Program solving Aspect

- The techniques of Greedy Search, Backtracking and Branch and Bound evaluations are based on this basic dynamic programming idea.

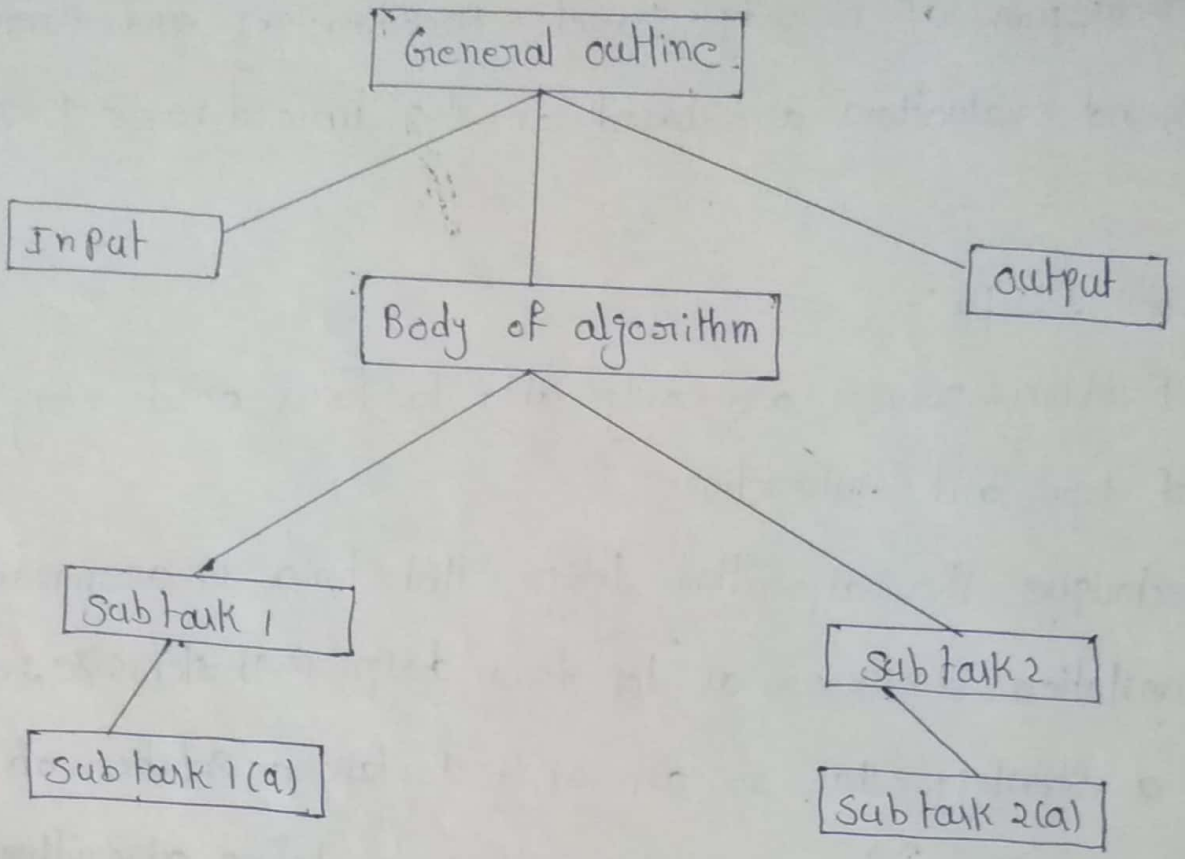
Top-Down Design :- (Follow steps 1, 2, 3 etc but not 1, 3, 2 ...) i.e. the limitation is

- The Problem-solvers are only able to focus on a very limited span of logic and instructions. (divide and conquer)
- A technique for algorithm design that tries to accommodate this limitation is known as "top-down design" (or) "step-wise refinement"
- It is a strategy that we can apply to take a solution of a computer problem from a given outline to define algorithm and problem implementation.
- It allows us to build our solutions to a problem in a step wise fashion.
- The structure and relationships among the various parts of the problem are verified. They are:

  a) Breaking a problem into sub problems
  b) choice of suitable data structure.
  c) construction of loops

a) Breaking a Problem into Sub Problems :-

- The basic idea is to divide the original problem into two or more subproblems which can be solved more efficiently.



b) choice of a suitable data structure! -

- The key to solve effetively many problems is to make appropriate choices about the associated data structures.
- A small change in data organization can have a significant influence on the algorithm required to solve the problem.
- The questions araised in setting up a data structures are:
  (1) can the data structure be easily searched?
  (2) can the data structure be easily updated?
  (3) Does the data structure involve the excessive use of storage?

organising the data element is called data structure

if we want to save int, float, char together, we use a data structure called structure

c) Construction of Loops:-

- when the series of iterative actions to be executed are the loops.
- To construct any loop we must take in to account three things, the initially conditions that need to apply before the loop begins to execute, the invariant relatively that must apply after each iteration of the loop, and the conditions under the iterative process must "terminate".

