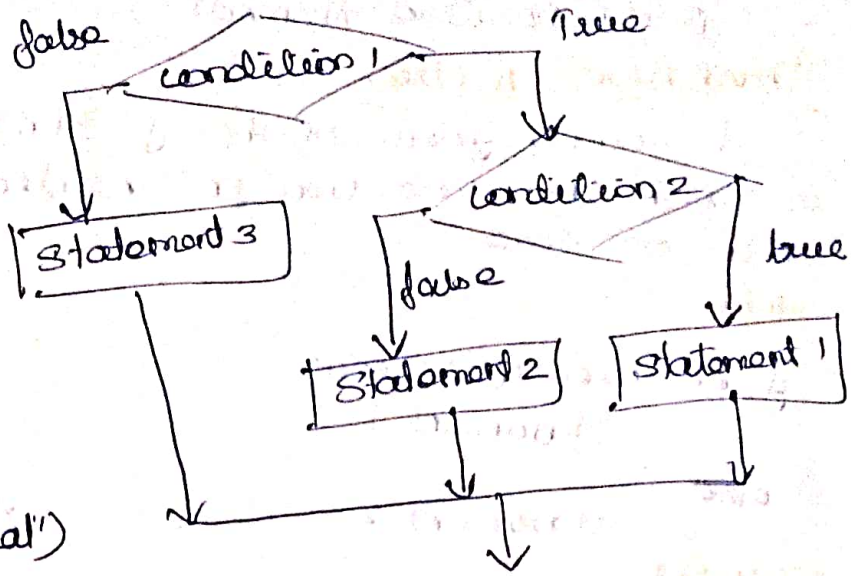


Nested conditionals

conditionals can be nested within another

Syntax

```
if booleanexpression1:
    statement1
else:
    if booleanexpression2:
        statement2
    else:
        statement3
```



Example

```
if x == y:
    Print("x and y are equal")
else:
    if x < y:
        Print('x is less than y')
    else:
        Print ("x is greater than y")
```

The outer conditional contains two branches. The first branch contains a simple statement

The second branch contains another if statement which has two branches of its own.

Those 2 branches are both simple statements although they could have been conditional statement as well.

Logical operators provide a way to simplify nested conditional statements.

Iteration statements

An iteration statement allows a code block to be repeated a certain number of times.

State

It is possible to have more than one assignment for the same variable.

The value which is assigned at the last is given to the variable.

The new assignment replaces the old value with the new value.

Example

```
x = 5
y = 3
a = 4
Print(x)
Print(y)
```

o/p

4
3

While

A while loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

```
while expression:
    statement(s)
```

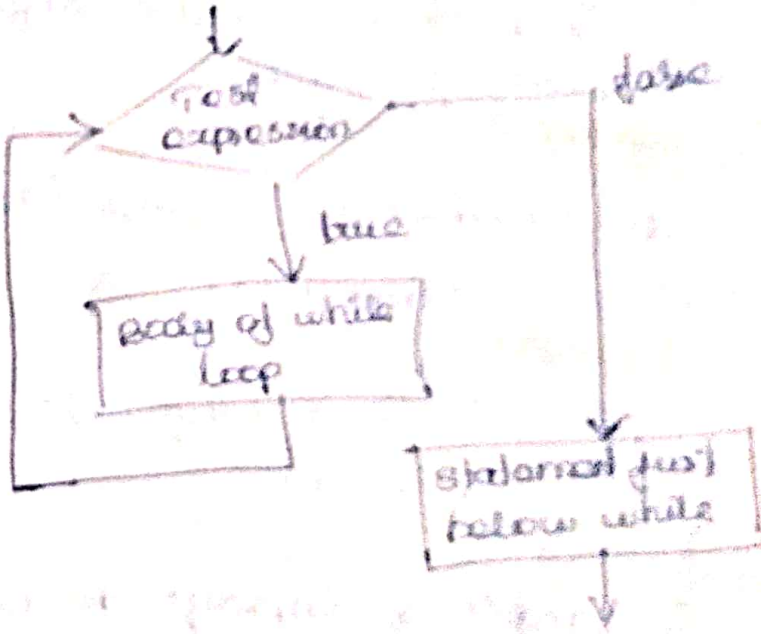
Here, statements may be a single statement or a block of statement.

The condition may be any expression and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false program control passes to the line immediately following the loop.

Program

```
count = 0
while(count < 9):
    Print ("The count is:", count)
    count = count + 1
Print ("Good bye")
```



o/p

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye
```

For

For statement iterates over the items of any sequence in the order that they appear in the sequence

Syntax

```
for <variable> in <Sequence>:
        <statement>
else:
        <statement>
```

The items of the sequence object are assigned one after the other to the loop variable to be precise the variable points to the items the body of the loop is executed.

Example

```
>>> languages = ["c", "c++", "perl", "Python"]
>>> for x in languages:
        Print (x)
```

output

```
c
c++
Perl
Python
```

The range() function

The built in function range() is used to iterate over a sequence of numbers.

Syntax

```
for variablename in range (start value, end value, increment/
                                                    decrement val):
        Statement(s)
```

Example

