

GE8151 - PROBLEM SOLVING AND PYTHON
PROGRAMMING

REGULATIONS – 2017

UNIT – I

UNIT I

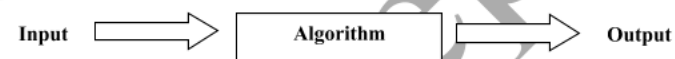
ALGORITHMIC PROBLEM SOLVING

1.1 ALGORITHMS

In computing, we focus on the type of problems categorically known as algorithmic problems, where their solutions are expressible in the form of algorithms.

The term 'algorithm' was derived from the name of Mohammed al-Khowarizmi, a Persian mathematician in the ninth century. Al-Khowarizmi → Algorismus (in Latin) → Algorithm.

An algorithm is a well-defined computational procedure consisting of a set of instructions that takes some value or set of values, as input, and produces some value or set of values, as output. In other word, an algorithm is a procedure that accepts data; manipulate them following the prescribed steps, so as to eventually fill the required unknown with the desired value(s).



People of different professions have their own form of procedure in their line of work, and they call it different names. For instance, a cook follows a procedure commonly known as a recipe that converts the ingredients (input) into some culinary dish (output), after a certain number of steps.

1.1.1 The Characteristics of a Good Algorithm

- **Precision** – the steps are precisely stated (defined).
- **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- **Finiteness** – the algorithm stops after a finite number of instructions are executed.
- **Effectiveness** – algorithm should be most effective among many different ways to solve a problem.
- **Input** – the algorithm receives input.
- **Output** – the algorithm produces output.
- **Generality** – the algorithm applies to a set of inputs.

1.2 BUILDING BLOCKS OF ALGORITHM (INSTRUCTIONS, STATE, CONTROL FLOW, FUNCTIONS)

An algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a *function*. Starting from an initial *state* and initial input, the *instructions* describe a computation that, when executed,

proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; that algorithms, known as **randomized algorithms**.

An **instruction** is a single operation, that describes a computation. When it executed it convert one state to other.

Typically, when an algorithm is associated with processing information, data can be read from an input source, written to an output device and stored for further processing. Stored data are considered as part of the internal *state* of the entity performing the algorithm. In practice, the state is stored in one or more data structures.

The **state** of an algorithm is defined as its condition regarding stored data. The stored data in an algorithm are stored as variables or constants. The state shows its current values or contents.

Because an algorithm is a precise list of precise steps, the order of computation is always crucial to the functioning of the algorithm. *Instructions* are usually assumed to be listed explicitly, and are described as starting "from the top" and going "down to the bottom", an idea that is described more formally by *flow of control*.

In computer science, **control flow** (or **flow of control**) is the order in which individual statements, instructions or function calls of an algorithm are executed or evaluated.

For complex problems our goal is to divide the task into smaller and simpler *functions* during algorithm design. So a set of related sequence of steps, part of larger algorithm is known as **functions**.

1.2.1 Control Flow

Normally Algorithm has three control flows they are:

- Sequence
- Selection
- Iteration

Sequence: A sequence is a series of steps that occur one after the other, in the same order every time. Consider a car starting off from a set of lights. Imagine the road in front of the car is clear for many miles and the driver has no need to slow down or stop. The car will start in first gear, then move to second, third, fourth and finally fifth. A good driver (who doesn't have to slow down or stop) will move through the gears in this sequence, without skipping gears.

Selection: A selection is a decision that has to be made. Consider a car approaching a set of lights. If the lights turn from green to yellow, the driver will have to decide whether to stop or continue through the intersection. If the driver stops, she will have to wait until the lights are green again.

Iteration: Iteration is sometimes called repetition, which means a set of steps which are repeated over and over until some event occurs. Consider the driver at the red light, waiting

for it to turn green. She will check the lights and wait, check and wait. This sequence will be repeated until the lights turn green.

1.3 NOTATION OF ALGORITHM

There are four different Notation of representing algorithms:

- Step-Form
- Pseudocode
- Flowchart
- Programming Language

1.3.1 Algorithm as Step-Form

It is a **step – by – step procedure** for solving a task or problem. The steps must be ordered, unambiguous and finite in number. It is English like representation of logic which is used to solve the problem.

Some guidelines for writing Step-Form algorithm are as follows:

- Keep in mind that algorithm is a step-by-step process.
- Use begin or start to start the process.
- Include variables and their usage.
- If there are any loops, try to give sub number lists.
- Try to give go back to step number if loop or condition fails.
- Use jump statement to jump from one statement to another.
- Try to avoid unwanted raw data in algorithm.
- Define expressions.
- Use break and stop to terminate the process.

For accomplishing a particular task, different algorithms can be written. The different algorithms differ in their requirements of time and space. The programmer selects the best suited algorithm for the given task to be solved.

Let as look at two simple algorithms to find the greatest among three numbers, as follows:

Algorithm 1.1:

Step 1: Start.
Step 2: Read the three numbers A,B,C.
Step 3: Compare A and B. If A is greater perform step 4 else perform step 5.
Step 4: Compare A and C. If A is greater, output "A is greater" else output "C is greater".
Then go to step 6
Step 5: Compare B and C. If B is greater, output "B is greatest" else output "C is greatest".
Step 6: Stop.

Algorithm 1.2:

Step 1: Start.
Step 2: Read the three numbers A,B,C.
Step 3: Compare A and B. If A is greater, store A in MAX, else store B in MAX.
Step 4: Compare MAX and C. If MAX is greater, output "MAX is greater" else output "C is greater".
Step 5: Stop.

Both the algorithms accomplish same goal, but in different ways. The programmer selects the algorithm based on the advantages and disadvantages of each algorithm. For example, the first algorithm has more number of comparisons, whereas in second algorithm an additional variable MAX is required.

In the **Algorithm 1.1**, **Algorithm 1.2**, step 1 and 2 are in **sequence logic** and step 3, 4, and 5 are in **selection logic**. In order to illustrate iteration logic, consider one more example.

Design an algorithm for calculating total mark of specified number of subjects given as: 86, 99, 98, 87, 89

Algorithm 1.3:

Step 1: Start
Step 2: Read Number of subjects as N
Step 3: Total = 0, i=1
Step 4: Get single subject mark as mark
Step 5: Total = Total + mark
Step 6: i=i+1
Step 7: If i<=N, THEN go to step 4. Otherwise, go to step 8
Step 8: Output the Total
Step 9: Stop

In the **Algorithm 1.3** step 7 have a go to statement with a back ward step reference, so it means that **iteration logic**.

1.3.2 Pseudocode

Pseudocode ("sort of code") is another way of describing algorithms. It is called "pseudo" code because of its strong resemblance to "real" program code. **Pseudocode is essentially English with some defined rules of structure and some keywords that make it appear a bit like program code.**

Some guidelines for writing pseudocode are as follows.

Note: These are not "strict" guidelines. Any words used that have similar form and function may be used in an emergency - however, it would be best to stick to the pseudocode words used here.







- for start and finish **BEGIN MAINPROGRAM, END MAINPROGRAM** - this is often abbreviated to **BEGIN** and **END** - especially in smaller programs
- for initialization **INITIALISATION, END INITIALISATION** - this part is optional, though it makes your program structure clearer
- for subprogram **BEGIN SUBPROGRAM, END SUBPROGRAM**
- for selection **IF, THEN, ELSE, ENDIF**
- for multi-way selection **CASEWHERE, OTHERWISE, ENDCASE**
- for pre-test repetition **WHILE, ENDWHILE**
- for post-test repetition **REPEAT, UNTIL**
- Keywords are written in capitals.
- Structural elements come in pairs, eg for every BEGIN there is an END, for every IF there is an ENDIF, etc.
- Indenting is used to show structure in the algorithm.
- The names of **subprograms are underlined**. This means that when refining the solution to a problem, a word in an algorithm can be underlined and a subprogram developed. This feature is to assist the use of the 'top-down' development concept.





1.3.3 Flowcharts

Flowcharts are a diagrammatic method of representing algorithms. They use an intuitive scheme of showing operations in boxes connected by lines and arrows that graphically show the flow of control in an algorithm.

Table 1.1 lists the flowchart symbol drawing, the name of the flowchart symbol in Microsoft Office (with aliases in parentheses), and a short description of where and how the flowchart symbol is used.

Table 1.1: Flowchart Symbols

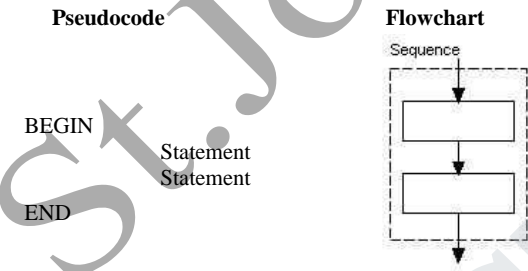
SYMBOL	NAME (ALIAS)	DESCRIPTION
	Flow Line (Arrow, Connector)	Flow line connectors show the direction that the process flows.
	Terminator (Terminal Point, Oval)	Terminators show the start and stop points in a process.
	Data (I/O)	The Data flowchart shape indicates inputs to and outputs from a process. As such, the shape is more often referred to as an I/O shape than a Data shape.
	Document	Pretty self-explanatory - the Document flowchart symbol is for a process step that produces a document.
	Process	Show a Process or action step. This is the most common symbol in flowcharts.
	Decision	Indicates a question or branch in the process flow. Typically, a Decision flowchart shape is used when there are 2 options (Yes/No, No/No-Go, etc.)

	Connector (Inspection)	This symbol is typically small and is used as a Connector to show a jump from one point in the process flow to another. Connectors are usually labeled with capital letters (A, B, AA) to show matching jump points. They are handy for avoiding flow lines that cross other shapes and flow lines. They are also handy for jumping to and from a sub-processes defined in a separate area than the main flowchart.
	Predefined Process (Subroutine)	A Predefined Process symbol is a marker for another process step or series of process flow steps that are formally defined elsewhere. This shape commonly depicts sub-processes (or subroutines in programming flowcharts).
	Preparation	As the names states, any process step that is a Preparation process flow step, such as a set-up operation. Eg, Used in For Loop.
	Magnetic Disk (Database)	The most universally recognizable symbol for a data storage location, this flowchart shape depicts a database.

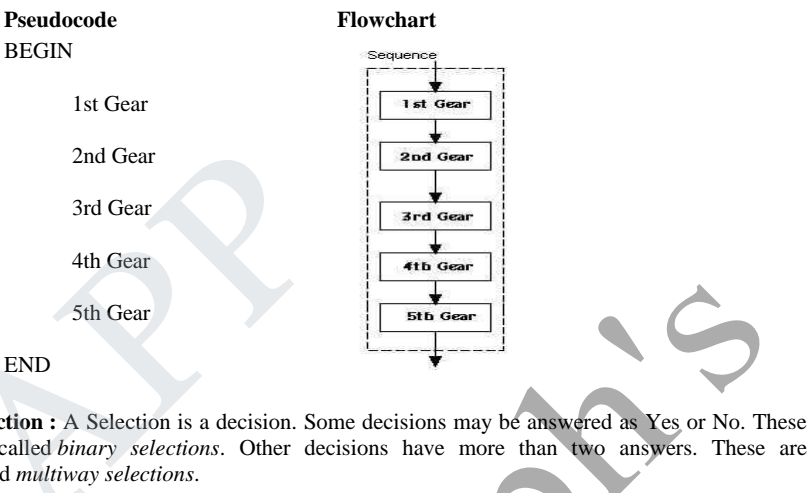
1.3.4 Control Structures of Pseudocode and Flowcharts

Each control structures can be built from the basic elements as shown below.

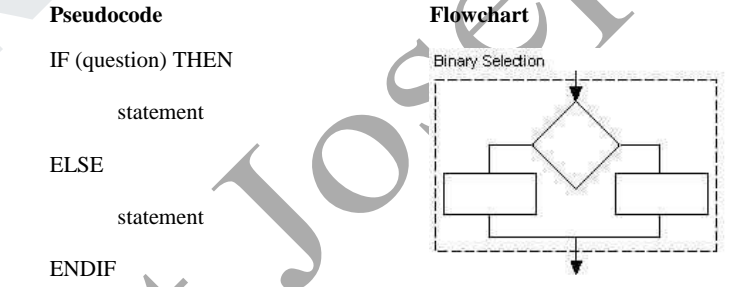
Sequence: A sequence is a series of steps that take place one after another. Each step is represented here by a new line.



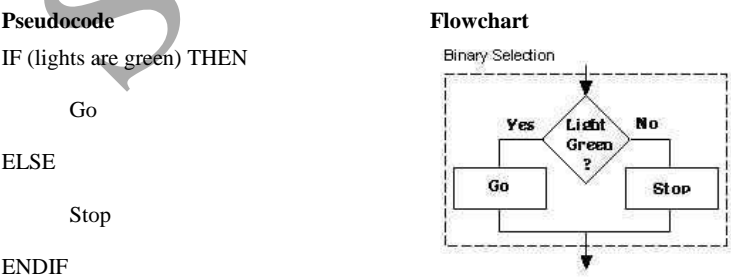
Our sequence example of changing gears could be described as follows :



A **binary selection** may be described in pseudocode and flow chart as follows :



An example of someone at a set of traffic lights follows :



A **Multiway Selection** may be described as follows :

Pseudocode

CASEWHERE (question)

Alternative 1: Statement

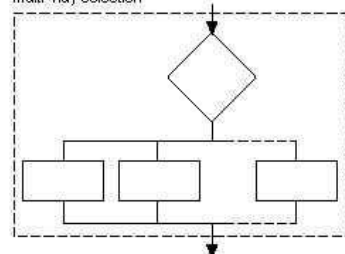
Alternative 2 : Statement

OTHERWISE : Statement

ENDCASE

Flowchart

Multi-way selection



An example of someone at a set of traffic lights follows :

Pseudocode

CASEWHERE Lights are :

Green : Go

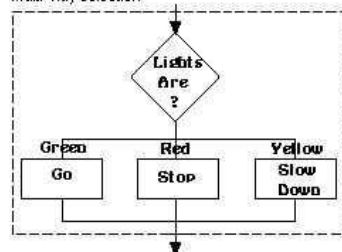
Amber : Slowdown

Red : Stop

ENDCASE

Flowchart

Multi-way selection



Repetition: A sequence of steps which are repeated a number of times, is called repetition. For a repeating process to end, a *decision* must be made. The decision is usually called a *test*. The position of this test divides repetition structures into two types : Pre-test and Post-test repetitions.

Pre-test repetitions (sometimes called guarded loops) will perform the test before any part of the loop occurs.

Post-test repetitions (sometimes called un-guarded loops) will perform the test after the main part of the loop (the body) has been performed at least once.

Pre-test repetitions may be described in as follows :

Pseudocode

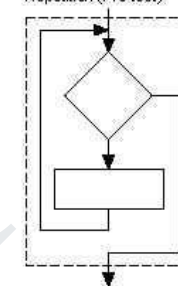
WHILE (question)

Statement

ENDWHILE

Flowchart

Repetition (Pre-test)



A traffic lights example follows :

Pseudocode

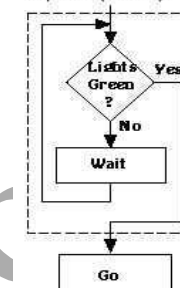
WHILE (lights are not green)

wait

ENDWHILE

Flowchart

Repetition (Pre-test)



Post-test repetitions may be described as follows:

Pseudocode

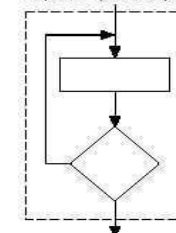
REPEAT

Statement

UNTIL (Question)

Flowchart

Repetition (Post-test)



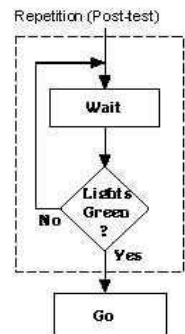
A traffic lights example follows :

Pseudocode

REPEAT

Wait

UNTIL lights are green

Flowchart

Sub-Programs: A sub-program is a self-contained part of a larger program. The use of sub-programs helps with "Top-Down" design of an algorithm, by providing a structure through which a large and unwieldy solution may be broken down into smaller, more manageable, components.

A sub-program is described as:

Pseudocodesubprogram**Flowchart**

Consider the total concept of driving a car. This activity is made up of a number of sub-processes which each contribute to the driver arriving at a destination. Therefore, driving may involve the processes of "Starting the car"; "Engaging the gears"; "Speeding up and slowing down"; "Steering"; and "Stopping the car".

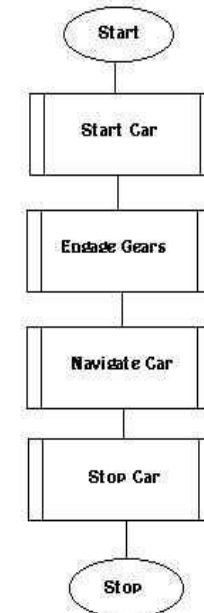
A (not very detailed) description of driving may look something like:

Pseudocode

BEGIN

Start CarEngage GearsNavigate CarStop Car

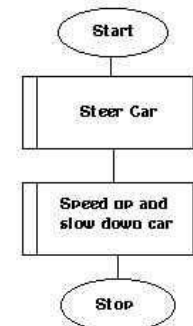
END

Flowchart

Note that the sub-program "Navigate Car" could be further developed into:

PseudocodeBEGIN SUBPROGRAM NavigateSteer CarSpeed up and slow down car

ENDSUBPROGRAM

Flowchart

In the example above, each of the sub-programs would have whole algorithms of their own, incorporating some, or all, of the structures discussed earlier.

1.3.5 Programming Language

Computers won't understand your algorithm as they use a different language. It will need to be translated into code, which the computer will then follow to complete a task. This code is written in a **programming language**. There are many programming languages available. Some of you may come across are C, C++, Java, Python, etc... Each of this language is suited to different things. In this book we will learn about Python programming language.

1.4 RECURSION

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem.

1.4.1 Example

For positive values of n , let's write $n!$, as we know $n!$ is a product of numbers starting from n and going down to 1. $n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1$. But notice that $(n-1) \cdot \dots \cdot 2 \cdot 1$ is another way of writing $(n-1)!$, and so we can say that $n! = n \cdot (n-1)!$. So we wrote $n!$ as a product in which one of the factors is $(n-1)!$. You can compute $n!$ by computing $(n-1)!$ and then multiplying the result of computing $(n-1)!$ by n . You can compute the factorial function on n by first computing the factorial function on $n-1$. So computing $(n-1)!$ is a subproblem that we solve to compute $n!$.

Let's look at an example: computing $5!$.

- You can compute $5!$ as $5 \cdot 4!$.
- Now you need to solve the subproblem of computing $4!$, which you can compute as $4 \cdot 3!$.
- Now you need to solve the subproblem of computing $3!$, which is $3 \cdot 2!$.
- Now $2!$, which is $2 \cdot 1!$.
- Now you need to compute $1!$. You could say that $1! = 1$, because it's the product of all the integers from 1 through 1. Or you can apply the formula that $1! = 1 \cdot 0!$. Let's do it by applying the formula.
- We defined $0!$ to equal 1.
- Now you can compute $1! = 1 \cdot 0! = 1$.
- Having computed $1! = 1$, you can compute $2! = 2 \cdot 1! = 2$.
- Having computed $2! = 2$, you can compute $3! = 3 \cdot 2! = 6$.
- Having computed $3! = 6$, you can compute $4! = 4 \cdot 3! = 24$.
- Finally, having computed $4! = 24$, you can finish up by computing $5! = 5 \cdot 4! = 120$.

So now we have another way of thinking about how to compute the value of $n!$, for all nonnegative integers n :

- If $n=0$, then declare that $n!=1$.
- Otherwise, n must be positive. Solve the subproblem of computing $(n-1)!$, multiply this result by n , and declare $n!$ equal to the result of this product.

When we're computing $n!$ in this way, we call the first case, where we immediately know the answer, the *base case*, and we call the second case, where we have to compute the same function but on a different value, the *recursive case*.

Base case is the case for which the solution can be stated non-recursively (ie. the answer is known). **Recursive case** is the case for which the solution is expressed in terms of a smaller version of itself.