Implementation of Algorithms:-

- If an algorithm has been properly designed, the path execution should flow in a straight line from top to bottom.
- Programs implemented in top-to-bottom rule are usually much easier to understand and debug.
- Following rules should apply for the implementation of algorithms.

a) use of Procedures to emphasize modularity.

b) choice of variable names.

c) Documentation of Programs.

d) Debugging Programs.

e) Program Testing.

Implementation of Algorithms:

a) use of Procedures to emphasize modularity:

- To assist with both the development of the implementation and the readability of the main program it is usually helpful to modularize the program along the lines that follow naturally from the top-down design.
- This Practice allows us to implement a set of independent Procedures to perform specific and well-defined tasks.
- In applying modularization in an implementation one thing to watch is that the Proces is not taken too far, to a Point at which the implementation again becomes difficult to read because of the fragmentation.
- When it is necessary to implement some what larger software Projects a good strategy is to first complete the overall design in a top-down fashion.

2. Choice of Variable Names:

- Another implementation detail that can make programs more meaningful and easier to understand is to choose appropriate Variable and constants names.
- For example, if we have to make manipulations on days of the week, we are much better off using the variable day rather than the single letter 'd' or some other variable.

3. Documentation of Programs:

- Another useful documenting practice that can be employed in 'c' is to associate a brief but accurate comment (//) with each begin statement used.
- A related part of program documentation is the information that the program presents to the user during the execution phase.
- A good programming practice is to always write programs so that they can be executed and used by other people unfamiliar with the workings and input requirements of the program.
- This means that the program must specify during execution exactly what responses it requires from the user.
- Also the program should "catch" incorrect responses to its requests and inform the user in an appropriate manner.

4. Debugging Programs:

- In Implementing an algorithm it is always necessary to carry out a number of tests to ensure that the program is behaving correctly according to its specifications.
- Even with small programs it is likely that there will be logical errors that do not show up in the compilation phase.
- To make the task of detecting logical errors some what easier it is a good idea to build into the program a set of statements that will printout information at certain points in computation.

5. Program Testing:

- In attempting to test whether or not a program will handle all variations of the problem it was designed to solve we must make every effort to be sure that it will cope with the limiting and unusual cases.
- Some of the things we might check are whether the program solves the smallest possible problem, whether it handles the case ~~and~~ when all data values are the same, and so on.
- ~~unusual cases like~~

* General Problem Solving Strategies:

The most commonly used problem solving strategies are:

1. Divide - conquer strategy.
2. Binary doubling strategy.
3. Dynamic Programming.

1. Divide - conquer strategy:

- The basic idea is to divide the problem into several subproblem beyond which cannot be further subdivided.
- Then solve the sub problems efficiently and join them together to get the solution for the main program.

Program Verification:

For Program verification, following areas should be verified

1 computer model for Program execution:

- To pursue this goal of Program verification, we must fully appreciate what happens when a program is executed under the influence of given input conditions.
- A program may have a variety of execution paths leading to succesful termination. For a given set of input conditions only one of these paths will be followed.
- Tests are used to bring about a change in the sequential flow of execution. This model for program execution provides us with a foundation on which to construct correctness proofs of algorithms.

2. Input and output assertions:

- For a given input we should get the corresponding correct output.
- when we consider division problem if the remainder is not equal to 0, we will get some result but when the remainder is 0 we will not get any output.
- It doesnot mean that the program is wrong. So we should specify for the given input corresponding output should be displayed.

3. Implications and symbolic execution:

- The Problem of actually verifying a program can be formulated as a set of implications which must be shown to be logically true. These implications have the general form:

a / b

- where the logical connective "/" is read as "divide". a is termed and b are termed as variables.

4. Verification of straight-line Program segments:

- The Program should be verify in straight-line i.e in top-down approach, so that the effieciency of the Program will be increased and the user can get easily understand about the Progra

5. Verification of Program segments with branches:

- To handle Program segments that contain branches it is necessary to set up and Prove verification conditions for each branch separately.