```
>>> for i in range(5):
        Print (i)
```

o/p

```
0
1
2
3
4
```

range(5,10) displays the numbers from 5 to 9 //5,6,7,8,9

range (0,10,3) displays the numbers from 0 to 10 by incrementing by 3 // 0,3,6,9

range (-10,-100,-30) displays the numbers from -10 to -100 by decrementing by 30, -10, -40, -70)

Break

It breaks out of the innermost enclosing for or while loop

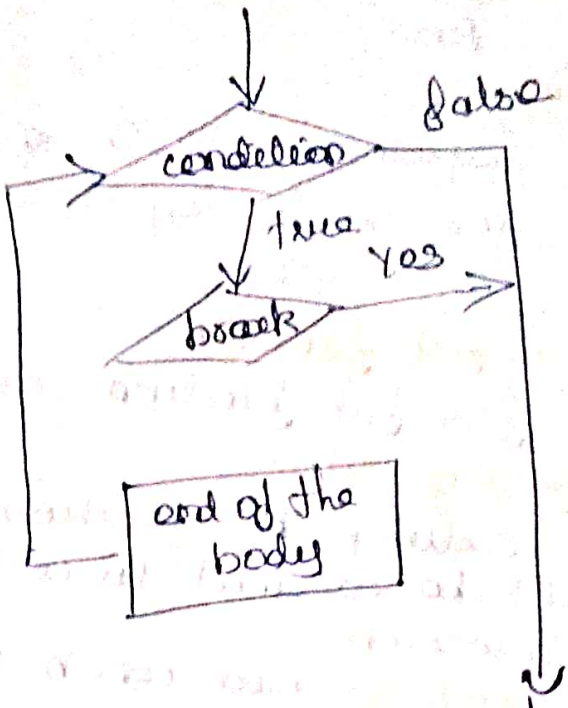
The break statement breaks the loop statement and transfers flow of execution to the statement immediately following the loop.

Syntax

```
break
```

working of breaking statement

```
for variablename in sequence
    if condition
        break
statement # outside loop
```



Program

```
for val in "string"
    if val == "i":
        break
    Print (val)
Print ("end")
```

o/p

```
s
t
r
end
```

continue

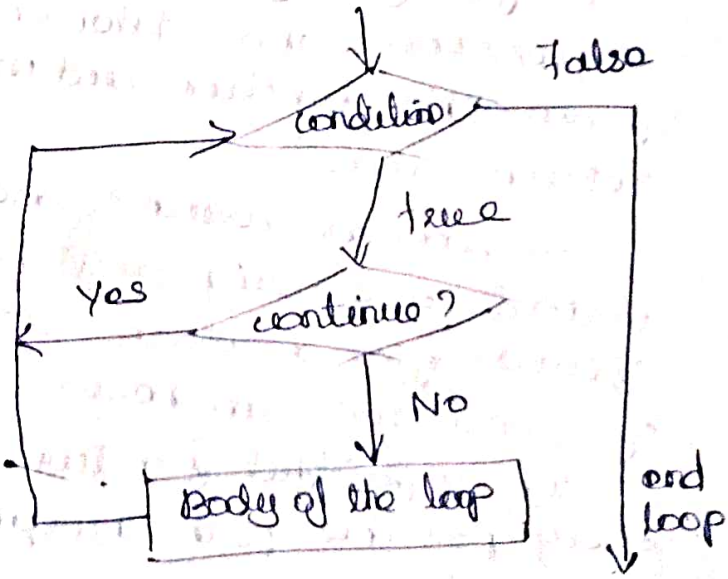
It causes the loop to skip the rest of the body and immediately retest its condition before reiterating

Syntax

```
continue
```

working of continue statement

```
for variablename in sequence
    if condition:
        continue
statement # outside loop
```



Program

```
for val in "string":
    if val == "i":
        continue
    Print (val)
Print ("end")
```

o/p

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

Pass

The pass statement do nothing. It can be used when a statement is required syntactically but the program requires no action.

The pass statement can be used in places where the program code cannot be left as blank but that can be written in future.

Syntax

pass

Program

sequence = {'p', 'a', 's', 's'}

for val in sequence

pass

o/p

no output will be displayed

Fruitful function

fruitful function are functions that return values

Return values

Calling the function generates the return value, which will be assigned to a variable or use as part of an expression.

Example area which returns the area of a circle with the given radius

```
def area (radius)
    a = math.pi * radius * * 2
    return a
```

In fruitful function the return statement includes an expression. This statement means "Return immediately from this function and use the following expression as a return value."

As soon as return statements runs the function terminate without executing any subsequent statement. 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.

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

This function is incorrect because if x happens to be 0 neither condition is true and the function ends without hitting a return statement

Parameters

Inside the function the arguments are assigned to variables called parameters. The function that takes an argument is as follows

```
def Print_twice(bruce):
    Print(bruce)
    Print(bruce)
```

This function assigns the argument to a parameter named bruce when the function is called, it prints the value of the parameter twice

```
>>> Print_twice('Spam')
Spam
Spam
>>> Print_twice(42)
42
42
```

Local and Global Scope

All variables in the program may not be accessible at all locations in that program This depends on the location of declaration statement. Scope of a variable determines the portion of the program where a variable can be accessed There are 2 scopes

- Local Scope
- Global Scope

Local Variables

- Variables declared inside the function. Its lifetime and scope is within a block/function
- Cannot access outside the function
- If try to access then error will be displayed by interpreter

Global Variables

- Variables declared in main function or above fun definition
- Its lifetime is within the full program

Program

```
a=10
b=5
def add():
Print("a+b",(a+b))
def sub():
a=3
b=2
Print("a-b",a-b)
```

Function composition

This is the ability to call one function from within another function. It is the way of combining functions such that the result of each function is passed as the argument of the next function.

The composition of 2 functions f and g is denoted by f(g(x)). x is the argument of g and the result of g(x) is the argument for f

Program