1.4.2 Recursion in Step Form

Recursion may be described as follows:

Syntax for recursive algorithm

Step 1: Start

Step 2: Statement(s)

Step 3: Call Subprogram(argument)

Step 4: Statement(s)

Step 5: Stop

Step 1: BEGIN Subprogram(argument)

Step 2: If base case then return base solution

Step 3: Else return recursive solution, include call subprogram(smaller version of argument)

Recursive algorithm solution for factorial problem

Step 1: Start

Step 2: Read number n

Step 3: Call factorial(n) and store the result in f

Step 4: Print factorial f

Step 5: Stop

Step 1: BEGIN factorial(n)

Step 2: If $n==0$ then return 1

Step 3: Else Return $n \cdot \text{factorial}(n-1)$

Step 4: END factorial

1.4.3 Recursion in Flow Chart

Syntax for recursive flow chart

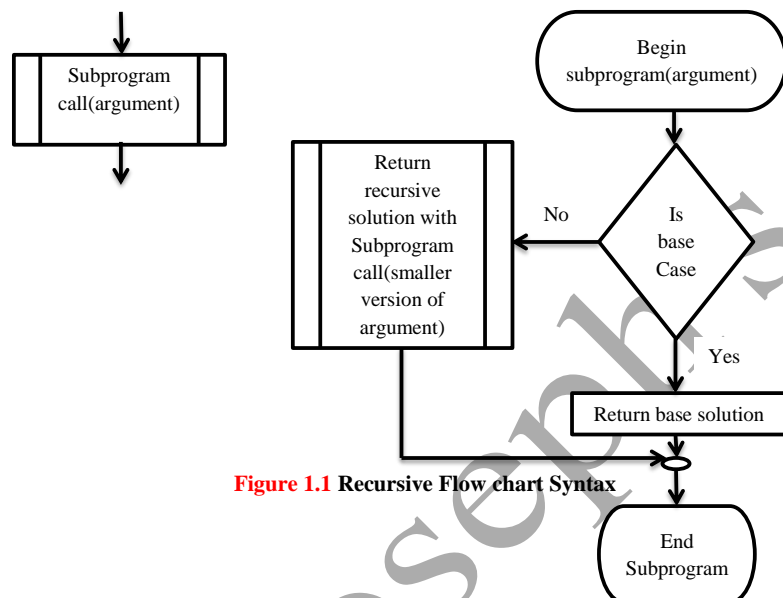


Figure 1.1 Recursive Flow chart Syntax

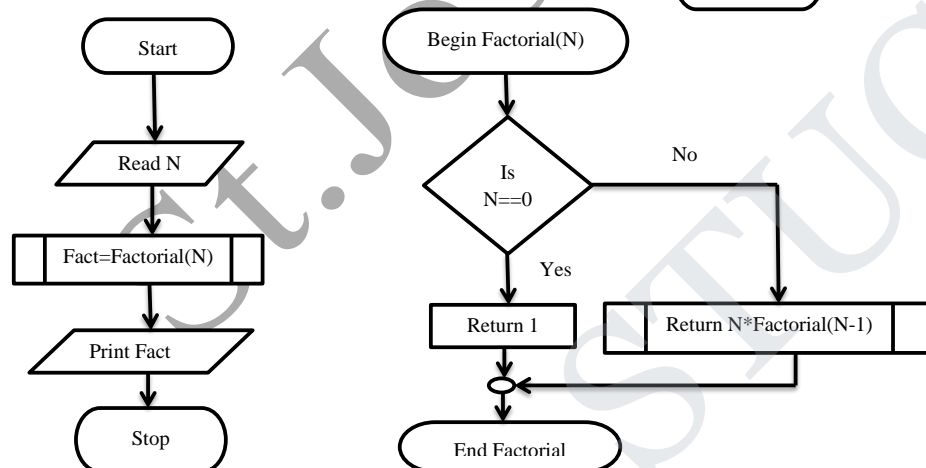


Figure 1.2 Recursive flow chart solution for factorial problem

1.4.4 Recursion in Pseudocode

Syntax for recursive Pseudocode

```

SubprogramName(argument)  BEGIN SUBPROGRAM SubprogramName(argument)
                             Is base Case
                             Return base case solution
                             Else
                             Return recursive solution with
                             SubprogramName (smaller version of
                             argument)
                             ENDSUBPROGRAM
  
```

Recursive Pseudocode solution for factorial problem

```

BEGIN Factorial              BEGIN SUBPROGRAM Fact(n)
    READ N                    IF n==0 THEN RETURN 1
    f= Fact(N)                ELSE RETURN n*Fact(n-1)
    PRINT f                    ENDSUBPROGRAM
END
  
```

1.5 ALGORITHMIC PROBLEM SOLVING

We can consider algorithms to be procedural solutions to problems. These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines. In particular, this distinguishes it from theoretical mathematics, whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution's properties. Let us discuss a sequence of steps one typically goes through in designing and analysing an algorithm

1. Understanding the problem
2. Ascertain the capabilities of the computational device
3. Exact /approximate solution
4. Decide on the appropriate data structure
5. Algorithm design techniques
6. Methods of specifying an algorithm
7. Proving an algorithms correctness
8. Analysing an algorithm

Understanding the problem:

The problem given should be understood completely. Check if it is similar to some standard problems & if a Known algorithm exists. Otherwise a new algorithm has to be developed.

Ascertain the capabilities of the computational device:

Once a problem is understood, we need to know the capabilities of the computing device. This can be done by knowing the type of the architecture, speed & memory availability.

Exact /approximate solution:

Once an algorithm is developed, it is necessary to show that it computes an answer for all the possible legal inputs. The solution is stated in two forms, exact solution or approximate solution. Examples of problems where an exact solution cannot be obtained are i) Finding a square root of a number. ii) Solutions of nonlinear equations.

Decide on the appropriate data structure:

Some algorithms do not demand any ingenuity in representing their inputs. Some others are in fact predicted on ingenious data structures. A data type is a well-defined collection of data with a well-defined set of operations on it. A data structure is an actual implementation of a particular abstract data type. The Elementary Data Structures are **Arrays**: These let you access lots of data fast. (good). You can have arrays of any other data type. (good). However, you cannot make arrays bigger if your program decides it needs more space. (bad). **Records**: These let you organize non-homogeneous data into logical packages to keep everything together. (good). These packages do not include operations, just data fields (bad, which is why we need objects). Records do not help you process distinct items in loops (bad, which is why arrays of records are used). **Sets**: These let you represent subsets of a set with such operations as intersection, union, and equivalence. (good). Built-in sets are limited to a certain small size. (bad, but we can build our own set data type out of arrays to solve this problem if necessary).

Algorithm design techniques:

Creating an algorithm is an art which may never be fully automated. By mastering these design strategies, it will become easier for you to develop new and useful algorithms. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other than computer science such as operation research and electrical engineering. Some important design techniques are linear, nonlinear and integer programming.

Methods of specifying an algorithm:

There are mainly two options for specifying an algorithm: use of natural language or pseudocode & Flowcharts. A Pseudo code is a mixture of natural language & programming language like constructs. A flowchart is a method of expressing an algorithm by a collection of connected geometric shapes.

Proving algorithms correctness:

Once an algorithm is developed, it is necessary to show that it computes an answer for all the possible legal inputs. We refer to this process as algorithm validation. The process of

validation is to assure us that this algorithm will work correctly independent of issues concerning programming language it will be written in.

Analysing algorithms:

As an algorithm is executed, it uses the computer's central processing unit to perform operation and its memory (both immediate and auxiliary) to hold the program and data. Analysis of algorithms and performance analysis refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area in which sometimes requires great mathematical skill. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another. Another result is that it allows you to predict whether the software will meet any efficiency constraint that exists.

1.6 ILLUSTRATIVE PROBLEMS

1.6.1 Find Minimum in a List

Consider the following requirement specification:

A user has a list of numbers and wishes to find the minimum value in the list. An Algorithm is required which will allow the user to enter the numbers, and which will calculate the minimum value that are input. The user is quite happy to enter a count of the numbers in the list before entering the numbers.

Finding the minimum value in a list of items isn't difficult. Take the first element and compare its value against the values of all other elements. Once we find a smaller element we continue the comparisons with its value. Finally we find the minimum.

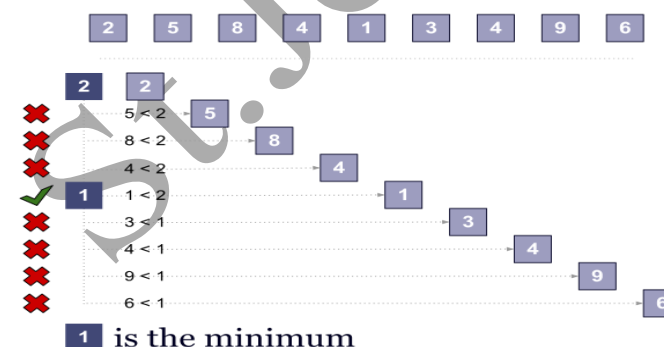


Figure 1.3 Steps to Find Minimum Element in a List

Step base algorithm to find minimum element in a list

Step 1: Start
 Step 2: READ total number of element in the List as N
 Step 3: READ first element as E
 Step 4: MIN =E
 Step 5: SET i=2
 Step 6: IF i>n go to Step 11 ELSE go to step 7
 Step 7: READ ith element as E
 Step 8: IF E < MIN THEN SET MIN = E
 Step 9: i=i+1
 Step 10: go to step 6
 Step 11: Print MIN
 Step 12: Stop

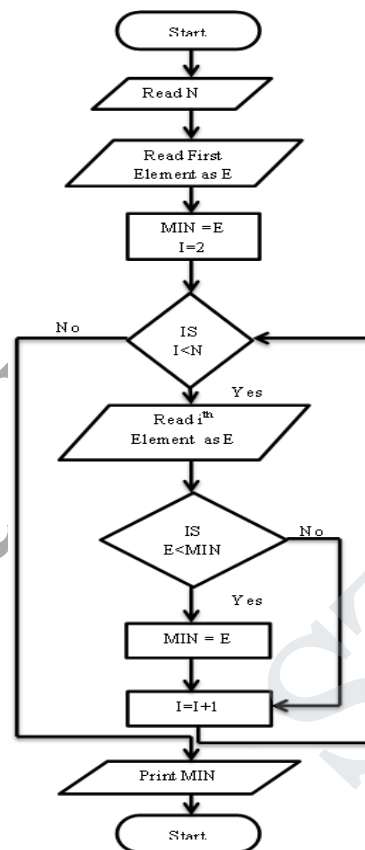


Figure 1.4 Flow Chart to Find Minimum Element in a List

Pseudocode to find minimum element in a list

```

READ total number of element in the List as N
READ first element as E
SET MIN =E
SET i=2
WHILE i<=n
    READ ith element as E
    IF E < MIN THEN
        SET MIN = E
    ENDIF
    INCREMENT i by ONE
ENDWHILE
PRINT MIN
  
```

1.6.2 Insert a Card in a List of Sorted Cards

Insert a Card in a List of Sorted Cards – is same as inserting an element into a sorted array.

We start from the high end of the array and check to see if that's where we want to insert the data. If so, fine. If not, we move the preceding element up one and then check to see if we want to insert x in the "hole" left behind. We repeat this step as necessary.

Thus the search for the place to insert x and the shifting of the higher elements of the array are accomplished together.

Position	0	1	2	3	4	5	Element To Be Insert
Original List	4	6	9	10	11		
7>11	×	4	6	9	10	11	7
7>10	×	4	6	9	10	11	
7>9	×	4	6	9	10	11	
7>6	√	4	6	7	9	10	11

Figure 1.5 Steps to Insert a Card in a List of Sorted Cards

Step base algorithm to Insert a Card in a List of Sorted Cards

Step 1: Start
 Step 2: Declare variables N, List[], i, and X.
 Step 3: READ Number of element in sorted list as N
 Step 4: SET i=0
 Step 5: IF i<N THEN go to step 6 ELSE go to step 9
 Step 6: READ Sorted list element as List[i]
 Step 7: i=i+1
 Step 8: go to step 5
 Step 9: READ Element to be insert as X
 Step 10: SET i = N-1
 Step 11: IF i>=0 AND X<List[i] THEN go to step 12 ELSE go to step 15
 Step 12: List[i+1]=List[i]
 Step 13: i=i-1

Step 14: go to step 11
Step 15: List[i+1]=X

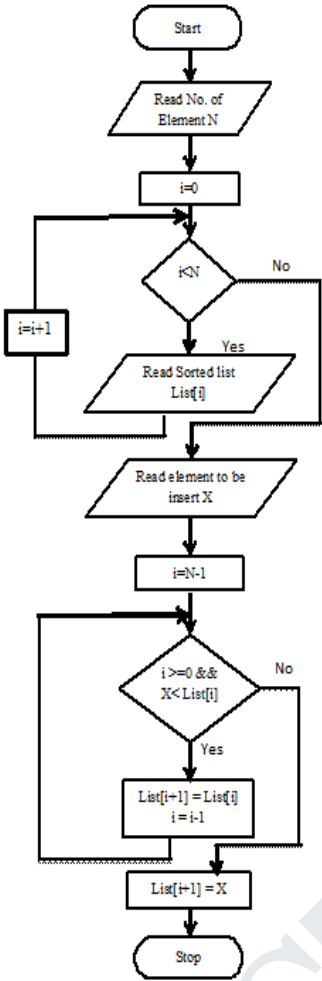


Figure 1.6 Flow Chart to Insert a Card in a List of Sorted Cards

Pseudocode to Insert a Card in a List of Sorted Cards

```
READ Number of element in sorted list as N
SET i=0
WHILE i<N
    READ Sorted list element as List[i]
```

```
i=i+1
ENDWHILE
READ Element to be insert as X
SET i = N-1
WHILE i >=0 and X < List[i]
    List[i+1] =List[i]
    i = i - 1
END WHILE
List[i+1] = X
```

1.6.3 Guess an Integer Number in a Range

Let's play a little game to give an idea of how different algorithms for the same problem can have wildly different efficiencies. The computer is going to randomly select an integer from 1 to N and ask you to guess it. The computer will tell you if each guess is too high or too low. We have to guess the number by making guesses until you find the number that the computer chose. Once found the number, think about what technique used, when deciding what number to guess next?

Maybe you guessed 1, then 2, then 3, then 4, and so on, until you guessed the right number. We call this approach **linear search**, because you guess all the numbers as if they were lined up in a row. It would work. But what is the highest number of guesses you could need? If the computer selects N, you would need N guesses. Then again, you could be really lucky, which would be when the computer selects 1 and you get the number on your first guess. How about on average? If the computer is equally likely to select any number from 1 to N, then on average you'll need N/2 guesses.

But you could do something more efficient than just guessing 1, 2, 3, 4, ..., right? Since the computer tells you whether a guess is too low, too high, or correct, you can start off by guessing N/2. If the number that the computer selected is less than N/2, then because you know that N/2 is too high, you can eliminate all the numbers from N/2 to N from further consideration. If the number selected by the computer is greater than N/2, then you can eliminate 1 through N/2. Either way, you can eliminate about half the numbers. On your next guess, eliminate half of the remaining numbers. Keep going, always eliminating half of the remaining numbers. We call this halving approach **binary search**, and no matter which number from 1 to N the computer has selected, you should be able to find the number in at most log₂N+1 guesses with this technique. The following table shows the maximum number of guesses for linear search and binary search for a few number sizes:

Highest Number	Max Linear Search Guesses	Max Binary Search Guesses
10	10	4
100	100	7
1,000	1,000	10

10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20

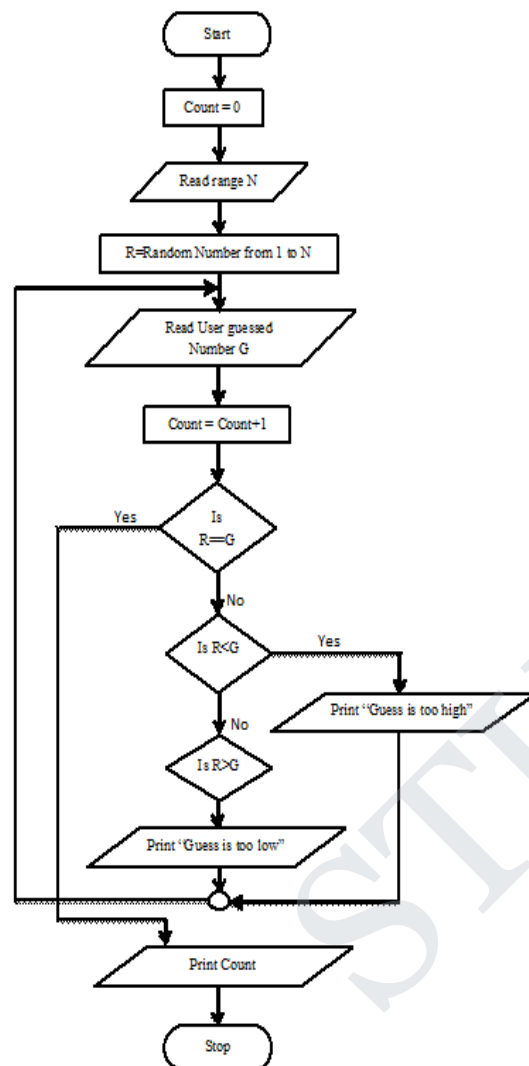


Figure 1.7 Flow Chart for Guess an Integer Number in a Range

Step base algorithm to Guess an Integer Number in a Range

Step 1: Start
 Step 2: SET Count = 0
 Step 3: READ Range as N
 Step 4: SELECT an RANDOMNUMBER from 1 to N as R
 Step 5: READ User Guessed Number as G
 Step 6: Count = Count + 1
 Step 7: IF R == G THEN go to step 10 ELSE go to step 8
 Step 8: IF R < G THEN PRINT "Guess is Too High" AND go to step 5 ELSE go to step 9
 Step 9: PRINT "Guess is Too Low" AND go to step 5
 Step 10: PRINT Count as Number of Guesses Took

Pseudocode to Guess an Integer Number in a Range

```

SET Count = 0
READ Range as N
SELECT an RANDOM NUMBER from 1 to N as R
WHILE TRUE
  READ User guessed Number as G
  Count = Count + 1
  IF R == G THEN
    BREAK
  ELSEIF R < G THEN
    DISPLAY "Guess is Too High"
  ELSE
    DISPLAY "Guess is Too Low"
  ENDIF
ENDWHILE
DISPLAY Count as Number of guesses Took
  
```

1.6.4 Tower of Hanoi

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted in **Figure 1.8**. These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one.

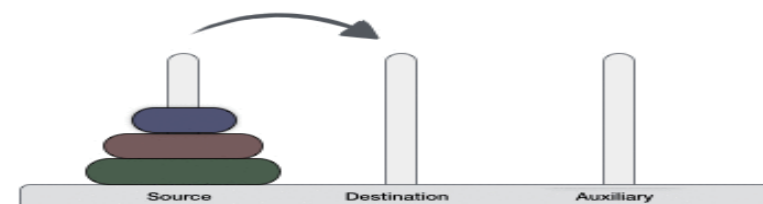


Figure 1.8 Tower of Hanoi

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are:

Rules

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

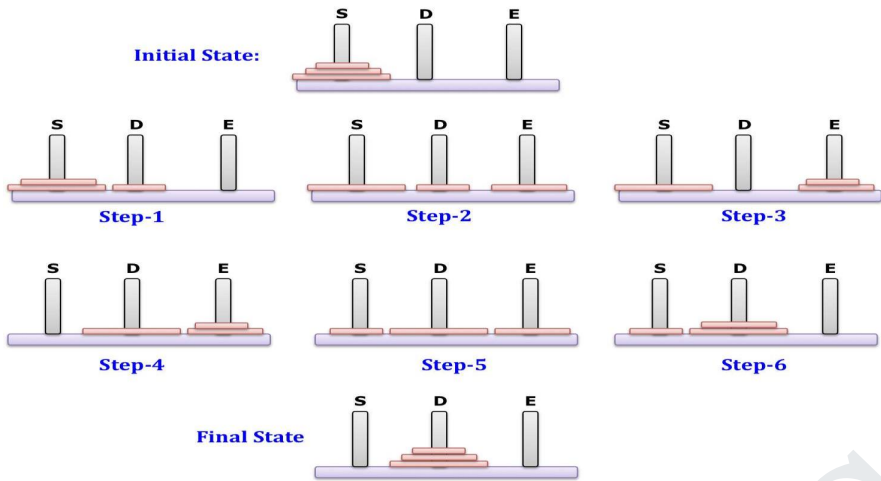


Figure 1.9 Step by Step Moves in Solving Three Disk Tower of Hanoi Problem

Algorithm:

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say \rightarrow 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination tower.

If we have 2 disks –

- First, we move the smaller (top) disk to aux tower.
- Then, we move the larger (bottom) disk to destination tower.
- And finally, we move the smaller disk from aux to destination tower.

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n-1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are –

- Step 1 – Move n-1 disks from source to aux
- Step 2 – Move nth disk from source to dest
- Step 3 – Move n-1 disks from aux to dest

A recursive Step based algorithm for Tower of Hanoi can be driven as follows –

```
Step 1: BEGIN Hanoi(disk, source, dest, aux)
Step 2: IF disk == 1 THEN go to step 3 ELSE go to step 4
Step 3: move disk from source to dest AND go to step 8
Step 4: CALL Hanoi(disk - 1, source, aux, dest)
Step 5: move disk from source to dest
Step 6: CALL Hanoi(disk - 1, aux, dest, source)
Step 7: END Hanoi
```

A recursive Pseudocode for Tower of Hanoi can be driven as follows –

```
Procedure Hanoi(disk, source, dest, aux)
  IF disk == 1 THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest) // Step 1
    move disk from source to dest // Step 2
    Hanoi(disk - 1, aux, dest, source) // Step 3
  END IF
END Procedure
```

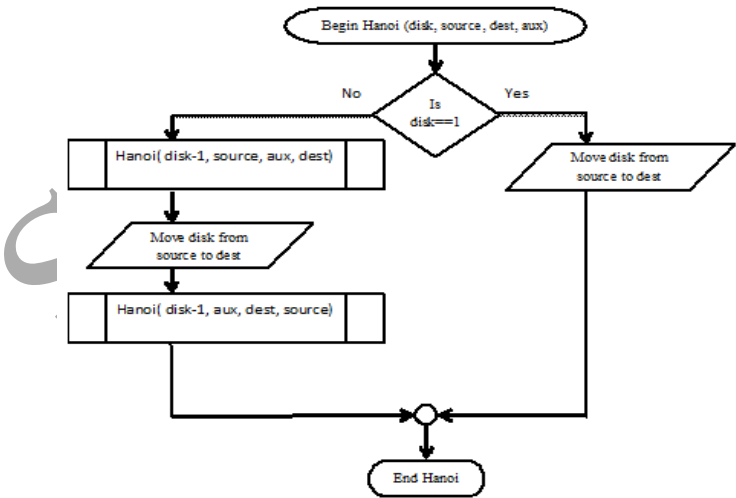


Figure 1.10 Flow Chart for Tower of Hanoi Algorithm

Tower of Hanoi puzzle with n disks can be solved in minimum $2^n - 1$ steps.