6. Verification of Program segments with loops:

- Verification of Program segments with loops is to verify loop segments directly because the number of iterations required is usually arbitrary.

7. Proof of termination:

- To Prove that a Program terminates, it is necessary to show that it accomplishes its stated objective in a finite number of steps.

The Efficiency of Algorithms:

- Efficiency considerations for algorithms are inherently tied in with the design, implementation, and analysis of algorithms
- Every algorithm must use up some of a computer's resources to complete its task.
- The resources most relevant in relation to efficiency are CPU time and internal memory.

1 Redundant Computations:

- Most of the inefficiencies that creep in to implementation of algorithms come about because redundant computations are made or unnecessary storage is used.
- The effects of redundant computations are most serious when they are embedded within a loop that must be executed many times. The example below illustrates this point.

```
x := 0;
for i := 1 to n do
 begin
   x := x + 0.01;
   y := (a*a*a+c)*x*x+b*b*x;
   writeln ('x = ', x, 'y = ', y)
   end
```

- This loop does twice the number of multiplications necessary to

complete the computation (calculation). The unnecessary multiplications and additions can be removed by precomputing two other constants a3c and b2 before executing the loop:

```
a3c := a*a*a+c;
b2 := b*b;
x := 0;
for i := 1 to n do
 begin
  x := x+0.01;
  y := a3c*x*x +b2*x;
  writeln ('x = ', x, 'y = ', y)
 end.
```

2. Inefficiency due to late termination:

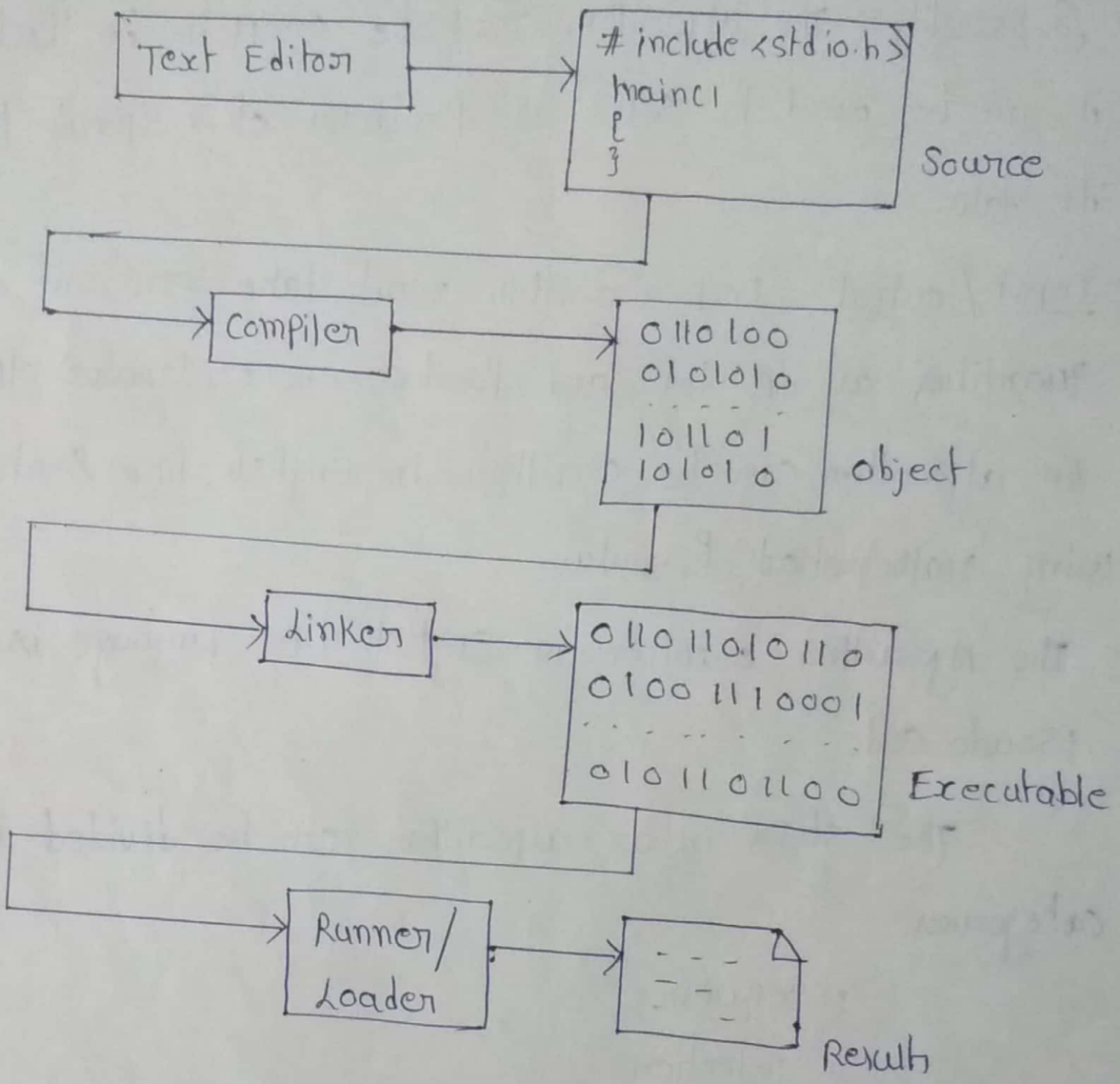
- Another place inefficiencies can come into an implementation is where considerably more tests are done than are required to solve the problem at hand.
- For Ex, to find whether a given no is prime or not, we have to find the factors.
- It is enough that if ~~you~~ we consider the factors till $N/2$.
- No need to iterate the loop till N times.
- In this way, we have to early detect the desired output conditions.