```
import math
def distance (xc, yc, xp, yp):
    dx = xp - xc
    dy = yp - yc
    r = dx ** 2 + dy ** 2
    radius = math.sqrt (r)
    return radius
def area (radius):
    return (math.pi * radius * radius)
xc = int (input ("enter xc"))
yc = int (input ("enter yc"))
xp = int (input ("enter xp"))
yp = int (input ("enter yp"))
a = area (distance (xc, yc, xp, yp))
Print ("Area of circle:", a)
```

Recursion

Recursion function is a function which calls itself again and again until the condition is true. It has a termination condition

Advantages

- Simplicity
- The length of the program can be reduced.
- A complex task can be broken into simpler subproblems using recursion
- Sequence generation is easier

Disadvantages

- The logic behind recursion is sometimes hard to understand.
- Recursive calls are inefficient as they take more memory & time.
- Recursive function are hard to debug.

Example

- Factorial
- Fiboracci Series

Infinite recursion

It happens when recursive function call fails to stop. The program with infinite recursion never terminates. Python display the error message on infinite recursion.

Example

```
def display():
    display()
display()
```

Strings

A string is a sequence of zero or more values enclosed within single or double quotes.

Ex

S = 'WELCOME' S1 = "Hello123"

An empty string contains no characters and has a length 0

Ex: S = ""

The index starts at 0 from the left end and -1 from the right end

Ex:
```
>>> S = "Hello"
>>> Print (S[0])
H
>>> Print (S[-1])
O
```

String Slices

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

The operator [n:m] returns the part of the string from the n^{th} character to the m^{th} character including the first but excluding the last.

If the first index is omitted the slice starts at the beginning of the string. // S[:3]

If the second index is omitted the slice goes to the end of the string. // S[3:]

Syntax

Stringname[start index : end index]

stringname[start index : end index : increment / decrement value]

Example

```
>>> s = "Hello Python"
>>> Print (S[1:3])
el
>>> Print (S[2:9:2])
lopt
>> Print (S[3:])
lo Python
```

Immutability

Strings are immutable. It means that modification is not allowed in strings.

The use of [] operator on the left side of an assignment operator is to change a character in a string.

Ex
```
>>> greeting = 'Hello World!'
>>> greeting[0] = 'J'
```

Error will be displayed. The reason for the error is that strings are immutable which means existing string cannot be modified.

One solution is to create a new string that is a variation on the original.

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

String Functions and methods

String methods

Strings provide methods that perform a variety of useful operations.

Syntax

stringname.functionname(parameter)

This form of dot notation specifies the name of the method and the name of the string to which it is applied.

The empty parentheses indicate that this method takes no arguments. A method call is called as invocation

Ex
```
s.lower()
len(s)
```

upper()

Returns a string with all uppercase letters

stringname.upper()

lower()

Returns a string with all lowercase letters

stringname.lower()

len()

Returns the length of the given string

len(stringname)

find()

Return the index of given substring

- If the given substring is not found then it returns -1
- By default find() starts at the beginning of the string

Syntax

```
Stringname.find(substring)
Stringname.find(substring, startindex)
Stringname.find(substring, startindex, endindex)
```

index()

Returns index of substring

Syntax

```
Stringname.index(substring)
```

Ex

```
>>> s="Hello"
>>> Print(s.index('e'))
1
```

islower()

Returns true if all alphabets in the given string are in lowercase otherwise returns false

Syntax

```
stringname.islower()
```

Ex

```
>>> s="hello"
>>> Print(a.islower())
True
```

isupper()

Returns true if all alphabets in the given string are in uppercase otherwise returns false.

Syntax

```
Stringname.isupper()
```

Ex

```
>>> s="hello"
>>> Print(s.isupper())
False
```

count()

Returns the number of occurrences of the given substring

Syntax

```
stringname.count(substring)
```

replace()

Returns a copy of the string where old substring is replaced with the new substring. The original string is unchanged.

Syntax

```
Stringname.replace(oldstring, newstring)
```

split()

splits string from left

Syntax

```
Stringname.split()
```

join()

Returns a concatenated string

casefold()

converts casefolded strings

Syntax

```
Stringname.casefold()
```

Lists as Arrays

Arrays

Array is a collection of values of same datatype

List

- List is a collection of values of different datatype
- Arrays and lists are both used in Python to store data
- They both can be used to store any data type
- indexed & iterated.

Example

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

o/p

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

Similarities b/w arrays & lists

- Both are mutable
- Arrays & lists are indexed & sliced identically
- The len command works just as well on arrays
- Both have sort & reverse attributes

Illustrative Programs

Square root

```
import math
n=int(input("enter a number"))
s = math.sqrt(n)
Print("square root", s)
```

o/p

```
enter a number 4
The square root is 2.0
```

GCD of two numbers

```
num1 = int(input("enter first number"))
num2 = int(input("enter second number"))
if num1>num2:
    smaller = num2
else:
    smaller = num1
for i in range(1,smaller+1)
    if (num1 % i == 0) and (num2 % i == 0):
        gcd = i
Print("The GCD of", num1, "and", num2, "is", gcd)
```

output

```
Enter first number : 6
Enter second number : 2
The GCD of 6 and 2 is : 2
```

Sum an Array of Numbers

```
n= int (input ("enter the range "))
l=[ ]
for i in range (0,n):
    n= int (input ("enter numbers %d :" %i))
    l.append (n)
s=0
Print ('Array of Numbers ", l)
for i in range (0, len (l)):
    S = S+l[i]
Print (" sum of Array of Numbers', s)
```

o/p

```
Enter the range 5
enter number 0 :1
enter number 1 : 2
enter number 2 : 3
enter number 3 :4
enter number 4 :5
Array of Numbers [1,2,3,4,5]
Sum of Array of Numbers : 15
```

Linear Search

```
items=[ ]
n= int ( input ("enter the no of items"))
l=0
for i in range (0,n):
    a = int (input ("enter the item %d :" %i))
    items. append (a)
Key = int (input ("enter search value "))
for i in range (0,n):
    if (items[i] == Key):
        l=1
if (l==1):
    Print ("Element found")
else:
    Print ("element not found")
```

o/p