1.7 MORE ILLUSTRATIVE PROBLEMS

1.7.1 Temperature Conversion

The commonly used scales for representing temperature are the Celsius scale, denoted in °C and the Fahrenheit scale (°F). Conversion from one scale to another can be done using the following equation.

$$F = (C * 1.8) + 32$$

$$C = (F - 32) * 1/1.8$$

Let us look at the algorithm, Pseudocode, and flow chart for the Temperature conversion problem.

Algorithm in step form

Step 1: Start
Step 2: READ the value in Celsius scale as C
Step 3: Use the formula $F = (C * 1.8) + 32$ to convert Celsius scale to Fahrenheit scale
Step 4: PRINT F
Step 5: READ the value in Fahrenheit scale as F
Step 6: Use the formula $C = (F - 32) * 1/1.8$ to convert Fahrenheit scale to Celsius scale
Step 7: PRINT C
Step 8: Stop

Pseudocode

```
READ Celsius value as C
COMPUTE F = (C*1.8)+32
PRINT F as Fahrenheit
READ Fahrenheit value as F
COMPUTE C = (F-32)*1/1.8
PRINT C as Celsius
```

Flow Chart

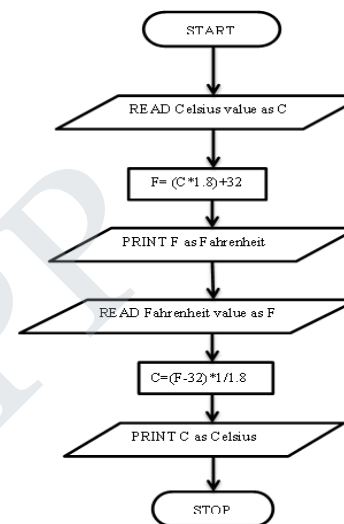


Figure 1.11 Flow Chart for Temperature Conversion

1.7.2 Swap Two Values without Temporary Variable

Swapping is the process of exchanging values of two variables. It can be performed in with temporary variable and without temporary variable method. Here we are going to discuss about the algorithm, Pseudocode, and flow chart for swapping without using temporary variable.

Algorithm in step form

Step 1: Start
Step 2: READ the values of A, B
Step 3: $A = A + B$
Step 4: $B = A - B$
Step 5: $A = A - B$
Step 6: PRINT A, B
Step 8: Stop

Pseudocode

```
READ two values say A, B
COMPUTE A = A+B
COMPUTE B = A-B
COMPUTE A = A-B
PRINT Swapped values of A, B
```

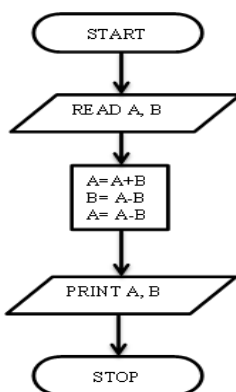


Figure 1.12 Flow Chart for Swapping Two Values without Temporary Variable

1.7.3 Calculate Area and Circumference of a Circle

Area and Circumference of a Circle can be calculated using following formula

$$\text{Area} = 3.14 * r * r$$

$$\text{Circumference} = 2 * 3.14 * r$$

Where r is the radius of a circle.

Following are the algorithm, Pseudocode, and flow chart for the calculation of area and circumference of a circle.

Algorithm in step form

Step1: Start
 Step2: READ radius of the circle as r
 Step3: CALCULATE $\text{area} = 3.14 * r * r$
 Step4: CALCULATE $\text{circumference} = 2 * 3.14 * r$
 Step5: PRINT area
 Step6: PRINT circumference
 Step 7: Stop

Pseudocode

```

READ radius of the circle as r
CALCULATE  $\text{area} = 3.14 * r * r$ 
CALCULATE  $\text{circumference} = 2 * 3.14 * r$ 
PRINT area
PRINT circumference
  
```

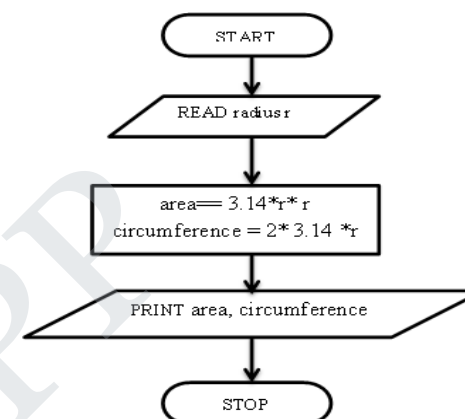


Figure 1.13 Flow Chart for Calculating Area and Circumference of Circle

1.7.4 Eligible for Vote or Not

If a person's age is 18 or greater than 18 then he is eligible to vote, otherwise he is not eligible to vote. Following are the algorithm, Pseudocode, and flow chart for check whether a person is eligible to vote or not.

Algorithm in step form

Step1: Start
 Step2: READ age of a person as age
 Step3: IF $\text{age} \geq 18$ then go to step 4 otherwise go to step 5
 Step4: PRINT "Eligible to Vote" then go to Step 6
 Step5: PRINT "Not Eligible to Vote" then go to Step 6
 Step6: Stop

Pseudocode

```

READ age of a person as age
IF  $\text{age} \geq 18$  THEN
    PRINT "Eligible to Vote"
ELSE
    PRINT "Not Eligible to Vote"
END IF
  
```

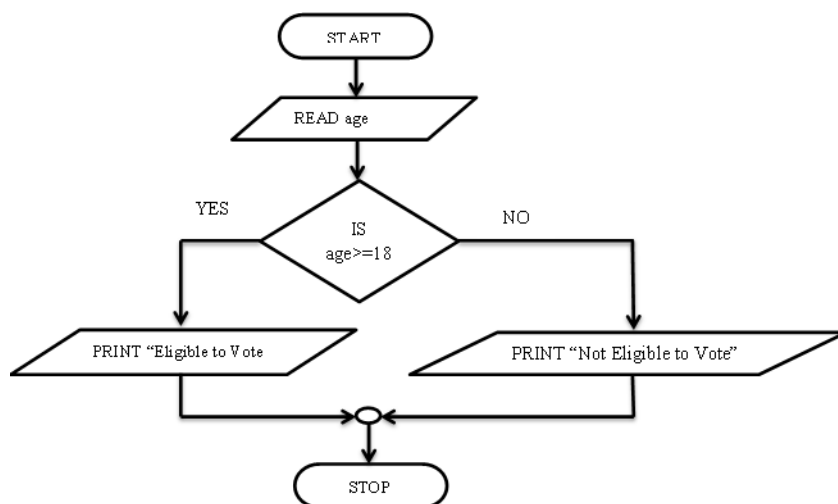


Figure 1.14 Flow Chart to Check Whether a Person is Eligible to Vote or Not.

1.7.5 Biggest Among Three Numbers

Following algorithm, Pseudocode, and flow chart are designed to print the biggest number among the given 3 numbers.

Algorithm in step form

Step 1: Start.
 Step 2: Read the three numbers A, B, C.
 Step 3: Compare A and B. If A is greater perform step 4 else perform step 5.
 Step 4: Compare A and C. If A is greater, output "A is greater" else output "C is greater".
 Then go to step 6
 Step 5: Compare B and C. If B is greater, output "B is greatest" else output "C is greatest".
 Then go to step 6
 Step 6: Stop.

Pseudocode

```

READ three numbers as A, B, C
IF A > B THEN
  IF A > C THEN
    PRINT "A is Big"
  ELSE
    PRINT "C is Big"
ELSE
  IF B > C THEN
    PRINT "B is Big"
  ELSE
    PRINT "C is Big"
  
```

```

ELSE
  PRINT "C is Big"
END IF
  
```

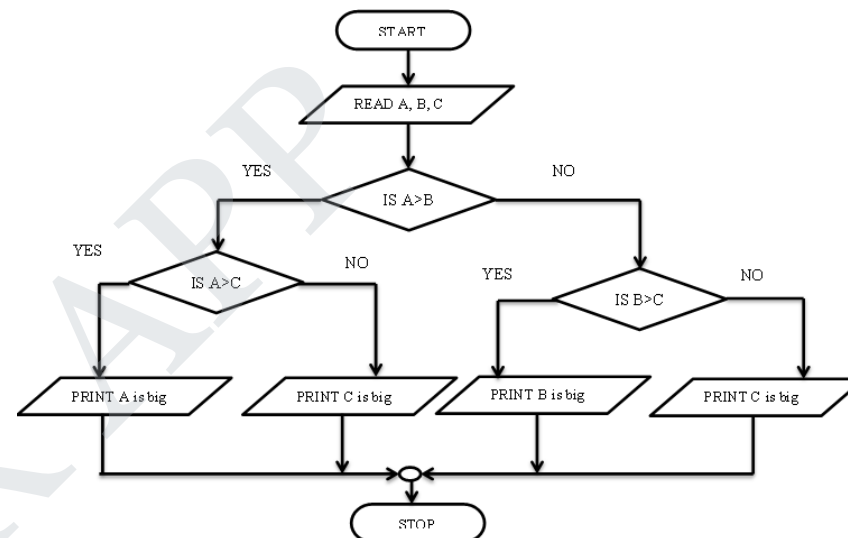


Figure 1.15 Flow Chart to Print Biggest Among Three Numbers

1.7.6 Armstrong Number

Armstrong numbers are numbers, if the sum of the cubes of its digits is equal to the number itself. For example, 153 is an Armstrong number since $1^3 + 5^3 + 3^3 = 153$

Following algorithm, Pseudocode, and flow chart are used to check whether the given number is Armstrong number or not.

Algorithm in step form

Step 1: Start.
 Step 2: READ a number as N
 Step 3: sum=0
 Step 4: temp=N
 Step 5: IF temp > 0 go to step 5.1 ELSE go to step 6
 Step 5.1: digit=temp%10
 Step 5.2: sum=sum+digit*digit*digit
 Step 5.3: temp=temp/10
 Step 5.4: go to step 5
 Step 6: IF N==sum THEN PRINT "Given Number is Armstrong Number" ELSE PRINT "Given Number is Not Armstrong Number"

Step 7: Stop

Pseudocode

```

READ a number as N
SET sum=0
ASSIGN temp=N
WHILE temp>0 THEN
    digit=temp%10
    sum =sum+digit*digit*digit
    temp=temp/10
END WHILE
IF N==sum
    PRINT "Given Number is Armstrong Number"
ELSE
    "Given Number is Not Armstrong Number"
  
```

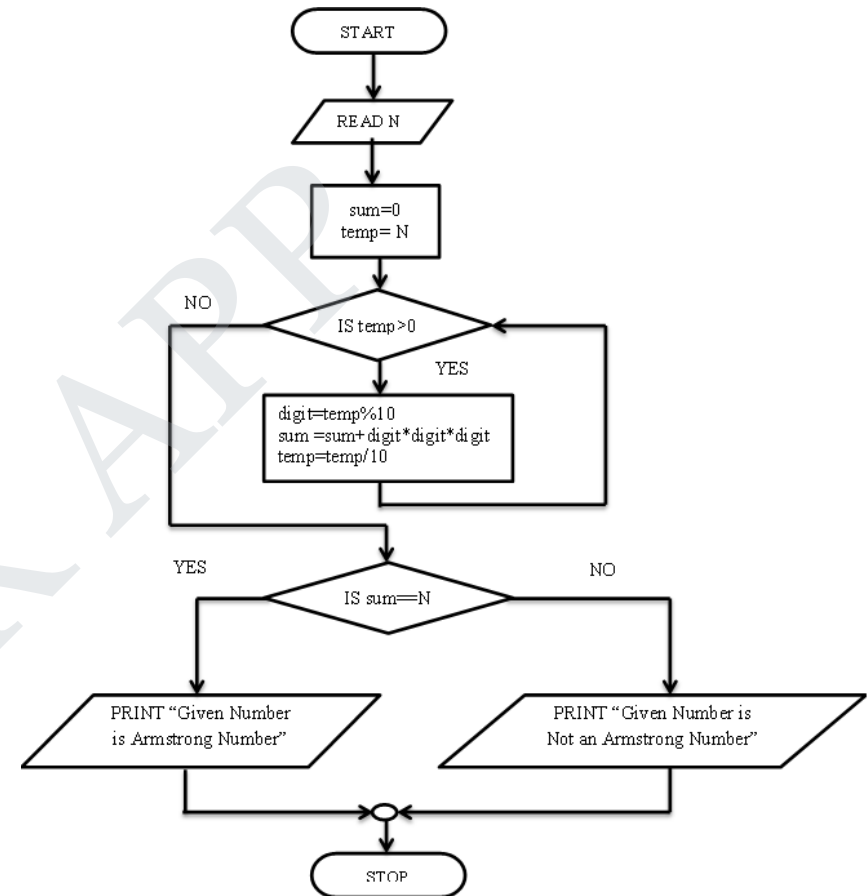


Figure 1.16 Flow Chart to Check Whether the Given Number is Armstrong Number or Not

1.7.7 Number Palindrome

Palindrome Number is a number that remains the same when its digits are reversed. For example 12521 is a Palindrome Number.

Following algorithm, Pseudocode, and flow chart are used to check whether the given number is Palindrome Number or not.

Algorithm in step form

Step 1: Start.

Step 2: READ a number as N

Step 3: reverse=0
 Step 4: temp=N
 Step 5: IF temp> 0 go to step 5.1 ELSE go to step 6
 Step 5.1: digit=temp%10
 Step 5.2: reverse = reverse*10+digit
 Step 5.3: temp=temp/10
 Step 5.4: go to step 5
 Step 6: IF N== reverse THEN PRINT "Given Number is Palindrome Number" ELSE PRINT
 "Given Number is Not Palindrome Number"
 Step 7: Stop

Pseudocode

```

READ a number as N
SET reverse =0
ASSIGN temp=N
WHILE temp>0 THEN
    digit=temp%10
    reverse = reverse*10+digit
    temp=temp/10
END WHILE
IF N== reverse
    PRINT "Given Number is Palindrome Number"
ELSE
    "Given Number is Not Palindrome Number"
  
```

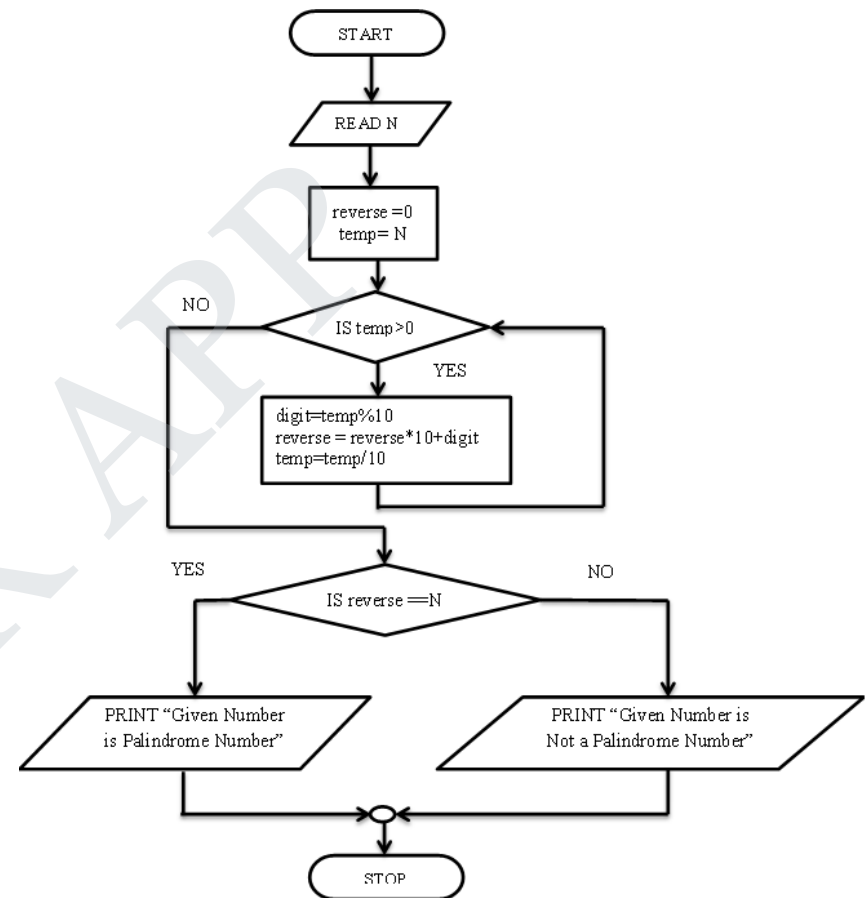


Figure 1.17 Flow Chart to Check Whether the Given Number is Palindrome Number or Not

1.7.8 Calculate the Total and Average of Marks

Algorithm, Pseudocode, and flow chart for calculate Total and average of given number of subjects mark.

Algorithm in step form

Step 1: Start
 Step 2: READ Number of subjects as N
 Step 3: SET i=0
 Step 4: IF i<N THEN go to step 5 ELSE go to step 8
 Step 5: READ single subject mark as List[i]
 Step 6: i=i+1

Step 7: go to step 4
 Step 8: SET $i=0$, $total=0$
 Step 9: IF $i < N$ THEN go to step 10 ELSE go to step 12
 Step 10: $total = total + List[i]$
 Step 11: $i = i + 1$, go to 9
 Step 12: $average = total / N$
 Step 13: PRINT total as Total Marks
 Step 14: PRINT average as Average Marks

Pseudocode

```

READ Number of subjects as N
SET i=0
WHILE i<N
    READ single subject mark as List[i]
    i=i+1
ENDWHILE
SET i=0, total=0
WHILE i<N
    total=total+List[i]
    i=i+1
ENDWHILE
COMPUTE average=total/N
PRINT total as Total Marks
PRINT average as Average Marks
  
```

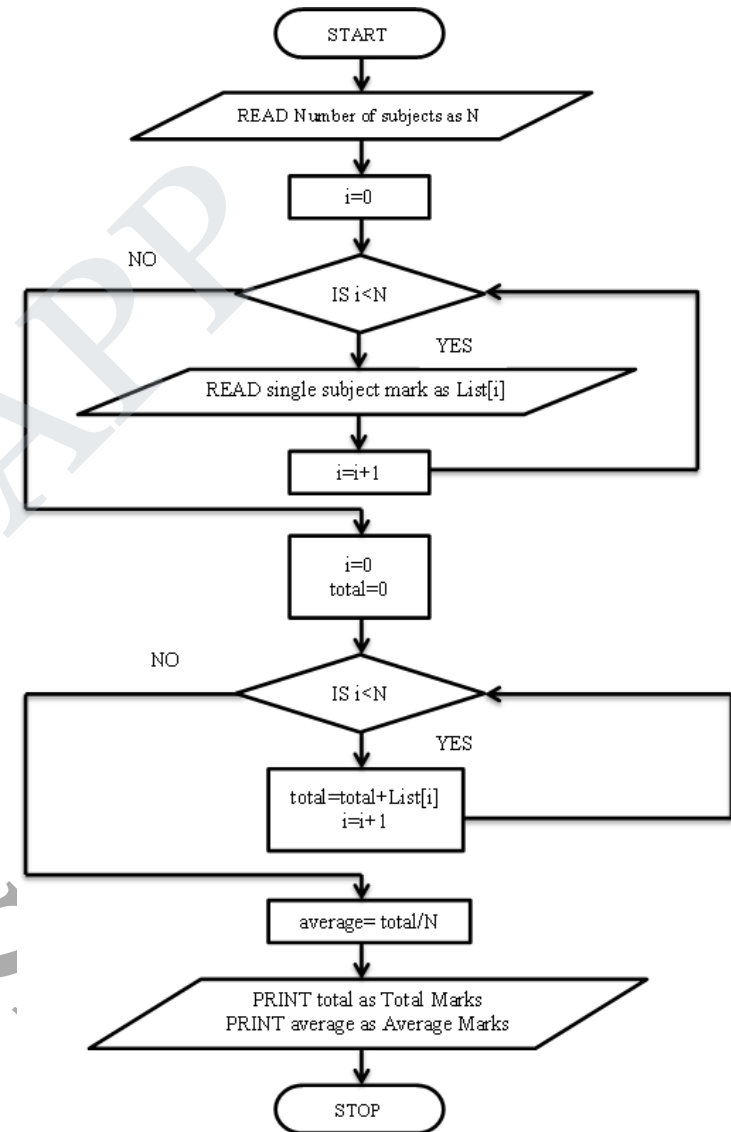


Figure 1.18 Flow Chart to Calculate the Total and Average of Marks

1.7.9 Check Whether a Number is Prime or Not

If a number is divisible only by itself and 1 then it is called prime Number. Following are the Algorithm, Pseudocode, and flow chart to Check Whether a Number is Prime or not.

