

**UNIT I ALGORITHMIC PROBLEM SOLVING****9**

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

**Algorithm**

Definition: An algorithm is procedure consisting of a finite set of unambiguous rules (instructions) which specify a finite sequence of operations that provides the solution to a problem. In other word, an algorithm is a step-by-step procedure to solve a given problem

Definition: An **algorithm** is a finite number of clearly described, unambiguous “double” steps that can be systematically followed to produce a desired result for given input in a finite amount of time.

**Building blocks of algorithm**

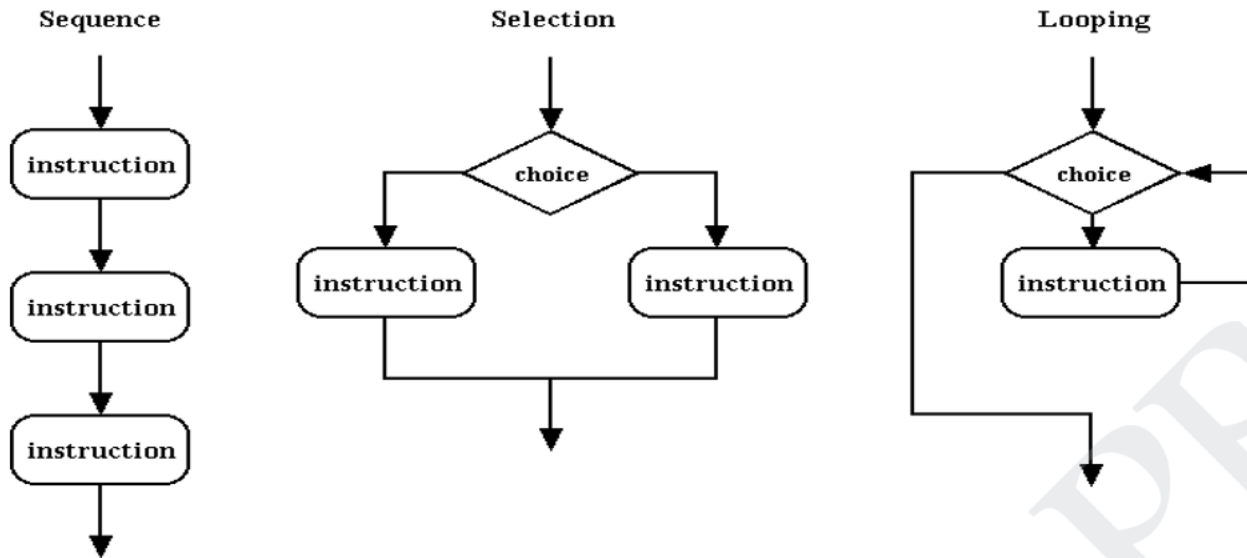
It has been proven that any algorithm can be constructed from just three basic building blocks. These three building blocks are Sequence, Selection, and Iteration.

Building Block	Common name
Sequence	Action
Selection	Decision
Iteration	Repetition or Loop

A sequence is one of the basic logic structures in computer programming. In a *sequence* structure, an action, or event, leads to the next ordered action in a predetermined order. The sequence can contain any number of actions, but no actions can be skipped in the sequence. Once running, the program must perform each action in order without skipping any.

A selection (also called a decision) is also one of the basic logic structures in computer programming. In a *selection* structure, a question is asked, and depending on the answer, the program takes one of two courses of action, after which the program moves on to the next event

An iteration is a single pass through a group/set of instructions. Most programs often contain loops of instructions that are executed over and over again. The computer repeatedly executes the loop, iterating through the loop



Write an algorithm to add two numbers entered by user.

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

Write an algorithm to find the largest among three different numbers entered by user.

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

Write an algorithm to find all roots of a quadratic equation  $ax^2 + bx + c = 0$ .

Step 1: Start  
 Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

$D \leftarrow b^2 - 4ac$

Step 4: If  $D \geq 0$

$r_1 \leftarrow (-b + \sqrt{D}) / 2a$

$r_2 \leftarrow (-b - \sqrt{D}) / 2a$

Display  $r_1$  and  $r_2$  as roots.

Else

Calculate real part and imaginary part

$rp \leftarrow -b / 2a$

$ip \leftarrow \sqrt{-D} / 2a$

Display  $rp + j(ip)$  and  $rp - j(ip)$  as roots

Step 5: Stop

**Write an algorithm to find the factorial of a number entered by user.**

Step 1: Start

Step 2: Declare variables  $n$ , factorial and  $i$ .

Step 3: Initialize variables

factorial  $\leftarrow 1$

$i \leftarrow 1$

Step 4: Read value of  $n$

Step 5: Repeat the steps till  $i = n$

5.1: factorial  $\leftarrow$  factorial \*  $i$

5.2:  $i \leftarrow i + 1$

Step 6: Display factorial

Step 7: Stop

**Write an algorithm to check whether a number entered by user is prime or not.**

Step 1: Start

Step 2: Declare variables  $n, i, \text{flag}$ .

Step 3: Initialize variables

flag  $\leftarrow 1$

$i \leftarrow 2$

Step 4: Read  $n$  from user.

Step 5: Repeat the steps till  $i < (n/2)$

5.1 If remainder of  $n \div i$  equals 0

flag  $\leftarrow 0$

Go to step 6

5.2  $i \leftarrow i + 1$

Step 6: If flag = 0

Display  $n$  is not prime

else

Display  $n$  is prime

Step 7: Stop

**Write an algorithm to find the Fibonacci series till term  $\leq 1000$ .**

Step 1: Start

Step 2: Declare variables first\_term, second\_term and temp.  
 Step 3: Initialize variables first\_term ← 0 second\_term ← 1  
 Step 4: Display first\_term and second\_term  
 Step 5: Repeat the steps until second\_term ≤ 1000  
     5.1: temp ← second\_term  
     5.2: second\_term ← second\_term + first term  
     5.3: first\_term ← temp  
     5.4: Display second\_term  
 Step 6: Stop

**Pseudo code:**

Pseudo code is a detailed yet readable description of what a computer program or algorithm must do, expressed in a formally-styled natural language rather than in a programming language. Pseudo code is sometimes used as a detailed step in the process of developing a program

**Compute the area of a rectangle:**

GET THE length, l, and width, w  
 COMPUTE area = l\*w  
 DISPLAY area

**Compute the perimeter of a rectangle:**

READ length, l  
 READ width, w  
 COMPUTE Perimeter = 2\*l + 2\*w  
 DISPLAY Perimeter of a rectangle

**Iteration:**

**Iteration** is the act of repeating a process, either to generate an unbounded sequence of outcomes, or with the aim of approaching a desired goal, target or result. Each repetition of the process is also called an "iteration", and the results of one iteration are used as the starting point for the next iteration.

```
a = 0
for i from 1 to 3      // loop three times
{
    a = a + i          // add the current value of i to a
}
print a               // the number 6 is printed (0 + 1; 1 + 2; 3 + 3)
```

**Recursion**

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), In order/Preorder/Post order Tree Traversals, DFS of Graph, etc.

```
int fact(int n)
{
```

```

if (n <= 1) // base case
    return 1;
else
    return n*fact(n-1);
}

```

In the above example, base case for  $n \leq 1$  is defined and larger value of number can be solved by converting to smaller one till base case is reached.

### Disadvantages of Recursion over iteration

Note that both recursive and iterative programs have same problem solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. Recursive program has greater space requirements than iterative program as all functions will remain in stack until base case is reached. It also has greater time requirements because of function calls and return overhead.

### Advantages of Recursion over iteration

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems it is preferred to write recursive code. We can write such codes also iteratively with the help of stack data structure. For example refer Inorder Tree Traversal without Recursion, Iterative Tower of Hanoi.











### Flow Charts

A Flowchart is a diagram that graphically represents the structure of the system, the flow of steps in a process, algorithm, or the sequence of steps and decisions for execution a process or solution a problem.

Flow charts are easy-to-understand diagrams that show how the steps of a process fit together. American engineer Frank Gilbert is widely believed to be the first person to document a process flow, having introduced the concept of a "Process Chart" to the American Society of Mechanical Engineers in 1921.

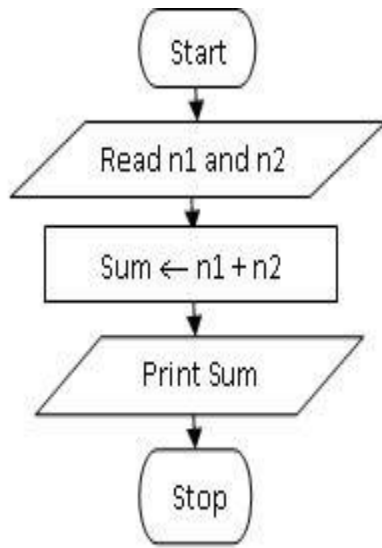
Flow charts tend to consist of four main symbols, linked with arrows that show the direction of flow:

1. Elongated circles, which signify the start or end of a process.
2. Rectangles, which show instructions or actions.
3. Diamonds, which highlight where you must make a decision.
4. Parallelograms, which show input and output. This can include materials, services or people.

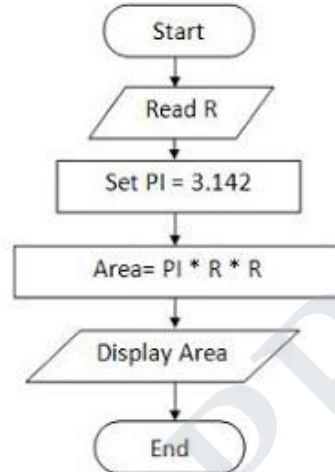
Name	Symbol	Description
Process		Process or action step
Flow line		Direction of process flow
Start/ terminator		Start or end point of process flow
Decision		Represents a decision making point
Connector		Inspection point
Inventory		Raw material storage
Inventory		Finished goods storage
Preparation		Initial setup and other preparation steps before start of process flow
Alternate process		Shows a flow which is an alternative to normal flow
Flow line(dashed)		Alternate flow direction of information flow