3. Early detection of desired output conditions:

- It sometimes happens, due to the nature of the input data, that the algorithm establishes the desired output condition before the general conditions for termination have been met.

4. Trading storage for efficiency gains:

- A trade between storage and efficiency is often used to improve the performance of an algorithm.
- what usually happens in this type of trade off is that we precompute or save some intermediate results and avoid having to do a lot of unnecessary testing and computation (calculation) later on. (ex. printf, scanf)
- one strategy that it sometimes used to try to speed up an algorithm is to implement it using the least number of loops
- while this is usually possible, inevitably (incapable) it makes program much harder to read and debug.
- It is therefore usually better to stick to the rule of having one loop do one job just as we have one variable doing one job.



Algorithm :

Def : A step-by-step procedure for solving the problem or performing any task is defined as an algorithm. Each step in the algorithm can be called as an 'instruction'.

An algorithm posses the following properties :

1. Finiteness :- An algorithm must terminate in a finite no. of steps.
2. Definiteness :- Each step of an algorithm must be clear and easy to understand (unambiguous).
3. Effectiveness :- Each step must be effective, in the sense that

should be Primitive (easily convertible to Program).

4. Generality :- The algorithm must be complete in itself so that it can be used to solve all problems of a specific type for an i/p data.

5. Input / output :- Each algorithm must take zero, one or more quantities as i/p data and Produce one or more o/p values.

- An algorithm can be written in English, like Sentences and using mathematical formulas.
- The Algorithm written in English like language is known as "Pseudo code".

The steps in an algorithm can be divided in to three categories.

1. Sequence.
2. Selection.
3. Iteration.

1. Sequence :- A series of steps that are perform one after the other.

2. Selection :- Making a choice from multiple available options.

3. Iteration :- Performing repetitive tasks.

Eg : 1 An algorithm to perform the addition of two numbers.

Step 1 : Start.

Step 2 : Let a, b, c are three integers.

Step 3 : Display the message "Enter a, b values".

step 4: Read two integers and stores in a, b.

step 5: compute c = a+b.

step 6: Display "The addition is :" c

step 7: Stop

Eg-2: Algorithm to find the average of three numbers.

step 1: start

step 2: Let a, b, c are three numbers.

step 3: let d is float.

step 4: Print the message "Enter any three integers"

step 5: Read the integers and stores in a, b, c

step 6: Compute d = (a+b+c)/3.0

step 7: Print "The avg is :" d.

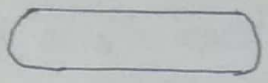
step 8: End

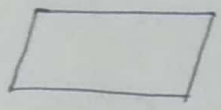
Flow chart:-

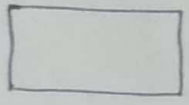
Flow chart is the pictorial / diagrammatic representation of an algorithm. It is constructed using different types of boxes and symbols. The operation to be performed is written in the box.

- All the symbols are interconnected by arrows to indicate the flow of information and processing.

Different types of symbols in the flowchart are:

1. OVAL  Terminal

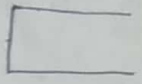
2. PARALLEOGRAM  input / out put

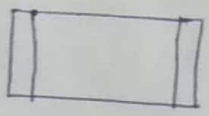
3. RECTANGLE  Process.