```
enter the no of items : 3
enter the item 0 : 1
enter the item 1 : 2
enter the item 2 : 5
enter search value : 2
element found
```

Binary Search

```
items=[]
n=int(input("enter the no of items"))
f=0
mid=0
for i in range(0,n):
    a=int(input("enter the item%d:"%i))
    items.append(a)
key=int(input("enter search value "))
first=0
last=len(items)-1
while(first<=last):
    mid=(first+last)//2
    if(items[mid]==key:
        f=1
        break
    elif(items[mid]<key):
        first=mid+1
    else:
        last=mid-1
if(f==1):
    Print("element found")
else:
    Print("element is not found")
```

Output

```
enter the no of items:3
enter the item0:1
enter the item1:2
enter the item2:5
enter search value:2
element found.
```

UNIT - IV

Lists, Tuples, Dictionaries

List

A list is a sequence of values of any type.
The values in a list are called elements or items

a = [2, 3.14, True, '3']

List index

- Any integer expression can be used as an index
- To read or write an element that does not exist index Error will be displayed
- If an index has a negative value it counts backward from the end of the list

Creating of list

To create a list the elements are enclosed in square brackets. A list that contains no elements is called an empty list. This can be created with empty brackets [].

Syntax

[elements] or []

Example

['c', 2, 3.14, True] or List1 = []

Assigning list values to variables

List values can be assigned by using listname

Syntax

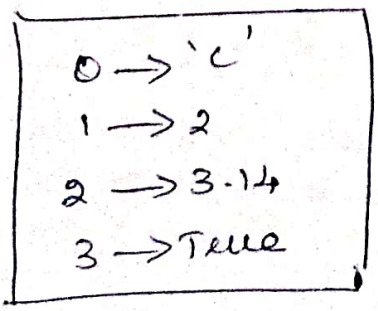
listname = [list of values]

Example

```
>>> List1 = ['c', 2, 3.14, True]
>>> Print (list1)
```

o/p

['c', 2, 3.14, True]



Accessing values in list

List values can be accessed by using listname followed by index or indices within square brackets

Syntax

listname[index]

Example

```
>>> list1 = ['Python', 'hello', 2014]
>>> print ("list1[0]:", list1[0])
```

o/p

list1[0]: Python

Deleting elements in the list

There are several ways to delete elements from a list

pop()

It modifies the list & returns the element that was removed. If an index is not given it deletes and returns the last element

Syntax

```
variable name = listname.pop(index)
```

Example

```
>>> t=['a','b','c']
>>> x = t.pop(1)
>>> t
['a','c']
>>>x
'b'
```

del

If the index is known and the removed value is not needed, del operator can be used to delete the element at the given index

Syntax

```
del listname[index]
```

Example

```
>>> t = ['a','b','c']
>>>del t[1]
>>> t
['a','c']
```

remove()

If the element to be deleted is known and the removed value is not needed, remove() can be used to remove the given element

Syntax

```
listname.remove(element)
```

Example

```
>>> t = ['a','b','c']
>>> t.remove('b')
>>> t
['a','c']
```

List operations

The + operator concatenates lists

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

The * operator repeats a list a given number of time

```
>>> [0] * 4
[0, 0, 0, 0]
```

This repeats [0] four times

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

This repeats the list [1, 2, 3] three times

List slices

The slice operator also works on list

Syntax

listname[start index : end index]

Example

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
```

If the first index is omitted, the slice starts at the beguning

```
>>> t[:4]
['a', 'b', 'c', 'd']
```

If the second index is omitted the slice goes to the end

```
>>> t[:4]
['a', 'b', 'c', 'd']
```

```
>>> t[3:]
['d', 'e', 'f']
```

List methods

Python provides methods that operate on lists

append()

It adds new element to the end of a list

Syntax

listname.append(element)

Example

```
>>> t = ['a', 'b', 'c']
```

extend()

It takes a list as an argument and appends all of the elements

Syntax

listname1.extend(listname2)

Example

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

sort()

arranges the elements of the list in ascending order

Syntax

listname.sort()

Example

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

reverse()

used to reverse the entire list

Syntax

listname.reverse()

Example

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.reverse()
>>> t
['a', 'b', 'e', 'c', 'd']
```

count()

returns the number of occurences of the given substring

Syntax

listname.count(element)

insert()

used to insert the given element in the given position

pop()

removes and returns the last element in the list.

listname.pop()

pop(index)

removes and returns the element at given index in the list. listname.pop(index)

remove()

If the element to be deleted is known and the removed value is not needed, remove() can be used to remove the given element.

listname.remove(element)

Traversing a list or List loops

The most common way to traverse the element of a list is with a for loop. This is used to read the elements of the list.

Syntax

for variable in listname:

Print(variablename)

Example

list 1 = ['python', 'hello', 2017]

for i in list 1:

Print(i)

- The built in functions range and len can also be used
- The function len returns the number of elements in the list
- The function range returns a list of indices from 0 to n-1 where n is the length of the list.

Example

list 1 = ['python', 'hello', 2017]

for i in list 1:

Print(i)

numbers = [1, 2, 3, 4]

for i in range(len(numbers)): number[i] = numbers[i]*2

This loop traverses the list and updates each element

A for loop over an empty list never runs the statements inside the loop.

Mutability

Lists are mutable. This means the elements in the list can be modified.

The syntax for accessing the elements is to use the index value inside the square bracket

-The index starts at 0 from the left and and -1 from the right and.

```
>>> list1 = [25, 10]
>>> Print (list 1)
[25, 10]
>>> List 1[1]=5
>>> Print (List 1)
[25, 5]
```

List1 → 0 → 25, 1 → 10

List 1 → 0 → 25, 1 → ~~10~~ 5

Aliasing

The association of a variable with an object is called a reference.