### Flow Chart Examples:

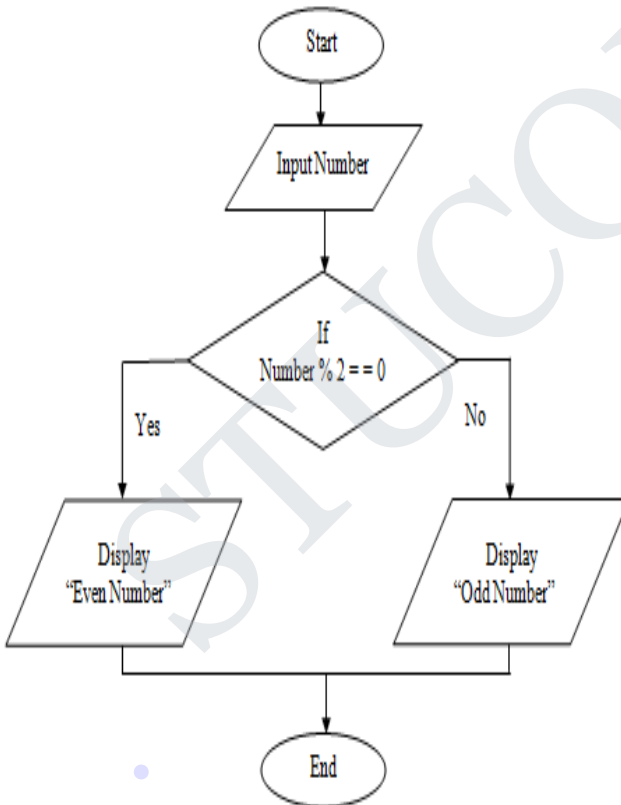
Sequence to find the sum of two number



Area of triangle

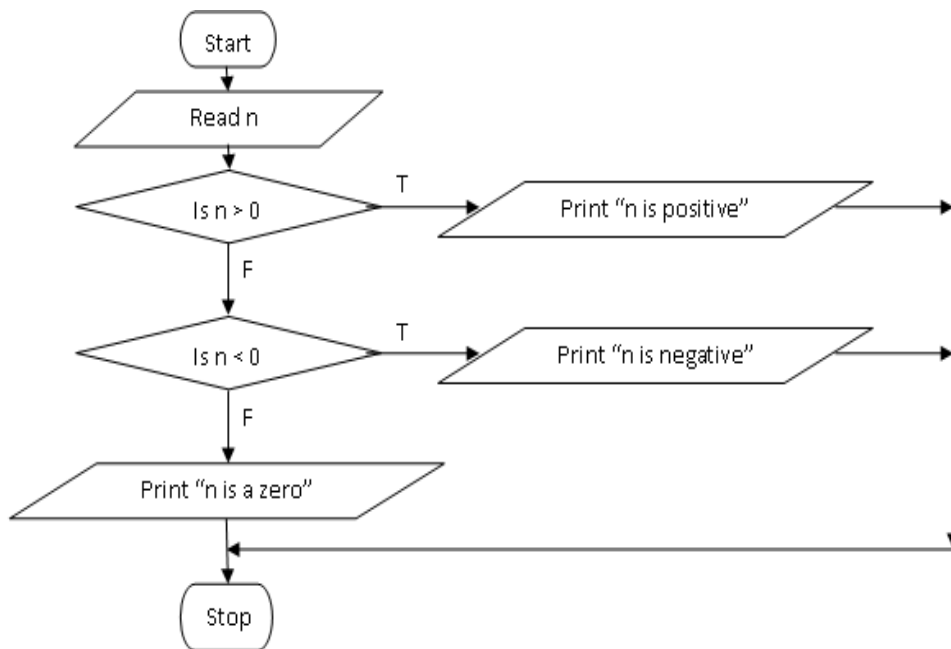


Flowchart to find the number is odd or Even

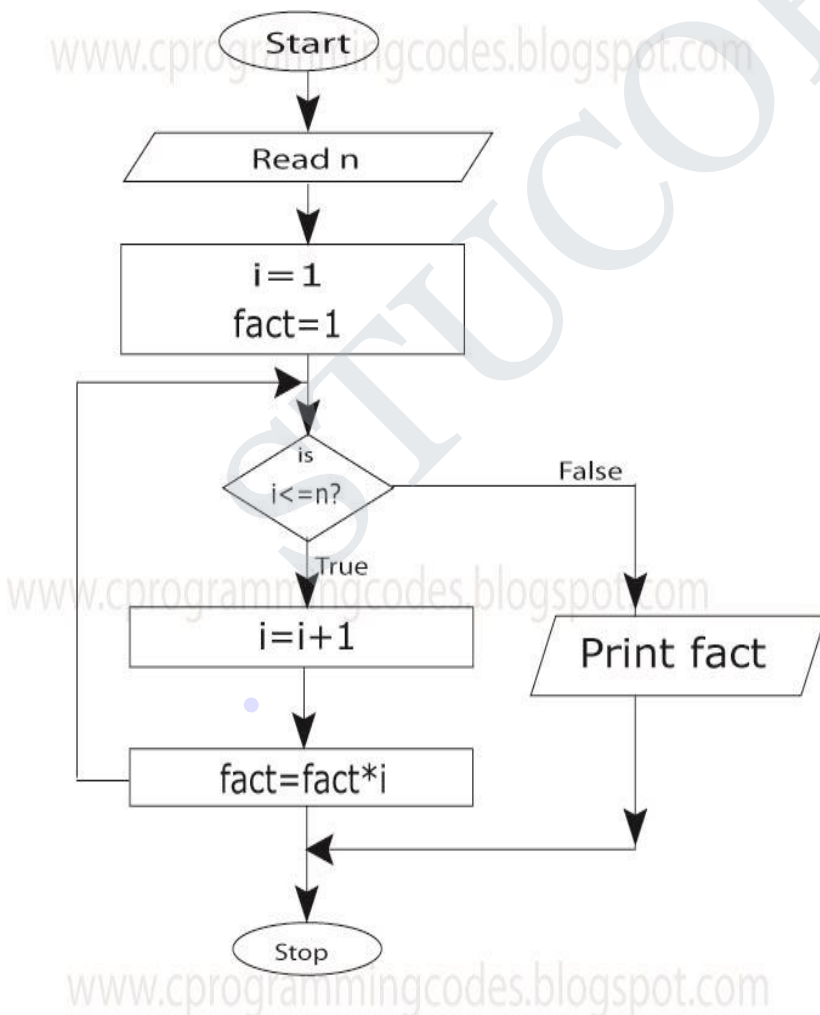


Flow Chart to find Positive or Negative





Factorial of the given Number



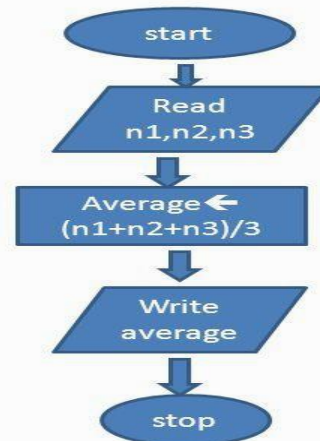


**Find the biggest of two numbers**

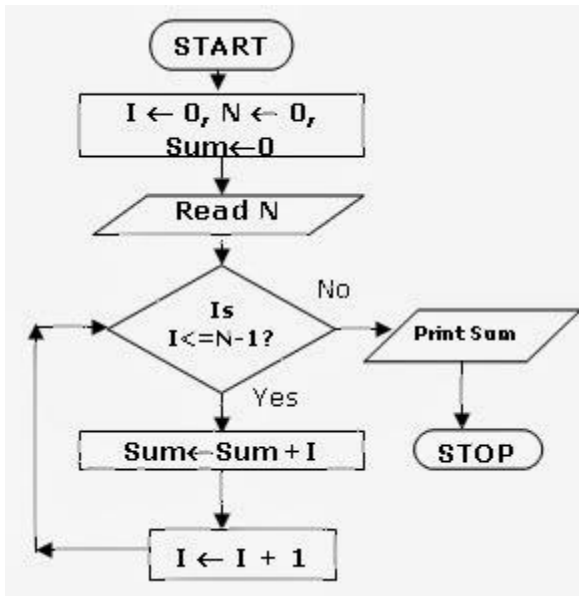
### Flowchart algorithm3

Draw a flowchart that will accept 3 numbers and find and display their average

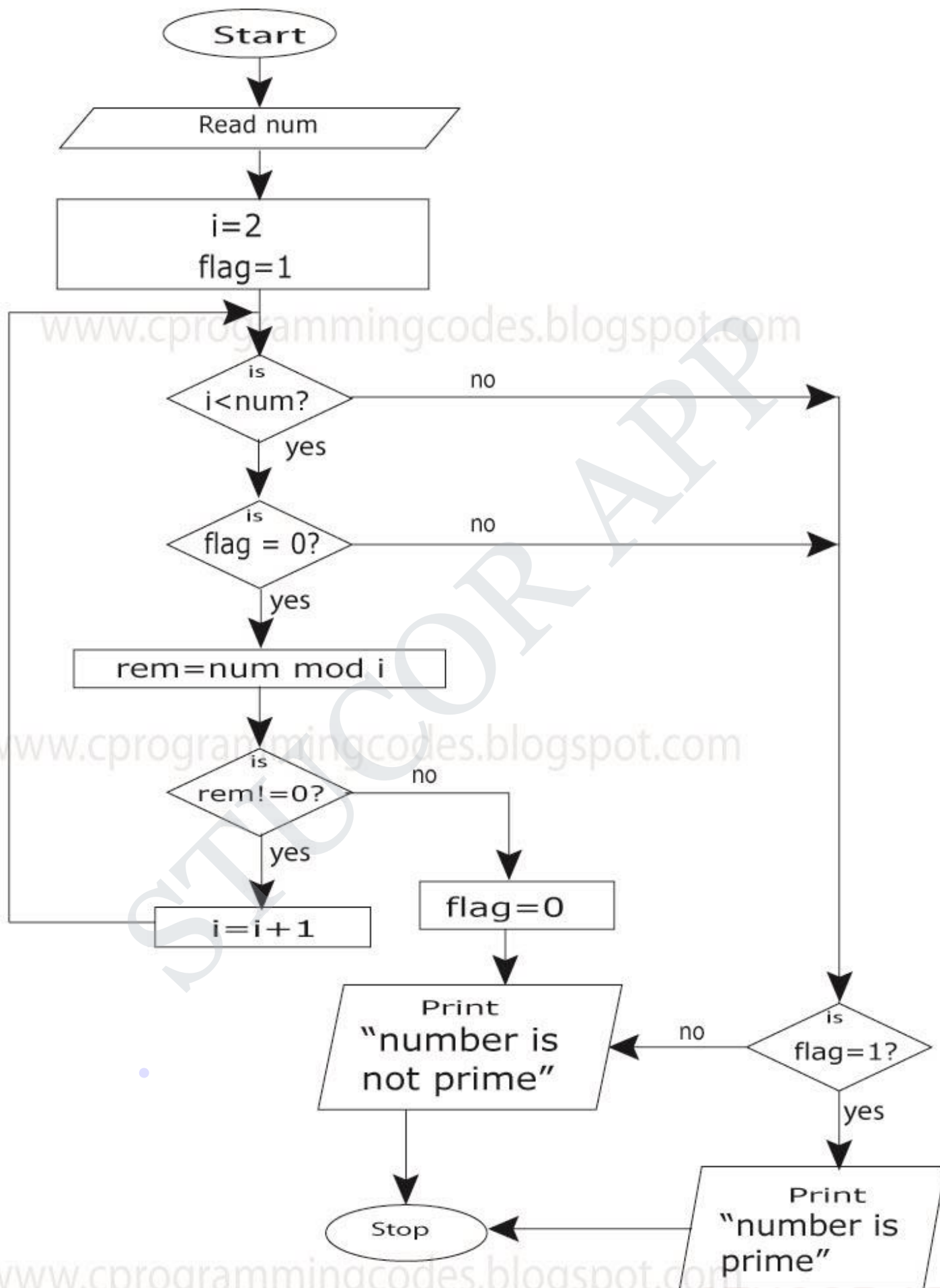
Start  
Read n1,n2,n3  
 $\text{Average} \leftarrow (n1+n2+n3)/3$   
Write average  
stop

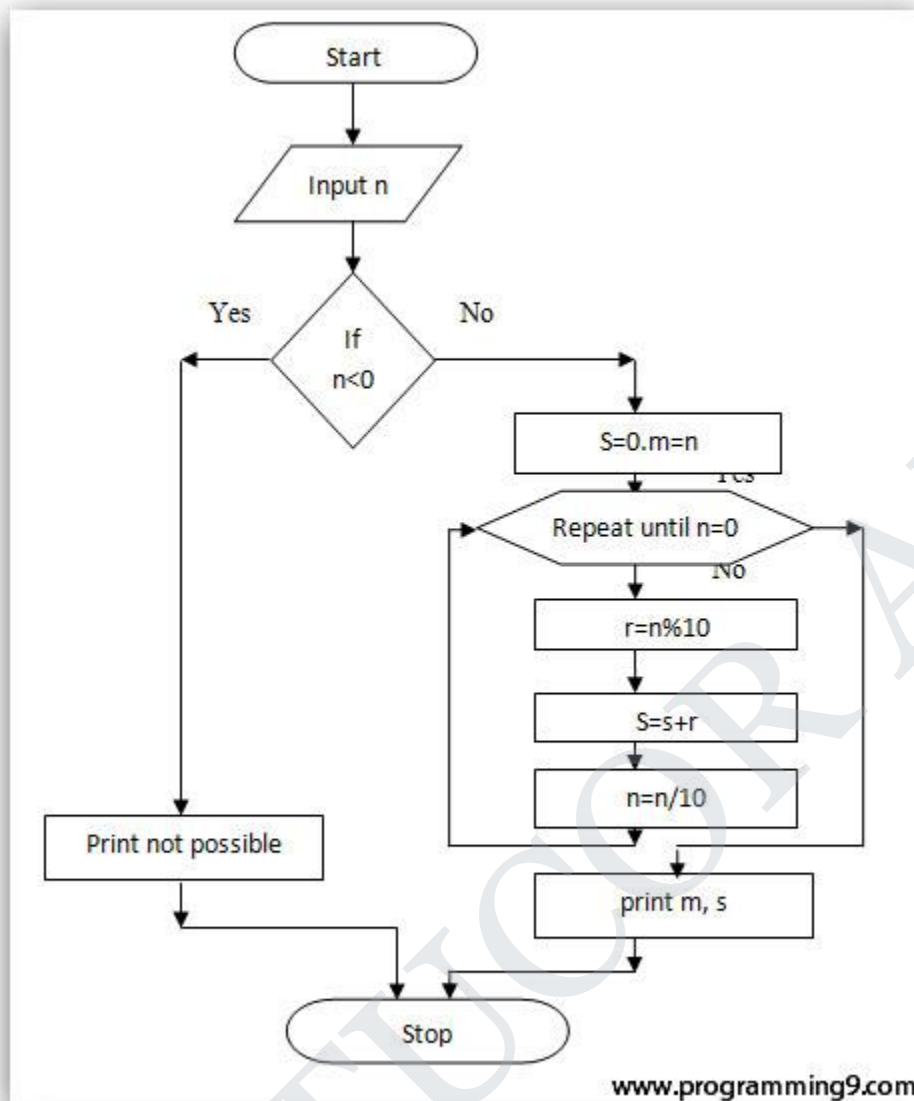


Flow chart to find the sum of n numbers



Flow chart for finding Prime number



**Flowchart for finding sum of digits of a given number**

Illustrative problems: find minimum in a list, insert a card in a list of sorted cards, guess an integer number in a range, Towers of Hanoi.

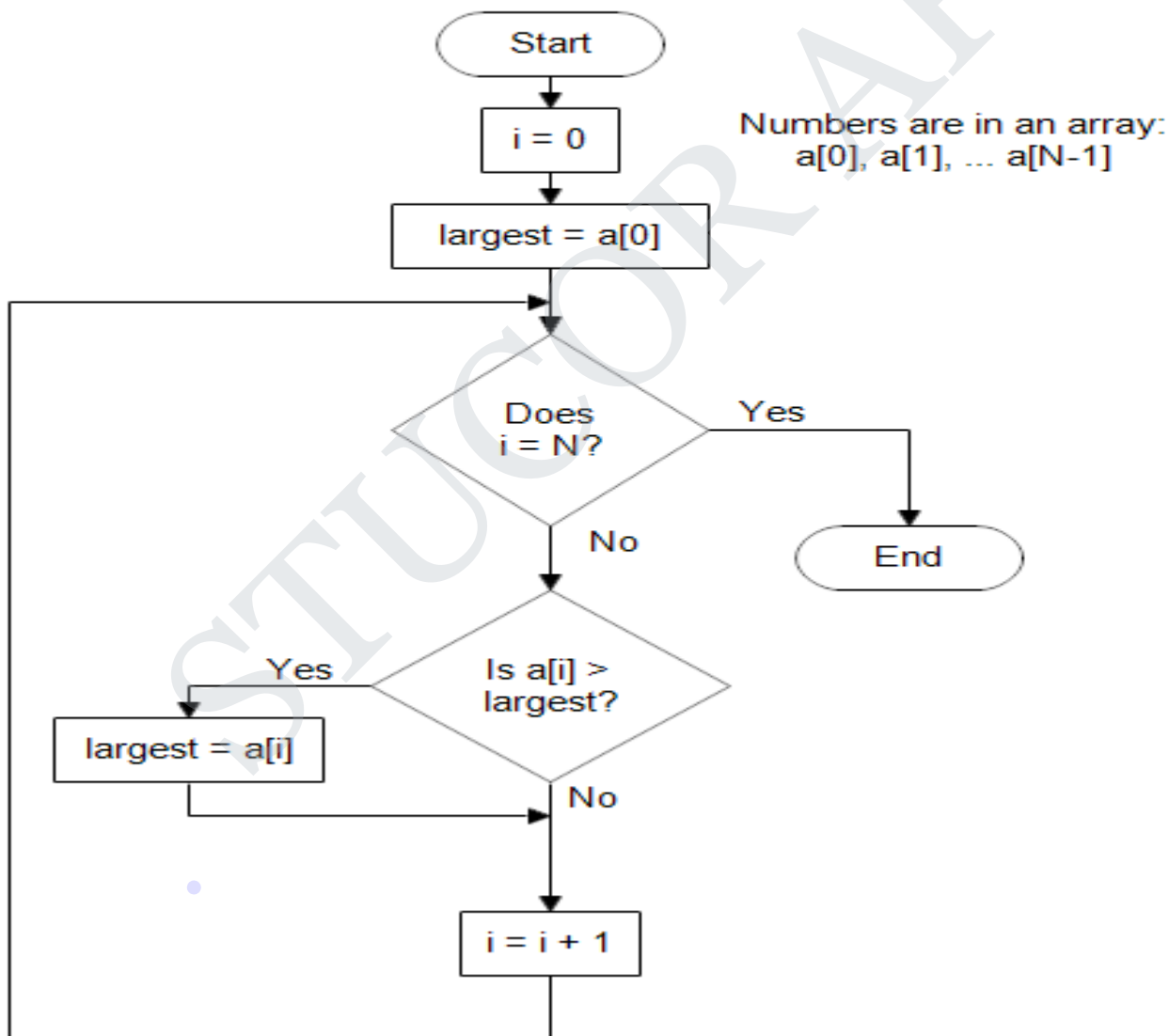
Find minimum in a list:

•

```

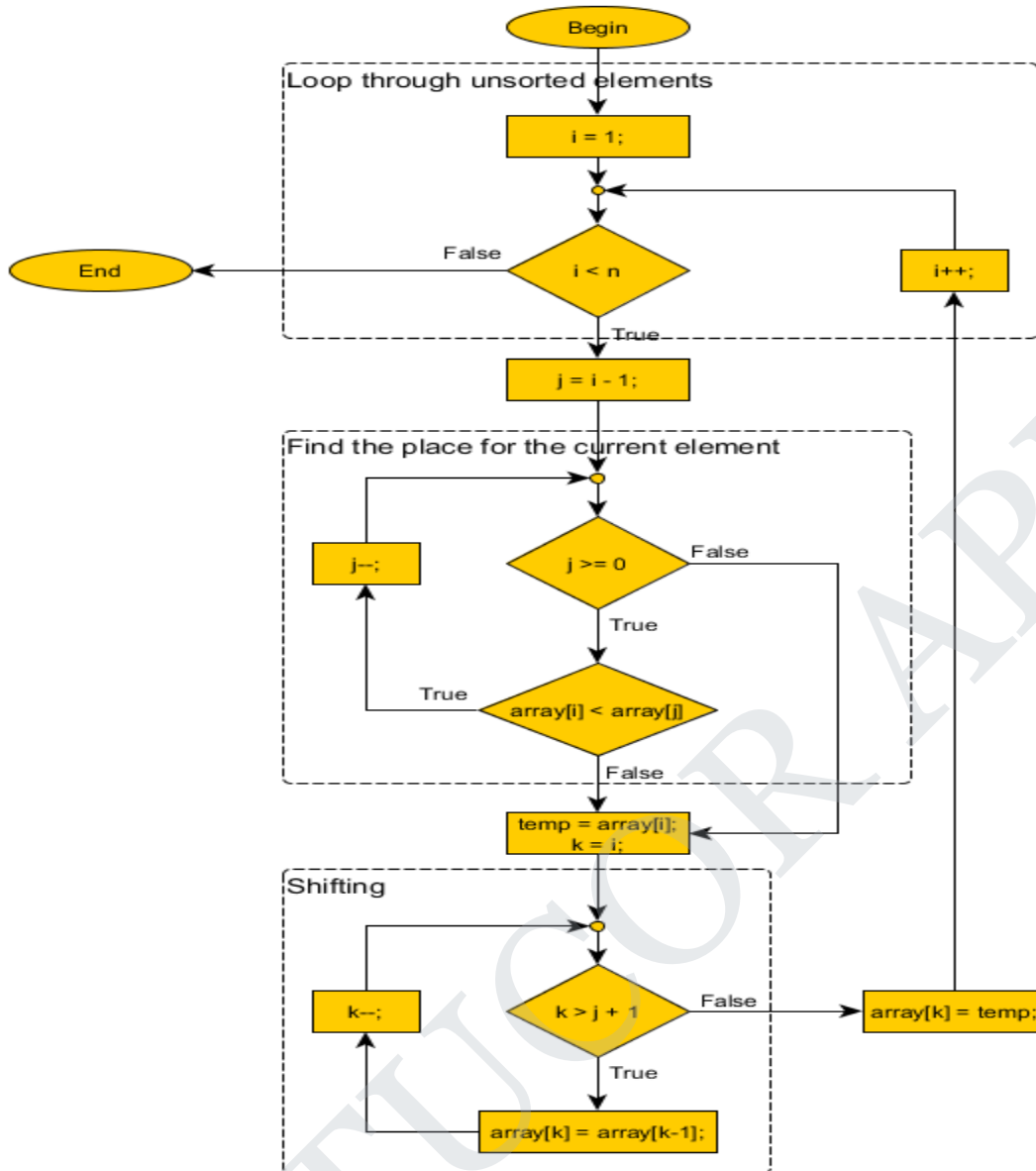
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
  Max = Min = a [1];
  For i = 2 to n do
  {
    If (a [i] > Max) then Max = a [i];
    If (a [i] < Min) then Min = a [i];
  }
}
    
```

## Finding the Largest Number in a List of Numbers



To find the minimum from the list the same flowchart but change the sign

## Insertion Sort



Sorting is ordering a list of objects. We can distinguish two types of sorting. If the number of objects is small enough to fit into the main memory, sorting is called *internal sorting*. If the number of objects is so large that some of them reside on external storage during the sort, it is called *external sorting*. In this chapter we consider the following internal sorting algorithms

- Bucket sort
- Bubble sort
- Insertion sort
- Selection sort
- Heapsort
- Mergesort

### Guess a Number in python

```
import random
```

```
x = random.randrange(1, 201)

print(x)

guess = int(input("Please enter your guess: "))

if guess == x:
    print("Hit!")
elif guess < x:
    print("Your guess is too low")
else:
    print ("Your guess is too high")
```

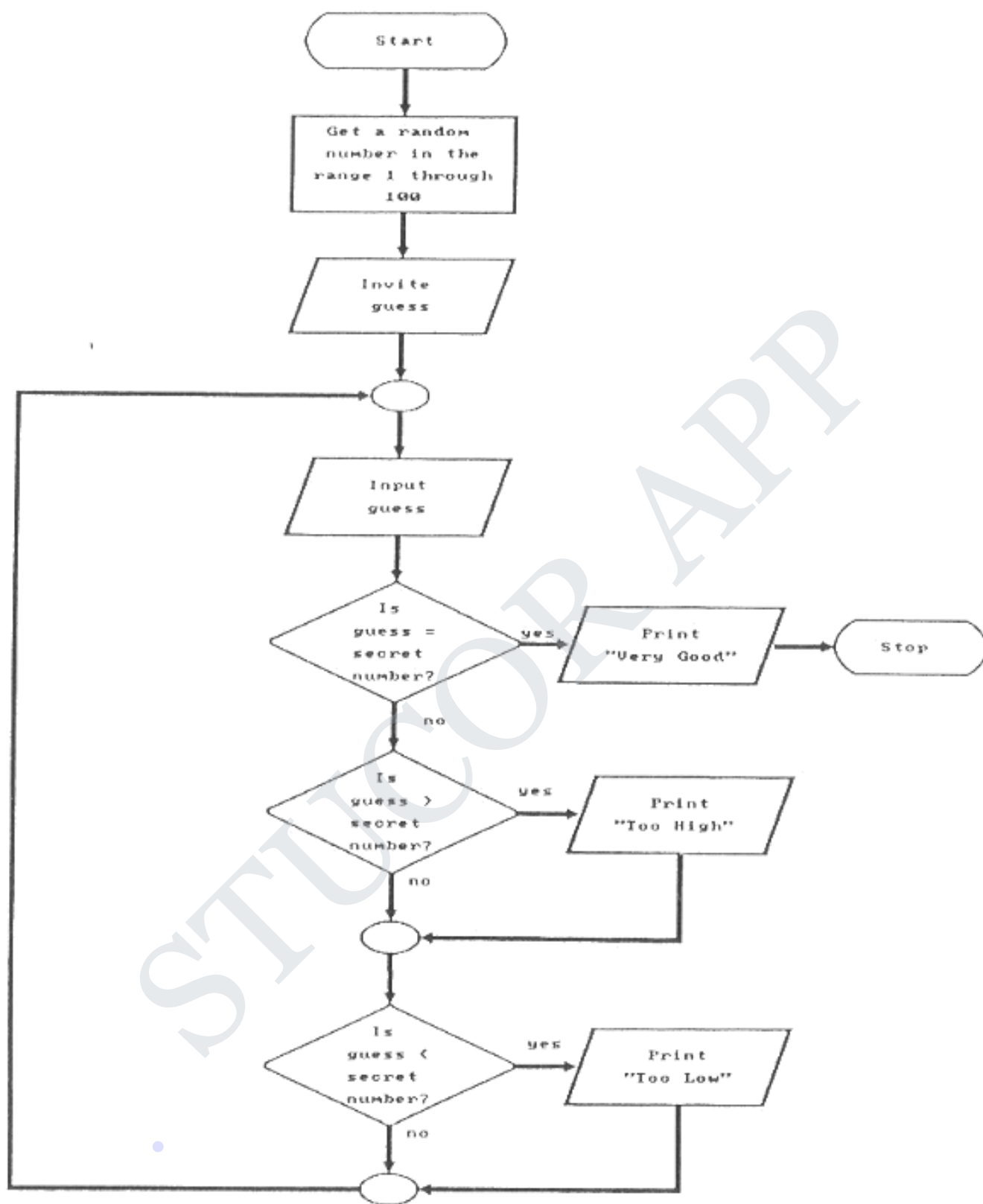
### **OUTPUT**

```
>>> runfile('C:/Users/admin/.anaconda/navigator/gu.py', wdir='C:/Users/admin/.anaconda/navigator')
158
Please enter your guess: 200
Your guess is too high
>>> runfile('C:/Users/admin/.anaconda/navigator/gu.py', wdir='C:/Users/admin/.anaconda/navigator')
59
Please enter your guess: 20
Your guess is too low
```

### **Flow Chart**

.





**Range function in python**

the range() function has two sets of parameters, as follows:

range(stop)

- stop: Number of integers (whole numbers) to generate, starting from zero. eg. range(3) == [0, 1, 2].

range([start], stop[, step])

- start: Starting number of the sequence.
- stop: Generate numbers up to, but not including this number.
- step: Difference between each number in the sequence.

Note that:

- All parameters must be integers.
- All parameters can be positive or negative.
- range() (and Python in general) is 0-index based, meaning list indexes start at 0, not 1. eg. The syntax to access the first element of a list is mylist[0]. Therefore the last integer generated by range() is up to, but not including, stop. For example range(0, 5) generates integers from 0 up to, but not including, 5.
- Python's range() Function Examples

```
>>> # One parameter
```

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

```
...     print(i)
```

```
...
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
>>> # Three parameters
```

```
>>> for i in range(4, 10, 2):
```

```
...     print(i)
```

```
...
```

```
4
```

```
6
```

```
8
```

```
>>> # Going backwards
```

```
>>> for i in range(0, -10, -2):
```

```
...     print(i)
```

```
...
```

```
0
```

```
-2
```

```
-4
```

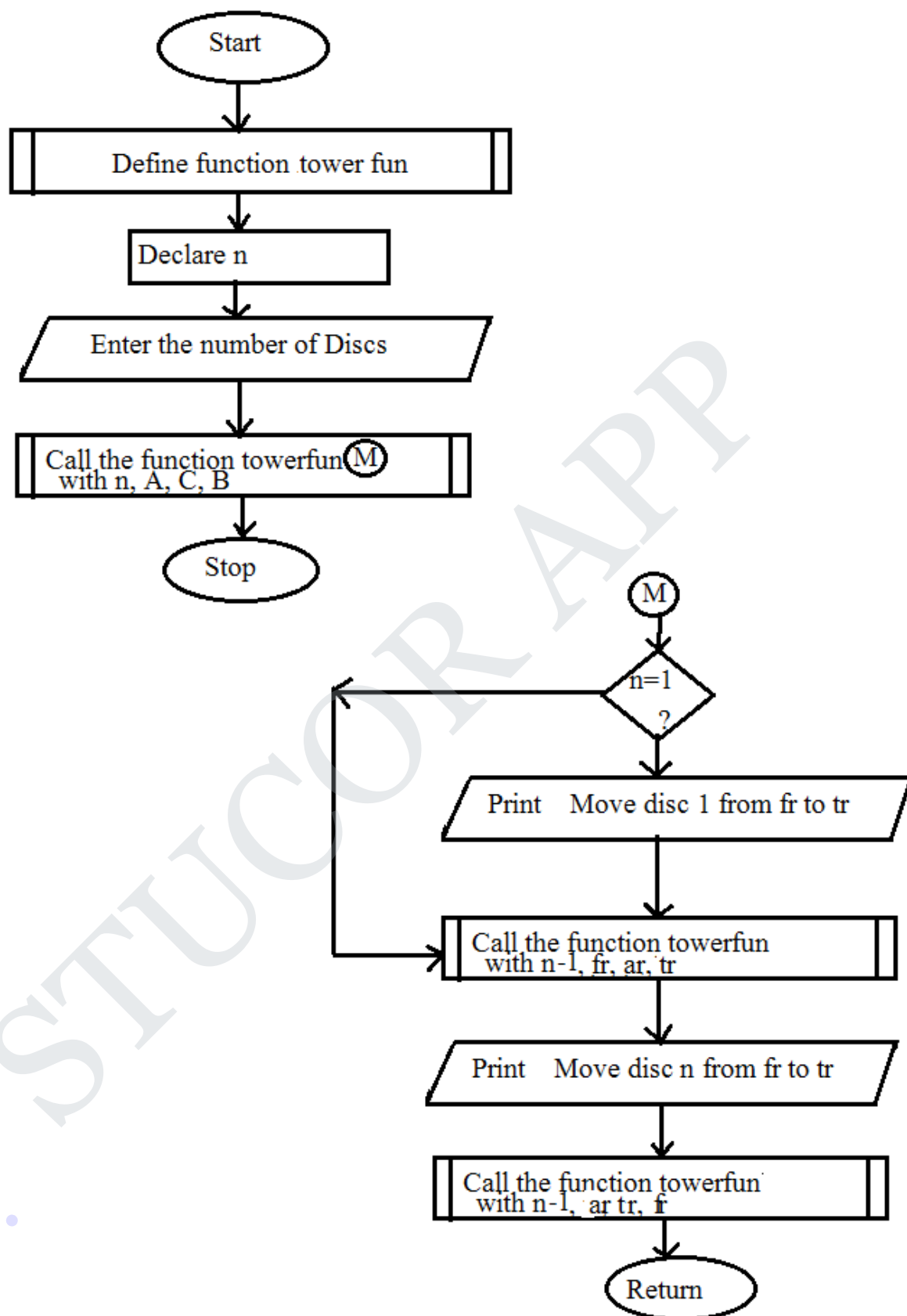
```
-6
```

```
-8
```

**Tower of the Hanoi**

STUCOR APP

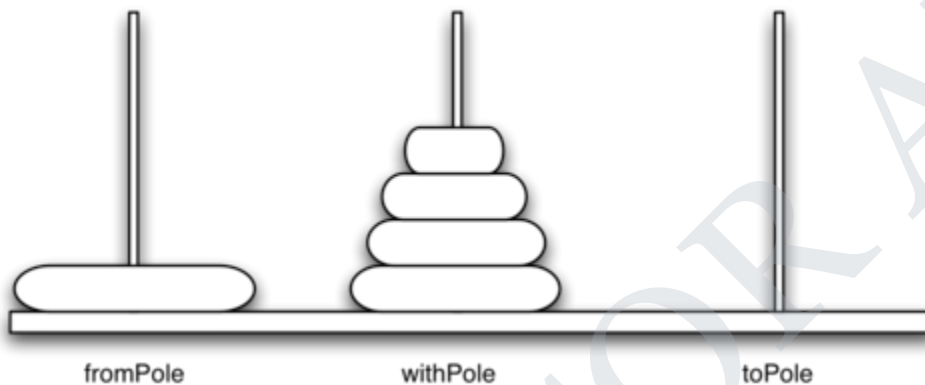
.



The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. He was inspired by a legend that tells of a Hindu temple where the puzzle was presented to young priests. At the beginning of time, the priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it. Their assignment was to transfer all 64 disks from one of the three poles to another, with two important constraints. They could only move one disk at a time, and they could never place a larger disk on top of a smaller one. The priests worked very efficiently, day and night, moving one disk every second. When they finished their work, the legend said, the temple would crumble into dust and the world would vanish.

Although the legend is interesting, you need not worry about the world ending any time soon. The number of moves required to correctly move a tower of 64 disks is  $2^{64}-1=18,446,744,073,709,551,615$ . At a rate of one move per second, that is 584,942,417,355 years! Clearly there is more to this puzzle than meets the eye.

Figure 1 shows an example of a configuration of disks in the middle of a move from the first peg to the third. Notice that, as the rules specify, the disks on each peg are stacked so that smaller disks are always on top of the larger disks. If you have not tried to solve this puzzle before, you should try it now. You do not need fancy disks and poles—a pile of books or pieces of paper will work.



How do we go about solving this problem recursively? How would you go about solving this problem at all? What is our base case? Let's think about this problem from the bottom up. Suppose you have a tower of five disks, originally on peg one. If you already knew how to move a tower of four disks to peg two, you could then easily move the bottom disk to peg three, and then move the tower of four from peg two to peg three. But what if you do not know how to move a tower of height four? Suppose that you knew how to move a tower of height three to peg three; then it would be easy to move the fourth disk to peg two and move the three from peg three on top of it. But what if you do not know how to move a tower of three? How about moving a tower of two disks to peg two and then moving the third disk to peg three, and then moving the tower of height two on top of it? But what if you still do not know how to do this? Surely you would agree that moving a single disk to peg three is easy enough, trivial you might even say. This sounds like a base case in the making.

Here is a high-level outline of how to move a tower from the starting pole, to the goal pole, using an intermediate pole:

1. Move a tower of height-1 to an intermediate pole, using the final pole.
2. Move the remaining disk to the final pole.
3. Move the tower of height-1 from the intermediate pole to the final pole using the original pole.