Algorithm in step form

- Step 1: Start
- Step 2: READ a Number to check as N
- Step 3: SET i=2, flag=0
- Step 4: IF $i \leq N/2$ THEN go to step 4.1 ELSE go to step 5
 - Step 4.1: IF $N \% i == 0$ THEN SET flag=1 and go to step 5 ELSE go to step 4.2
 - Step 4.2: $i=i+1$, go to step 4
- Step 5: IF flag==1 THEN PRINT "Given Number is Not Prime" ELSE PRINT "Given Number is Prime"
- Step 6: Stop

Pseudocode

```
READ a Number to check as N
SET i=2, flag=0
WHILE i<=N/2
    IF N%i==0 THEN
        flag=1
        break;
    ELSE
        i=i+1
END WHILE
IF flag==1 THEN
    PRINT "Given Number is Not Prime"
ELSE
    PRINT "Given Number is Prime"
```

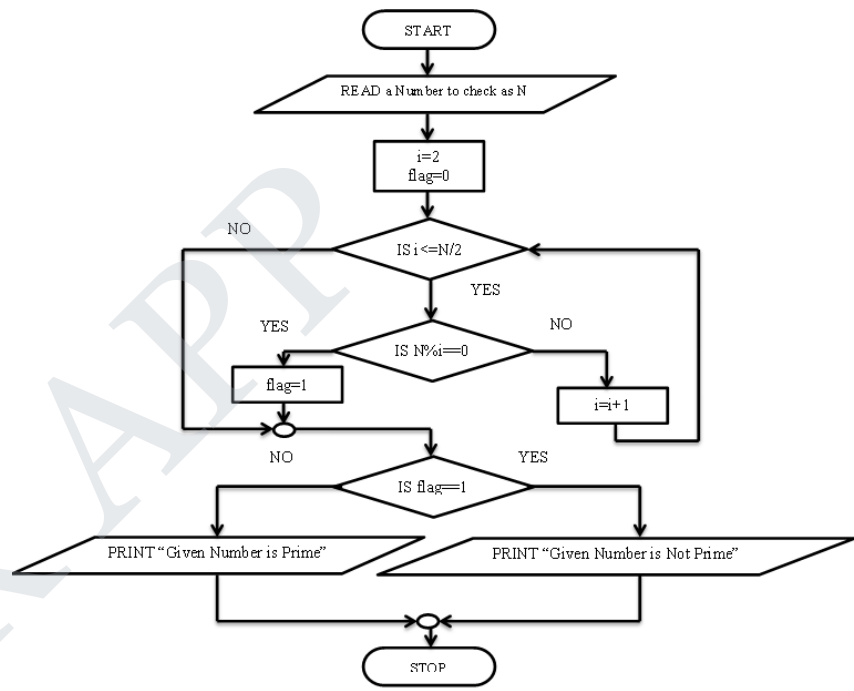


Figure 1.19 Flow Chart to Check Whether a Number is Prime or Not

GE8151 - PROBLEM SOLVING AND PYTHON PROGRAMMING

REGULATIONS – 2017

UNIT – II

Prepared By :

Mr. Vinu S, ME,
Assistant Professor,
St. Joseph's College of Engineering,
Chennai -600119.

UNIT II

DATA, EXPRESSIONS, STATEMENTS

2.1 THE PYTHON INTERPRETER

The Python programming language was conceived in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to the ABC programming language. When he began implementing Python, Guido van Rossum was also reading the published scripts from “**Monty Python’s Flying Circus**”, a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

Python is a cross-platform programming language, meaning, it runs on multiple platforms like Windows, Mac OS X, Linux, Unix and has even been ported to the Java and .NET virtual machines. It is free and open source.

2.1.1 Starting the Interpreter

The Python **interpreter** is a program that reads and executes Python code. Even though most of today’s Linux and Mac have Python preinstalled in it, the version might be out-of-date. So, it is always a good idea to install the most current version.

After installation, the python interpreter lives in the installed directory.

By default it is /usr/local/bin/pythonX.X in Linux/Unix and C:\PythonXX in Windows, where the 'X' denotes the version number. To invoke it from the shell or the command prompt we need to add this location in the search path.

Search path is a list of directories (locations) where the operating system searches for executables. For example, in Windows command prompt, we can type set path=%path%;c:\python27 (python27 means version 2.7, it might be different in your case) to add the location to path for that particular session. In Mac OS we need not worry about this as the installer takes care about the search path.

Now there are two ways to start Python.

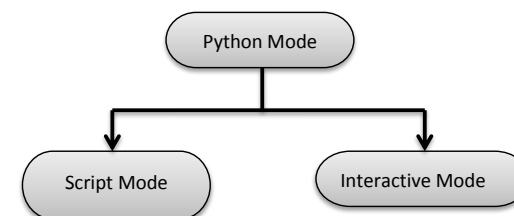


Figure 2.1 Modes of Python Interpreter

2.1.2 Interactive Mode

Typing *python* in the command line will invoke the interpreter in interactive mode. When it starts, you should see output like this:

```
PYTHON 2.7.13 (V2.7.13:A06454B1AFA1, DEC 17 2016, 20:42:59) [MSC V.1500 32 BIT
(INTEL)] ON WIN32
TYPE "COPYRIGHT", "CREDITS" OR "LICENSE()" FOR MORE INFORMATION.
>>>
```

The first three lines contain information about the interpreter and the operating system it's running on, so it might be different for you. But you should check that the version number, which is 2.7.13 in this example, begins with 2, which indicates that you are running Python 2. If it begins with 3, you are running Python 3.

The last line is a **prompt** that indicates that the interpreter is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result:

```
>>> 5 + 4
9
```

This prompt can be used as a calculator. To exit this mode type **exit()** or **quit()** and press enter.

2.1.3 Script Mode

This mode is used to execute Python program written in a file. Such a file is called a **script**. Scripts can be saved to disk for future use. Python scripts have the **extension .py**, meaning that the filename ends with *.py*.

For example: *helloWorld.py*

To execute this file in script mode we simply write *python helloWorld.py* at the command prompt.

2.1.4 Integrated Development Environment (IDE)

We can use any text editing software to write a Python script file.

We just need to save it with the *.py* extension. But using an IDE can make our life a lot easier. IDE is a piece of software that provides useful features like code hinting, syntax highlighting and checking, file explorers etc. to the programmer for application development.

Using an IDE can get rid of redundant tasks and significantly decrease the time required for application development.

IDEL is a graphical user interface (GUI) that can be installed along with the Python programming language and is available from the official website.

We can also use other commercial or free IDE according to our preference. *PyScripter IDE* is one of the Open source IDE.

2.1.5 Hello World Example

Now that we have Python up and running, we can continue on to write our first Python program.

Type the following code in any text editor or an IDE and save it as *helloWorld.py*

```
print("Hello world!")
```

Now at the command window, go to the location of this file. You can use the **cd** command to change directory.

To run the script, type, *python helloWorld.py* in the command window. We should be able to see the output as follows:

```
Hello world!
```

If you are using *PyScripter*, there is a green arrow button on top. Press that button or press *Ctrl+F9* on your keyboard to run the program.

In this program we have used the built-in function *print()*, to print out a string to the screen. String is the value inside the quotation marks, i.e. *Hello world!*.

2.2 VALUES AND TYPES

A **value** is one of the basic things a program works with, like a letter or a number. Some example values are 5, 83.0, and 'Hello, World!'. These values belong to different **types**: 5 is an **integer**, 83.0 is a **floating-point number**, and 'Hello, World!' is a **string**, so-called because the letters it contains are strung together. If you are not sure what type a value has, the interpreter can tell you:

```
>>>type(5)
<class 'int'>
>>>type(83.0)
<class 'float'>
>>>type('Hello, World!')
<class 'str'>
```

In these results, the word "class" is used in the sense of a category; a type is a category of values. Not surprisingly, integers belong to the type *int*, strings belong to *str* and floating-point numbers belong to *float*. What about values like '5' and '83.0'? They look like numbers, but they are in quotation marks like strings.

```
>>>type('5')
<class 'str'>
>>>type('83.0')
<class 'str'>
```

They're strings. When you type a large integer, you might be tempted to use commas between groups of digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```
>>> 1,000,000
(1, 0, 0)
```

That's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. We'll learn more about this kind of sequence later.

2.2.1 Standard Data Types

Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has **five** standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

2.2.1.1 Python Numbers

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

```
var1 =1
var2 =10
```

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

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

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

```
del var
del var_a,var_b
```

Python supports four different numerical types –

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

Examples

Here are some examples of numbers –

Table 2.1: Different number format example

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEI	32.3+e18	.876j

-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating-point numbers denoted by $x + yj$, where x and y are the real numbers and j is the imaginary unit.

2.2.1.2 Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example –

```
str = 'Python Programming'
print str                # Prints complete string
print str[0]             # Prints first character of the string
print str[-1]            # Prints last character of the string
print str[2:5]           # Prints characters starting from 3rd to 5th
print str[2:]            # Prints string starting from 3rd character
print str * 2            # Prints string two times
print str + " Course"    # Prints concatenated string
```

This will produce the following result –

```
Python Programming
P
g
tho
thon Programming
Python ProgrammingPython Programming
Python Programming Course
```

2.2.1.3 Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The

plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.
For example –

```
list = [ 'Hai', 123 , 1.75, 'vinu', 100.25 ]
smalllist = [251, 'vinu']
print list                # Prints complete list
print list[0]             # Prints first element of the list
print list[-1]            # Prints last element of the list
print list[1:3]           # Prints elements starting from 2nd till 3rd
print list[2:]            # Prints elements starting from 3rd element
print smalllist * 2       # Prints list two times
print list + smalllist    # Prints concatenated lists
```

This produces the following result –

```
['Hai', 123, 1.75, 'vinu', 100.25]
Hai
100.25
[123, 1.75]
[1.75, 'vinu', 100.25]
[251, 'vinu', 251, 'vinu']
['Hai', 123, 1.75, 'vinu', 100.25, 251, 'vinu']
```

2.2.1.4 Python Boolean

A Boolean type was added to Python 2.3. Two new constants were added to the `__builtin__` module, **True** and **False**. True and False are simply set to integer values of 1 and 0 and aren't a different type.
The type object for this new type is **named bool**; the constructor for it takes any Python value and converts it to True or False.

```
>>>bool(1)
True
>>>bool(0)
False
```

Python's Booleans were added with the primary goal of making code clearer. For example, if you're reading a function and encounter the statement `return 1`, you might wonder whether the 1 represents a Boolean truth value, an index, or a coefficient that multiplies some other quantity. If the statement is `return True`, however, the meaning of the return value is quite clear.

Python's Booleans were *not* added for the sake of strict type-checking. A very strict language such as Pascal would also prevent you performing arithmetic with Booleans, and would require that the expression in an if statement always evaluate to a Boolean result. Python is not this strict and never will be. This means you can still use any expression in an if statement, even ones that evaluate to a list or tuple or some random object. The Boolean type is a subclass of the int class so that arithmetic using a Boolean still works.

```
>>> True + 1
```

```
2
>>> False + 1
1
>>> False * 85
0
>>> True * 85
85
>>> True+True
2
>>> False+False
0
```

We will discuss about data types like Tuple, Dictionary in [Unit IV](#).

2.2.2 Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Table 2.2: Data Type Conversion

Function	Description
<code>int(x [,base])</code>	Converts x to an integer. base specifies the base if x is a string.
<code>long(x [,base])</code>	Converts x to a long integer. base specifies the base if x is a string.
<code>float(x)</code>	Converts x to a floating-point number.
<code>complex(real [,imag])</code>	Creates a complex number.
<code>str(x)</code>	Converts object x to a string representation.
<code>repr(x)</code>	Converts object x to an expression string.
<code>eval(str)</code>	Evaluates a string and returns an object.
<code>list(s)</code>	Converts s to a list.
<code>chr(x)</code>	Converts an integer to a character.

unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

2.3 VARIABLES

A variable is a name that refers to a value. Variable reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory.

2.3.1 Assignment Statements

An assignment statement creates a new variable and gives it a value:

```
>>>message = 'Introducing Python Variable'
>>>num = 15
>>>radius = 5.4
```

This example makes three assignments. The first assigns a string to a new variable named message; the second gives the integer 15 to num; the third assigns floating point value 5.4 to variable radius.

2.3.2 Variable Names

Programmers generally choose names for their variables that are meaningful

The Rules

- Variables names must start with a letter or an underscore, such as:
 - `_mark`
 - `mark_`
- The remainder of your variable name may consist of letters, numbers and underscores.
 - `subject1`
 - `my2ndsubject`
 - `un_der_scores`
- Names are case sensitive.
 - `case_sensitive`, `CASE_SENSITIVE`, and `Case_Sensitive` are each a different variable.
- Can be any (reasonable) length

- There are some reserved (**KeyWords**) words which you cannot use as a variable name because Python uses them for other things.

The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python 3 has these keywords:

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

You don't have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you'll know.

If you give a variable an illegal name, you get a syntax error:

```
>>>lbook = 'python'
SyntaxError: invalid syntax
>>>more@ = 1000000
SyntaxError: invalid syntax
>>>class = 'Fundamentals of programming'
SyntaxError: invalid syntax
```

`lbook` is illegal because it begins with a number. `more@` is illegal because it contains an illegal character, `@`. `class` is illegal because it is a keyword.

Good Variable Name

- Choose meaningful name instead of short name. `roll_no` is better than `rn`.
- Maintain the length of a variable name. `Roll_no_of_a_student` is too long?
- Be consistent; `roll_no` or `orRollNo`
- Begin a variable name with an underscore(`_`) character for a special case.

2.4 EXPRESSIONS AND STATEMENTS

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```
>>> 50
50
```

```
>>> 10<5
False
>>> 50+20
70
```

When you type an expression at the prompt, the interpreter evaluates it, which means that it finds the value of the expression.

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

```
>>> n = 25
>>> print(n)
```

The first line is an assignment statement that gives a value to n. The second line is a print statement that displays the value of n. When you type a statement, the interpreter executes it, which means that it does whatever the statement says. In general, statements don't have values.

2.4.1 Difference Between a Statement and an Expression

A statement is a complete line of code that performs some action, while an expression is any section of the code that evaluates to a value. Expressions can be combined "horizontally" into larger expressions using operators, while statements can only be combined "vertically" by writing one after another, or with block constructs. Every expression can be used as a statement, but most statements cannot be used as expressions.

2.5 OPERATORS

In this section, you'll learn everything about different types of operators in Python, their syntax and how to use them with examples.

Operators are special symbols in Python that carry out computation. The value that the operator operates on is called the operand.

For example:

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

Here, + is the operator that performs addition. 10 and 5 are the operands and 15 is the output of the operation. Python has a number of operators which are classified below.

- Arithmetic operators
- Comparison (Relational) operators
- Logical (Boolean) operators
- Bitwise operators

- Assignment operators
- Special operators

2.5.1 Arithmetic Operators

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

Table 2.3: Arithmetic operators in Python

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Example

```
x = 7
y = 3
print('x + y =',x+y)
print('x - y =',x-y)
print('x * y =',x*y)
print('x / y =',x/y)
print('x // y =',x//y)
print('x % y =',x%y)
print('x ** y =',x**y)
```

When you run the program, the output will be:

```
x + y = 10
x - y = 4
x * y = 21
x / y = 2.3333333333333335
x // y = 2
x % y = 1
x ** y = 343
```

2.5.2 Comparison or Relational Operators

Comparison operators are used to compare values. It either returns True or False according to the condition.

Table 2.4: Comparison Operators in Python

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

Example

```
x = 5
y = 7
print('x > y is',x>y)
print('x < y is',x<y)
print('x == y is',x==y)
print('x != y is',x!=y)
print('x >= y is',x>=y)
print('x <= y is',x<=y)
```

When you run the program, the output will be:

```
x > y is False
x < y is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

2.5.3 Logical Operators

Logical operators are the **and**, **or**, **not** operators.

Table 2.5: Logical operators in Python

Operator	Meaning	Example
and	True if both the operands are true	<code>x and y</code>
or	True if either of the operands is true	<code>x or y</code>
not	True if operand is false (complements the operand)	<code>not x</code>

Example

```
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

When you run the program, the output will be:

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

2.5.4 Bitwise Operators

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

In Table 2.6: Let `x = 10` (0000 1010 in binary) and `y = 4` (0000 0100 in binary)

Table 2.6: Bitwise operators in Python

Operator	Meaning	Example
&	Bitwise AND	<code>x & y = 0</code> (0000 0000)
	Bitwise OR	<code>x y = 14</code> (0000 1110)
~	Bitwise NOT	<code>~x = -11</code> (1111 0101)
^	Bitwise XOR	<code>x ^ y = 14</code> (0000 1110)
>>	Bitwise right shift	<code>x >> 2 = 2</code> (0000 0010)
<<	Bitwise left shift	<code>x << 2 = 40</code> (0010 1000)

Example

```
x=10
y=4
print('x&y=',x&y)
print('x|y=',x|y)
print('~x=',~x)
print('x^y=',x^y)
print('x>>2=',x>>2)
print('x<<2=',x<<2)
```

When you run the program, the output will be:

```
x&y= 0
x|y= 14
~x= -11
x^y= 14
```

```
x>>> 2= 2
x<<< 2= 40
```

2.5.5 Assignment Operators

Assignment operators are used in Python to assign values to variables.

`a = 10` is a simple assignment operator that assigns the value 10 on the right to the variable `a` on the left.

There are various compound operators in Python like `a += 10` that adds to the variable and later assigns the same. It is equivalent to `a = a + 10`.

Table 2.7: Assignment operators in Python

Operator	Example	Equivalent to
=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 5</code>	<code>x = x + 5</code>
-=	<code>x -= 5</code>	<code>x = x - 5</code>
*=	<code>x *= 5</code>	<code>x = x * 5</code>
/=	<code>x /= 5</code>	<code>x = x / 5</code>
%=	<code>x %= 5</code>	<code>x = x % 5</code>
//=	<code>x //= 5</code>	<code>x = x // 5</code>
**=	<code>x **= 5</code>	<code>x = x ** 5</code>
&=	<code>x &= 5</code>	<code>x = x & 5</code>
=	<code>x = 5</code>	<code>x = x 5</code>
^=	<code>x ^= 5</code>	<code>x = x ^ 5</code>
>>=	<code>x >>= 5</code>	<code>x = x >> 5</code>
<<=	<code>x <<= 5</code>	<code>x = x << 5</code>

2.5.6 Special Operators

Python language offers some special type of operators like the identity operator or the membership operator. They are described below with examples.

2.5.6.1 Identity Operators

is and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Table 2.8: Identity operators in Python

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	<code>x is True</code>
is not	True if the operands are not identical (do not refer to the same object)	<code>x is not True</code>

Example

```
x1 = 7
y1 = 7
x2 = 'Welcome'
y2 = 'Welcome'
x3 = [1,2,3]
y3 = [1,2,3]
print(x1 is not y1)
print(x2 is y2)
print(x3 is y3)
```

When you run the program, the output will be:

```
False
True
False
```

Here, we see that `x1` and `y1` are integers of same values, so they are equal as well as identical. Same is the case with `x2` and `y2` (strings). But `x3` and `y3` are list. They are equal but not identical. Since list are mutable (can be changed), interpreter locates them separately in memory although they are equal.

2.5.6.2 Membership Operators

in and **not in** are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

Table 2.9: Membership operators in Python

Operator	Meaning	Example
in	True if value/variable is found in the sequence	<code>5 in x</code>
not in	True if value/variable is not found in the sequence	<code>5 not in x</code>

Example

```
x = 'Python Programming'
print('Program' not in x)
print('Program' in x)
print('program' in x)
```

When you run the program, the output will be:

```
False
True
False
```

Here, 'Program' is in `x` but 'program' is not present in `x`, since Python is case sensitive.

2.6 PRECEDENCE OF PYTHON OPERATORS

The combination of values, variables, operators and function calls is termed as an expression. Python interpreter can evaluate a valid expression. When an expression contains more than one operator, the order of evaluation depends on the **Precedence** of operations.

For example, multiplication has higher precedence than subtraction.

```
>>> 20 - 5*3
5
```

But we can change this order using parentheses () as it has higher precedence.

```
>>> (20 - 5) * 3
45
```

The operator precedence in Python are listed in the following table. It is in descending order, upper group has higher precedence than the lower ones.

Table 2.10: Operator precedence rule in Python

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

2.7 ASSOCIATIVITY OF PYTHON OPERATORS

We can see in the above table that more than one operator exists in the same group. These operators have the same precedence.

When two operators have the same precedence, associativity helps to determine which the order of operations.

Associativity is the order in which an expression is evaluated that has multiple operator of the same precedence. Almost all the operators have left-to-right associativity.

For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, left one is evaluated first.

```
>>> 10 * 7 // 3
23
>>> 10 * (7//3)
20
>>> (10 * 7)//3
23
```

We can see that $10 * 7 // 3$ is equivalent to $(10 * 7) // 3$.

Exponent operator ** has right-to-left associativity in Python.

```
>>> 5 ** 2 ** 3
390625
>>> (5** 2) **3
15625
>>> 5 ** (2 **3)
390625
```

We can see that $2 ** 3 ** 2$ is equivalent to $2 ** (3 ** 2)$.

2.7.1 Non Associative Operators

Some operators like assignment operators and comparison operators do not have associativity in Python. There are separate rules for sequences of this kind of operator and cannot be expressed as associativity.

For example, $x < y < z$ neither means $(x < y) < z$ nor $x < (y < z)$. $x < y < z$ is equivalent to $x < y$ and $y < z$, and is evaluated from left-to-right.

Furthermore, while chaining of assignments like $x = y = z$ is perfectly valid, $x = y += z$ will result into error.

2.8 TUPLE ASSIGNMENT

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

```
>>> 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, you 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'
```

2.9 COMMENTS

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they start with the **# symbol**:

```
# compute Area of a triangle using Base and Height
area= (base*height)/2
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
area= (base*height)/2 # Area of a triangle using Base and Height
```

Everything from the # to the end of the line is ignored—it has no effect on the execution of the program. Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out what the code does; it is more useful to explain why.

This comment is redundant with the code and useless:

```
base= 5 # assign 5 to base
```

This comment contains useful information that is not in the code:

```
base = 5 # base is in centimetre.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade off.

If we have comments that extend **multiple lines**, one way of doing it is to use hash (#) in the beginning of each line. For example:

```
#This is a long comment
#and it extends
#to multiple lines
```

Another way of doing this is to use triple quotes, either ''' or ''''.

These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a
perfect example of
multi-line comments"""
```

2.9.1 Docstring in Python

Docstring is short for documentation string. It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring. Triple quotes are used while writing docstrings. For example:

```
def area(r):
    """Compute the area of Circle"""
    return 3.14159*r**2
```

Docstring is available to us as the attribute `__doc__` of the function. Issue the following code in shell once you run the above program.

```
>>>print(area.__doc__)
Compute the area of Circle
```

2.10. MODULES AND FUNCTIONS

2.10.1 Functions

In the context of programming, a **function** is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes code reusable.