```
>>> a = [5, 10]
>>> b= a
>>> b is a
True
```

0 - 25
1 - 10

Cloning lists

- Cloning operation is used to create a copy of existing list
- The copy contains the same elements as the original list
- The change made in one copy of list will not affect the other

Syntax

newlistname = list(oldlistname)

Example

```
a = [5, 10]
b: List (a)
Print ("Original List : a)
print ("cloning List : b)
```

o/p

original list [25, 0]

List parameters

when a list is passed to a function the function gets a reference to the list. If the function modifies the list list will be modified.

Example

delete_head removes the first element from a list

```
def delete_head(t):
    del t[0]
>>> z = ['a','b','c']
>>> delete_heads(z)
>>> letters
```

Lists and Strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters list() can be used.

Syntax

```
listname = list(stringname)
```

Example

```
>>> s = 'Hello'
>>> t = list(s)
>>> t
['H','e','l','l','o']
```

The list function breaks a string into individual letters. To break a string into words, the split method can be used.

```
>>> s = 'Hello world'
>>> t = s.split()
>>> t
['Hello', 'world']
```

An optional argument called a delimiter specifies which characters to use as word boundaries. Example uses a hyphen as a delimiter

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

List comprehension

comprehensions are constructs that allows sequences to be built from other sequences List comprehension can be used for filtering contain following parts -

- An input sequence
- Variables representing members of the input sequence
- An optional predicate expression
- An output expression producing elements of the output

Syntax

[expression for item in list if conditional]

This is equivalent to

```
for item in list:
    if condition:
        expression
```

Example

```
>>>[i * 2 for i in range(5) if i == 1]
```

o/p

16

Tuples

A tuple is a sequence of values. The values can be any type and they are indexed by integers Tuples are immutable tuple is a name of a built in function so it cannot be used as Keywords

Example

```
>>> t1 = ('a', 1, 2.0, True)  or  t1='a', 1, 2.0, True
>>> Print (t1)
('a', 1, 2.0, True)
```

Creation of tuples

A tuple is created by placing all the items inside a parentheses (), separated by comma. The parentheses are optional

A tuple is a comma-separated list of values

Syntax

```
tuplename = sequence of values
```

Example

```
t = t = 'a', 1, 2.0, True
```

Creation of tuples with single element

To create a tuple with a single element, final comma should be included.

Syntax

```
tuplename = value,
```

Example

```
>>> t1='a'
>>> type(t1)
<class 'tuple'>
```

A value in parenthesis is not a tuple

```
>>> t2=('a')
>>> type(t2)
<class 'str'>
```

Accessing elements in the tuple

In tuple the index starts from 0 from the left end and -1 from the right end. The index operator [] to access an item in a tuple where the index starts from 0

Syntax

```
tuplename[index]
```

example

```
>>> t=('a','b','c','d','e')
>>> t[0]
'a'
>>> t[-1]
'e'
```

Slicing

The slice operator can also be used in tuples. It selects a range of elements.

Syntax

```
tuplename[index-range]
```

Example

```
>>> t[1:3]
('b','c')
```

Assigning values to variables

Tuples are immutable so elements in the tuple cannot be modified and also values cannot be assigned to variables

after creation

Deletion of tuples

Tuples can be deleted using keyword del. Since tuples are immutable, individual elements cannot be deleted.

Tuple method

Since tuple is immutable add or remove elements in tuple is not possible

- count (element)
- index (element)

Program

```
>>>m=('a','p','p','l','e')
>>>Print (my_tuple.count ('p'))
2
>>> Print (my_tuple.index ('l'))
3
```

Tuple assignment

It allows a left side tuple to be assigned values from right side tuple. Tuple Assignment is often useful to swap the values of 2 variables

```
>>> a,b = b,a
```

Program

```
x=int (input ('enter x value'))
y= int (input ("enter y value"))
Print (" before swapping")
Print ("x=",x,"y=",y)
x,y = y,x
Print ("After swapping")
Print ("x=",x,"y=",y)
```

o/p

```
enter x value
10
enter y value
20
before swapping
  x=10  y=20
After swapping
  x=20  y=10
```

Tuples as return values

The left side is a tuple of variables, the right side is a tuple of expressions. Each value is assigned to its respective variable.

All the expressions on the right side are evaluated before any of the assignments.

Program

max and min are built-in functions that find the largest and smallest elements of a sequence. maximum computes both and returns a tuple of 2 values

```
def maximum(t):
    return max(t), min(t)
t = (20, 50, 0, 100)
Print ('Tuple', t)
maximum, minimum = maximum(t)
Print ("Maximum", maximum)
Print ("Minimum", minimum)
```

o/p

Tuple (20, 50, 0, 100)
Maximum 100
Minimum 0

Variable - length argument tuples

- Functions can take a variable number of arguments
- A parameter name that begins with * gathers arguments into a tuple

Example

```
>>> t = (7, 3)
>>> divmod(*t)
(2, 1)
```

Advantages of Tuples

- Generally used for heterogeneous datatypes
- Since tuple are immutable, iterating through tuple is faster than with list
- Tuples contain immutable elements can be used as key for a dictionary

Dictionary

- Dictionary is a mutable, associative data structure of variable length
- Dictionary is an unordered collection of items with key value pair.
- Dictionaries are optimized to retrieve values when the key is known
- In a dictionary, items can be of any type

Example

```
d = {1: 'apple', 2: 'ball'}
```

Creation of dictionary

A dictionary is created by placing the items inside curly braces { } separated by comma.

An item has a key and the corresponding value expressed as a key:value pair.

Syntax