As long as we always obey the rule that the larger disks remain on the bottom of the stack, we can use the three steps above recursively, treating any larger disks as though they were not even there. The only thing missing from the outline above is the identification of a base case. The simplest Tower of Hanoi problem is a tower of one disk. In this case, we need move only a single disk to its final destination. A tower of one disk will be our base case. In addition, the steps outlined above move us toward the base case by reducing the height of the tower in steps 1 and 3. Listing 1 shows the Python code to solve the Tower of Hanoi puzzle.

```
def moveTower(height,fromPole, toPole, withPole):  
    if height >= 1:  
        moveTower(height-1,fromPole,withPole,toPole)  
        moveDisk(fromPole,toPole)  
        moveTower(height-1,withPole,toPole,fromPole)  
def moveDisk(fp,tp):  
    print("moving disk from",fp,"to",tp)  
moveTower(3,"A","B","C")
```

## **OUTPUT**

```
runfile('C:/Users/admin/.anaconda/navigator/toh.py',wdir='C:/Users/admin/.anaconda/navigator')  
moving disk from A to B  
moving disk from A to C  
moving disk from B to C  
moving disk from A to B  
moving disk from C to A  
moving disk from C to B  
moving disk from A to B
```

**UNIT II DATA, EXPRESSIONS, STATEMENTS****9**

Python interpreter and interactive mode; values and types: int, float, boolean, string, and list; variables, expressions, statements, tuple assignment, precedence of operators, comments; modules and functions, function definition and use, flow of execution, parameters and arguments; Illustrative programs: exchange the values of two variables, circulate the values of n variables distance between two points.

**Unit-II****Two Marks****1. Define values and types**

A value is one of the basic things in a program like a letter or a number. The values are belonging to different types. For example

```
>>> type('Hello')
```

```
<type 'str'>
```

```
>>> type(17)
```

```
<type 'float'>
```

**2. Define variables.**

A variable is a name that refers to a value. They can contain both letters and numbers but they do not begin with a letter. The underscore '\_' can appear in a name.

Example:

```
>>> n=176
```

Here n is a variable name. The value of that variable is 176.

**3. Define Boolean Operators.**

A Boolean expression is an expression that is either true (or) false. The Operator == which compares two operands and produces True if they are equal otherwise it is false.

```
>>> 5==5
```

```
True
```

```
>>> 5==6
```

```
False
```

**4. Define String**

A string is a sequence of characters. We can access the character one at a time.

```
>>> fruit='banana'
```



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

```
>>> print letter
```

Output:

B

## 5. Define List.

A List is a sequence of values. The values are characters in a list they can be of any type. The values in a list are called elements.

Examples

[10,20,30,40] → number List

['a','b','c','d'] → Character List

['a',20,5,25.5,[20,30]] → Nested List

[] → Empty List

## 6. What do you mean by mutable list.

The values in the list are changed. So it is called mutable list.

```
>>> n=[17,35]
```

```
>>> n[0]=5
```

```
>>> n=[5,35] → n[0] is changed as 5.
```

## 7. Define Expression

An expression is a combination of values, variables and operators. A value all by itself is considered an expression.

## 8. Define Tuples

A Tuple is a sequence of values. The values can be of any type, They are indexed by integers.

Tuples are immutable. A tuple is a comma separated list of values.

```
>>> t=('a','b','c')
```

## 9. What is the function of slice in tuple?

It prints the values from starting to end-1.

```
>>> t[1:3]
```

```
>>('a','b')
```

10. Write a program for swapping of two numbers?

```
a=5
```

```
b=6
```

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

```
print (a,b)
```

11. What are different Membership Operators in python?

In – Evaluates to true, if it finds a variable in the specified sequence and false otherwise.

Not in- Evaluates to true, if it does not finds a variable in the specified sequence and false otherwise.

12. Define Functions

A function is a named sequence of statements that performs a computation. A function takes an argument and returns a result. The result is called the return value.

Example

```
def print_1():
```

```
    print ("welcome")
```

13. Define Module

A module is a file that contains a collection of related functions.

```
>>> import math
```

```
>>> print math
```

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

14. What are the different types of arguments?

1. Function arguments

2. keyword arguments

3. Default arguments

4. variable length arguments.

**PART-B (16 marks)**

1. Define Operators. What are the different types of Operators in python? Explain in detail.

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

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

## Types of Operator

Python language supports the following types of operators.

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

## Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a ** b = 10$ to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	$9 // 2 = 4$ and $9.0 // 2.0 = 4.0$ , $-11 // 3 = -4$ , $-11.0 // 3 = -4.0$

## Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	
>	If the value of left operand is greater than the value of right operand, then (a > b) is not true. condition becomes true.	
<	If the value of left operand is less than the value of right operand, then (a < b) is true. condition becomes true.	
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

### Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+=	Add It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a ac /= a is equivalent to c = c / a
%=	Modulus It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	Floor Division It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

## Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if  $a = 60$ ; and  $b = 13$ ; Now in binary format they will be as follows –

$a = 0011\ 1100$

$b = 0000\ 1101$

-----

$a \& b = 0000\ 1100$

$a | b = 0011\ 1101$

$a \wedge b = 0011\ 0001$

$\sim a = 1100\ 0011$

There are following Bitwise operators supported by Python language

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

## Python Logical Operators

There are following logical operators supported by Python language. Assume variable  $a$  holds 10 and variable  $b$  holds 20 then

Used to reverse the logical state of its operand.

## Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below

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

### Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below:

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

### Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += **=	Assignment operators
**=	
is is not	Identity operators
in not in	Membership operators

not or and

Logical operators

### 1. Define function?.What are the different types of arguments in python

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.

### Defining a Function

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

### Syntax

```
def functionname( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

### Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
    "This prints a passed string into this function"
    print str
    return
```

### Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call `printme()` function –

```
#!/usr/bin/python
```



```
def printme( str ):
    "This prints a passed string into this function"
    print str
    return;

printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

I'm first call to user defined function!  
Again second call to the same function

### Pass by reference vs value

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

```
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist.append([1,2,3,4]);
    print "Values inside the function: ", mylist
    return

mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

### Function Arguments

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

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

### Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

```
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
```

```

    print str
    return;
printme()

```

When the above code is executed, it produces the following result:

```

Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)

```

## Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways –

```

#!/usr/bin/python

# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str
    return;

# Now you can call printme function
printme( str = "My string")
My string

```

## Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

```

def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print "Name: ", name
    print "Age ", age
    return;

printinfo( age=50, name="miki" )
printinfo( name="miki" )
Name: miki
Age 50
Name: miki
Age 35

```

## Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this –

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example –

```
# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return;
```

```
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
Output is:
10
Output is:
70
60
50
```

### 1.Simple Calculator Program

```
def sum(a,b):
```

```
    return a+b
```

```
def sub(a,b):
```

```
    return a-b
```

```
def mul(a,b):
```

```
    return a*b
```

```
def div(a,b):
```

```
    return(a//b)

print ("Addition", sum(5,6))

print ("Subtraction",sub(8,4))

print ("Multiplication",mul(8,4))

print ("Division",div(8,2))
```

**Output:**

Addition 11

Subtraction 4

Multiplication 32

Division 4

**2.Circulate n values in a List**

```
def rotate(l,y=1):

    if(len(l)==0):

        return 1

    y=y%len(l)

    return l[y:]+l[:y]

print(rotate([1,2,3,4]))

print(rotate([2,3,4,1]))

print(rotate([3,4,1,2]))

print(rotate([3,2,1,4]))
```

**Output:**

[2, 3, 4, 1]

[3, 4, 1, 2]

[4, 1, 2, 3]

[2, 1, 4, 3]

**3. Swapping of two variables**

```
def swap(a,b):
```

```
print("Values before swapping", a,b)

a,b=b,a

print("Values after swapping", a,b)

swap(4,5)
```

**Output:**

Values before swapping 4 5

Values after swapping 5 4

**4. Distance between two points**

```
import math

p1 = [4, 0]

p2 = [6, 6]

distance = math.sqrt( ((p1[0]-p2[0])**2)+((p1[1]-p2[1])**2) )

print(distance)
```

**Output:**

**Distance between two points**

**6.324555320336759**

**5.Sum of n numbers**

```
def sum(n):

    s=0

    for i in range(1,n):

        s=s+i

    return s

n=input("Enter the input value of n")

n=int(n)

print("Sum of n numbers", sum(n))
```

**Output:**

**Enter the input value of n 10**

**Sum of n numbers 45****6. Sum of odd and even numbers**

```
def sumofevenodd(n):
```

```
    se=0
```

```
    so=0
```

```
    for i in range(1,n):
```

```
        if i%2==0:
```

```
            se=se+i
```

```
        else:
```

```
            so=so+i
```

```
    return se,so
```

```
n=input("Enter the input value of n")
```

```
n=int(n)
```

```
print("Sum of n numbers", sumofevenodd(n))
```

**Output:**

**Enter the input value of n 10**

**Sum of n numbers (20, 25)**

**7. Variable Length arguments**

```
def sum(farg,*args):
```

```
    s=0
```

```
    for i in args:
```

```
        s=s+i
```

```
    print ("Sum of three numbers",s)
```

```
sum(1,4,5,6)
```

**Output:**

Sum of three numbers 15

```
def someFunction(**kwargs):
```

```
if 'text' in kwargs:  
    print (kwargs['text'])  
someFunction(text='welcome')
```

**Output:**

Welcome

**9. Keyword arguments**

```
def print_info(name,age=25):  
    print ("Name:",name)  
    print("Age",age)  
print_info(name='Uma',age=35)  
print_info(name='Reka')
```

**Output:**

Name: Uma

Age 35

Name: Reka

Age 25

**UNIT III CONTROL FLOW, FUNCTIONS****9**

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

**UNIT 3****Conditionals:****Boolean values:**

Boolean values are the two constant objects False and True. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively.

A string in Python can be tested for truth value.

The return type will be in Boolean value (True or False)

To see what the return value (True or False) will be, simply print it out.

```
str="Hello World"
```

```
print str.isalnum()      #False    #check if all char are numbers
print str.isalpha()      #False    #check if all char in the string are alphabetic
print str.isdigit()      #False    #test if string contains digits
print str.istitle()      #True     #test if string contains title words
print str.isupper()      #False    #test if string contains upper case
print str.islower()      #False    #test if string contains lower case
print str.isspace()      #False    #test if string contains spaces
print str.endswith('d') #True     #test if string ends with a d
print str.startswith('H') #True    #test if string starts with H
```

The if statement

Conditional statements give us this ability to check the conditions. The simplest form is the if statement, which has the general form:



if BOOLEAN EXPRESSION:

STATEMENTS

A few important things to note about if statements:

The colon (:) is significant and required. It separates the header of the compound statement from the body.

The line after the colon must be indented. It is standard in Python to use four spaces for indenting.

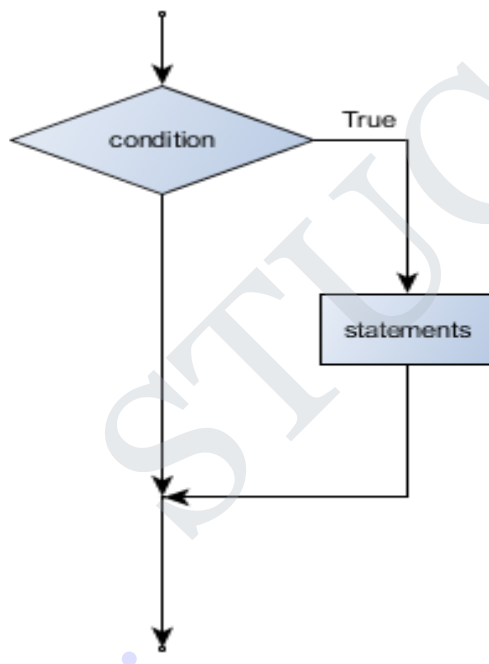
All lines indented the same amount after the colon will be executed whenever the `BOOLEAN_EXPRESSION` is true.

Here is an example:

If `a>0`:

Print ("a is positive")

Flow Chart:



The header line of the if statement begins with the keyword `if` followed by a boolean expression and ends with a colon (:).

The indented statements that follow are called a block. The first unintended statement marks the end of the block. Each statement inside the block must have the same indentation.

if else statement

when a condition is true the statement under if part is executed and when it is false the statement under else part is executed

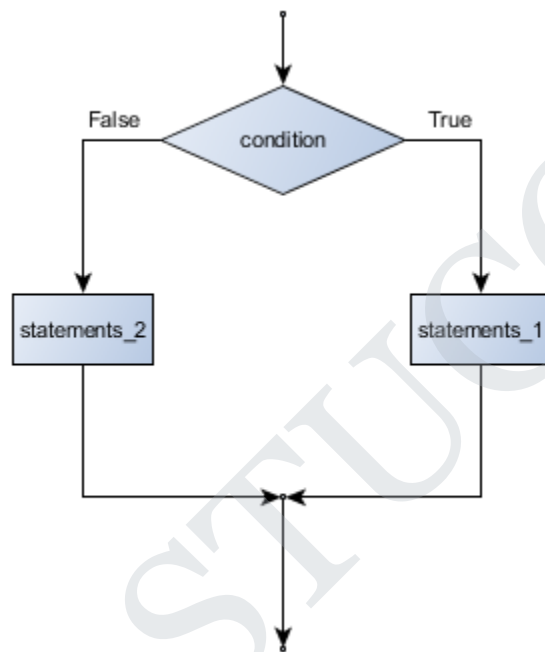
if  $a > 0$ :

print("a is positive")

else:

print("a is negative")

Flow Chart



The syntax for an if else statement looks like this:

if BOOLEAN EXPRESSION:

STATEMENTS\_1      # executed if condition evaluates to True

else:

STATEMENTS\_2      # executed if condition evaluates to False

Each statement inside the if block of an if else statement is executed in order if the boolean expression evaluates to True. The entire block of statements is skipped if the boolean expression evaluates to False, and instead all the statements under the else clause are executed.

```
if True:      # This is always true

    pass      # so this is always executed, but it does nothing

else:

    pass
```

### Chained Conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

```
if x < y:

    STATEMENTS_A

elif x > y:

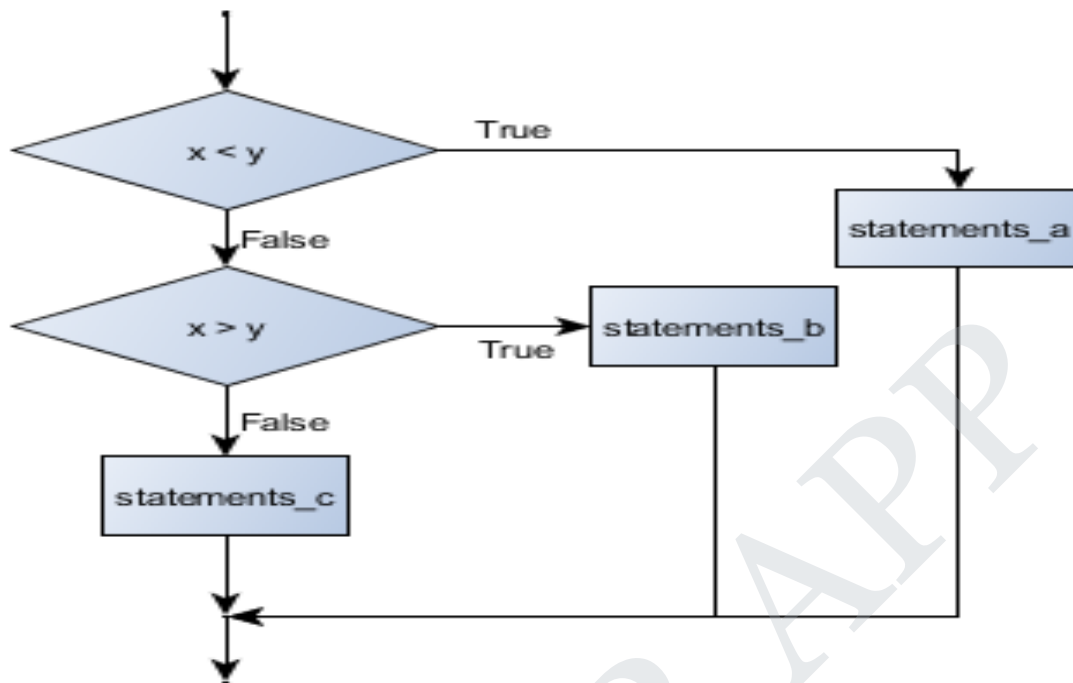
    STATEMENTS_B

else:

    STATEMENTS_C
```