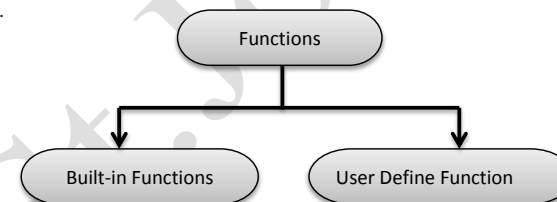


Figure 2.2 Types of Functions

Functions can be classified into

- Built-in Functions
- User Defined Functions

2.10.1.1 Built-in Functions

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. We have already seen one example of a **function call**: `type()`

```
>>>type(25)
<class 'int'>
```

The name of the function is type. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument. It is common to say that a function “takes” an argument and “returns” a result. The result is also called the **return value**.

Python provides functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>>int('25')
25
>>>int('Python')
ValueError: invalid literal for int(): Python
```

`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>>int(9.999999)
9
>>>int(-2.3)
-2
```

`float` converts integers and strings to floating-point numbers:

```
>>>float(25)
25.0
>>>float('3.14159')
3.14159
```

Finally, `str` converts its argument to a string:

```
>>>str(25)
'25'
>>>str(3.14159)
'3.14159'
```

2.10.1.1.1. `range()` – function

The `range()` constructor returns an immutable sequence object of integers between the given start integer to the stop integer.

Python's `range()` Parameters

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(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>range(5,10)
[5, 6, 7, 8, 9]
>>>range(10,1,-2)
[10, 8, 6, 4, 2]
```

2.10.1.1.2. Printing to the Screen

In **python 3**, `print` function will print as strings everything in a comma-separated sequence of expressions, and it will separate the results with single blanks by default. Note that you can mix types: anything that is not already a string is automatically converted to its string representation.

Eg.

```
>>> x=10
>>> y=7
>>> print('The sum of', x, 'plus', y, 'is', x+y)
The sum of 10 plus 7 is 17
You can also use it with no parameters:
```

`print()`

to just advance to the next line.

In **python 2**, simplest way to produce output is using the `print` statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows

```
>>> x=10
>>> y=7
>>> print 'The sum of', x, 'plus', y, 'is', x+y
The sum of 10 plus 7 is 17
```

2.10.1.1.3. Reading Keyboard Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are –

- `raw_input`
- `input`

The `raw_input` Function

The `raw_input([prompt])` function reads one line from standard input and returns it as a string.

```
>>>str = raw_input("Enter your input: ");
Enter your input: range(0,10)
>>> print "Received input is : ", str
Received input is : range(0,10)
```

This prompts you to enter any string and it would display same string on the screen.

The `input` Function

The `input([prompt])` function is equivalent to `raw_input`, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
>>>str = input("Enter your input: ");
Enter your input: range(0,10)
>>> print "Received input is : ", str
Received input is : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Input data is evaluated and the list is generated
```

2.10.1.2 User-defined functions

As you already know, Python gives you many built-in functions like `print()`, `input()`, `type()` etc. but you can also create your own functions. These functions are called *user-defined functions*.

2.10.1.2.1 Function Definition and Use

Syntax of Function definition

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

Above shown is a function definition which consists of following components.

1. Keyword **def** marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. **Parameters** (arguments) through which we pass values to a function. They are **optional**.
4. A colon (:) to mark the end of function header.
5. **Optional documentation string** (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An **optional return** statement to return a value from the function.

Example of a function

```
def welcome(person_name):
    """This function welcome
    the person passed in as
    parameter"""
    print(" Welcome ", person_name , " to Python Function Section")
```

Using Function or Function Call

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> welcome('Vinu')
Welcome Vinu to Python Function Section
```

The return statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

```
return [expression_list]
```

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the `None` object.

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num
print(absolute_value(5))
print(absolute_value(-7))
```

When you run the program, the output will be:

```
5
7
```

2.10.2 Flow of Execution

To ensure that a function is defined before its first use, you have to know the order statements run in, which is **called the flow of execution**. Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that

statements inside the function don't run until the function is called. A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off. **Figure 2.3** show the flow of execution

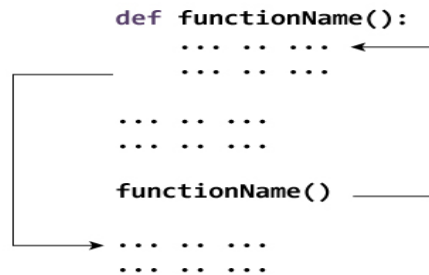


Figure 2.3 Flow of execution when function Call

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function! Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

2.10.3 Parameters and Arguments

Some of the functions we have seen require **arguments**. For example, when you call `type()` you pass a variable or value as an argument. Some functions take more than one argument: eg, `range()` function take one or two or three arguments.

Inside the function, the **arguments** are assigned to variables called **parameters**. Here is a definition for a function that takes an argument:

```
def welcome(person_name):
    print(" Welcome ", person_name , " to Python Function Section")
```

This function assigns the argument to a parameter named `person_name`. When the function is called, it prints the value of the parameter (whatever it is). This function works with any value that can be printed.

```
>>> welcome("Vinu")
Welcome Vinu to Python Function Section
>>> welcome(100)
Welcome 100 to Python Function Section
>>> welcome(50.23)
Welcome 50.23 to Python Function Section
```

The argument is evaluated before the function is called, so in the below examples the expressions `'vinu'*3` is evaluated.

```
>>> welcome('vinu'*3)
Welcome vinuvinuvinu to Python Function Section
```

You can also use a variable as an argument:

```
>>> student_name='Ranjith'
>>> welcome(student_name)
Welcome Ranjith to Python Function Section
```

2.10.4 Function Arguments

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

- Required arguments
- Default arguments
- Keyword 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.

```
def add(a,b):
    return a+b
a=10
b=20
print "Sum of ", a ,"and ", b, "is" , add(a,b)
```

To call the function **add()**, you definitely need to pass two argument, otherwise it gives a syntax error as follows

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

```
Sum of 10 and 20 is 30
```

If we miss to give an argument it will show syntax error. Example

```
def add(a,b):
    return a+b
a=10
b=20
print "Sum of ", a ,"and ", b, "is" , add(a)
It will produce Error message as follows
Sum of 10 and 20 is
```

Traceback (most recent call last):

```
File "G:/class/python/code/required_arguments.py", line 5, in <module>
    print "Sum of ", a ,"and ", b, "is" , add(a)
TypeError: add() takes exactly 2 arguments (1 given)
```

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. The following example gives an idea on default arguments, it add default value of b if it is not passed while calling.

```
def add(a,b=0):
    print "Sum of ", a ,"and ", b, "is" ,a+b
a=10
b=20
add(a,b)
add(a)
```

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

```
Sum of 10 and 20 is 30
Sum of 10 and 0 is 10
```

When default argument is used in program, Non default arguments should come before default arguments.

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 add() function in the following ways –

```
def add(a,b):
    print "Sum of ", a ,"and ", b, "is" ,a+b
a=10
b=20
add(b=a,a=b)
```

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

```
Sum of 20 and 10 is 30
```

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

```
def printvalues( arg1, *vartuple ):
    print "Output is: "
    print arg1
    for var in vartuple:
        print var

printvalues( 20 )
```

```
printvalues( 50, 60, 55 )
```

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

```
Output is:
20
Output is:
50
60
55
```

2.10. 5 The Anonymous Functions or Lambda Functions

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

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of **lambda** functions contains only a single statement, which is as follows

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

Following is the example to show how lambda form of function works

```
sum = lambda a, b: a + b;
print "Sum is : ", sum( 5, 10 )
print "Sum is : ", sum( 30, 50 )
```

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

```
Sum is: 15
Sum is: 80
```

2.10.6 Modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A **module** is a file that contains a collection of related functions. Python has lot of built-in modules; math module is one of them. math module provides most of the familiar mathematical functions.

Before we can use the functions in a module, we have to import it with an **import statement**:

```
>>> import math
```

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

```
>>> 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**.

```
>>> math.log10(200)
2.3010299956639813
>>> math.sqrt(10)
3.1622776601683795
```

Math module have functions like log(), sqrt(), etc... In order to know what are the functions available in particular module, we can use **dir()** function after importing particular module. Similarly if we want to know detail description about a particular module or function or variable means we can use help() function.

Eg.

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsun', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> help(pow)
Help on built-in function pow in module __builtin__:
```

```
pow(...)
pow(x, y[, z]) -> number
```

With two arguments, equivalent to $x^{**}y$. With three arguments, equivalent to $(x^{**}y) \% z$, but may be more efficient (e.g. for longs).

2.10.6.1 Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named addModule.py with the following code:

```
def add(a, b):
    result = a + b
    print(result)
add(10,20)
```

If you run this program, it will add 10 and 20 and print 30. We can import it like this:

```
>>> import addModule
30
```

Now you have a module object addModule

```
>>> addModule
<module 'addModule' from 'G:/class/python/code/addModule.py'>
```

The module object provides add():

```
>>> addModule.add(120,150)
270
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    add(10,20)
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `'__main__'`; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped. Modify addModule.py file as given below.

```
def add(a, b):
    result = a + b
    print(result)
if __name__ == '__main__':
    add(10,20)
```

Now while importing addModule test case is not running

```
>>> import addModule
```

`__name__` has module name as its value when it is imported. Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed. If you want to reload a module, you can use the built-in function reload, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

2.11 ILLUSTRATIVE PROGRAMS

2.11.1 Exchange the Value of two Variables

In python exchange of values of two variables (swapping) can be done in different ways

- Using Third Variable
- Without Using Third Variable
 - Using Tuple Assignment method
 - Using Arithmetic operators
 - Using Bitwise operators

2.11.1.1 Using Third Variable

```
1 var1 = input("Enter value of variable1: ")
2 var2 = input("Enter value of variable2: ")
3 temp = var1
4 var1 = var2
5 var2 = temp
6 print("After swapping:")
7 print("First Variable =",var1,)
8 print("Second Variable=",var2,)
```

When you run the program, the output will be:

```
Enter value of variable1: 5
Enter value of variable2: 10
After swapping:
First Variable = 10
Second Variable= 5
```

In the above program line number 3,4,5, are used to swap the values of two variables. In this program, we use the *temp* variable to temporarily hold the value of *var1*. We then put the value of *var2* in *var1* and later *temp* in *var2*. In this way, the values get exchanged. In this method we are using one more variable other than *var1*, *var2*. So we calling it as swapping with the use of third variable.

2.11.1.2 Without Using Third Variable

We can do the same swapping operation even without the use of third variable, There are number of ways to do this, they are.

2.11.1.2.1 Using Tuple Assignment method

In this method in the above program instead of line number 3,4,5 use a single tuple assignment statement

```
var1, var2 = var2, var1
```

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.

2.11.1.2.2 Using Arithmetic operators

If the variables are both numbers, Swapping can be performed using simple mathematical addition subtraction relationship or multiplication division relationship.

Addition and Subtraction

In this method in the above program instead of line number 3,4,5 use the following code

```
x = x + y
y = x - y
x = x - y
```

Multiplication and Division

In this method in the above program instead of line number 3,4,5 use the following code

```
x = x * y
y = x / y
x = x / y
```

2.11.1.2.3 Using Bitwise operators

If the variables are integers then we can perform swapping with the help of bitwise XOR operator. In order to do this in the above program instead of line number 3,4,5 use the following code

```
x = x ^ y
y = x ^ y
x = x ^ y
```

2.11.2 Circulate the Value of N Variables

Problem of circulating a Python list by an arbitrary number of items to the right or left can be easily performed by List slicing operator.

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Figure 2.4.a Example List

Consider the above list Figure 2.4.a; circulation of the above list by *n* position can be easily achieved by slicing the array into two and concatenating them. Slicing is done as ***n*th element to end element + beginning element to *n*-1th element**. Suppose *n*=2 means, given list is rotated 2 positions towards left side as given in Figure 2.4.b

3	4	5	6	7	1	2
---	---	---	---	---	---	---

Figure 2.4.b Left Circulated List

Suppose *n* = - 2 means, given list is rotated 2 position towards right side as given in Figure 2.4.c

6	7	1	2	3	4	5
---	---	---	---	---	---	---

Figure 2.4.c Right Circulated List

So the simple function to perform this circulation operation is

```
def circulate(list, n):
    return list[n:] + list[:n]

>>> circulate([1,2,3,4,5,6,7], 2)
[3, 4, 5, 6, 7, 1, 2]
>>> circulate([1,2,3,4,5,6,7], -2)
[6, 7, 1, 2, 3, 4, 5]
```

2.11.3 Test for Leap Year

In order to check whether a year is leap year or not python provide a *isleap()* function in calendar module. So if you want to check whether a year is leap year or not, first import calendar module. Then use the *isleap()* function.

```
>>> import calendar
>>> calendar
<module 'calendar' from 'C:\Python27\lib\calendar.pyc'>
>>> calendar.isleap(2003)
False
>>> import calendar
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2004)
True
```

```
>>> calendar.isleap(2003)
False
>>> calendar.isleap(1900)
False
>>> calendar.isleap(2020)
True
```

GE8151 - PROBLEM SOLVING AND PYTHON
PROGRAMMING

REGULATIONS – 2017

UNIT – III

Prepared By :

Mr. Vinu S, ME,
Assistant Professor,
St. Joseph's College of Engineering,
Chennai -600119.

UNIT III

CONTROL FLOW, FUNCTIONS

3.1 CONDITIONALS

Flow of execution of instruction can be controlled using conditional statements. Conditional statements have some Boolean expression. Boolean expression can have relational operators or logical operators or both.

3.1.1 Boolean Expressions

A **boolean expression** is an expression its result is either true or false. The following examples use the operator `==`, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 9 == 9
True
>>> 9 == 6
False
```

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

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

Boolean expression can have relational operators or logical operators. The `==` operator is one of the relational operators; the others are:

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

More explanation can found in [2.5.2](#). Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a relational operator. There is no such thing as `=<` or `=>`.

3.1.2 Logical operators

There are three logical operators: **and**, **or**, and **not**. The semantics (meaning) of these operators is similar to their meaning in English. This was explained in [2.5.3](#)

For example:

$x > 0$ and $x < 10$ is true only if x is greater than 0 and less than 10.

$n\%2 == 0$ or $n\%3 == 0$ is true if either or both of the conditions is true, that is, if the number is divisible by 2 or 3.

not operator negates a boolean expression, so **not** ($x > y$) is true if $x > y$ is false, that is, if x is less than or equal to y .

Note:- Any nonzero number is interpreted as True

```
>>>95 and True
True
>>>mark=95
>>>mark>0 and mark<=100
True
>>> mark=102
>>> mark>0 and mark<=100
False
```

3.2 SELECTION

In Unit I, you were introduced to the concept of flow of control: the sequence of statements that the computer executes. In procedurally written code, the computer usually executes instructions in the order that they appear. However, this is not always the case. One of the ways in which programmers can change the flow of control is the use of selection control statements.

Now we will learn about selection statements, which allow a program to choose when to execute certain instructions. For example, a program might choose how to proceed on the basis of the user's input. As you will be able to see, such statements make a program more versatile.

In python selection can be achieved through

- if statement
- The elif Statement
- if...elif...else
- Nested if statements

3.2.1 Conditional Execution

In order to write useful programs, we almost always need the ability to check conditions and change the behaviour of the program accordingly. **Selection or Conditional** statements give us this ability. The simplest form is the **if statement**:

General Syntax of if statement is

if TEST EXPRESSION:

STATEMENT(S)

executed if condition evaluates to True

Here, the program evaluates the TEST EXPRESSION and will execute statement(s) only if the text expression is True. If the text expression is False, the statement(s) is not executed.

A few important things to note about if statements:

1. The colon (:) is significant and required. It separates the **header** of the **compound statement** from the **body**.
2. The line after the colon must be indented. It is standard in Python to use four spaces for indenting.
3. All lines indented the same amount after the colon will be executed whenever the BOOLEAN_EXPRESSION is true.
4. Python interprets non-zero values as True. None and 0 are interpreted as False.

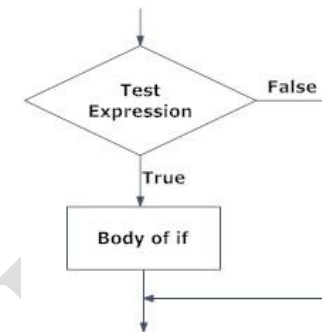


Figure 3.1 if Statement Flowchart

Here is an example:

```
mark = 102
if mark >= 100:
    print(mark, " is a Not a valid mark.")
print("This is always printed.")
```

Output will be:

```
102 is a Not a valid mark.
This is always printed.
```

The boolean expression after the if statement (here $mark \geq 100$) is called the **condition**. If it is true, then all the indented statements get executed.

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually

as a place keeper for code you haven't written yet). In that case, you can use the **pass** statement, which does nothing.

```
if x < 0:
    pass                # TODO: need to handle negative values!
```

3.2.2 Alternative execution

A second form of the if statement is "alternative execution", in which there are two possibilities and the condition determines which one runs. In other words, it is frequently the case that you want one thing to happen when a condition is true, and **something else** to happen when it is false. For that we have the if else statement. The syntax looks like this:

General Syntax of if .. else statement is

```
if TEST 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 test expression evaluates to True. The entire block of statements is skipped if the test expression evaluates to False, and instead all the statements under the else clause are executed.

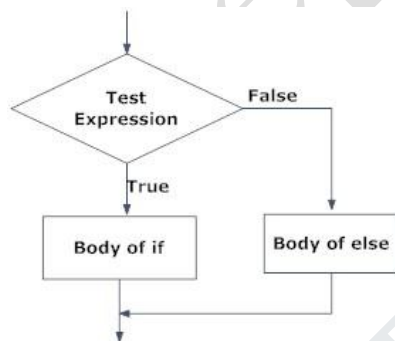


Figure 3.2 if..else Flowchart

There is no limit on the number of statements that can appear under the two clauses of an if else statement, but there has to be at least one statement in each block. Occasionally, it is useful to have a section with no statements, for code you haven't written yet. In that case, you can use the **pass** statement, which does nothing except act as a placeholder.

```
if True:
    pass                # This is always true
else:
    pass                # so this is always executed, but it does nothing
```

Here is an example:

```
age=21                                #age=17
if age >= 18:
    print("Eligible to vote")
else:
    print("Not Eligible to vote")
```

Output will be:

Eligible to vote

In the above example, when age is greater than 18, the test expression is true and body of if is executed and body of else is skipped. If age is less than 18, the test expression is false and body of else is executed and body of if is skipped. If age is equal to 18, the test expression is true and body of if is executed and body of else is skipped.

3.2.3 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**. The syntax looks like this:

General Syntax of if..elif...else statement is

```
if TEST EXPRESSION1:
    STATEMENTS_A
elif TEST EXPRESSION2:
    STATEMENTS_B
else:
    STATEMENTS_C
```

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

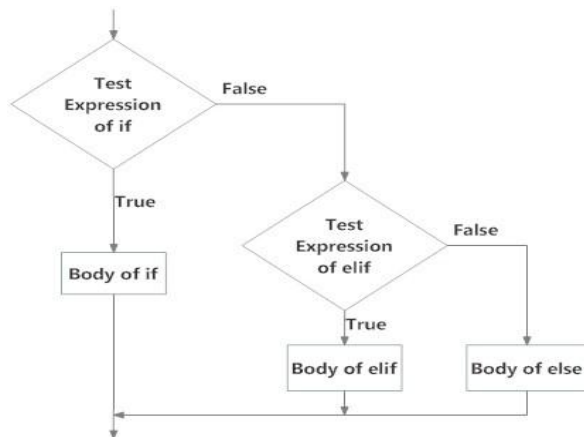


Figure 3.3 Flowchart of if...elif...else

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.

Here is an example:

```

time=17                                #time=10, time=13, time=17, time=22
if time<12:
    print("Good Morning")
elif time<15:
    print("Good Afternoon")
elif time<20:
    print("Good Evening")
else:
    print("Good Night")
  
```

Output will be:

Good Evening

When variable time is less than 12, Good Morning is printed. If time is less than 15, Good Afternoon is printed. If time is less than 20, Good Evening is printed. If all above conditions fails Good Night is printed.

3.2.4 Nested Conditionals

One conditional can also be nested within another. ie, We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting.

General Syntax of Nested if..else statement is

```

if TEST EXPRESSION1:
    if TEST EXPRESSION2:
        STATEMENTS_B
    else:
        STATEMENTS_C
else:
    if TEST EXPRESSION3:
        STATEMENTS_D
    else:
        STATEMENTS_E
  
```

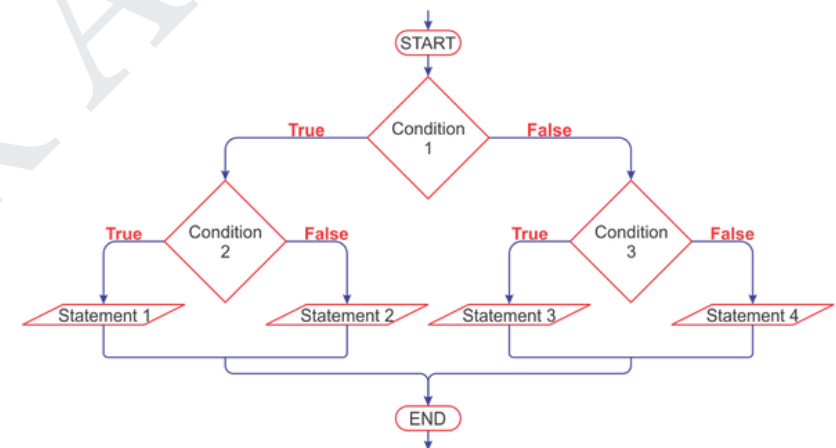


Figure 3.4 Flowchart of Nested if..else

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

Here is an example:

```

a=10
b=20
c=5
if a>b:
    if a>c:
        print("Greatest number is ",a)
    else:
  
```

```

        print("Greatest number is ",c)
    else:
        if b>c:
            print("Greatest number is ",b)
        else:
            print("Greatest number is ",c)

```

Output will be :
Greatest number is 20

Although the indentation of the statements makes the structure apparent, nested conditionals very quickly become difficult to read. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the above code using single if...elif.... else statement:

```

a=10
b=20
c=5
if a>b and a>c:
    print("Greatest number is ",a)
elif b>a and b>c:
    print("Greatest number is ",b)
else:
    print("Greatest number is ",c)

```

Another example, we can rewrite the following code using a single conditional:

```

if 0 < x:
    if x < 10:
        print('x is a positive single-digit number. ')

```

The print statement runs only if we make it pass both conditionals, so we can get the same effect with the and operator:

```

if 0 < x and x < 10:
    print('x is a positive single-digit number. ')

```

For this kind of condition, Python provides a more concise option:

```

if 0 < x < 10:
    print('x is a positive single-digit number. ')

```

3.3 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.

Repeated execution of a set of statements is called **iteration**. Python has two statements for iteration

- for statement
- while statement

Before we look at those, we need to review a few ideas.

Reassignment

As we saw back in the variable section, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```

age = 26
print(age)
age = 17
print(age)

```

The output of this program is

```

26
17

```

because the first time age is printed, its value is 26, and the second time, its value is 17.