4. DIAMOND  Decision.

5. CIRCLE  Connector.

6. ARROW Flow

7. BRACKET  Annotation.

8. Double Sided Rectangle  Pre-defined process.

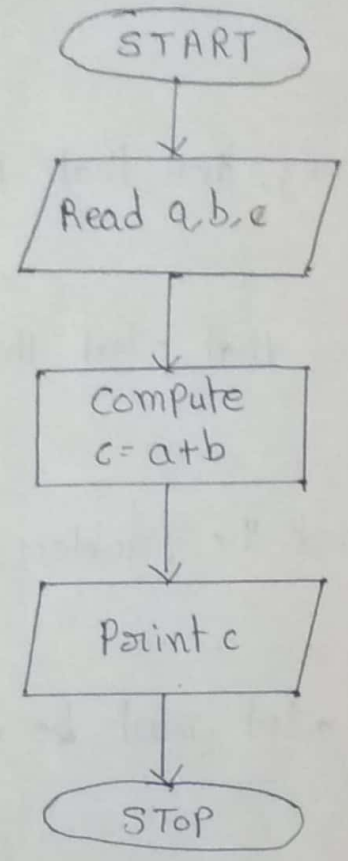
- Flowchart should be clear, neat and easy to follow.
- It should be logically correct.
- It should be verified for its validity with some test data.

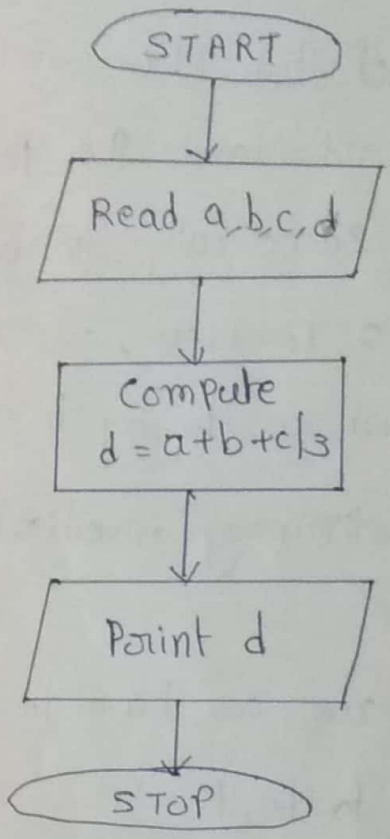
Adv:

- Logic of program is clearly represented.
- It is easy to follow logically the flowchart.

Limitations:

- Flowcharts are difficult to modify.
- Re-drawing of flow chart may be necessary.
- Translation of flow chart in to computer language is always not easy.





i) Sum of two numbers ii) Average of three numbers.

Introduction to Problem Solving :-

Problem Solving aspect :-

- It is widely recognized that Problem-Solving is a creative process which largely defines systematization and mechanization.
- The computer Problem Solving is about totally understanding.
- Some of the aspects of Problem solving are :

1. Problem Definition Phase.
2. Getting started on a Problem.
3. the use of specific examples.
4. Similarities among Problems.
5. Working backwards from the solution.

Python Programming

Literals :

1. Numeric
   - int - 10
   - float - 12.5
   - complex - 10+2.5j
2. String
   - Single line - t = 'hello'
   - Multi line - t = 'hello\ world'
3. boolean Literal
   - True
   - False
4. Special Literals - None.

- None is used to make the variable referenced to some memory location.

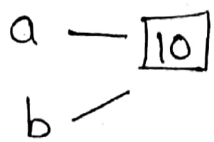
Ex : a = None

o/p : None i.e a contains None it means memory is allocated but it doesnot contain any value.

Literal collections : Tuples, Lists, Dictionary.

Variables : a = 10

b = 10



a = 11

b = 10

a = a++ // error i.e no updation concept in python.

a = a+1 // only Python has reinitialization.

bcz, if it accepts updation the impact will be there on other variables bcz both of them refers the same value in same

Identifiers:

Identifiers are the names given to the fundamental building blocks in a program. These can be variables, class, object, functions, lists, dictionaries etc.

Rules of an Identifiers:

1. An identifier is a long sequence of characters & numbers.
2. No special character except underscore (_) can be used as an identifiers.
3. Keyword should not be used as an identifier name.
4. First character of an identifier can be character, underscore(_), but not digit.