Flowchart of this chained conditional

•



elif is an abbreviation of else if. Again, exactly one branch will be executed. There is no limit of the number of elif statements but only a single (and optional) final else statement is allowed and it must be the last branch in the statement:

```
if choice == '1':
```

```
    print("Monday.")
```

```
elif choice == '2':
```

```
    print("Tuesday")
```

```
elif choice == '3':
```

```
    print("Wednesday")
```

```
else:
```

```
    print("Invalid choice.")
```

### Another Example

```
def letterGrade(score):
```

```
    if score >= 90:
```

```
letter = 'A'

else: # grade must be B, C, D or F

    if score >= 80:

        letter = 'B'

    else: # grade must be C, D or F

        if score >= 70:

            letter = 'C'

        else: # grade must D or F

            if score >= 60:

                letter = 'D'

            else:

                letter = 'F'

return letter
```

### **Program1:**

#### **Largest of two numbers**

```
x1 = int(input("Enter two numbers: "))
x2 = int(input("Enter two numbers: "))

if x1>x2:

    print("x1 is big")

else:

    print("x2 is big")
```

### **Program2:**

#### **Largest of three numbers**

```
num1 = int(input("enter first number"))

num2 = int(input("enter second number"))
```

```
num3 = int(input("enter third number"))  
if (num1 >= num2) and (num1 >= num3):  
    largest = num1  
elif (num2 >= num1) and (num2 >= num3):  
    largest = num2  
else:  
    largest = num3  
print("The largest number between",num1,",",num2,"and",num3,"is",largest)
```

### **Program 3: Roots of a quadratic equation**

```
import math  
a=int(input("Enter a"))  
b=int(input("enter b"))  
c=int(input("enter c"))  
d = b**2-4*a*c  
d=int(d)  
if d < 0:  
    print("This equation has no real solution")  
elif d == 0:  
    x = (-b+math.sqrt(b**2-4*a*c))/2*a  
    print("This equation has one solutions: ", x)  
else:  
    x1 = (-b+math.sqrt(b**2-4*a*c))/2*a  
    x2 = (-b-math.sqrt(b**2-4*a*c))/2*a  
    print ("This equation has two solutions: ", x1, " and", x2)
```

### **Program 4: Odd or even**

```
x=int(input("enter a number"))
```

```
if x%2==0:
```

```
    print("it is Even ")
```

```
else:
```

```
    print("it is Odd")
```

#### **Program 4: Positive or negative**

```
x=int(input("enter a number"))
```

```
if x>0:
```

```
    print("it is positive")
```

```
else:
```

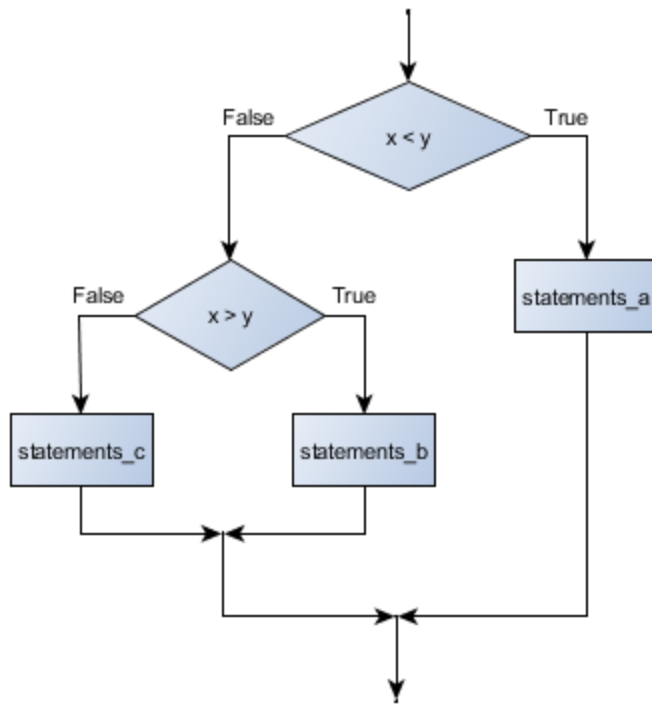
```
    print("it is negative")
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

#### **Nested Conditionals**

#### **Flowchart**

.



if  $x < y$ :

    STATEMENTS\_A

else:

    if  $x > y$ :

        STATEMENTS\_B

    else:

        STATEMENTS\_C

The outer conditional contains two branches. The second branch contains another if statement, which has two, branches of its own. Those two branches could contain conditional statements as well.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```

if 0 < x:           # assume x is an int here
    if x < 10:
        print("x is a positive single digit.")
  
```



## **Iteration**

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

### **While loop**

#### **While statement**

The general syntax for the while statement looks like this:

While BOOLEAN\_EXPRESSION:

STATEMENTS

Like the branching statements and the for loop, the while statement is a compound statement consisting of a header and a body. A while loop executes an unknown number of times, as long as the BOOLEAN\_EXPRESSION is true.

The flow of execution for a while statement works like this:

Evaluate the condition (BOOLEAN\_EXPRESSION), yielding False or True.

If the condition is false, exit the while statement and continue execution at the next statement.

If the condition is true, execute each of the STATEMENTS in the body and then go back to step 1.

#### **Program for Sum of digits**

```
n=int(input("enter n"))
s=0
while(n>0):
    r=n%10
    s=s+r
    n=n/10
print("sum of digits",int(s))
```

#### **OUTPUT**

enter n 122

sum of digits 5

**Program to check Armstrong Number**

```
n=int(input("enter n"))
a=n
a=int(a)
s=0
while(n>0):
    r=n%10
    s=s+r*r*r
    n=n//10
print(s)
if(s==a):
    print("It is an Armstrong number")
else:
    print("It is not an Armstrong number")
```

**Program to check for Number Palindrome**

```
n=int(input("enter n"))
a=n
a=int(a)
s=0
while(n>0):
    r=n%10
    s=s*10+r
    n=n//10
print(s)
if(s==a):
```

```
print("It is a Palindrome")  
  
else:  
  
    print("It is not a Palindrome")
```

### **for loop**

The for loop processes each item in a sequence, so it is used with Python's sequence data types - strings, lists, and tuples.

Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed.

The general form of a for loop is:

for LOOP\_VARIABLE in SEQUENCE:

STATEMENTS

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

```
...     print('i is now:', i)
```

```
i is now 0
```

```
i is now 1
```

```
i is now 2
```

```
i is now 3
```

```
i is now 4
```

```
>>>
```

### **Example 2:**

```
for x in range(13): # Generate numbers 0 to 12
```

```
    print(x, '\t', 2**x)
```

Using the tab character ('\t') makes the output align nicely.

```
0    1
```

```
1    2
```

2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096

Meaning	Math Symbol	Python Symbols
Less than	<	<
Greater than	>	>
Less than or equal to	≤	<=
Greater than or equal to	≥	>=
Equals	=	==
Not equal	≠	!=

### Program for Converting Decimal to Binary

```
def convert(n):
    if n > 1:
        convert(n//2)
    print(n % 2,end = "")

# enter decimal number
dec=int(input("enter n"))
```

```
convert(dec)
```

**OUTPUT**

```
enter n 25
```

```
11001
```

**Program for Converting Decimal to Binary, Octal, HexaDecimal**

```
dec = int(input("enter n"))
```

```
print("The decimal value of",dec,"is:")
```

```
print(bin(dec),"in binary.")
```

```
print(oct(dec),"in octal.")
```

```
print(hex(dec),"in hexadecimal.")
```

**OUTPUT**

```
enter n 25
```

```
The decimal value of 25 is:
```

```
0b11001 in binary.
```

```
0o31 in octal.
```

```
0x19 in hexadecimal.
```

**Program for finding the Factorial**

```
n= int(input("enter a number"))
```

```
f = 1
```

```
for i in range(1,n+1):
```

```
    f = f*i
```

```
print("The factorial of",n,"is",f)
```

**OUTPUT**

```
enter a number 7
```

```
The factorial of 7 is 5040
```

**Prime Number Generation between different intervals:**

```
start=int(input("enter start value"))
end=int(input("enter end value"))
print("Prime numbers between",start,"and ",end,"are:")
for num in range(start,end + 1):
    # prime numbers are greater than 1
    if num > 1:
        for i in range(2,num):
            if (num % i) == 0:
                break
        else:
            print(num)
```

**OUTPUT**

enter start value 0

enter end value 25

Prime numbers between 0 and 25 are:

2

3

5

7

11

13

17

19

23

**Program for swapping two numbers:**

```
a = int(input("enter a"))
b=int(input("enter b"))
print("before swapping\n a=", a, " b=", b)
temp = a
a = b
b = temp
print("\nafter swapping\n a=", a, " b=", b)
```

**Program for swapping two numbers without using temporary variable**

```
a = int(input("enter a"))
b=int(input("enter b"))
print("before swapping\n a=", a, " b=", b)
a,b=b,a
print("\nafter swapping\n a=", a, " b=", b)
```

**Program for Four Function Calculator**

```
print("Epr any two number: ")
n1 = int(input("enter first number"))
n2 = int(input("enter second number"))
ch = input("Enter operator (+,-,*,/): ")
if ch == '+':
    res = n1+n2
    print(n1, "+", n2, "=", res)
elif ch == '-':
    res = n1-n2
    print(n1, "-", n2, "=", res)
```

```

elif ch == '*':

    res = n1 * n2

    print(n1, "*", n2, "=", res)

elif ch == '/':

    res = n1 / n2

    print(n1, "/", n2, "=", res)

else:

    print("enter a valid operator")

```

### **Fruitful function**

**Fruitful function is a special function in which function is defined in the return statement.** It is a function with return type as expression. The first example is area, which returns the area of a circle with the given radius:

```

def cal(r1):

    return 3.14*r1*r1

r=int(input("enter r"))

print("ans=",cal(r))

```

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

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```

def absoluteValue(x):
    if x<0:
        return -x
    else:
        return x

```

Since these return statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.



To deal with increasingly complex programs, you might want to try a process called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$ . By the Pythagorean theorem, the distance is:

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

### **Program for Distance Calculation**

```
import math
def distance(x1, y1, x2, y2):
    dx=x2-x1
    dy=y2-y1
    return (math.sqrt(dx**2)+(dy**2))
```

```
x1=int(input("enter x1"))
y1=int(input("enter y1"))
x2=int(input("enter x2"))
y2=int(input("enetr y2"))
print(distance(x1,y1,x2,y2))
```

### **Fibonocci using recursion and fruitful function**

```
def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))
```

```
nterms = int(input("How many terms? "))
```

```
if nterms <= 0:
```

```
    print("Plese enter a positive integer")
```

```
else:
```

```
    print("Fibonacci sequence:")
```

```
for i in range(terms):

    print(recur_fibo(i))
```

### **Function Composition in python**

Function composition is a way of combining functions such that the result of each function is passed as the argument of the next function. For example, the composition of two functions  $f$  and  $g$  is denoted  $f(g(x))$ .  $x$  is the argument of  $g$ , the result of  $g$  is passed as the argument of  $f$  and the result of the composition is the result of  $f$ .

Let's define `compose2`, a function that takes two functions as arguments ( $f$  and  $g$ ) and returns a function representing their composition:

```
def compose2(f, g):

    return lambda x: f(g(x))
```

Example:

```
>>> def double(x):
...     return x * 2
...
>>> def inc(x):
...     return x + 1
...
>>> inc_and_double = compose2(double, inc)
>>> inc_and_double(10)
2
```

### **Syntax**

```
pass
```

Example

```
#!/usr/bin/python3
```

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

### Output

When the above code is executed, it produces the following result –

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

## Local and Global Scope variables

### Local Variables

Define the variables inside the function definition, they are local to this function by default. This means that anything will do to such a variable in the body of the function will have no effect on other variables outside of the function, even if they have the same name. This means that the function body is the scope of such a variable, i.e. the enclosing context where this name with its values is associated.

```
def f():
```

```
print(s)
```

```
s = "I love Paris in the summer!"
```

```
f()
```

Output:

I Love Paris in the summer.

The variable `s` is defined as the string "I love Paris in the summer!", before calling the function `f()`. The body of `f()` consists solely of the `"print(s)"` statement. As there is no local variable `s`, i.e. no assignment to `s`, the value from the global variable `s` will be used. So the output will be the string "I love Paris in the summer!". what will happen, if we change the value of `s` inside of the function `f()`?

```
def f():
```

```
    s = "I love London!"
```

```
    print(s)
```

```
s = "I love Paris!"
```

```
f()
```

```
print(s)
```

Output:

I love London!

I love Paris!

Even though the name of both global and local variables are same. The value of the local variables does not affect the global variables. If we are directly calling the global variable,

without assigning any values to the variable, if both local and global variables are same. It produces an **Unbound Local Error**.

For Example,

```
def f():
```

```
    print(s)
```

```
    s = "I love London!"
```

```
    print(s)
```

```
s = "I love Paris!"
```

```
f()
```

Unbound Local Error: local variable's' referenced before assignment. Because Local variable s referenced before assignment.

**To avoid such Problem,**

We want use global variable in python. We should give keyword global in python

```
def f():
```

```
    global s
```

```
    print(s)
```

```
    s = "Only in spring, but London is great as well!"
```

```
    print(s)
```

```
s = "I am looking for a course in Paris!"
```

```
f()
```

```
print(s)
```

**Output:**

I am looking for a course in Paris!

Only in spring, but London is great as well!

Only in spring, but London is great as well!

### **Recursive Functions:**

A recursive function is a function that calls itself. To prevent a function from repeating itself indefinitely, it must contain at least one selection statement. This statement examines a condition called a **base case** to determine whether to stop or to continue with another **recursive step**.

#### **1. Program for finding the factorial of a given no**

```
def fact(n):  
    if n==0:  
        return 1  
    else:  
        return n*fact(n-1)
```

```
print("The factorial of a given no is",fact(5))
```

#### **Output:**

The factorial of a given no is:120

#### **2. Finding nth Fibonacci Number**

```
def fib(n):  
    if n<3:  
        return 1  
    else:  
        return fib(n-1)+fib(n-2)  
    print("The nth fibonacci no is",fib(10))
```

#### **Output:**

The nth Fibonacci series is 55.

### **Strings**

A string is a sequence of characters.

```
fruit='banana'
```

```
fruit[1]
```

```
Out[3]: 'a'
```

The second statement selects character number 1 from fruit and assigns it to the variable.

The expression in brackets is called an index. The index indicates which character in the Sequence.

### **Len**

Len is a inbuilt function. It returns number of characters in a string.

```
len(fruit)
```

```
Out[4]: 6
```

### **String Slices:**

A segment of a string is called string slices.

```
Fruit='banana'
```

```
Fruit[1:4]
```

It returns the output as 'ana'. It includes the first character and excludes the last character in a string.

```
Fruit[:3]
```

It includes from the first character. It produces the output as 'ban'.