Here is what **reassignment** looks like in a **state snapshot**:

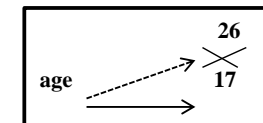


Figure 3.5 State Diagram for Reassignment

With reassignment it is especially important to distinguish between an assignment statement and a boolean expression that tests for equality. Because Python uses the equal token (=) for assignment, it is tempting to interpret a statement like a = b as a boolean test. Unlike mathematics, Remember that the Python token for the equality operator is ==.

Note too that an equality test is symmetric, but assignment is not. For example, if a == 7 then 7 == a. But in Python, the statement a = 7 is legal and 7 = a is not.

Furthermore, in mathematics, a statement of equality is always true. If a == b now, then a will always equal b. In Python, an assignment statement can make two variables equal, but because of the possibility of reassignment, they don't have to stay that way:

```

a = 5
b = a           # after executing this line, a and b are now equal
a = 3           # after executing this line, a and b are no longer equal

```

The third line changes the value of a but does not change the value of b, so they are no longer equal.

Updating variables

When an assignment statement is executed, the right-hand-side expression (i.e. the expression that comes after the assignment token) is evaluated first. Then the result of that evaluation is written into the variable on the left hand side, thereby changing it.

One of the most common forms of reassignment is an update, where the new value of the variable depends on its old value.

```
n = 5
n = 3 * n + 1
```

The second line means “get the current value of n, multiply it by three and add one, and put the answer back into n as its new value”. So after executing the two lines above, n will have the value 16.

If you try to get the value of a variable that doesn’t exist yet, you’ll get an error:

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

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

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

This second statement — updating a variable by adding 1 to it. It is very common. It is called an **increment** of the variable; subtracting 1 is called a **decrement**.

3.3.1 The while statement

The **while loop** in Python is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don't know beforehand, the number of times to iterate.

The general syntax for the while statement :

```
while TEST_EXPRESSION:
    STATEMENTS
```

In while loop, test expression is checked first. The body of the loop is entered only if the TEST_EXPRESSION evaluates to True. After one iteration, the test expression is checked again. This process continues until the TEST_EXPRESSION evaluates to False.

In Python, the body of the while loop is determined through indentation. Body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as False.

Flowchart of while Loop

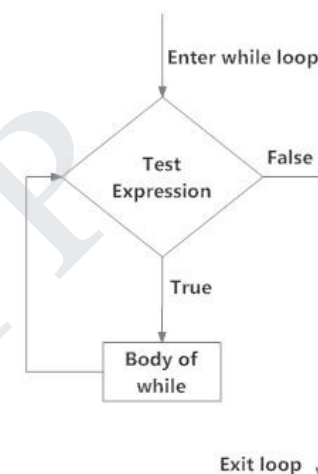


Figure 3.6 Flowchart of while Loop

Example:

Python program to find sum of first n numbers using while loop

```
n = 20
sum = 0                                # initialize sum and counter
i = 1
while i <= n:
    sum = sum + i
    i = i + 1                          # update counter
print("The sum is", sum)              # print the sum
```

When you run the program, the output will be:

```
The sum is 210
```

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (20 in our program).

We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop). Finally the result is displayed.

3.3.2 The for Statement

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

The general syntax for the while statement:

```
for LOOP_VARIABLE in SEQUENCE:  
    STATEMENTS
```

Here, LOOP_VARIABLE is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

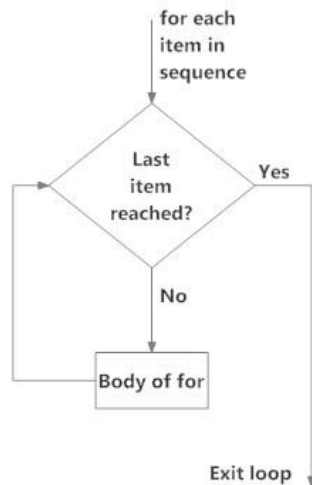


Figure 3.7 Flowchart of for Loop

Example

```
marks = [95,98,89,93,86]
total = 0
for subject_mark in marks:
    total = total+subject_mark
print("Total Mark is ", total)
```

when you run the program, the output will be:

```
Total Mark is 461
```

We can use the range() function in for loops to iterate through a sequence of numbers.

Example:

```
sum=0
for i in range(20):
    sum=sum+i
```

```
print("Sum is ", sum)
```

Output will be:

```
Sum is 190
```

3.3.3 Break, Continue, Pass

You might face a situation in which you need to exit a loop completely when an external condition is triggered or there may also be a situation when you want to skip a part of the loop and start next execution.

Python provides **break** and **continue** statements to handle such situations and to have good control on your loop. This section will discuss the *break*, *continue* and *pass* statements available in Python.

3.3.3.1 The break Statement

The **break** statement in Python terminates the current loop and resumes execution at the next statement. The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

Example:

```
for letter in 'Welcome':
    if letter == 'c':
        break
    print('Current Letter :', letter)
var = 10
while var > 0:
    print('Current variable value :', var)
    var = var -1
    if var == 5:
        break
print "End!"
```

This will produce the following output:

```
Current Letter : W
Current Letter : e
Current Letter : l
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
End!
```

3.3.3.2 The continue Statement:

The **continue** statement in Python returns the control to the beginning of the while loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The **continue** statement can be used in both *while* and *for* loops.

Example:

```
for letter in 'Welcome':           # First Example
    if letter == 'c':
        continue
    print('Current Letter :', letter)
var = 10                           # Second Example
while var > 0:
    print('Current variable value :', var)
    var = var - 1
    if var == 5:
        continue
print "End!"
```

This will produce the following **output**:

```
Current Letter : W
Current Letter : e
Current Letter : l
Current Letter : o
Current Letter : m
Current Letter : e
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 5
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
End!
```

3.3.3.3 The else Statement Used with Loops

Python supports to have an **else** statement associated with a loop statements.

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

while loop with else

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

Here is an example to illustrate this.

```
counter = 0
```

```
while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Output

```
Inside loop
Inside loop
Inside loop
Inside else
```

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

for loop with else

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

Here is an example to illustrate this.

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

When you run the program, the **output will be**:

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

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints

3.3.3.4 The pass Statement

In Python programming, **pass** is a null statement. The difference between a **comment** and **pass** statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when pass is executed. It results into no operation (NOP).

Syntax of pass

```
pass
```

We generally use it as a placeholder.

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

Example

```
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

We can do the same thing in an **empty function**

```
def function(args):
    pass
```

3.4 FRUITFUL FUNCTIONS

3.4.1 Return Values

The built-in functions we have used, such as *abs*, *pow*, *int*, *max*, and *range*, have produced results. Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.

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

Here, we are going to write more functions that return values, which we will call **fruitful functions**, for want of a better name. The first example is *area*, which returns the area of a circle with the given radius:

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

We have seen the *return* statement before, but in a fruitful function the *return* statement includes a **return value**. This statement means: evaluate the return expression, and then return it immediately as the result of this function. The expression provided can be arbitrarily complicated, so we could have written this function like this:

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

On the other hand, **temporary variables** like *b* above often make debugging easier.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional. We have already seen the built-in *abs*, now we see how to write our own:

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

```
else:
    return x
```

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

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

Think about this version and convince yourself it works the same as the first one.

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

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

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

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

```
>>> print(bad_absolute_value(0))
None
```

All Python functions return *None* whenever they do not return another value.

It is also possible to use a return statement in the middle of a *for* loop, in which case control immediately returns from the function. Let us assume that we want a function which looks through a list of words. It should return the first 2-letter word. If there is not one, it should return the empty string:

```
def find_first_2_letter_word(xs):
    for wd in xs:
        if len(wd) == 2:
            return wd
    return ""
```

While running output will be:

```
>>> find_first_2_letter_word(["This", "is", "a", "dead", "parrot"])
'is'
>>> find_first_2_letter_word(["I", "like", "cheese"])
"
```

Single-step through this code and convince yourself that in the first test case that we've provided, the function returns while processing the second element in the list: it does not have to traverse the whole list.

3.4.2 Incremental Development

To deal with increasingly complex programs, we are going to suggest a technique 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 we 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:

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

The first step is to consider what a *distance* function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function that captures our thinking so far:

```
def distance(x1, y1, x2, y2):  
    return 0.0
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated.

To test the new function, we call it with sample values:

```
>>> distance(1, 2, 4, 6)  
0.0
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be — in the last line we added.

A logical first step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. We will refer to those values using temporary variables named *dx* and *dy*.

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    return 0.0
```

If we call the function with the arguments shown above, when the flow of execution gets to the return statement, *dx* should be 3 and *dy* should be 4. We can check that this is the case in **PyScripter** by putting the cursor on the return statement, and running the program to break execution when it gets to the cursor (using the *F4* key). Then we inspect the variables *dx* and *dy* by hovering the mouse above them, to confirm that the function is getting the right parameters and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of *dx* and *dy*:

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquared = dx*dx + dy*dy  
    return 0.0
```

Again, we could run the program at this stage and check the value of *dsquared* (which should be 25).

Finally, using the fractional exponent *0.5* to find the square root, we compute and return the result:

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquared = dx*dx + dy*dy  
    result = dsquared**0.5  
    return result
```

If that works correctly, you are done. Otherwise, you might want to inspect the value of *result* before the return statement.

When you start out, you might add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger conceptual chunks. Either way, stepping through your code one line at a time and verifying that each step matches your expectations can save you a lot of debugging time. As you improve your programming skills you should find yourself managing bigger and bigger chunks: this is very similar to the way we learned to read letters, syllables, words, phrases, sentences, paragraphs, etc., or the way we learn to chunk music — from individual notes to chords, bars, phrases, and so on.

The key aspects of the process are:

1. Start with a working skeleton program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
2. Use temporary variables to refer to intermediate values so that you can easily inspect and check them.
3. Once the program is working, relax, sit back, and play around with your options.
4. You might want to consolidate multiple statements into one bigger compound expression, or rename the variables you've used, or see if you can make the function shorter. A good guideline is to aim for making code as easy as possible for others to read.

Here is another version of the function. It makes use of a square root function that is in the *math* module

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

While running output will be:

```
>>> distance(1, 2, 4, 6)
5.0
```

3.4.3 Debugging with print

Another powerful technique for debugging (an alternative to single-stepping and inspection of program variables), is to insert *extra print* functions in carefully selected places in your code. Then, by inspecting the output of the program, you can check whether the algorithm is doing what you expect it to. Be clear about the following, however:

- You must have a clear solution to the problem, and must know what should happen before you can debug a program. Work on *solving* the problem on a piece of paper (perhaps using a flowchart to record the steps you take) *before* you concern yourself with writing code. Writing a program doesn't solve the problem — it simply *automates* the manual steps you would take. So first make sure you have a pen-and-paper manual solution that works. Programming then is about making those manual steps happen automatically.
- Do not write **chatterbox** functions. A chatterbox is a fruitful function that, in addition to its primary task, also asks the user for input, or prints output, when it would be more useful if it simply shut up and did its work quietly.

For example, we've seen built-in functions like *range*, *max* and *abs*. None of these would be useful building blocks for other programs if they prompted the user for input, or printed their results while they performed their tasks.

So a good tip is to avoid calling *print* and *input* functions inside fruitful functions, *unless the primary purpose of your function is to perform input and output*. The one exception to this rule might be to temporarily sprinkle some calls to *print* into your code to help debug and understand what is happening when the code runs, but these will then be removed once you get things working.

3.4.4 Composition

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

As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables *xc* and *yc*, and the perimeter point is in *xp* and *yp*. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we've just written a function, *distance*, that does just that, so now all we have to do is use it:

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

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

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

Wrapping that up in a function, we get:

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

We called this function *area2* to distinguish it from the *area* function defined earlier.

The temporary variables *radius* and *result* are useful for development, debugging, and single-stepping through the code to inspect what is happening, but once the program is working, we can make it more concise by composing the function calls:

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

3.5 SCOPE: GLOBAL AND LOCAL

Namespace is a collection of names. In Python, you can imagine a namespace as a mapping of every name; you have defined, to corresponding objects. Different namespaces can co-exist at a given time but are completely isolated.

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.

Scope is the portion of the program from where a namespace can be accessed directly without any prefix.

At any given moment, there are at least three nested scopes.

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.

If there is a function inside another function, a new scope is nested inside the local scope.

Example

```
def outer_function():
    b = "India"
    def inner_func():
        c = "TamilNadu"
    a = "World"
```

Here, the variable *a* is in the global namespace. Variable *b* is in the local namespace of *outer_function()* and *c* is in the nested local namespace of *inner_function()*.

When we are in inner_function(), c is local to us, b is nonlocal and a is global. We can read as well as assign new values to c but can only read b and a from inner_function().

If we try to assign as a value to b, a new variable b is created in the local namespace which is different than the nonlocal b. Same thing happens when we assign a value to a.

However, if we declare a as global, all the reference and assignment go to the global a. Similarly, if we want to rebind the variable b, it must be declared as nonlocal. The following example will further clarify this.

```
def outer_function():
    a = "i am in India"
    def inner_function():
        a = "i am in TamilNadu"
        print('a =',a)
    inner_function()
    print('a =',a)
a = "i am in World"
outer_function()
print('a =',a)
```

As you can see, the **output** of this program is

```
a = i am in TamilNadu
a = i am in India
a = i am in World
```

In this program, three different variables a are defined in separate namespaces and accessed accordingly. While in the following program,

```
def outer_function():
    global a
    a = "i am in India"
    def inner_function():
        global a
        a = "i am in TamilNadu"
        print('a =',a)
    inner_function()
    print('a =',a)
a = "i am in World"
outer_function()
print('a =',a)
```

The **output** of the program is.

```
a = i am in TamilNadu
a = i am in TamilNadu
a = i am in TamilNadu
```

Here, all reference and assignment are to the global a due to the use of keyword global.

3.6 RECURSION

A recursive function is a function which calls itself with "smaller (or simpler)" input values. Generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive function to solve that problem.

3.6.1 Base Condition in Recursion

Base case is the smallest version of the problem, which cannot be expressed in terms of smaller problems. Also this base case has a predefined solution. In a recursive program, the solution to the base case is provided and the solution of a bigger problem is expressed in terms of smaller problems.

For Example:

Following is an example of a recursive function to find the factorial of an integer.

For positive values of n, let's write n!, as we know n! is a product of numbers starting from n and going down to 1. $n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1$. But notice that $(n-1) \cdot \dots \cdot 2 \cdot 1$ is another way of writing $(n-1)!$, and so we can say that $n! = n \cdot (n-1)!$. So we wrote n! as a product in which one of the factors is $(n-1)!$. You can compute n! by computing $(n-1)!$ and then multiplying the result of computing $(n-1)!$ by n. You can compute the factorial function on n by first computing the factorial function on $n-1$. So computing $(n-1)!$ is a subproblem that we solve to compute n!.

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

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Python program to find factorial of a given number using a recursive function is.

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return (n * factorial(n-1))
num = 5
print("The factorial of", num, "is", factorial(num))
```

Output will be:

```
The factorial of 7 is 120
```

In the above example, *factorial()* is a recursive function as it calls itself. When we call this function with a positive integer, it will recursively call itself by decreasing

the number. Each function call multiplies the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
factorial(5)           # 1st call with 5
5 * factorial(4)       # 2nd call with 4
5 * 4 * factorial(3)   # 3rd call with 3
5 * 4 * 3 * factorial(2) # 4th call with 2
5 * 4 * 3 * 2 * factorial(1) # 5th call with 1
5 * 4 * 3 * 2 * 1      # return from 5th call as n==1
5 * 4 * 3 * 2          # return from 4th call
5 * 4 * 6              # return from 3rd call
5 * 24                 # return from 2nd call
120                   # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls **itself infinitely**

Advantages of recursion

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

Disadvantages of recursion

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

3.7 STRINGS

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

```
>>> subject = 'python'
>>> letter = subject [1]
```

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

```
>>> letter
'y'
```

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

```
>>> letter = subject [0]
>>> letter
'p'
```

So p is the 0th letter ("zero-eth") of 'python', y is the 1th letter ("one-eth"), and t is the 2th letter ("two-eth").

As an index you can use an expression that contains variables and operators:

```
>>> i = 1
>>> subject [i]
'y'
>>> subject [i+1]
't'
```

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

```
>>> letter =subject [1.5]
TypeError: string indices must be integers
```

3.7.1 len()

len is a built-in function that returns the number of characters in a string:

```
>>> subject = 'python'
>>> len(subject)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(subject)
>>> last = subject [length]
IndexError: string index out of range
```

The reason for the IndexError is that there is no letter in 'python' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>> last = subject [length-1]
>>> last
't'
```

Or you can use negative indices, which count backward from the end of the string. The expression **subject [-1]** yields the last letter, **subject [-2]** yields the second to last, and so on.

3.7.2 Traversal with a for Loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a while loop:

```
index = 0
while index < len(subject):
    letter = subject [index]
    print(letter)
```

```
index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(subject)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop doesn't run. The last character accessed is the one with the index `len(subject)-1`, which is the last character in the string.

Another way to write a traversal is with a for loop:

```
for letter in subject:
```

```
    print(letter)
```

subject	→	p	y	t	h	o	n	,
Index		0	1	2	3	4	5	

Figure 3.8 Slice indices

Each time through the loop, the next character in the string is assigned to the variable `letter`. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a for loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *MakeWay for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
```

```
suffix = 'ack'
```

```
for letter in prefixes:
```

```
    print(letter + suffix)
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Quack
```

3.7.3 String Slices

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

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

The operator `[n:m]` returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing between the characters, as in **Figure 3.8**. If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> subject = 'python'
>>> subject[:3]
'.pyt'
>>> subject[3:]
'hon'
```

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

```
>>> subject = 'python'
>>> subject[3:3]
''
```

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

Continuing this example, `subject[:]` gives entire string.

```
>>> subject = 'python'
>>> subject[:]
python
```

3.7.4 Strings are Immutable

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

```
>>> greeting = 'Hello, world!'
```

```
>>> greeting[0] = 'J'
```

```
TypeError: 'str' object does not support item assignment
```

The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

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

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

3.7.5 String Methods

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

```
>>> word = 'python'
>>> new_word = word.upper()
>>> new_word
'PYTHON'
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no arguments.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on `word`.

As it turns out, there is a string method named `find` that is remarkably similar to the function we wrote:

```
>>> word = 'python'
>>> index = word.find('t')
>>> index
2
```

In this example, we invoke find on word and pass the letter we are looking for as a parameter. Actually, the find method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('ho')
3
```

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

```
>>> word='welcome'
>>> word.find('e')
1
>>> word.find('e',2)
6
```

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

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

This search fails because b does not appear in the index range from 1 to 2, not including 2. Searching up to, but not including, the second index makes find consistent with the slice operator.

3.7.6 More String Methods

Python includes the following built-in methods to manipulate strings

SN	Methods	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	endswith(suffix,beg=0, end=len(string))	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	expandtabs(tabsize=8)	Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.

8	find(str, beg=0 end=len(string))	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	index(str, beg=0, end=len(string))	Same as find(), but raises an exception if str not found.
10	isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	isdigit()	Returns true if string contains only digits and false otherwise.
13	islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	isnumeric()	Returns true if a unicode string contains only numeric characters and false otherwise.
15	isspace()	Returns true if string contains only whitespace characters and false otherwise.
16	istitle()	Returns true if string is properly "titlecased" and false otherwise.
17	isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	join(seq)	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	len(string)	Returns the length of the string
20	ljust(width[, fillchar])	Returns a space-padded string with the original string left-justified to a total of width columns.
21	lower()	Converts all uppercase letters in string to lowercase.
22	lstrip()	Removes all leading whitespace in string.
23	maketrans()	Returns a translation table to be used in translate function.
24	max(str)	Returns the max alphabetical character from

		the string str.
25	min(str)	Returns the min alphabetical character from the string str.
26	replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	rfind(str, beg=0, end=len(string))	Same as find(), but search backwards in string.
28	rindex(str, beg=0, end=len(string))	Same as index(), but search backwards in string.
29	rjust(width[, fillchar])	Returns a space-padded string with the original string right-justified to a total of width columns.
30	rstrip()	Removes all trailing whitespace of string.
31	split(str='', num=string.count(str))	Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	splitlines(num=string.count('\n'))	Splits string at all (or num) NEWLINES and returns a list of each line with NEWLINES removed.
33	startswith(str, beg=0, end=len(string))	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.
34	strip([chars])	Performs both lstrip() and rstrip() on string
35	swapcase()	Inverts case for all letters in string.
36	title()	Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	translate(table, deletechars='')	Translates string according to translation table str(256 chars), removing those in the del string.
38	upper()	Converts lowercase letters in string to uppercase.
39	zfill(width)	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	isdecimal()	Returns true if a unicode string contains only decimal characters and false otherwise.