Python Keywords:

1. Keywords are the Reserved words which provides (Convey) a special meaning to the compiler/Interpreter.
2. Each Keyword has a special meaning & a specific operation.

True False None and as
assert def class continue break.
else finally elif del except
global for if from import
raise try or return pass
nonlocal in not is lambda.

Operators in Python:

An operator is a Symbol that Performs an operation An operator ach on Some variables called operands.

1. Arithmetic operators.
2. Assignment operators.
3. unary minus operators.
4. Relational operators.
5. Logical operators.
6. Boolean operators
7. Bitwise operators.
8. Membership operators.
9. Identity operators.

1. Arithmetic operators:

These operators are used to Perform basic arithmetic operations like addition, Subtraction, division etc.

Ex: Let's assume a = 13 and b = 5

operator	Meaning	Example	Result
+	Addition operator	a+b	18
-	Subtraction operator	a-b	8
*	Multiplication operator	a*b	65
/	Fractional Division	a/b	2.6
%	Modulus operator	a%b	3
**	Exponent operator	a**b	371293
//	Integer Division	a//b	2

2. Assignment operators:

These operators are useful to store the right side value into a left side variable.

Ex: Let assume x=20, y=10 and z=5

Operator	Meaning	Example	Result
=	Assignment operator	z=x+y	z=30
+=	Addition Assignment "	z+=x	z=25
-=	Subtraction "	z-=x	z=-15
*=	Multiplication "	z*=x	z=100
/=	Division Assignment "	z/=x	z=0.25
%=	Modulus Assignment "	z%=x	z=5
**=	Exponentiation "	z**=y z=z**y	z=9765625
//=	Floor division "	z//=y z=z//y	z=0

3. Unary Minus Operator:

The unary minus operator is denoted by the symbol minus (-). When this operator is used before a variable, its value is negated.

Ex: n=10

Print (-n) # displays -10

num = -10

num = - num

Print (num) # displays 10

4. Relational operators:

Relational operators are used to show the relation b/w expressions. Let us assume a=1 and b=2

operator	Example	Meaning	Result
>	a>b	Greaterthan operator	False
>=	a>=b	Greaterthan or equal to	False.
<	a<b	Lessthan operator	True
<=	a<=b	Lessthan or equal to operator	True
==	a==b	Equals operator	False
!=	a!=b	Not equals operator	True

5. Logical operators:

Logical operators are useful to construct compound conditions. Logical operators are used in b/w the expression containing relational operators.

Operators	Description
and	Logical AND
or	Logical OR
not	Logical NOT (compliment the condition i.e reverse)

Ex: a = 5>4 and 3>2 o/p:

Print a — True

b = 5>4 or 3<2

Print b — True

c = not (5>4)

Print c — False

6. Membership operators :

The membership operators are useful to test for membership in a sequence such as strings, lists, tuples and dictionaries.

Operators	Description
in	Returns true if a ele is found in specified squence i.e list etc.
~~not in~~	
not in	Returns true if a element is not found in a sequence.

ex: a = 10

b = 20

list = [10, 20, 30, 40, 50];

if (a in list):

o/p Print ("a is in given list")

else:

Print ("a is not in given list")

if (b not in list):

o/p Print ("b is not given in list")

else:

Print ("b is given in list")

7. Identity operators:

These operators compare the memory location of two objects

operators	Description
is	Return true if identity of two operands are same, else false.
is not	Return true if identity of two operands are not same, else false.

Ex: a = 20

b = 20

a is b

o/p: True

b = 10

a isnot b

o/p: ~~False~~. True.

8. Bitwise operators:

These operators act on individual bits (0 and 1) of the operands. We can use bitwise operators directly on binary nos.

There are 6 types of bitwise operators as shown below:

1. Bitwise Complement operator (~)
2. Bitwise AND operator (&)
3. Bitwise OR operator (|)
4. Bitwise XOR operator (^)
5. Bitwise Left Shift operator (<<)
6. Bitwise Right Shift operator (>>)

1. Bitwise complement operator (~):

This operator gives the complement form of a given number.

x	y
0	1
1	0

2. Bitwise AND operator (&):

This operator performs AND operation on the individual bits of numbers.

x	y	x&y
0	0	0
0	1	0
1	0	0
1	1	1

Ex: x = 10 => 00001010
y = 11 => 00001011
x&y => 00001010
=> 10 //

3. Bitwise OR operator (|):

This operator performs OR operation on the bit of the numbers. The symbol of bitwise operator is called Pipe symbo

x	y	x|y
0	0	0
0	1	1
1	0	1
1	1	1

Ex: x = 10 => 00001010
y = 11 => 00001011
x|y => 00001011
=> 11 //

4. Bitwise XOR operator (^):

This operator performs exclusive XOR operation on the bits of numbers. The symbol is called cap, Carat or circumflex.

x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

Ex:
x = 10 => 00001010
y = 11 => 00001011
00000001
=> 1 //

9. Boolean operators:

Boolean operators act upon 'bool' type literals and they provide 'bool' type output. Lets x = True and y = False.