```
dictionaryname = {key:value pairs}
```

Example

```
>>>d = {1:'apple', 2:'ball'}
>>>print(d)
{1:'apple', 2:'ball'}
```

Accessing elements

Dictionary uses keys to access the elements. Key can be used either inside square brackets or with the get() method. If the key is not found get() returns None instead of KeyError

Syntax

```
dictionaryname[key]
dictionary.get(key)
```

Adding/updating elements in dictionary

Dictionary are mutable. Using assignment operator, new items can be added to dictionary or existing items can be modified.

Syntax

```
dictionaryname[key] = value
```

Deleting elements in dictionary

There are 4 ways to delete elements from dictionary

- del
- pop()
- popitem()
- clear()

Dictionary operations & methods

dict() - creates a new dictionary

dict(sequence) - creates a new dictionary with key values and their associated values

len() - used to return the length of the dictionary

items() - returns a list of items (key:value) pairs

keys() - Returns a list of keys in dictionary
values() - Returns the list of values in the dictionary
pop(key) - Return and remove the item associated with the given key in a dictionary
popitem() - can be used to remove and return an arbitrary item (key,value) from the dictionary
clear() - All the items in the dictionary can be removed at once.
get(key) - returns the element associated with the given key
sorted(dictionary) - returns the sorted list of keys in the dictionary
key in d - used to check whether the given key is found in dictionary

Illustrative program

Selection Sort

```
def SelectionSort(alist):
    pos=0
    for i in range(0,len(alist)):
        for j in range(i+1,len(alist)):
            if alist[pos]>alist[j]:
                pos=j
        temp=alist[i]
        alist[i]=alist[pos]
        alist[pos]=temp
        print("Iteration:",i+1)
        pos=i+1
alist=[10,0,3,5,3]
print("before sorting",alist)
SelectionSort(alist)
print("After sorting",alist)
```

o/p

Before sorting [10, 0, 3, 5, 3]
After sorting [0, 3, 5, 8, 10]

Insertion sort

```
def insertionSort(alist):
    for index in range(1,len(alist)):
        currentvalue=alist[index]
        position=index
        while position and alist[position-1]>currentvalue:
            alist[position]=alist[position-1]
            position=position-1
```

alist[position] = current value

```
alist = [10,0,8,5,3]
Print ("before sorting", alist)
insertion sort (alist)
Print ("After sorting", alist)
```

O/P

Before sorting [10,0,8,5,3]
After sorting [0,3,5,8,10]

Mergesort

```
def mergesort (alist):
    if len(alist)>1:
        mid = len(alist)
        left half = alist[:mid]
        right half = alist[mid:]
        mergesort (left half)
        mergesort (right half)
        i=0
        j=0
        K=0
        while i < len(left half) and j < len(right half):
            if left half[i] < right half[j]:
                alist[K] = left half[i]
                i = i+1
            else:
                alist[K] = right half[j]
                j = j+1
            K = K+1
        while i < len(left half)
            alist[K] = left half[i]
            i = i+1
            K = K+1
        while j < len(right half):
            alist[K] = right half[j]
            j = j+1
            K = K+1
alist = [10,0,8,5,3]
Print ("before merge sort:", alist)
mergesort (alist)
Print ("After merge sort : alist)
```

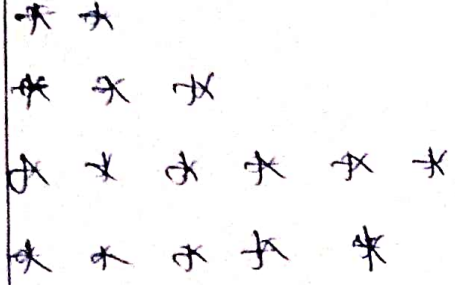
output

before merge sort [10,0,8,5,3]
After merge sort [0,3,5,8,10]

Histogram

```
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



UNIT-5

Files, Modules, Packages

File

A file is a collection of data stored in a particular area on the disk. To keep the data permanent files are used.
There are 2 types of file

- Text file
- Binary file

Text files

- A text file is a sequence of characters stored on a disk
- In Python the default file type is text file.

File modes

modes	description
r	opens a file for reading only
rb	opens a file for reading only in binary format
r+	opens a file for both reading & writing
rb+	opens a file for both reading and writing in binary format.
w	opens a file for writing only
wb	opens a file for writing only in binary format
w+	opens a file for both writing and reading
wb+	opens a file for both writing and reading in binary format
a	opens a file for appending
ab	opens a file for appending in binary format
a+	opens a file for both appending and reading
ab+	opens a file for both appending and reading in binary format.

File operations

A file operations take place in the following order.

- open a file

- Read or write
- close the file

opening a file

The built-in function open() is used to open a file. It returns a file object that provides methods for working with the file.

Syntax

```
fileobject = open(filename, mode)
```

Example

```
>>> fout = open('output.txt', 'w')
```

fileobject - fout
filename - output.txt
mode - w (write mode)

Reading and writing

In order to perform read or write operation in a file, a file must be opened. After performing all operations file must be closed.

write operation

The built in function write() is used to write data into the file. It returns the number of characters that were written to the file.

To write a file you have to open it with mode 'w'

If the file already exists opening it in write mode clears out the old data. If the file doesn't exist a new one is created.