```
Fruit[3:]
```

It includes the end of the string. It returns output as 'ana'

```
Fruit[3:3]
```

If first index is greater than or equal to the second index. Then it returns empty string as output'

```
Fruit[:]
```

It returns the entire string. Out put is 'banana'.

```
Fruit[-1]
```

It returns the last character in a string. Output 'a'.

### **String Immutable**

We cant change the value of the strings.

```
fruit[0]='w'
```

Traceback (most recent call last):

File "<ipython-input-10-839a456c8838>", line 1, in <module>

```
fruit[0]='w'
```

TypeError: 'str' object does not support item assignment

Strings are immutable.

String Methods

### **Built-in String Methods**

A method is similar to a function. It takes an argument and returns the values. A method call is called invocation.

Python includes the following built-in methods to manipulate strings –

SN	Methods with Description
1	capitalize() Capitalizes first letter of string
2	center(width, fillchar) Returns a space-padded string with the original string centered to a total of width columns.
3	count(str, beg= 0,end=len(string)) Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	decode(encoding='UTF-8',errors='strict') Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	encode(encoding='UTF-8',errors='strict') Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.



6	<code>endswith(suffix, beg=0, end=len(string))</code> Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	<code>expandtabs(tabsize=8)</code> Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	<code>find(str, beg=0 end=len(string))</code> Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	<code>index(str, beg=0, end=len(string))</code> Same as find(), but raises an exception if str not found.
10	<code>isalnum()</code> Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	<code>isalpha()</code> Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	<code>isdigit()</code> Returns true if string contains only digits and false otherwise.
13	<code>islower()</code> Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	<code>isnumeric()</code>

	Returns true if a unicode string contains only numeric characters and false otherwise.
15	<p>isspace()</p> <p>Returns true if string contains only whitespace characters and false otherwise.</p>
16	<p>istitle()</p> <p>Returns true if string is properly "titlecased" and false otherwise.</p>
17	<p>isupper()</p> <p>Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.</p>
18	<p>join(seq)</p> <p>Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.</p>
19	<p>len(string)</p> <p>Returns the length of the string</p>
20	<p>ljust(width[, fillchar])</p> <p>Returns a space-padded string with the original string left-justified to a total of width columns.</p>
21	<p>lower()</p> <p>Converts all uppercase letters in string to lowercase.</p>
22	<p>lstrip()</p> <p>removes all leading whitespace in string.</p>
23	<p>maketrans()</p> <p>Returns a translation table to be used in translate function.</p>

24	<code>max(str)</code> Returns the max alphabetical character from the string str.
25	<code>min(str)</code> Returns the min alphabetical character from the string str.
26	<code>replace(old, new [, max])</code> Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	<code>rfind(str, beg=0, end=len(string))</code> Same as find(), but search backwards in string.
28	<code>rindex( str, beg=0, end=len(string))</code> Same as index(), but search backwards in string.
29	<code>rjust(width,[, fillchar])</code> Returns a space-padded string with the original string right-justified to a total of width columns.
30	<code>rstrip()</code> Removes all trailing whitespace of string.
31	<code>split(str="", num=string.count(str))</code> Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	<code>splitlines( num=string.count('\n'))</code> Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.

33	<p><code>startswith(str, beg=0, end=len(string))</code></p> <p>Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.</p>
34	<p><code>strip([chars])</code></p> <p>Performs both <code>lstrip()</code> and <code>rstrip()</code> on string</p>
35	<p><code>swapcase()</code></p> <p>Inverts case for all letters in string.</p>
36	<p><code>title()</code></p> <p>Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.</p>
37	<p><code>translate(table, deletechars="")</code></p> <p>Translates string according to translation table str(256 chars), removing those in the del string.</p>
38	<p><code>upper()</code></p> <p>Converts lowercase letters in string to uppercase.</p>
39	<p><code>zfill (width)</code></p> <p>Returns original string leftpadded with zeros to a total of width characters; intended for numbers, <code>zfill()</code> retains any sign given (less one zero).</p>
40	<p><code>isdecimal()</code></p> <p>Returns true if a unicode string contains only decimal characters and false otherwise.</p>

### **The string module**

This module contains a number of functions to process standard Python strings. In recent versions, most functions are available as string methods as well (more on this below).

### Example: Using the string module

# File: string-example-1.py

```
import string
text = "Monty Python's Flying Circus"
print "upper", "=>", string.upper(text)
print "lower", "=>", string.lower(text)
print "split", "=>", string.split(text)
print "join", "=>", string.join(string.split(text), "+")
print "replace", "=>", string.replace(text, "Python", "Java")
print "find", "=>", string.find(text, "Python"), string.find(text, "Java")
print "count", "=>", string.count(text, "n")
```

```
upper => MONTY PYTHON'S FLYING CIRCUS
lower => monty python's flying circus
split => ['Monty', 'Python's', 'Flying', 'Circus']
join => Monty+Python's+Flying+Circus
replace => Monty Java's Flying Circus
find => 6 -1
count => 3
```

### Example: Using string methods instead of string module functions

# File: string-example-2.py

```
text = "Monty Python's Flying Circus"
print "upper", "=>", text.upper()
print "lower", "=>", text.lower()
print "split", "=>", text.split()
print "join", "=>", "+".join(text.split())
print "replace", "=>", text.replace("Python", "Perl")
print "find", "=>", text.find("Python"), text.find("Perl")
```

```
print "count", "=>", text.count("\n")
```

```
upper => MONTY PYTHON'S FLYING CIRCUS
```

```
lower => monty python's flying circus
```

```
split => ['Monty', 'Python's', 'Flying', 'Circus']
```

```
join => Monty+Python's+Flying+Circus
```

```
replace => Monty Perl's Flying Circus
```

```
find => 6 -1
```

```
count => 3
```

In addition to the string manipulation stuff, the **string** module also contains a number of functions which convert strings to other types:

**Example: Using the string module to convert strings to numbers**

```
# File: string-example-3.py
```

```
import string
```

```
print int("4711"),
```

```
print string.atoi("4711"),
```

```
print string.atoi("11147", 8), # octal
```

```
print string.atoi("1267", 16), # hexadecimal
```

```
print string.atoi("3mv", 36) # whatever...
```

```
print string.atoi("4711", 0),
```

```
print string.atoi("04711", 0),
```

```
print string.atoi("0x4711", 0)
```

```
print float("4711"),
```

```
print string.atof("1"),
```

```
print string.atof("1.23e5")
```

```
4711 4711 4711 4711 4711
```

```
4711 2505 18193
4711.0 1.0 123000.0
```

### Programs

1. Write a function that takes a string as an argument and displays the letters backward, one per line.

```
def revword(s):
    for c in reversed(s):
        print(c)
```

```
revword('string')
```

Output:

```
g
n
i
r
t
s
```

### 3. Searching a letter present in word

```
def find(word, letter):
    index=0
    while index<len(word):
        if word[index]==letter:
            return index
        index=index+1
    return -1

print("the letter present", find('malayalm','l'))
```

Output:

The letter present 2

### 4. Program for count the number of characters present in a word.

```
def find(word,ch):
    count=0
    for letter in word:
        if letter==ch:
            count=count+1
    return count
print("The total count for the given letter", find('malayalm','l'))
```

Output:

The total count for the given letter 2

### 3. String Palindrome

```
def palin(word):
    y=word[::-1]
    if word==y:
        print("Palindrome")
    else:
        print("Not Palindrome")
palin('good')
```

Output:

Not Palindrome.

### 5. Write a Program for GCD of two numbers

```
def ComputeGCD(x,y):
    while y:
        x,y=y,x%y
    return x
print("GCD of two numbers", ComputeGCD(24,12))
```

Output:

GCD of two numbers 12

### 6. Exponent of a number

```
def expo(x,y):
    exp=x**y
```



```
print(exp)
expo(5,2)
```

**Output**

25

STUCOR APP

.

**UNIT IV LISTS, TUPLES, DICTIONARIES**

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

**LISTS, TUPLES, DICTIONARIES****4.1 LISTS:**

A list is a sequence of values. In a string, the values are characters but in a list, list values can be any type.

**Elements or Items:**

The values in a list are called elements or items. list must be enclosed in square brackets ([and]).

**Examples:**

```
>>>[10,20,30]
>>>['hi','hello','welcome']
```

**Nested List:**

A list within another list is called nested list.

**Example:**

```
['good',10,[100,99]]
```

**Empty List:**

A list that contains no elements is called empty list. It can be created with empty brackets, [].

**Assigning List Values to Variables:****Example:**

```
>>>numbers=[40,121]
>>>characters=['x','y']
>>>print(numbers,characters)
```

**Output:**

```
[40,121]['x','y']
```

**4.1.1 LIST OPERATIONS:****Example 1:**

The + operator concatenates lists:

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

**Output:** [1,2,3,4,5,6]

**Example 2:**

The \* operator repeats a list given number of times:

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

**Output:**

```
[0,0,0,0]
```

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

**Output:**

```
[1,2,3,1,2,3,1,2,3]
```

The first example repeats [0] four times. The second example repeats [1,2,3] three times.

**4.1.2 LIST SLICES:**

List slicing is a computationally fast way to methodically access parts of given data.

**Syntax:**

Listname[start:end:step] where **:end** represents the first value that is not in the selected slice. The difference between end and start is the number of elements selected (if step is 1, the default). The start and end may be a negative number. For negative numbers, the count starts from the end of the array instead of the beginning.

**Example:**

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

Lists are mutable, so it is useful to make a copy before performing operations that modify this. A slice operator on the left side of an assignment can update multiple elements.

**Example:**

```
>>>t[1:3]=['x','y']
```

```
>>>t
```

**Output:**

```
['a','x','y','d','e','f']
```

**4.1.3 LIST METHODS:**

Python provides methods that operate on lists.

**Example:**

1. append adds a new element to the end of a list.

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

```
>>>t.append('e')
```

```
>>>t
```

**Output:**

```
['a','b','c','d','e']
```

2. extend takes a list as an argument and appends all the elements.

```
>>>t1=['a','b']
```

```
>>>t2=['c','d']
```

```
>>>t1.extend(t2)
```

```
>>>t1
```

**Output:**

```
['a','b','c','d']
```

t2 will be unmodified.

**4.1.4 LIST LOOP:**

List loop is traversing the elements of a list with a for loop.

**Example 1:**

```
>>>mylist=[[1,2],[4,5]]
```

```
>>>for x in mylist:
```

```
    if len(x)==2:
```

```
        print x
```

**Output:**

```
[1,2]
```

```
[4,5]
```

**Example 2:**

```
>>>for i in range(len(numbers)):
```

```
    Numbers[i]=numbers[i]*2
```

Above example is used for updating values in numbers variables. Above loop traverses the list and updates each element. Len returns number of elements in the list. Range returns a list of indices from 0 to n-1, where n is the length of the list. Each time through the loop i gets the index of next element. A for loop over an empty list never runs the body:

**Example:**

```
for x in []:
    print('This won't work')
```

A list inside another list counts as a single element. The length of the below list is four:

```
['spam',1,['x','y'],[1,2,3]]
```

**Example 3:**

```
colors=["red","green","blue","purple"]
for i in range(len(colors)):
    print(colors[i])
```

**Output:**

```
red
green
blue
purple
```

**4.1.5 MUTABILITY:**

Lists are mutable. Mutable means, we can change the content without changing the identity. Mutability is the ability for certain types of data to be changed without entirely recreating it. Using mutable data types can allow programs to operate quickly and efficiently.

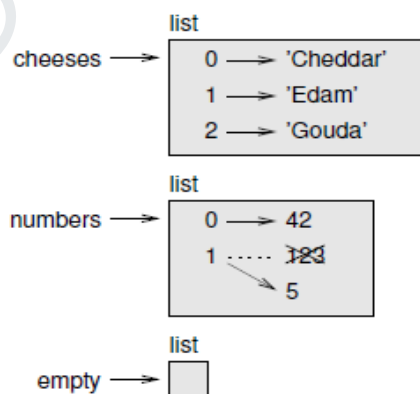
**Example for mutable data types are:**

List, Set and Dictionary

**Example 1:**

```
>>>numbers=[42,123]
>>>numbers[1]=5
>>>numbers
[42,5]
```

Here, the second element which was 123 is now turned to be 5.



**Figure 4.1 shows the state diagram for cheeses, numbers and empty:**

Lists are represented by boxes with the word “list” outside and the elements of the list inside. cheeses refers to a list with three elements indexed 0, 1 and 2.

numbers contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. empty refers to a list with no elements.

The in operator also works on lists.

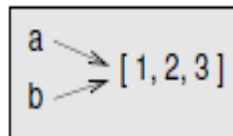
```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

#### 4.1.6 ALIASING

If a refers to an object and we assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

The state diagram looks like Figure 4.2.



**Figure 4.2 State Diagram**

The association of a variable with an object is called a **reference**. In this example, there are two references to the same object. An object with more than one reference has more than one name, so we say that the object is **aliased**. If the aliased object is mutable, changes made with one alias affect the other:

#### 4.1.7 CLONING LISTS Copy list v Clone list

```
veggies=["potatoes","carrots","pepper","parsnips","swedes","onion","minehead"]
veggies[1]="beetroot"
# Copying a list gives it another name
daniel=veggies
# Copying a complete slice CLONES a list
david=veggies[:]
daniel[6]="emu"
# Daniel is a SECOND NAME for veggies so that changing Daniel also changes veggies.
# David is a COPY OF THE CONTENTS of veggies so that changing Daniel (or
veggies) does NOT change David.
# Changing carrots into beetroot was done before any of the copies were made, so will
affect all of veggies, daniel and david
for display in (veggies, daniel, david):
    print(display)
```

#### Output:

```
['potatoes', 'beetroot', 'pepper', 'parsnips', 'swedes', 'onion', 'emu']
['potatoes', 'beetroot', 'pepper', 'parsnips', 'swedes', 'onion', 'emu']
['potatoes', 'beetroot', 'pepper', 'parsnips', 'swedes', 'onion', 'minehead']
```

#### 4.1.8 LIST PARAMETERS

When we pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, delete\_head removes the first element from a list:

**Example:**

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
```

**Output:**

```
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object. The stack diagram looks like Figure 4.3. Since the list is shared by two frames, I drew it between them. It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list.