Operator	Description
and	x and y are True, then returns True.
or	x or y is True, then returns True.
not	If x is True it returns False.

ex: a = True

b = False

```
>>> a and a
 True
>>> a and b
 False
>>> a or a
 True
>>> a or b
  True
>>> b or b
  False.
>>> not a
 False.
>>> not b   => True.
```

5. Bitwise left shift operator (<<):

This operator shifts the bits of the no. towards left a specified number of positions.

Ex: $x = 10 \Rightarrow 00001010$, Now $x << 2$

$\Rightarrow 00101000$ (or) $x \cdot 2^y \Rightarrow 10 * 2^2$

$\Rightarrow 40$ $\Rightarrow 40$

6. Bitwise right shift operator (>>):

This operator shifts the bits of the no. towards right a specified number of positions.

Ex: $x = 10 \Rightarrow 00001010$

$\Rightarrow 00000010$ (or) $\frac{x}{2^y} \Rightarrow \frac{10}{2^2}$

$\Rightarrow 2$ $\Rightarrow 10/4 \Rightarrow 2$

~~operator Preced~~

Expressions:

Expression is a set of operators and operands which is used to perform computation. The following are the types of expressions:

1. Unary expression.
2. Binary expression.
3. Primary expression.

Type Casting:

In Python implicit Promotion (i.e Promotion w.r.to expression not w.r.to assignment) is there but implicit demotion is not there due to its own constraints like automatic type of data allocation for the variable.

But both explicit Promotion and demotion is available in Python.

Ex: 1. x = 10
y = 12.5
c = x + y

o/p: 22.5

Implicit Promotion (w.r.to exp)

2. x = 5
y = 2
c = x/y

o/p: 2.5

So, No Implicit demotion.
expected o/p is 2.

(or)

x = 12.5
y = x

o/p: 12.5 instead of 12, i.e no demotion

3. x = 5
y = float(x)

o/p: 5.0

Explicit Promotion

4. x = 5.5
y = int(x)

o/p: 5

Explicit demotion.

Def: Converting one type to another type (i.e data type) is called "Type casting".

Datatypes in Python:

A datatype represents the type of data stored into a variable or memory. The classification of data types is as follows:

1. None Type
2. Numeric Types
3. Boolean Types
4. Sequence Types
5. Sets.
6. Mapping.

1. None Type: In Python the 'None' datatype represents an object that does not contain any value.

2. Numeric Types:

The Numeric Types represents numbers. There are three sub types.

1. int
2. float
3. complex

1. The int data type represents an integer number.

Ex: a = 57

2. The float datatype represents an floating point numbers.

Ex: b = 12.5

3. The Complex number is a no. that is written in the form of a+bj or a+bJ. Here 'a' represents the real Part and 'b' represents the imaginary part. The suffix 'j' or 'J' represents the square root value of -1. The Parts 'a' and 'b' may contain integers or floats.

Ex: c1 = 2.5 + 3j

3. Boolean Types:

The bool datatype in Python represents boolean values. There are only two boolean values True or False that can be represented by this datatype. Python internally represents True as 1 and False as 0. A blank string like "" also represented as False.

4. Sequence Types:

A sequence represents a group of elements or items. For ex, a group of integers will form a sequence. They are as follows

1. Str
2. list
3. Tuple
4. range.

i) Str: In Python, Str represents string data type. A string is represented by a group of characters.

Ex: str1 = "Hello"

str2 = 'welcome'

ii) list Datatype:

Lists in Python are similar to arrays in C or Java. A list represents a group of elements. The main difference b/w a list and an array is that a list can store different types of elements but an array can store only one type of elements. Also, lists can grows dynamically in memory. It is an ordered list.