Syntax

```
fileobject = open(filename, 'w')
fileobject.write(string)
```

Example

```
>>> fout = open('output.txt', 'w')
>>> line1 = "Problem Solving"
>>> fout.write(line1)
```

It returns the number of characters that were written to the file

If the file is called again to perform write operation it adds the new data to the end of the file

Example

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

Read operation

The built in function read() is used to read the contents from the file

To perform read operation the file should be opened with mode 'r'

Syntax

```
fileobject = open(filename, 'r')
fileobject.read()
```

Example

```
>>> fout = open('output.txt', 'r')
>>> fout.read()
'Problem Solving and Python Programming'
```

Closing a file

- After performing file operations, file must be closed
- The built in function close() is used to close the file

Syntax

```
filevariablename.close()
```

Example

```
>>>fout.close()
```

File methods

Method	Description
close()	close an open file. It has no effect if the file is already closed.
read(n)	Read atmost n characters from the file. Reads till end of file if it is negative or zero.

readable()	Returns a true if the file stream can be read from
readline(n, ...)	Read and return one line from the file. Reads in atmost n bytes if specified.
readlines(n, ...)	Read and return a list of lines from the file. Reads in at most n bytes / characters if specified.
tell()	Returns the current file location
writable()	Returns true if the file stream can be written to.
write(s)	Write string to the file and return the number of characters written.
writelines(lines)	Write a list of lines to the file

Format Operator

The write() accept only string as arguments. To give other values in write(), the values must be converted to string.

The built-in function str() is used to convert other values to strings

Syntax

fileobject.write(str(variable name))

Example

```
>>> f = open('output.txt', 'w')
>>> x = 52
>>> f.write(str(x))
```

format operator

An alternative is to use the format operator %. The first operand contains one or more format sequence, which specify how the second operand is formatted. The result is a string

format sequence

%d - to format an integer

%g - to format a floating point number

%s - To format a string

The format sequence %d means that the second operand should be formatted as a decimal integer.

Example

```
>>> x = 15
>>> '%d' % x
'15'
```

The result is the string '15'

A format sequence can appear anywhere in the string

Example

```
>>> a = 30
>>> print ("The bag contains %d apples" %a)
The bag contains 30 apples
```

If there is more than one format sequence in the string the second argument has to be a tuple

Example

```
>>> "Ram scored %d %s in sem %d" %(9, "GPA", 1)
'Ram scored 9 GPA in sem 1'
```

Command line arguments

- Command line arguments are the arguments passed into the program in the command line
- While executing the program command line arguments are given along with program name.
- In Python during supporting command line arguments are stored in sys module.
- While using command line arguments, sys module should be imported

import sys

- sys.argv is the list in sys module, which contains the command line arguments passed to the program.

Syntax

len (sys.argv)

Program

To find the occurrence of each word (word count) in a text file using command line arguments

exercise

```
Print ('finding word count using command line Arguments')
import sys
if len(sys.argv) != 2:
    Print ('error filename missing')
    sys.exit()
filename = sys.argv[1]
file = open(filename, 'r')
wordcount = { }
for word in file.read().split():
    if word not in wordcount:
        wordcount[word] = 1
    else:
        wordcount[word] += 1
for k,v in wordcount.items():
Print (k,v)
```

Sample txt

```
This is Python Program
```

Output

```
open the command prompt (start->cmd)
>Python exio.py sample.txt
finding word count using command line Arguments
This 1
is 1
Python 1
Program 1
```

Errors and Exceptions

Errors

It is also known as bugs They are always the mistake made by programmer There are two types of error

- logical error
- Syntax error

Logical error

Logical errors occurs due to the mistake in the logic of the program

Examples

- using integer division instead of floating point division
- Indenting the block in a wrong level.

Program

```
i=1
n=0
while (i<=5)
        n=n+2
        i=i+1
print ('Result:',n)
```

output

```
Result : 0
```

Syntax error

In python syntax errors are identified only during program execution. It display error message and stop without continuing execution process.

Examples

- misspelling a keyword
- leaving out a symbol such as colon, comma, brackets
- putting a keyword in a wrong place

Exceptions are also called as runtime errors. Exceptions are unusual conditions that occurs during execution

Examples

- Division by zero
- performing operations with incompatible types
- using an identifier which has not been defined
- accessing a element which is not in list or tuple or dictionary

- Access to a file or file object which does not exist

Program

```
a=10
Print(b)
```

output

NameError : b is not defined

Exception Name	Description
Name Error	Raised when an identifier is not found in the local or global namespace.
Syntax Error	Raised when there is an error in Python syntax
Type Error	Raised when an operation or function is attempted that is invalid for the specified data type
Key Error	Raised when the specified key is not found in the dictionary
Index Error	Raised when an index is not found in a sequence

Handling Exceptions

- The simplest way to handle exceptions is with a "try-except" block
- Exceptions that are caught in try blocks are handled in except blocks
- If an error is encountered a try block code execution is stopped and control transferred down to except block

Syntax

```
try:
    statements
    - - - - -
except Exceptional:
    if there is exceptional then execute this block.
```

else

If there is no exception then execute this block

The try statement works as follows

First the try clause is executed

If no exception occurs the except clause is skipped and execution of the try statement is finished.

If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed and then execution continues after the try statement.

If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found it is an unhandled exception and execution stops with a message.

Example

```
a,b=5,0
try:
  z=a/b
except:
    Print("Mathematic Exception")
else:
    Print("successfully Done")
```

output

Arithmatic Exception

Declaring multiple Exception

Multiple exceptions can be declared using the same exception statement

Syntax

```
try:
  statements
```

except Exception1, Exception 2, ... Exception N:
    execute this code in case any Exception of these occurs
else:
    execute code in case no exception occurred.

Example

```
try:
    a=10/0;
except Arithmatic Error, Standard Error:
    Print ("Arithmetic Exception")
else:
    Print ("successfully Done")
```

Finally

Finally block will always be executed irrespective of the execution

Syntax

```
try:
    statement
finally
    code which is must to be executed
```

Example

```
try:
    a=10/0;
    Print "exception occured"
finally:
    Print "code to be executed"
```

Modules

A module is a collection of functions that are grouped together in a single file

Functions in a module are usually related to each other. There are 2 types of modules

- Built in modules
- User Defined modules

Importing modules

Before using a function in the module, import the module using import keyword.