3.7.7 The in Operator

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 't' in 'python'
True
>>> 'jan' in 'python'
False
```

For example, the following function prints all the letters from word1 that also appear in word2:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Here’s what you get if you compare 'django' and 'mongodb'

```
>>> in_both('django','mongodb')
d
n
g
o
```

3.7.8 String Comparison

The relational operators work on strings. To see if two strings are equal:

```
if word == 'python':
    print('All right, python.')
```

Other relational operations are useful for putting words in alphabetical order:

```
if word < 'python':
    print('Your word, ' + word + ', comes before python.')
elif word > 'python':
    print('Your word, ' + word + ', comes after python.')
else:
    print('All right, python.')
```

Python does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Python, comes before c language

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison.

3.8 LISTS AS ARRAYS

Most of programs work not only with variables. They also use lists of variables. For example, a program can handle an information about students in a class by reading the list of students from the keyboard or from a file. A change in the number of students in the class must not require modification of the program source code.

To store such data, in Python you can use the data structure called list (in most programming languages the different term is used — “array”).

Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. A list (array) is a set of objects. Individual objects can be accessed using ordered indexes that represent the position of each object within the list (array).

The list can be set manually by enumerating of the elements the list in square brackets, like here:

```
Primes = [2, 3, 5, 7, 11, 13]
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
```

The list Primes has 6 elements, namely: Primes[0] == 2, Primes[1] == 3, Primes[2] == 5, Primes[3] == 7, Primes[4] == 11, Primes[5] == 13. The list Rainbow has 7 elements, each of which is the string.

Like the characters in the string, the list elements can also have negative index, for example, Primes[-1] == 13, Primes[-6] == 2. The negative index means we start at the last element and go left when reading a list.

You can obtain the number of elements in a list with the function len (meaning length of the list), e.g. len(Primes) == 6.

Unlike strings, the elements of a list are changeable; they can be changed by assigning them new values.

Consider several ways of creating and reading lists. First of all, you can create an empty list (the list with no items, its length is 0), and you can add items to the end of your list using append. For example, suppose the program receives the number of elements in the list n, and then n elements of the list one by one each at the separate line. Here is an example of input data in this format:

```
a = [] # start an empty list
n = int(input('Enter No of Elements')) # read number of element in the list
for i in range(n):
    new_element = int(input('Enter Element :')) # read next element
    a.append(new_element) # add it to the list
# the last two lines could be replaced by
one: # a.append(int(input('Enter Element :'))))
print(a)
```

Output will be

```
Enter No of Elements:5
Enter Element :2
Enter Element :7
Enter Element :4
Enter Element :3
Enter Element :8
[2, 7, 4, 3, 8]
```

In the demonstrated example the empty list is created, then the number of elements is read, then you read the list items line by line and append to the end. The same thing can be done, saving the variable n:

Next method to creating and reading lists is, first, consider the size of the list and create a list from the desired number of elements, then loop through the variable i starting with number 0 and inside the loop read i-th element of the list:

```
a = [0] * int(input('Enter No of Elements :'))
for i in range(len(a)):
    a[i] = int(input('Enter Element : '))
print (a)
```

Output will be

```
Enter No of Elements :3
Enter Element : 2
Enter Element : 4
Enter Element : 3
[2, 4, 3]
```

You can print elements of a list a with print(a); this displays the list items surrounded by square brackets and separated by commands. In general, this is inconvenient; in common, you are about to print all the elements in one line or one item per line. Here are two examples of that, using other forms of loop:

```
a = [1, 2, 3, 4, 5]
for i in range(len(a)):
    print(a[i])
```

Here the index i is changed, then the element a[i] is displayed.

```
a = [1, 2, 3, 4, 5]
for elem in a:
    print(elem, end=' ')
```

In this example, the list items are displayed in one line separated by spaces, and it's not the index that is changed but rather the value of the variable itself (for example, in the loop for elem in ['red', 'green', 'blue'] variable elem will take the values 'red', 'green', 'blue' successively.

3.9 ILLUSTRATIVE PROGRAMS:

3.9.1 Square Roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it. For example, one way of computing square roots is **Newton's method**. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:

$$Y = \frac{x + \frac{a}{x}}{2}$$

For example, if a is 4 and x is 3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

The result is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

After a few more updates, the estimate is almost exact:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

When $y == x$, we can stop. Here is a program that starts with an initial estimate, $x = 0.5*a$, and improves it until it stops changing:

```
def square_root(a):
    x = .5*a
    while True:
        print(x)
        y = (x + a/x) / 2
        if y == x:
            break
    x = y
    print("Square Root of ", a, " is ", y)
square_root(25)
```

Output will be:

```
12.5
7.25
5.349137931034482
5.011394106532552
5.000012953048684
5.00000000016778
5.0
Square Root of 25 is 5.0
```

3.9.2 GCD

3.9.2.1 Euclidean algorithm

This algorithm is based on the fact that GCD of two numbers divides their difference as well. In this algorithm, we divide the greater by smaller and take the remainder. Now, divide the smaller by this remainder. Repeat until the remainder is 0.

For example, if we want to find the GCD of 108 and 30, we divide 108 by 30. The remainder is 18. Now, we divide 30 by 18 and the remainder is 12. Now, we divide 18 by 12 and the remainder is 6. Now, we divide 12 by 6 and the remainder is 0. Hence, 6 is the required GCD.

```
def computeGCD(x, y):
    while(y):
        x, y = y, x % y
    return x

#can also be written as follows
#remainder=x%y
# x=y
#y=remainder

a=108
b=30
print('GCD of ',a,' and ', b, ' is ',computeGCD(a,b))
```

Output will be:

```
GCD of 108 and 30 is 6
```

Here we loop until y becomes zero. The statement $x, y = y, x \% y$ does swapping of values in Python. In each iteration, we place the value of y in x and the remainder ($x \% y$) in y , simultaneously. When y becomes zero, we have GCD in x .

3.9.3 Exponentiation

Simplest way to find the exponentiation of a number x^y in python is, the use of `**` operator

```
>>> 4**3
64
```

Another way to find the exponentiation of a number x^y in python is, the use of `pow()` function that is available in `math` module. eg

```
>>> import math
>>> math.pow(4,3)
64.0
```

We can also write our own function to find the exponentiation of a number x^y using looping statement.

```
def my_pow(x,y):
    powered = x
    if y == 0:
        powered=1
    else:
        while y > 1:
            powered *= x
            y -= 1
```

```

return powered
a=4
b=3
print(a, 'Power', b, 'is', my_pow(a,b))
Output will be:
4 Power 3 is 64

```

3.9.4 Sum of a List (Array) of Numbers

Suppose that you want to calculate the sum of a list of numbers such as: [1, 3, 5, 7, 9]. An iterative function that computes the sum is shown in **Iterative Summation program**. The function uses an accumulator variable (*Sum*) to compute a running total of all the numbers in the list by starting with 00 and adding each number in the list.

Iterative Summation:

```

def listsum(numList):
    Sum = 0
    for i in numList:
        Sum = Sum + i
    return Sum

my_list=[1,3,5,7,9]
print(listsum(my_list))
Output will be:
25

```

Pretend for a minute that you do not have *while* loops or *for* loops. How would you compute the sum of a list of numbers? If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers. To redefine the problem from adding a list to adding pairs of numbers, we could rewrite the list as a fully parenthesized expression. Such an expression looks like this:

$((((1+3)+5)+7)+9)$

We can also parenthesize the expression the other way around,

$(1+(3+(5+(7+9))))$

Notice that the innermost set of parentheses, (7+9), is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

$total = (1+(3+(5+(7+9))))$

$total = (1+(3+(5+16)))$

$total = (1+(3+21))$

$total = (1+24)$

$total = 25$

How can we take this idea and turn it into a Python program? First, let's restate the sum problem in terms of Python lists. We might say the sum of the list *numList* is

the sum of the first element of the list (*numList[0]*), and the sum of the numbers in the rest of the list (*numList[1:]*). To state it in a functional form:

$listSum(numList) = first(numList) + listSum(rest(numList))$

In this equation *first(numList)* returns the first element of the list and *rest(numList)* returns a list of everything but the first element. This is easily expressed in Python as shown in **Recursive Summation program**.

Recursive Summation:

```

def listsum(numList):
    if len(numList) == 1:
        return numList[0]
    else:
        return numList[0] + listsum(numList[1:])
my_list=[1,3,5,7,9]
print(listsum(my_list))

```

Output will be:
25

There are a few key ideas in this listing to look at. First, on **line 2** we are checking to see if the list is one element long. This check is crucial and is our escape (base case) clause from the function. The sum of a list of length 1 is trivial; it is just the number in the list. Second, on line 5 our function calls itself! This is the reason that we call the *listsum* algorithm recursive. A recursive function is a function that calls itself.

Figure 3.9 shows the series of **recursive calls** that are needed to sum the list [1,3,5,7,9]. You should think of this series of calls as a series of simplifications. Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller.

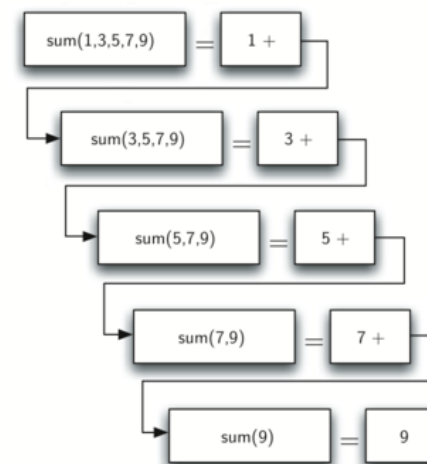


Figure 3.9 Recursive Calls in List Sum

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. **Figure 3.10** shows the additions that are performed as *listsum* works its way backward through the series of calls. When *listsum* returns from the topmost problem, we have the solution to the whole problem.

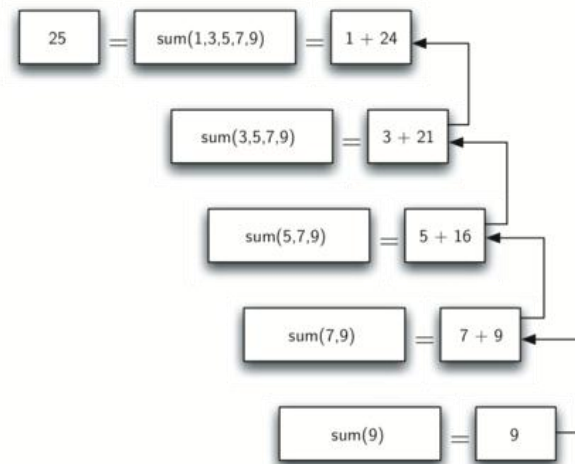


Figure 3.10 Recursive Return for List Sum

3.9.5 Linear Search

Given an array `arr[]` of `n` elements, write a function to search a given element `x` in `arr[]`. A simple approach is to do **linear search**, i.e

- Start from the leftmost element of `arr[]` and one by one compare `x` with each element of `arr[]`
- If `x` matches with an element, return the index.
- If `x` doesn't match with any of elements, return -1.

Example:

56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9
56	3	249	518	7	26	94	651	23	9

Figure 3.11 Linear Search : Searching for 9 in 10 Element Array.

```

def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

data = []
n = int(input('Enter Number of Elements in the Array: '))
for i in range(0, n):
    data.append(input('Enter the Element :'))
x = input('Enter the Element to be Search ')
found=linear_search(data,x)
if found!=-1:
    print('Element ', x, ' Found at Position ',found+1)
else:
    print('Element ',x, ' is Not Found in the Array ')
  
```

Output will be:

```

Enter Number of Elements in the Array: 5
Enter the Element : 2
Enter the Element : 7
Enter the Element : 4
Enter the Element : 9
Enter the Element : 1
Enter the Element to be Search 9
Element 9 Found at Position 4
  
```

The time complexity of above algorithm is $O(n)$.

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

3.9.6 Binary Search

Given a sorted array `arr[]` of `n` elements, write a function to search a given element `x` in `arr[]`. A simple approach is to do *linear search*. The time complexity of above algorithm is $O(n)$. Another approach to perform the same task is using Binary Search.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example:

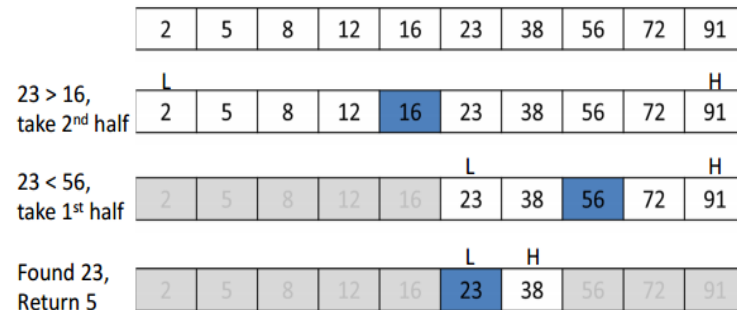


Figure 3.12 Binary Search: Searching for 23 in 10 Element Array.

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$. We basically ignore half of the elements just after one comparison.

1. Compare `x` with the middle element.
2. If `x` matches with middle element, we return the mid index.
3. Else If `x` is greater than the mid element, then `x` can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (`x` is smaller) recur for the left half.

Recursive implementation of Binary Search

```
def binarySearch(arr, left, right, x):
    if right >= left:
        mid = left + (right - left) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binarySearch(arr, left, mid - 1, x)
        else:
            return binarySearch(arr, mid + 1, right, x)
    else:
        return -1

data = []
n = int(input('Enter Number of Elements in the Array: '))
for i in range(0, n):
    data.append(input('Enter the Element :'))
```

```
x = input('Enter the Element to be Search ')
result = binarySearch(data, 0, len(data), x)
if result != -1:
    print("Element is present at index ", result+1)
else:
    print("Element is not present in array")
```

Output will be:

```
Enter Number of Elements in the Array: 5
Enter the Element :2
Enter the Element :7
Enter the Element :9
Enter the Element :11
Enter the Element :14
Enter the Element to be Search 11
Element is present at index 4
```

Iterative Implementation of Binary Search

```
def binarySearch(arr, left, right, x):
    while left <= right:
        mid = left + (right - left) // 2;
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1

data = []
n = int(input('Enter Number of Elements in the Array: '))
for i in range(0, n):
    data.append(input('Enter the Element :'))
x = input('Enter the Element to be Search ')
result = binarySearch(data, 0, len(data), x)
if result != -1:
    print("Element is present at index ", result+1)
else:
    print("Element is not present in array")
```

Auxiliary Space: $O(1)$ in case of iterative implementation. In case of recursive implementation, $O(\log n)$ recursion call stack space.

GE8151 - PROBLEM SOLVING AND PYTHON
PROGRAMMING

REGULATIONS – 2017

UNIT – IV

Prepared By :

Mr. Vinu S, ME,
Assistant Professor,
St. Joseph's College of Engineering,
Chennai -600119.

UNIT IV

LISTS, TUPLES, DICTIONARIES

In this Unit we will discuss about Python's built-in compound types like Lists, Tuples, Dictionaries and their use.

4.1 LISTS

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes **items**. There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]): It can have any number of items and they may be of different types (integer, float, string etc.).

Syntax

[]	# empty list
[1, 2, 3]	# list of integers
['physics', 'chemistry', 'computer']	# list of strings
[1, "Hello", 3.4]	# list with mixed datatypes

Also, a list can even have another list as an item. This is called **nested list**.

my_list = ["mouse", [8, 4, 6], ['a']]	# nested list
---------------------------------------	---------------

As you might expect, you can assign list values to variables:

```
>>>subject= ['physics', 'chemistry','computer']
>>>mark=[98,87,94]
>>>empty=[]
>>> print(subject,mark,empty)
['physics', 'chemistry', 'computer'], [98, 87, 94], []
```

4.1.1 Lists are Mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> subject[0]
'physics'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>>mark=[98,87,94]
>>> mark[2]=100
>>> mark
[98, 87, 100]
```

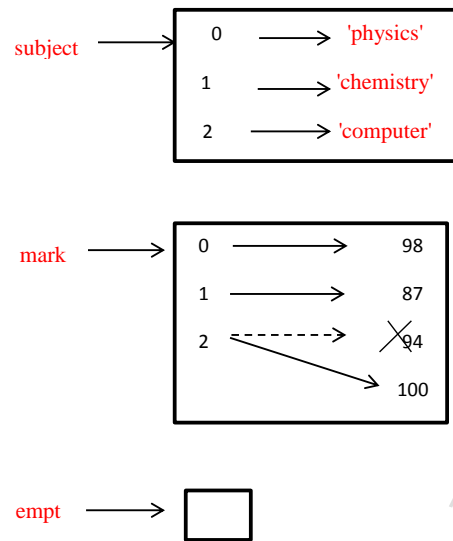


Figure 4.1 State diagrams

Figure 4.1 shows the state diagram for the lists `subject`, `mark`, and `empty`. 2nd element of `mark`, which used to be 94, is now 100

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The **in operator** also works on lists.

```
>>> subject= ['physics', 'chemistry', 'computer']
>>> 'chemistry' in subject
True
>>> 'english' in subject
False
```

4.1.2 Traversing a List, List Loop

The most common way to traverse the elements of a list is with a `for` loop.

The Syntax is :

for VARIABLE in LIST:

STATEMENT(S) USING VARIABLE

Elements in the List are stores in VARIABLE one by one for each iteration.

WE can use the element stored in VARIABLE for further processing inside the loop body

Example:

```
>>> for s in subject:
    print(s)

physics
chemistry
computer
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the built-in functions `range` and `len`:

The Syntax is :

for ITERATION_VARIABLE in range(len(LIST_NAME)):

STATEMENT(S) Using LIST_NAME[ITERATION_VARIABLE]

ITERATION_VARIABLE holds value from 0 to length of LIST_NAME one by one.

Using that ITERATION_VARIABLE we can access individual element of LIST_NAME for reading and writing

```
>>> for i in range(len(mark)):
    mark[i] = mark[i] * 2
>>> mark
[196, 174, 200]
```

This loop traverses the list and updates each element. `len` returns the 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 the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A `for` loop over an empty list never runs the body:

```
for x in []:
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of the following list is 3:

```
>>> my_list = ["mouse", [8, 4, 6], ['a']]
>>> len(my_list)
3
```

4.1.3 List Operations

The `+` operator concatenates lists:

```
>>> first=[100,200,300]
>>> second=[55,65]
>>> third=first+second
>>> third
[100, 200, 300, 55, 65]
```

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

```
>>> [5]*3
[5, 5, 5]
>>> [55,65]*3
[55, 65, 55, 65, 55, 65]
```

4.1.4 List Slices

The slice operator also works on lists:

```
>>> w=['w','e','t','c','o','m','e']
>>> w[2:5]
['t', 'c', 'o']
>>> w[:3 ]
['w', 'e', 't']
```

```
['w', 'e', 'l']
>>> w[5:]
['m', 'e']
>>> w[:]
['w', 'e', 'l', 'c', 'o', 'm', 'e']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

4.1.5 List Methods

Python includes following list methods.

SN	Methods with Description	Example
1	list.append(obj) Appends object obj to list	<pre>>>> t = ['a', 'b', 'c'] >>> t.append('d') >>> t ['a', 'b', 'c', 'd']</pre>
2	list.count(obj) Returns count of how many times obj occurs in list	<pre>>>> aList = [123, 'xyz', 'zara', 'abc', 123] >>> aList.count(123) 2 >>> aList.count('xyz') 1</pre>
3	list.extend(seq) Appends the contents of seq to list	<pre>>>> t1 = ['a', 'b', 'c'] >>> t2 = ['d', 'e'] >>> t1.extend(t2) >>> t1 ['a', 'b', 'c', 'd', 'e']</pre> <p>This example leaves t2 unmodified.</p>
4	list.index(obj) Returns the lowest index in list that obj appears	<pre>aList = [123, 'xyz', 'zara', 'abc', 123] >>> aList.index(123) 0 >>> aList.index('xyz') 1</pre>
5	list.insert(index, obj) Inserts object obj into list at offset index	<pre>>>> aList.insert(3, 2009) >>> aList [123, 'xyz', 'zara', 2009, 'abc', 123]</pre>
6	list.pop(obj=list[-1]) Removes and returns last object or obj from list	<pre>>>> aList = [123, 'xyz', 'zara', 'abc'] >>> aList.pop() 'abc' >>> aList.pop(2) 'zara' >>> aList [123, 'xyz']</pre>
7	list.remove(obj)	<pre>>>> aList = [123, 'xyz', 'zara', 'abc'] >>> aList.remove('xyz') >>> aList</pre>

	Removes object obj from list	[123, 'zara', 'abc']
8	list.reverse() Reverses objects of list in place	<pre>>>> aList = [123, 'xyz', 'zara', 'abc', 'xyz'] >>> aList.reverse() >>> aList ['xyz', 'abc', 'zara', 'xyz', 123]</pre>
9	list.sort([func]) Sorts objects of list, use compare func if given	<pre>>>> t = ['d', 'c', 'e', 'b', 'a'] >>> t.sort() >>> t ['a', 'b', 'c', 'd', 'e']</pre>

4.1.6 Deleting Elements

There are several ways to delete elements from a list. If you know the index of the element you want to delete, you can use **pop**:

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

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element. If you don't need the removed value, you can use the del operator:

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

If you know the element you want to remove (but not the index), you can use remove:

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

The return value from remove is None. To remove more than one element, you can use del with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

As usual, the slice selects all the elements up to but not including the second index.

4.1.7 Aliasing

If one is a refers to an object and you assign two = one, then both variables refer to the same object:

```
>>> one = [10, 20, 30]
>>> two = one
```

```
>>> two is one
True
```

The state diagram looks like **Figure 4.2**

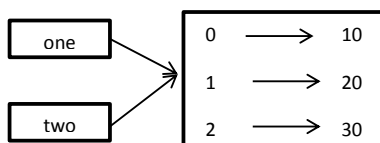


Figure 4.2 State diagrams

```
>>> two[1]=40
>>> one
[10, 40, 30]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

4.1.8 Cloning Lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator. Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.