**Example 1:**

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
```

**Output:**

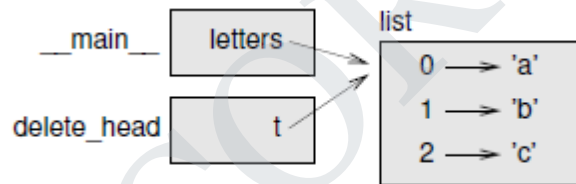
```
[1, 2, 3]
```

**Example 2:**

```
>>> t2
```

**Output:**

```
None
```



**Figure 4.3 Stack Diagram**

## 4.2. TUPLES

- A tuple is a sequence of values.
- The values can be any type and are indexed by integers, unlike lists.
- Tuples are immutable.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, we have to include a final comma:

```
>>> t1 = 'a',
>>> type(t1)
```

**Output:**

```
<class 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
```

**Output:**

```
<class 'str'>
```

Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple.

**Example:**

```
>>> t = tuple()
>>> t
```

**Output:** `()`

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

**Example:**

```
>>> t = tuple('lupins')
>>> t
```

**Output:** `('l', 'u', 'p', 'i', 'n', 's')`

Because tuple is the name of a built-in function, we should avoid using it as a variable name. Most list operators also work on tuples. The bracket operator indexes an element:

**Example:**

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

**Output:** `'a'` And the slice operator selects a range of elements.

**Example:**

```
>>> t[1:3]
```

**Output:** `('b', 'c')` But if we try to modify one of the elements of the tuple, we get an error:

```
>>> t[0] = 'A'
```

`TypeError: object doesn't support item assignment` Because tuples are immutable, we can't modify the elements. But we can replace one

tuple with another: **Example:**

```
>>> t = ('A',) + t[1:]
>>> t
```

**Output:**

```
('A', 'b', 'c', 'd', 'e')
```

This statement makes a new tuple and then makes `t` refer to it.

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

**Example 1:**

```
>>> (0, 1, 2) < (0, 3, 4)
```

**Output:**

```
True
```

**Example 2:**

```
>>> (0, 1, 2000000) < (0, 3, 4)
```

**Output:**

```
True
```

#### 4.2.1 TUPLE ASSIGNMENT

It is often useful to swap the values of two variables. With conventional assignments, we have to use a temporary variable. For example, to swap `a` and `b`:

**Example:**

```
>>> temp = a
>>> a = b
>>> b = temp
```

This solution is cumbersome; **tuple assignment** is more elegant:

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

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. The number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b = 1, 2, 3
```

ValueError: too many values to unpack

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, we could write:

```
>>> addr = 'monty@python.org'
```

```
>>> uname, domain = addr.split('@')
```

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

```
>>> uname
```

```
'monty'
```

```
>>> domain
```

```
'python.org'
```

## 4.2.2 TUPLE AS RETURN VALUES

A function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if we want to divide two integers and compute the quotient and remainder, it is inefficient to compute  $x/y$  and then  $x\%y$ . It is better to compute them both at the same time. The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. We can store the result as a tuple:

**Example:**

```
>>> t = divmod(7, 3)
```

```
>>> t
```

**Output:**

```
(2, 1)
```

Or use tuple assignment to store the elements separately:

**Example:**

```
>>> quot, rem = divmod(7, 3)
```

```
>>> quot
```

**Output:**

```
2
```

**Example:**

```
>>> rem
```

**Output:**

```
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
```

```
    return min(t), max(t)
```

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

## 4.3 DICTIONARIES

Dictionaries have a method called items that returns a sequence of tuples, where each tuple is a key-value pair.

### 4.3.1 OPERATIONS AND METHODS

**Example:**

```
>>> d = {'a':0, 'b':1, 'c':2}
```

```
>>> t = d.items()
```

```
>>> t
```



**Output:**

```
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

The result is a dict\_items object, which is an iterator that iterates the key-value pairs. We can use it in a for loop like this:

**Example:**

```
>>> for key, value in d.items():
...     print(key, value)
```

**Output:**

```
c 2
a 0
b 1
```

As we should expect from a dictionary, the items are in no particular order.

Going in the other direction, we can use a list of tuples to initialize a new dictionary:

**Example:**

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
```

**Output:**

```
{'a': 0, 'c': 2, 'b': 1}
```

Combining dict with zip yields a concise way to create a dictionary:

**Example:**

```
>>> d = dict(zip('abc', range(3)))
>>> d
```

**Output:**

```
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method update also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary. It is common to use tuples as keys in dictionaries (primarily because we can't use lists). **For example**, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined last, first and number, we could write: `directory [last, first] = number`

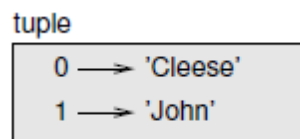
The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

```
For last, first in directory:
    print(first, last, directory[last,first])
```

This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple('Cleeese', 'John') would appear as in Figure 4.4. But in a larger diagram we might want to leave out the details. For example, a diagram of the telephone directory might appear as in Figure 4.5.

Here the tuples are shown using Python syntax as a graphical shorthand. The telephone number in the diagram is the complaints line for the BBC, so please don't call it.



**4.4 State Diagram**

dict

('Cleese', 'John')	→	'08700 100 222'
('Chapman', 'Graham')	→	'08700 100 222'
('Idle', 'Eric')	→	'08700 100 222'
('Gilliam', 'Terry')	→	'08700 100 222'
('Jones', 'Terry')	→	'08700 100 222'
('Palin', 'Michael')	→	'08700 100 222'

#### 4.5 State Diagram

#### 4.4 ADVANCED LIST PROCESSING

List processing is a list of programming codes, including abstract data structure, used to calculate specified variables in a certain order. A value may repeat more than once.

##### 4.4.1 LIST COMPREHENSION

The function shown below takes a list of strings, maps the string method capitalize to the elements, and returns a new list of strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

We can write this more concisely using a **list comprehension**:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the 'for' clause indicates what sequence we are traversing. The syntax of a list comprehension is a little awkward because the loop variable, s in this example, appears in the expression before we get to the definition.

List comprehensions can also be used for filtering. For example, this function selects only the elements of t that are upper case, and returns a new list:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

We can rewrite it using a list comprehension

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster.

But, list comprehensions are harder to debug because we can't put a print statement inside the loop. We can use them only if the computation is simple enough that we are likely to get it right the first time.

#### 4.5 ILLUSTRATIVE PROGRAMS SELECTION SORT

Selection sort is one of the simplest sorting algorithms. It is similar to the hand picking where we take the smallest element and put it in the first position and the second smallest at the second position and so on. It is also similar. We first check for smallest element in the list and swap it with the first element of the list. Again, we check for the smallest number in a sub list, excluding the first element of the list as it is where it should be (at the first position) and put it in

the second position of the list. We continue repeating this process until the list gets sorted.

#### ALGORITHM:

1. Start from the first element in the list and search for the smallest element in the list.
2. Swap the first element with the smallest element of the list.
3. Take a sub list (excluding the first element of the list as it is at its place) and search for the smallest number in the sub list (second smallest number of the entire list) and swap it with the first element of the list (second element of the entire list).
4. Repeat the steps 2 and 3 with new subsets until the list gets sorted.

#### PROGRAM:

```
a = [16, 19, 11, 15, 10, 12, 14]
i = 0
while i < len(a):
    #smallest element in the sublist
    smallest = min(a[i:])
    #index of smallest element
    index_of_smallest = a.index(smallest)
    #swapping
    a[i], a[index_of_smallest] = a[index_of_smallest], a[i]
    i = i + 1
print (a)
```

#### Output

```
>>>
```

```
[10, 11, 12, 14, 15, 16, 19]
```

#### 4.5.2 INSERTION SORT

Insertion sort is similar to arranging the documents of a bunch of students in order of their ascending roll number. Starting from the second element, we compare it with the first element and swap it if it is not in order. Similarly, we take the third element in the next iteration and place it at the right place in the sub list of the first and second elements (as the sub list containing the first and second elements is already sorted). We repeat this step with the fourth element of the list in the next iteration and place it at the right position in the sub list containing the first, second and the third elements. We repeat this process until our list gets sorted.

#### ALGORITHM:

1. Compare the current element in the iteration (say A) with the previous adjacent element to it. If it is in order then continue the iteration else, go to step 2.
2. Swap the two elements (the current element in the iteration (A) and the previous adjacent element to it).
3. Compare A with its new previous adjacent element. If they are not in order, then proceed to step 4.
4. Swap if they are not in order and repeat steps 3 and 4.
5. Continue the iteration.

#### PROGRAM:

```
a = [16, 19, 11, 15, 10, 12, 14]
#iterating over a
for i in a:
    j = a.index(i)
    #i is not the first element
    while j > 0:
        #not in order
        if a[j-1] > a[j]:
```

```

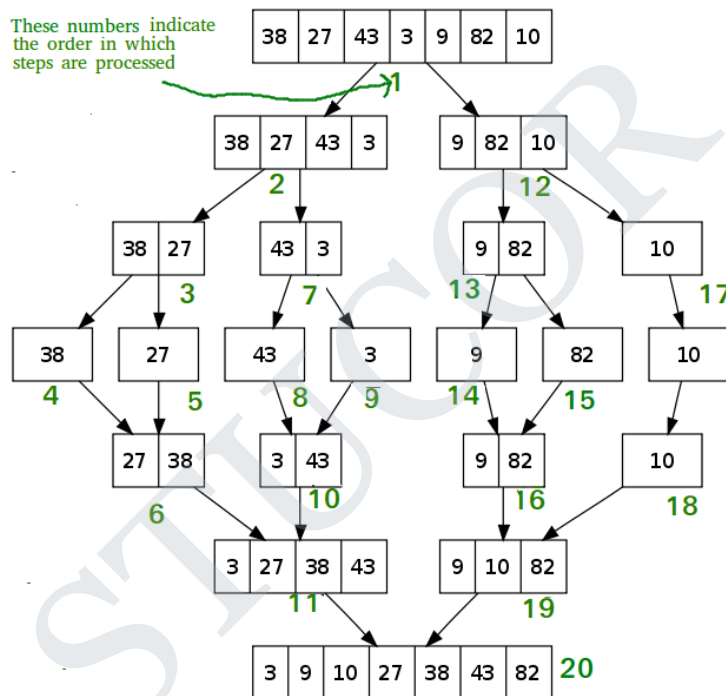
STUCOR APP
#swap
a[j-1],a[j] = a[j],a[j-1]
else:
    #in order
    break
j = j-1
print (a)
Output
>>>
[10, 11, 12, 14, 15, 16, 19]

```

### 4.5.3 MERGE SORT

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



### ALGORITHM:

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:  
middle m = (l+r)/2
2. Call mergeSort for first half:  
Call mergeSort(arr, l, m)
3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

### PROGRAM:

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
        mergeSort(lefthalf)
        mergeSort(righthalf)
        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1
        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)
n = input("Enter the size of the list: ")
n=int(n);
alist = []
for i in range(n):
    alist.append(input("Enter %dth element: "%i))
mergeSort(alist)
print(alist)
```

### Input:

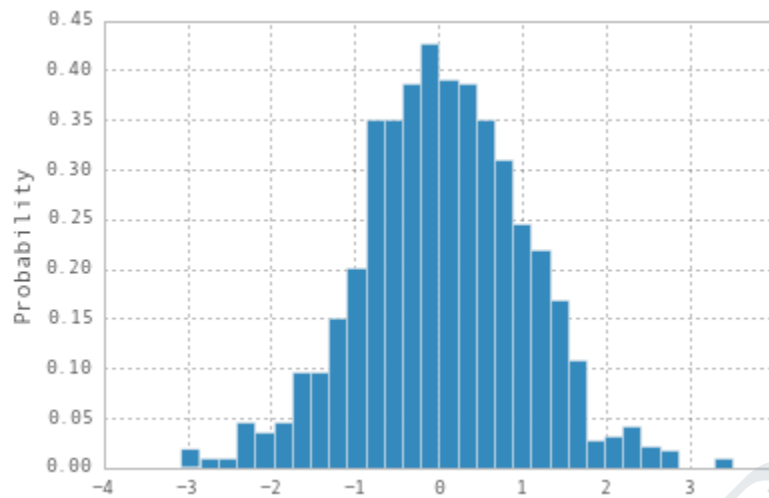
a = [16, 19, 11, 15, 10, 12, 14]

### Output:

```
>>>
[10, 11, 12, 14, 15, 16, 19]
```

### 4.5.5 HISTOGRAM

- A histogram is a visual representation of the Distribution of a Quantitative variable.
  - Appearance is similar to a vertical bar graph, but used mainly for continuous distribution
  - It approximates the distribution of variable being studied
- A visual representation that gives a discretized display of value counts.

**Example:**

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

The name of the function is histogram, which is a statistical term for a collection of counters (or frequencies).

The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character *c* is not in the dictionary, we create a new item with key *c* and the initial value 1 (since we have seen this letter once). If *c* is already in the dictionary we increment *d[c]*. Here's how it works:

**Example:**

```
>>> h = histogram('brontosaurus')
>>> h
```

**Output:**

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on. Dictionaries have a method called *get* that takes a key and a default value. If the key appears in the dictionary, *get* returns the corresponding value; otherwise it returns the default value. **For example:**

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

As an exercise, use *get* to write histogram more concisely. We should be able to eliminate the if statement.

If we use a dictionary in a for statement, it traverses the keys of the dictionary. For example, `print_hist` prints each key and the corresponding value:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Here's what the output looks like:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Again, the keys are in no particular order. To traverse the keys in sorted order, we can use the built-in function `sorted`:

```
>>> for key in sorted(h):
    ... print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

Write a function named `choose_from_hist` that takes a histogram as defined in Histogram given above and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

The function should return 'a' with probability 2/3 and 'b' with probability 1/3.

## **TWO MARKS**

1. What are elements in a list? Give example.

The values in a list are called elements or items.

A list must be enclosed in square brackets ([and]).

### **Examples:**

```
>>> [10,20,30]
>>> ['hi','hello','welcome']
```

2. What is a nested list? Give example.

A list within another list is called nested list.

### **Example:**

```
['good',10,[100,99]]
```

3. What is a empty list?

A list that contains no elements is called empty list. It can be created with empty brackets, [].

4. What is list slicing? Write its syntax.

List slicing is a computationally fast way to methodically access parts of given data. **Syntax:**

```
Listname [start:end:step]
```

5. What is mutability? What is its use?

Mutability is the ability for certain types of data to be changed without entirely recreating it. Using mutable data types can allow programs to operate quickly and efficiently.

6. List any three mutable data types. Give example for mutability.

**Example for mutable data types are:**

List, Set and Dictionary

**Example 1:**

```
>>>numbers=[42,123]
```

```
>>>numbers[1]=5
```

```
>>>numbers
```

**Output:** [42,5]

7. What is aliasing? Give example.

An object with more than one reference has more than one name, so we say that the object is **aliased**.

If the aliased object is mutable, changes made with one alias affect the other.

**Example:**

If a refers to an object and we assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
```

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

```
>>> b is a    True
```

8. What is the difference between copying and cloning lists?

Copying a list gives it another name but copying a complete slice clones a list.

9. Give example for copying and cloning lists.

**Example:**

```
veggies=["potatoes","carrots","pepper","parsnips","swedes","onion","min  
thead"]
```

```
veggies[1]="beetroot"
```

Copying a list

```
daniel=veggies
```

CLONES a list

```
david=veggies[:]
```

```
daniel[6]="emu"
```

10. Define Dictionaries. Give example.

A **dictionary** is an associative array (also known as hashes). Any key of the **dictionary** is associated (or mapped) to a value. The values of a **dictionary** can be any **Python** data type.

**Example:**

```
>>> d = {'a':0, 'b':1, 'c':2}
```

```
>>> t = d.items()
```

```
>>> t
```

**Output:**

```
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

### 11. What is list processing?

List processing is a list of programming codes, including abstract data structure, used to calculate specified variables in a certain order. A value may repeat more than once.



## UNIT V FILES, MODULES, PACKAGES

9

Files and exception: text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages; Illustrative programs: word count copy file.

### PERSISTENCE

Most of the programs we have seen are **transient** in the sense that they **run for a short time and produce some output**, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent: they run for a long time** (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

**Examples of persistent programs are operating systems**, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them. An alternative is to store the state of the program in a database.

### 5.1 FILES:TEXTFILE

A textfile is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM.

A **text file** is a **file containing characters, structured as individual lines of text**. In addition to printable characters, text files also contain the *nonprinting* newline character, `\n`, to denote the end of each text line. the newline character causes the screen cursor to move to the beginning of the next screen line. Thus, text files can be directly viewed and created using a text editor.

In contrast, **binary files** can **contain various types of data, such as numerical values, and are therefore not structured as lines of text**. Such files can only be read and written via a computer program.

### **Using Text Files**

Fundamental operations of all types of files include *opening* a file, *reading* from a file, *writing* to a file, and *closing* a file. Next we discuss each of these operations when using text files in Python.

### OPENING TEXT FILES

All files must first be opened before they can be read from or written to. In Python, when a file is (successfully) opened, a file object is created that provides methods for accessing the file.

All files must first be opened before they can be used. In Python, when a file is opened, a file object is created that provides methods for accessing the file.

### 5.1.2 OPENING FOR READING

The syntax to open a file object in Python is

*file\_object* = *open*("filename", "mode") where *file\_object* is the variable to add the file object.

To open a file for reading, the built-in open function is used as shown,

**input\_file=open('myfile.txt','r')**

The modes are:

- **'r'** – Read mode which is used when the file is only being read
- **'w'** – Write mode which is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated)
- **'a'** – Appending mode, which is used to add new data to the end of the file; that is new information is automatically amended to the end
- **'r+'** – Special read and write mode, which is used to handle both actions when working with a file

The **first argument is the file name to be opened, 'myfile.txt'**. The second argument, **'r'**, **indicates that the file is to be opened for reading.** (The second argument is optional when opening a file for reading.) If the file is successfully opened, a file object is created and assigned to the provided identifier, in this case identifier **input\_file**.

When opening a file for reading, there are a few reasons why an **I/O error may occur**. First, if the **file name does not exist, then the program will terminate with a “no such file or directory” error.**

```
... open('testfile.txt','r')
Traceback (most recent call last):
  File ", pyshell#1 . ", line 1, in , module .
    open('testfile.txt','r')
IOError: [Errno 2] No such file or directory:
```

This error can also occur if the file name is not found in the location looked for (uppercase and lowercase letters are treated the same for file names). When a file is opened, it is first searched for in the same folder/directory that the program resides in.

However, an alternate location can be specified in the call to open by providing a path to the file.