Ex: list1 = [10, -20, 15.5, 'Hello', "bye"]

iii) tuple Datatype:

A tuple is similar to a list. A tuple contains a group of elements which can be of different types. The elements in the tuple are enclosed in Parentheses (). whereas the list elements can be modified, it is not possible to modify the tuple elements.

tpl = (10, -20, 15.5, 'Vijay', "welcome")

Ex: tpl[0] = 99 // This will result in error.

iv) range Datatype:

The range datatype represents a sequence of numbers The no:s in the range are not modifiable.

* It is possible to convert range into list.

l = list(range(10))

it returns list l as l = [0 to 9].

* range (10) => ranges from 0 to 9. // default stepsize 1.
* range (2,10) => ranges from 2 to 9. [initial value 2, final 10
* range (2,10,2) => ranges from 2 to 9 as [2,4,6,8] stepsize
* range (10,-1,-1) => ranges from 10 to 0 as [10,...,0]

Generally, range is used for repeating a for loop for a specific number of times.

5. Set Data type:

A set is an unordered collection of elements much like a set in Mathematics. The order of elements is not maintained in the set. It means the elements may not appear in the same order as they are entered into the set. Moreover, a set doesn't accept duplicate elements. The elements in the set are enclosed in { }. curly braces.

Ex: S = {10, 20, 30, 40, 50}
print(S) # may display {50, 10, 20, 30}.

6. Mapping Types:

A map represents a group of elements in the form of "key value" pairs. So that when the key is given, we can retrieve the value associated with it. The dict datatype is an example for map here 'dict' represents dictionary.

Ex: d = {10: "Karthik"}
key value

Input and output Functions:

output Statements:

1. The Print() statement:

When the Print() function is called simply, it will throw cursor to the next line. It means that a blank line will be displayed.

2. The Print("String") Statement:

When the string is passed to the Print() function the string is displayed as it is.

Ex: Print ("Hello")

o/p: Hello

3. The Print (variable list) statement:

Print() function can also display the values of variable

Ex: a = 10

Print(a)

o/p: 10

4. The values in the o/p are seperated by a space by default. To seperate the o/p with comma or any other character, we should use 'sep' attribute. 'sep' represents Separator.

Syn: sep = "character" which is used to seperate values in the o/p.

ex: a, b = 2, 4

Print (a, b)

o/p: 2 4

Print (a, b, sep = ", ")

o/p: 2, 4

Print (a, b, sep = ":")

o/p: 2 : 3

'end' attribute is used to, not to throw the cursor into the next line by Print() function.

ex: Print ("Hello", end = ' ')

Print ("welcome")

o/p: Hello welcome

Syn: end = "characters".

4. The Print (object) statement:

We can pass objects like lists, tuples or dictionaries to the Print() function.

ex: lst = [10, 'A', 'Hai']

Print (lst) o/p: [10 'A' 'Hai']

↳ object

5. The Print ("String", Variable list) statement:

The most common use of the Print() function is to use strings along with the variables inside the Print() function

ex: a = 2

Print (a, "is even number")

o/p: 2 is even number

Print ('You typed', a, 'as input')

o/p: you typed 2 as input.

6. The Print (Formatted string) statement:

The o/p displayed by the Print() function can be formatted as we like. The special operator '%' (Percent) can be used for this purpose. It joins a string with a variable or value.

syn: Print ("Formatted String" % (variable list))

ex: 1. x = 10

Print ("value = %i" % x)

o/p: value = 10

When more than one variable is to be displayed, then parantheis are needed as:

2. x, y = 10, 15

Print ("x = %i y = %d" %. (x, y))

o/p: x = 10 y = 15

Input Statements:

To accept input from the keyboard python provides input() function. This function takes a value from the keyboard and returns as a string.

Ex: 1. str = input()

Karthik

Print (str)

Karthik

2. str = input ("enter ur name:")

Enter ur name : Karthik

Print (str)

Karthik.

3. str = input ("enter a no");

enter a no : 36

x = int (str) # str is converted into int

Print (x)

36

4. x = int (input ("enter a no:"))

enter a no : 36

Print (x)

36.

Evaluating an expression:

We can use eval() function along with input() function since the input() function accepts a value in the form of a string, the eval() function receives that string and evaluates it.

ex: 1. x = eval (input ("Enter an expression"))

Print ("Result = %d" % x)

o/p: enter an expression: 10 + 5 - 4

Result = 11

2. a, b = 5, 10

result = eval ("a+b-4")

Print (result)

o/p: 11

Formate() function:

Ex: 1. name, salary = "Ravi", 12500.75

Print ("Hello {n}, your salary is {s}". format (n=name, s=salary))

o/p: Hello Ravi, your salary is 12500.75