Syntax

import modulename

Example

```
>>> help(math)
```

Help on built-in module math:

```
NAME
math
FILE
(built-in)
DESCRIPTION
It provides access to the mathematical functions.
FUNCTIONS
sqrt(x)
    Return the square root of x
pow(x,y)
    Return x**y (x to the power of y).
```

By combining the module's name and function's name using a dot, the function in the module can be used

```
>>> math.sqrt(9)
3.0
```

Example

In built-in math module, floor function rounds a number down.

```
>>> import math
>>> import house
>>> floor(22.7)
```

It is not clear that the above floor function is taken from math module or house module. So function's name should be joined with the module's name using dot.

Example

```
>>> import math
```

```
>>> import house
>>> math.floor(22.7)
>>> house.floor(22.7)
```

The values can also be assigned to variables which is imported from modules.

```
>>> impod math
>>> math.pi = 3
>>> radius = 5
>>> print("circumference is", 2 * math * pi * radius)
circumference is 30
```

Combining the module's name with the names of the functions can also be used.

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

The statement import * includes everything from the module at once.

```
>>> from math impod *
>>> '%.6f' %.sqrt(8)
'2.828427'
```

Unimport / Reimport a module

once a module has been imported, it stays in memory until the program ends.

There are ways to 'unimport' a module or to reimport a module that has changed while the program is running

Defining your own module

Python allows us to define our own modules

The name of the module is the same as the name of the file, but without .Py extension

Program

```
import math
```

```
def distance(xc, yc, xp, yp):
    dx = xp - xc
    dy = yp - yc
    r = math.sqrt(dx**2 + dy**2)
    return r
def circle(radius):
    return (math.pi * radius * radius)
```

After creating this file, the file can be imported like any other module.

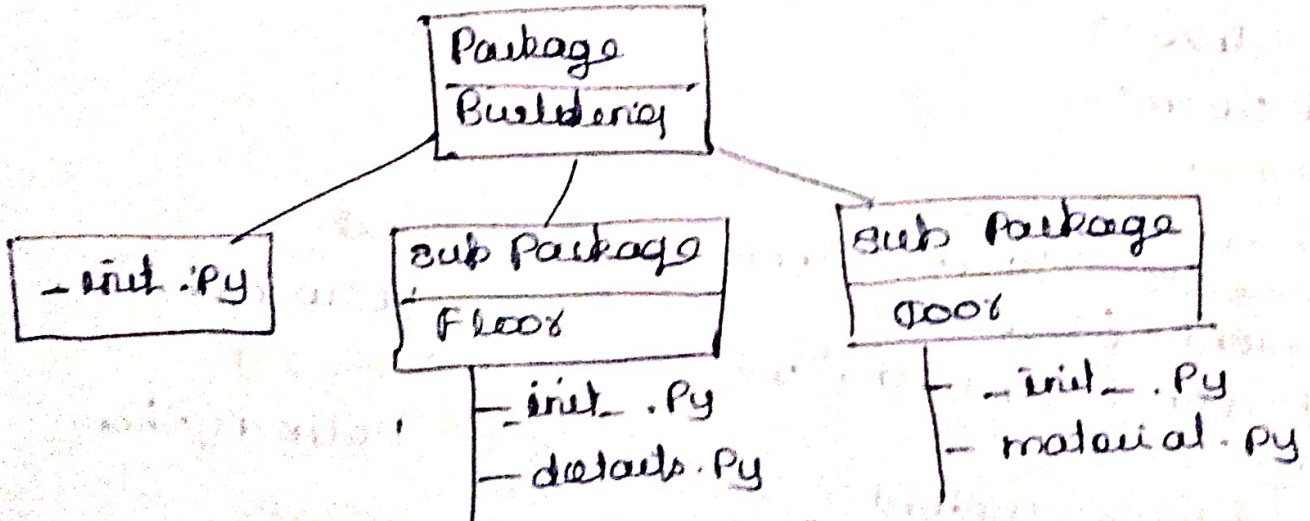
```
>>> import area
>>> area.circle(distance(0,3,3,4))
  6.28
```

Packages

A package is directory with Python files and a file with the name __init__.py. Packages contain multiple packages and modules.

Example

If a directory called building is created then a module named Floor can be created.



Importing module from a package

Modules can be imported from package using '.' operator

Example `import Building.Floor.details`

Illustrative Program (word count)

```
print("Finding word count in Text File.")
file = open("sample.txt", 'r+')
word count = 0
```

```
num_chars = 0
for word in file.read().split():
    num_chars += len(word)
    if word not in wordcount:
        wordcount[word] = 1
    else:
        wordcount[word] += 1
for k,v in wordcount.items():
    Print(str(k)+'_'+str(v))
file.close()
```

sample.txt

Hello world

output

Printing word count in Text file

Hello - 1

world - 1

File copy

```
import shutil
fs = open("s.txt","r+")
fs.write("Hello Python")
fd = open("copy.txt","w+")
try:
    shutil.copyfileobj(fs,fd)
    Print("File copied")
except:
    Print("unable to copy")
fs.close()
fd.close()
```

s.txt

Hello Python

copy.txt

Hello Python

output

file copied

Program

```
fs = open("s.txt","r+")
fs.write("Hello Python")
content = fs.read()
fd = open("copy.txt","w+")
try:
    fd.write(content)
    Print("File copied")
except:
    Print("unable to copy")
fs.close()
fd.close()
```

s.txt

Hello Python

copy.txt

Hello Python

output

File copied.