```
input_file=open('data/myfile.txt','r')
```

the file is searched for in a subdirectory called data of the directory in which the program is contained. Thus, its location is relative to the program location. (Although some operating systems use forward slashes, and other backward slashes in path names, directory paths in Python are always written with forward slashes and are automatically converted to backward slashes when required by the operating system executing on.) Absolute paths can also be provided giving the location of a file anywhere in the file system.

```
input_file= open('C:/mypythonfiles/data/myfile.txt','r')
```

When the program has finished reading the file, it should be closed by calling the close method on the file object,

```
input_file=open('C:/mypythonfiles/data/myfile.txt','r')
```

The read functions contains different methods,  
read(), readline() and readlines()

read() #return one big string

readline #return one line at a time

readlines #returns a list of lines

### 5.1.3 OPENING A FILE

To write a file

used to write s

the write method is

```
output_file=open('mynewfile.txt','w')
```

```
output_file.close()
```

```
f = open("test.txt","w") #opens file with name of "test.txt"
```

```
f.close()
```

```
f = open("test.txt","w") #opens file with name of "test.txt"
f.write("I am a test file.")
f.write("Welcome to python.")
f.write("Created by Guido van Rossum and first released in 1991 ")
f.write("Design philosophy that emphasizes code readability.")
f.close()
```

This method writes a sequence of strings to the file.

`write ()`    #Used to write a fixed sequence of characters to a file

`writelines()`            #writelines can write a list of strings.

### Appending to a file example

```
f = open("test.txt","w") #opens file with name of "test.txt"
f.close()
```

#### To open a text file,read mode:

```
fh = open("hello.txt", "r")
```

To read a text file:

```
fh = open("hello.txt","r")
```

```
print fh.read()
```

#### To read one line at a time:

```
fh = open("hello.txt", "r")
```

```
print fh.readline()
```

#### To read a list of lines:

```
fh = open("hello.txt", "r")
```

```
print fh.readlines()
```

#### To write to a file:

```
fh = open("hello.txt","w")
```

```
write("Hello World")
```

To writelines to a file:

#### **5.1.4 FORMAT OPERATOR**

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the format operator, %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the format string, which contains one or more format sequences, which specify how the second operand is formatted. The result is a string.

For example, the format sequence '%d' means that the second operand should be formatted as a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a

sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

## 5.2 COMMAND LINE ARGUMENTS

Command-line arguments in Python show up in `sys.argv` as a list of strings (so you'll need to import the `sys` module).

For example, if you want to print all passed command-line arguments:

```
import sys
print(sys.argv) # Note the first argument is always the script filename.
```

`sys.argv` is a list in Python, which contains the command-line arguments passed to the script.

With the `len(sys.argv)` function you can count the number of arguments.

If you are gonna work with command line arguments, you probably want to use `sys.argv`.

To use `sys.argv`, you will first have to import the `sys` module.

### Example

```
import sys
print "This is the name of the script: ", sys.argv[0]
print "Number of arguments: ", len(sys.argv)
print "The arguments are: ", str(sys.argv)
```

#### Output

f the script: sysargv.py

The arguments are: ['sysargv.py']

### **5.3 ERRORS AND EXCEPTIONS, HANDLING EXCEPTIONS**

Various error messages can occur when executing Python programs. Such errors are called exceptions. So far we have let Python handle these errors by reporting them on the screen. Exceptions can be “caught” and “handled” by a program, however, to either correct the error and continue execution, or terminate the program gracefully.

#### **5.3.1 WHAT IS AN EXCEPTION?**

An exception is a value (object) that is raised (“thrown”) signaling that an unexpected, or “exceptional,” situation has occurred. Python contains a predefined set of exceptions referred to as standard exceptions .

ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
IndexError	Raised when an index is not found in a sequence.

IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.

### **5.3.2 PYTHON EXCEPTION HANDLING - TRY, EXCEPT AND FINALLY**

#### **Exception handling can be done by using try statement**

```
try:
statements
except ExceptionType:
statements
except ExceptionType:
statements
```

Exception handling provides a means for functions and methods to report errors that cannot be corrected locally. In such cases, an exception (object) is raised that can be caught by its client code (the code that called it), or the client's client code, etc., until handled (i.e. the exception is caught and the error appropriately dealt with). If an exception is thrown back to the top-level code and never caught, then the program terminates displaying the exception type that occurred.



#### **5.3.3 The try statement works as follows.**

- First, the try clause (the statement(s) between the try and except keywords) 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**

```
import math
num=int(input('Enter the number'))
print('the factorial is',math.factorial(num))
```

**Output**

Enter the number-5

ValueError: factorial() not defined for negative values

**Example**

```
import math
num=int(input('Enter the number'))
valid_input=False;
while not valid_input:
    try:
        result=math.factorial(num);
        print('the factorial is',result)
        valid_input=True
    except ValueError:
        print('Cannot recompute reenter again')
        num=int(input('Please reenter'))
```

**Output**

Enter the number-5 Cannot recompute reenter again

Please reenter5

the factorial is 120

**5.4 Modules in Python**

A Python module is a file containing Python definitions and statements. The module that is directly executed to start a Python program is called the main module. Python provides standard (built-in) modules in the Python Standard Library.

Each module in Python has its own namespace: a named context for its set of identifiers. The fully

qualified name of each identifier in a module is of the form `modulename.identifier`.

### 5.4.1 MATHEMATICAL FUNCTIONS IN PYTHON

Python has a `math` module that provides most of the familiar mathematical functions. A module is a file that contains a collection of related functions.

Before we can use the module, we have to import it:

```
>>> import math
```

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

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

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

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example uses `log10` to compute a signal-to-noise ratio in decibels (assuming that `signal_power` and `noise_power` are defined). The `math` module also provides `log`, which computes logarithms base *e*.

The second example finds the sine of radians. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by  $2\pi$ :

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the `math` module. The value of this

variable is an approximation of  $\pi$ , accurate to about 15 digits.

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0    Output:0.707106781187
```

One of the most useful features of programming languages is their ability to take small building blocks and compose them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name.

```
>>> minutes = hours * 60 # right
>>> hours * 60 = minutes # wrong!
SyntaxError: can't assign to operator
```

In python a number of mathematical operations can be performed with ease by importing a module named “math” which defines various functions which makes our tasks easier.

**1. ceil()** :- This function returns the smallest integral value greater than the number. If number is already integer, same number is returned.

**2. floor()** :- This function returns the greatest integral value smaller than the number. If number is already integer, same number is returned.

```
# Python code to demonstrate the working of ceil() and floor()
# importing "math" for mathematical operations
import math
a = 2.3
# returning the ceil of 2.3
print ("The ceil of 2.3 is : ", end="")
print (math.ceil(a))
# returning the floor of 2.3
print ("The floor of 2.3 is : ", end="")
print (math.floor(a))
```

3. **fabs()** :- This function returns the **absolute value** of the number.

4. **factorial()** :- This function returns the **factorial** of the number. An error message is displayed if number is not integral.

```
# Python code to demonstrate the working of
# fabs() and factorial()
# importing "math" for mathematical operations
import math
a = -10
b = 5
# returning the absolute value.
print ("The absolute value of -10 is : ", end="")
print (math.fabs(a))
# returning the factorial of 5
print ("The factorial of 5 is : ", end="")
print (math.factorial(b))
Output:
The absolute value of -10 is : 10.0
The factorial of 5 is : 120
```

5. **copysign(a, b)** :- This function returns the number with the **value of 'a'** but with the **sign of 'b'**. The returned value is float type.

6. **gcd()** :- This function is used to compute the **greatest common divisor of 2 numbers** mentioned in its arguments.

```
# Python code to demonstrate the working of copysign() and gcd()
```

```
import math
```

```
a = -10
```

```
b = 5.5
```

```
c = 15
```

```
d = 5
```

```
# returning the copysigned value.
```

```
print ("The copysigned value of -10 and 5.5 is : ", end="")
```

```
print (math.copysign(5.5, -10))
```

```
# returning the gcd of 15 and 5
```

```
print ("The gcd of 5 and 15 is : ", end="")
```

```
print (math.gcd(5,15))
```

Output:

The copysigned value of -10 and 5.5 is : -5.5

**MATH MODULE**

This module contains a set of commonly-used mathematical functions, including number-theoretic functions (such as factorial); logarithmic and power functions; trigonometric (and hyperbolic) functions; angular conversion functions (degree/radians); and some special functions and constants (including pi and e). A selected set of function from the math module are presented here.

`math.ceil` returns the ceiling of x (smallest integer greater than or equal to x).

`math.fabs(x)` returns the absolute value of x

`math.factorial(x)` returns the factorial of x

`math.floor()` returns the floor of x (largest integer less than x).

`math.fsum(s)` returns an accurate floating-point sum of values in s (or other iterable).

`math.modf()` returns the fractional and integer parts of x.

`math.trunc(X)` returns the truncated value of s.

`math.exp(x)` returns  $e^x$ , for natural log base e.

`math.log(x,base)` returns log x for base. If base omitted, returns log x base e.

`math.sqrt(x)` returns the square root of x.

`math.cos(x)` returns cosine of x radians.

`math.sin(x)` returns sine of x radians.

`math.tan(x)` returns tangent of x radians.

`math.acos(x)` returns arc cosine of x radians.

`math.asin(x)` returns arc sine of x radians.

`math.atan(x)` returns arc cosine of x radians.

`math.degrees(x)` returns x radians to degrees.

`math.radians(x)` returns x degrees to radians.

`math.pi` mathematical constant pi = 3.141592

`math.e` mathematical constant e = 2.718281

**IMPORTING MODULES**

A Python module is a file containing Python definitions and statements. When a Python file is directly executed, it is considered the main module of a program. Main modules are given the special name `__main__`. Main modules provide the basis for a complete Python program. They may import (include) any number of other modules (and each of those modules import other modules, etc.).

Main modules are not meant to be imported into other modules.

As with the main module, imported modules may contain a set of statements. The statements of imported modules are executed only once, the first time that the module is imported. The purpose of these statements is to perform any initialization needed for the members of the imported module. The Python Standard Library contains a set of predefined Standard (built-in) modules.

#### 1. **import modulename**

Makes the namespace of modulename available, but not part of, the importing module. All imported identifiers used in the importing module must be fully qualified:

```
import math  
print('factorial of 16 = ', math.factorial(16))
```

#### 2. **from modulename import identifier\_1, identifier\_2, ...**

identifier\_1, identifier\_2, etc. become part of the importing module's namespace:

```
from math import factorial  
print('factorial of 16 = ', factorial(16))
```

#### 3. **from modulename import Identifier\_1 as identifier\_2**

identifier\_1 becomes part of the importing module's namespace as identifier\_2

```
from math import factorial as fact  
print('factorial of 16 = ', fact(16))
```

#### 4. **from modulename import \***

All identifiers of modulename become part of the importing module's namespace (except those beginning with an underscore, which are treated as private).

#### 5. **from math import \***

```
print('factorial of 16 = ', fact(16))  
print('area of circle = ', pi*(radius**2))
```

### **5.5 PAC**

Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a change.

Each package in Python is a directory which MUST contain a special file called `__init__.py`. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported

### **5.5.1 STEPS TO CREATE A PYTHON PACKAGE**

1. Create a directory and give it your package's name.
2. Put your classes in it.
3. Create a `__init__.py` file in the directory

The `__init__.py` file is necessary because with this file, Python will know that this directory is a Python package directory other than an ordinary directory (or folder – whatever you want to call it).

### **5.5.2 EXAMPLE ON HOW TO CREATE A PYTHON PACKAGE**

In this tutorial, we will create an Animals package – which simply contains two module files named Mammals and Birds, containing the Mammals and Birds classes, respectively.

1. Step 1: Create the Package Directory
2. So, first we create a directory named Animals.
3. Step 2: Add Classes

Now, we create the two classes for our package. First, create a file named `Mammals.py` inside the Animals directory and put the following code in it:

```
class Mammals:
    def __init__(self):
        """ Constructor for this class. """
        # Create some member animals
        self.members = ['Tiger', 'Elephant', 'Wild Cat']
```

```
print("Printing members of the Mammals class")
```



The class has a property named members – which is a list of some mammals we might be interested in. It also has a method named printMembers which simply prints the list of mammals of this class!.When you create a Python package, all classes must be capable of being imported, and won't be executed directly.

Next we create another class named Birds. Create a file named Birds.py inside the Animals directory and put the following code in it:

```
class Birds:
    def __init__(self):
        """ Constructor for this class. """
        # Create some member animals
        self.members = ['Sparrow', 'Robin', 'Duck']
    def printMembers(self):
        print('Printing members of the Birds class')
        for member in self.members:
            print('\t%s ' % member)
```

This code is similar to the code we presented for the Mammals class.

Step 3: Add the \_\_init\_\_.py File

Finally, we create a file named `__init__.py` inside the Animals directory and put the following code in it:

```
from Mammals import Mammals
```

```
from Birds import Birds
```

That's it! That's all there is to it when you create a Python package. For testing, we create a simple file named `test.py` in the same directory where the Animals directory is located.

We place the following code in the `test.py` file:

```
# Import classes from your brand new package
```

```
from Animals import Mammals
```

```
from Animals import Birds
```

```
# Create an object of Mammals class & call a method of it
```

```
myMammal = Mammals()
```

```
myMammal.printMembers()
```

```
# Create an object of Birds class & call a method of it
```

```
myBird = Birds()
```

```
myBird.printMembers()
```

### **ILLUSTRATIVE PROGRAMS: WORD COUNT, COPY FILE**

#### **WORD COUNT IN A TEXT FILE**

```
fname = input("Enter file name: ")
```

```
num_words = 0
```

```
with open(fname, 'r') as f:
```

```
for line in f:
    words = line.split()
    num_words += len(words)
print("Number of words:")
print(num_words)
```

Case 1:

Contents of file:

Hello world

### OUTPUT:

Enter file name: data1.txt

Number of words:

2

Case 2:

Contents of file:

This programming language is

Python

Output:

Enter file name: data2.txt

Number of words:

5

### COPY FILE IN PYTHON

```
with open("test.txt") as f:
    with open("out.txt", "w") as f1:
        for line in f:
            f1.write(line)
```

### OUTPUT:

Case 1:

Contents of file(test.txt):

Hello world

Output(out.txt):

Hello world