Example.

```
a = [81, 82, 83]
b = a[:] # make a clone using slice
print(a == b)
print(a is b)
b[0] = 5
print(a)
print(b)
```

Output will be

```
True
False
[81, 82, 83]
[5, 82, 83]
```

Now we are free to make changes to b without worrying about a. Again, we can clearly see that a and b are entirely different list objects.

But the problem is if the list is a nested one this method won't work.

Example

```
a = [81, 82, [83,84]]
```

```
b = a[:] # make a clone using slice
print(a == b)
print(a is b)
b[2][0] = 5
print(a)
print(b)
```

Output will be:

```
True
False
[81, 82, [5, 84]]
[81, 82, [5, 84]]
```

Change in list b affect list a. In order to overcome this issue python provide a module called copy. This module provides **generic shallow and deep copy operations**.

4.1.8.1 Copy Module

Interface summary:

```
copy.copy(x)
    Return a shallow copy of x.
copy.deepcopy(x)
    Return a deep copy of x.
```

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Shallow copy Example:

```
import copy
a = [81, 82, [83,84]]
b = copy.copy(a) # make a clone using shallow copy()
print(a == b)
print(a is b)
b[1]=10
b[2][0] = 5
print(a)
print(b)
```

Output will be:

```
True
False
```

```
[81, 82, [5, 84]]
[81, 10, [5, 84]]
```

Deep copy Example:

```
import copy
a = [81, 82, [83, 84]]
b = copy.deepcopy(a)          # make a clone using deepcopy()
print(a == b)
print(a is b)
b[1] = 10
b[2][0] = 5
print(a)
print(b)
```

Output will be:

```
True
False
[81, 82, [83, 84]]
[81, 10, [5, 84]]
```

We can notice the difference between shallow copy and deep copy from the above examples. In shallow copy method change made in nested list b affect list a also it is same as copying using slice operator. But in deep copy method changes made in nested list b does not affect list a.

4.1.9 List Parameters

When you 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:

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

Here's how it is used:

```
>>> letters = ['x', 'y', 'z']
>>> delete_head(letters)
>>> letters
['y', 'z']
```

The parameter `t` and the variable `letters` are aliases for the same object. 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. Here's an example using `append`:

```
>>> t1 = [10, 20]
>>> t2 = t1.append(30)
>>> t1
[10, 20, 30]
>>> t2
None
```

The return value from `append` is `None`. Here's an example using the `+` operator:

```
>>> t3 = t1 + [40]
>>> t1
[10, 20, 30]
>>> t3
[10, 20, 30, 40]
```

The result of the operator is a new list, and the original list is unchanged. This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]                # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but that doesn't affect the caller.

```
>>> t4 = [10, 20, 30]
>>> bad_delete_head(t4)
>>> t4
[10, 20, 30]
```

At the beginning of `bad_delete_head`, `t` and `t4` refer to the same list. At the end, `t` refers to a new list, but `t4` still refers to the original, unmodified list. An alternative is to write a function that creates and returns a new list. For example, `tail` returns all except first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['x', 'y', 'z']
>>> rest = tail(letters)
>>> rest
['y', 'z']
```

4.2 TUPLES

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

4.2.1 Creating Tuples

Similar to List, Tuple is a sequence of values. The values can be any type, and they are indexed by integers. The important difference is that tuples are immutable. Syntactically, a tuple is a comma-separated list of values:

```
>>> message = 'h', 'a', 'i'
>>> type(message)
<type 'tuple'>
```

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

```
>>> message = ('h', 'a', 'i')
>>> type(message)
<type 'tuple'>
```

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

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

A value in parentheses is not a tuple:

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

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

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

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

```
>>> course=tuple('python')
>>> course
('p', 'y', 't', 'h', 'o', 'n')
```

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

```
>>> course = ('p', 'y', 't', 'h', 'o', 'n')
>>> course[0]
'p'
```

And the slice operator selects a range of elements.

```
>>> course[3:5]
('h', 'o')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> course[0]='P'
```

TypeError: 'tuple' object does not support item assignment

Because tuples are immutable, you can't modify the elements. But you can replace one tuple with another:

```
>>> course=('P',)+course[1:]
>>> course
('P', 'y', 't', 'h', 'o', 'n')
```

This statement makes a new tuple and then makes course 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.

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

4.2.2 Tuple Assignment

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

```
>>> 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, you could write:

```
>>> email='hodcse@stjosephs.ac.in'
>>> username, domain=email.split('@')
```

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

```
>>> username
'hodcse'
>>> domain
'stjosephs.ac.in'
```

4.2.3 Tuples as return Values

Strictly speaking, 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 you 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. You can store the result as a tuple:

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

Or use tuple assignment to store the elements separately:

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

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.

```
>>> quot
2
>>> rem
1
```

4.2.4 Variable-Length Argument Tuples

Functions can take a variable number of arguments. A parameter name that begins with ***** **gathers** arguments into a tuple. For example, `displayall` takes any number of arguments and prints them:

```
>>> def displayall(*args):
    print(args)
```

The gather parameter can have any name you like, but `args` is conventional. Here's how the function works:

```
>>> displayall('python',355.50,3)
('python', 355.5, 3)
```

The complement of gather is **scatter**. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the ***** operator. For example, `divmod` takes exactly two arguments; it doesn't work with a tuple:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
But if you scatter the tuple, it works:
>>> divmod(*t)
(2, 1)
```

Many of the built-in functions use variable-length argument tuples. For example, `max` and `min` can take any number of arguments:

```
>>> max(55,72,38)
72
```

But `sum` does not.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

4.2.5 Lists and Tuples

zip is a built-in function that takes two or more sequences and returns a list of tuples where each tuple contains one element from each sequence. The name of the function refers to a zipper, which joins and interleaves two rows of teeth.

This example zips a string and a list:

```
>>> s='hai'
>>> t=[0,1,2]
>>> zip(s,t)
[('h', 0), ('a', 1), ('i', 2)]
```

The result is a **zip object** that knows how to iterate through the pairs. The most common use of `zip` is in a for loop:

```
>>> for pair in zip(s, t):
    print(pair)
```

```
('h', 0)
```

```
('a', 1)
('i', 2)
```

A `zip` object is a kind of **iterator**, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.

If you want to use list operators and methods, you can use a `zip` object to make a list:

```
>>> list(zip(s, t))
[('h', 0), ('a', 1), ('i', 2)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result has the length of the shorter one.

```
>>> list(zip('Vinu', 'Ranjith'))
[('V', 'R'), ('i', 'a'), ('n', 'n'), ('u', 'j')]
```

You can use tuple assignment in a for loop to traverse a list of tuples:

```
>>> t=[('h',0),('a',1),('i',2)]
>>> for letter,number in t:
    print(number,letter)
```

```
(0, 'h')
(1, 'a')
(2, 'i')
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to `letter` and `number`.

If you combine `zip`, `for` and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, `has_match` takes two sequences, `t1` and `t2`, and returns `True` if there is an index `i` such that `t1[i] == t2[i]`:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function `enumerate`:

```
>>> for index, element in enumerate('hai'):
    print(index, element)
(0, 'h')
(1, 'a')
(2, 'i')
```

The result from `enumerate` is an `enumerate` object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence.

4.3 DICTIONARIES

Dictionaries are one of Python's best features; they are the building blocks of many efficient and elegant algorithms.

4.3.1 A Dictionary is a Mapping

A **dictionary** is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

A dictionary contains a collection of indices, which are called **keys**, and a collection of values. Each key is associated with a single value. The association of a key and a value is called a **key-value** pair or sometimes an **item**.

In mathematical language, a dictionary represents a **mapping** from keys to values, so you can also say that each key “maps to” a value. As an example, we’ll build a dictionary that maps from English to Tamil words, so the keys and the values are all strings.

The **function dict** creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

4.3.2 Creates Dictionary

```
>>> eng_tam=dict()
>>> eng_tam
{}

```

The squiggly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng_tam['two']='irantu'

```

This line creates an item that maps from the key 'two' to the value 'irantu'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> eng_tam
{'two': 'irantu'}

```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng_tam={'two':'irantu','three':'munru','four':'nanku'}

```

But if you print eng_tam, you might be surprised:

```
>>> eng_tam
{'four': 'nanku', 'two': 'irantu', 'three': 'munru'}

```

The order of the key-value pairs might not be the same. If you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that’s not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> eng_tam['two']
'irantu'

```

The key 'two' always maps to the value 'irantu' so the order of the items doesn’t matter. If the key isn’t in the dictionary, you get an exception:

```
>>> eng_tam['five']
KeyError: 'five'

```

The len function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng_tam)
3

```

The in operator works on dictionaries, too; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).

```
>>> 'two' in eng_tam
True
>>> 'irantu' in eng_tam
False

```

To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

```
>>> meaning = eng_tam.values()
>>> 'irantu' in meaning
True

```

The in operator uses different algorithms for lists and dictionaries. For lists, it searches the elements of the list in order. As the list gets longer, the search time gets longer in direct proportion. For dictionaries, Python uses an algorithm called a **hashtable** that has a remarkable property: the in operator takes about the same amount of time no matter how many items are in the dictionary.

4.3.3 Dictionary as a Collection of Counters

Suppose you are given a string and you want to count how many times each letter appears.

There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don’t have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
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:

```
>>> h=histogram('python programming')

```

```
>>> h
{'a': 1, ' ': 1, 'g': 2, 'i': 1, 'h': 1, 'm': 3, 'o': 2, 'n': 1, 'p': 2, 'r': 2, 't': 1, 'y': 1}
```

The histogram indicates that the letters 'a' and ' ' appear once; 'g' 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
>>>
```

4.3.4 Looping and Dictionaries

If you 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('programming')
>>> print_hist(h)
('a', 1)
('g', 2)
('i', 1)
('m', 2)
('o', 1)
('n', 1)
('p', 1)
('r', 2)
```

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

```
>>> for key in sorted(h):
        print(key, h[key])

('a', 1)
('g', 2)
('i', 1)
('m', 2)
('n', 1)
('o', 1)
('p', 1)
('r', 2)
```

4.3.5 Reverse Lookup

Given a dictionary `d` and a key `k`, it is easy to find the corresponding value `v = d[k]`. This operation is called a **lookup**.

But what if you have `v` and you want to find `k`? You have two problems: first, there might be more than one key that maps to the value `v`. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search.

Here is a function that takes a value and returns the first key that maps to that value:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, `raise`. The **raise statement causes an exception**; in this case it causes a `LookupError`, which is a built-in exception used to indicate that a lookup operation failed.

If we get to the end of the loop, that means `v` doesn't appear in the dictionary as a value, so we raise an exception.

Here is an example of a successful reverse lookup:

```
>>> h = histogram('programming')
>>> key = reverse_lookup(h, 2)
>>> key
'g'
```

And an unsuccessful one:

```
>>> key = reverse_lookup(h, 3)
```

Traceback (most recent call last):

```
File "<pyshell#38>", line 1, in <module>
    key = reverse_lookup(h, 3)
File "G:\class\python\code\dictionary.py", line 22, in reverse_lookup
    raise LookupError()
LookupError
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message. The `raise` statement can take a detailed error message as an optional argument. For example:

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

4.3.6 Dictionaries and Lists

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the

same frequency, each value in the inverted dictionary should be a list of letters. Here is a function that inverts a dictionary:

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

Each time through the loop, key gets a key from d and val gets the corresponding value. If val is not in inverse, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> hist = histogram('programming')
>>> hist
{'a': 1, 'g': 2, 'i': 1, 'm': 2, 'o': 1, 'n': 1, 'p': 1, 'r': 2}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'i', 'o', 'n', 'p'], 2: ['g', 'm', 'r']}
```

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

4.3.7 Advanced List Processing - List Comprehension

Python supports a concept called "**list comprehensions**". It can be used to construct lists in a very natural, easy way, like a mathematician is used to do. Following function 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.

4.4 ILLUSTRATIVE PROGRAMS

4.4.1 Sorting

It is an operation in which all the elements of a list are arranged in a predetermined order. The elements can be arranged in a sequence from smallest to largest such that every element is less than or equal to its next neighbour in the list. Such an arrangement is called ascending order. Assuming an array or List containing N elements, the ascending order can be defined by the following relation:

$$\text{List}[i] \leq \text{List}[i+1], 0 \leq i < N-1$$

Similarly in descending order, the elements are arranged in a sequence from largest to smallest such that every element is greater than or equal to its next neighbor in the list. The descending order can be defined by the following relation:

$$\text{List}[i] \geq \text{List}[i+1], 0 \leq i < N-1$$

It has been estimated that in a data processing environment, 25 per cent of the time is consumed in sorting of data. Many sorting algorithms have been developed. Some of the most popular sorting algorithms that can be applied to arrays are in-place sort algorithm. An **in-place** algorithm is generally a comparison-based algorithm that stores the sorted elements of the list in the same array as occupied by the original one. A detailed discussion on sorting algorithms is given in subsequent sections.

4.4.2 Selection Sort

It is a very simple and natural way of sorting a list. It finds the smallest element in the list and exchanges it with the element present at the head of the list as shows in

Figure 4.3

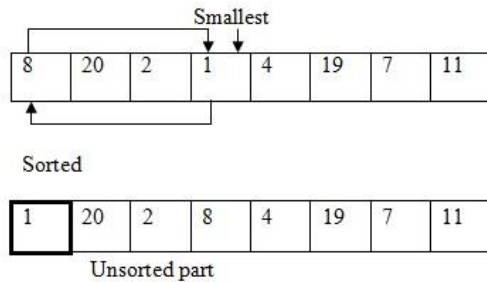


Figure 4.3 Selection sort (first pass)

It may be noted from **Figure 4.3** that initially, whole of the list was unsorted. After the exchange of the smallest with the element on the list, the list is divided into two parts: sorted and unsorted.

Now the smallest is searched in the unsorted part of the list, i.e., '2' and exchange with the element at the head of unsorted part, i.e., '20' as shown in **Figure 4.4**.

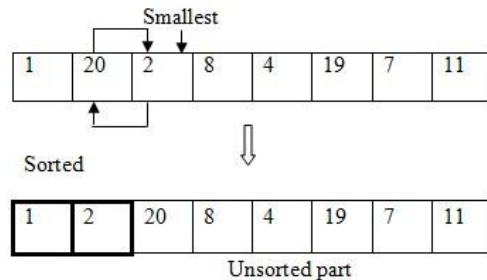


Figure 4.4 Selection sort (second pass)

This process of selection and exchange (i.e., a pass) continues in this fashion until all the elements in the list are sorted (see **Figure 4.5**). Thus, in selection sort, two steps are important – **selection and exchange**.

From **Figure 4.4** and **Figure 4.5**, it may be observed that it is a case of nested loops. The outer loop is required for passes over the list and inner loop for searching smallest element within the unsorted part of the list. In fact, for N number of elements, $N-1$ passes are made.

An algorithm for selection sort is given below. In this algorithm, the elements of a list stored in an array called $LIST[]$ are sorted in ascending order. Two variables called $Small$ and Pos are used to locate the smallest element in the unsorted part of the list. $Temp$ is the variable used to interchange the selected element with the first element of the unsorted part of the list.

Algorithm SelectionSort()

```

1. For I= 1 to N-1                                #Outer Loop
    1.1 small = List[I]
    1.2 Pos = I
    1.3 For J=I+1 to N                              # Inner Loop
        1.3.1 if (List[J] < small)
            1.3.1.1 small = List[J]
            1.3.1.2 Pos = J                        #Note the position of the smallest
    1.4 Temp= List[I]                                #Exchange Smallest with the
    Head
    1.5 List[I] = List [Pos]
    1.6 List [Pos] = Temp

2. Print the sorted list

```

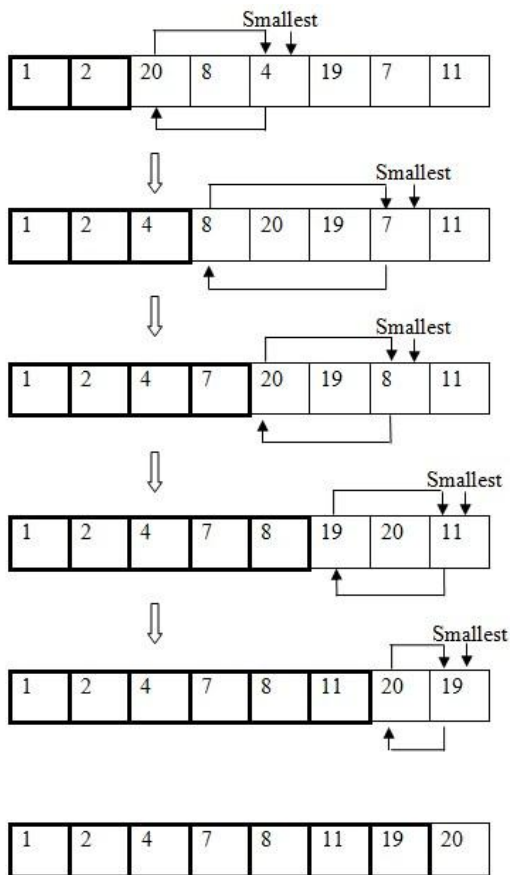


Figure 4.5 Selection sort

Program to sort a List in ascending order using Selection Sort

```
data = []
print('Selection Sort :')
n = int(raw_input('Enter Number of Elements in the Array: '))
for i in range(0, n):
    x = raw_input('Enter the Element %d :'%(i+1))
    data.append(x)
print('Original Array :')
print(data)
print('Intermediate Steps :')
for i in range(0, n-1):
    small=int(data[i])
    pos=i
```

```
for j in range(i+1,n):
    if int(data[j])<small:
        small=int(data[j])
        pos=j
    temp=data[i]
    data[i]=data[pos]
    data[pos]=temp
    print(data)
print('Sorted Array :')
print(data)
```

Output

Selection Sort :

Enter Number of Elements in the Array: 5

Enter the Element 1 :4

Enter the Element 2 :3

Enter the Element 3 :6

Enter the Element 4 :8

Enter the Element 5 :1

Original Array :

[4, '3', '6', '8', '1']

Intermediate Steps :

[1, '3', '6', '8', '4']

[1, '3', '6', '8', '4']

[1, '3', '4', '8', '6']

[1, '3', '4', '6', '8']

Sorted Array :

[1, '3', '4', '6', '8']

4.4.3 Insertion Sort

This algorithm mimics the process of arranging a pack of playing cards; the first two cards are put in correct relative order. The third is inserted at correct place relative to the first two. The fourth card is inserted at the correct place relative to the first three, and so on.

Given a list of numbers, it divides the list into two part – sorted part and unsorted part. The first element becomes the sorted part and the rest of the list becomes the unsorted part as Shown in Figure 4.6. It picks up one element from the front of the unsorted part as shown in Figure 4.6. It picks up one element from the front of the unsorted part and inserts it in proper position in the sorted part of the list. The insertion action is repeated till the unsorted part is exhausted.

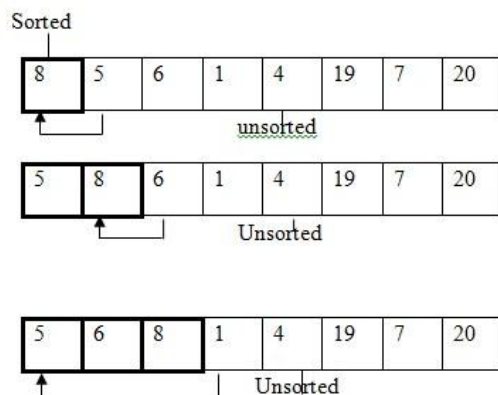


Figure 4.6 Insertion sort

It may be noted that the insertion operation requires following steps:

Step

1. Scan the sorted part to find the place where the element, from unsorted part, can be inserted. While scanning, shift the elements towards right to create space.
2. Insert the element, from unsorted part, into the created space.

This algorithm for the insertion sort is given below. In this algorithm, the elements of a list stored in an array called List[] are sorted in an ascending order. The algorithm uses two loops – the outer For loop and inner while loop. The inner while loop shifts the elements of the sorted part by one step to right so that proper place for incoming element is created. The outer For loop inserts the element from unsorted part into the created place and moves to next element of the unsorted part.

Algorithm insertSort()

1. For I = 2 to N #The first element becomes the sorted part

```
1.1 Temp = List[I] #Save the element from unsorted part into temp
1.2 J = I-1
1.3 While(Temp <= List[J] AND J >=0)
```

```
1.3.1 List[J+1] = List[J] #Shift elements towards right
```

```
1.3.2 J = J-1
```

```
1.4 List [J+1] = Temp
```

2. Print the list

3. Stop

Program to sort a List in ascending order using Selection Sort

```
data = []
print('Insertion Sort :')
n = int(raw_input('Enter Number of Elements in the Array: '))
for i in range(0, n):
    x = raw_input('Enter the Element %d :'%(i+1))
    data.append(x)
print('Original Array :')
print(data)
print('Intermediate Steps :')
for i in range(1,n):
    temp=int(data[i])
    j=i-1
    while temp<int(data[j]) and j>=0:
        data[j+1]=data[j]
        j=j-1
    data[j+1]=temp
    print(data)
print('Sorted Array is:')
print(data)
```

Output

```
Insertion Sort :
Enter Number of Elements in the Array: 5
Enter the Element 1 :3
Enter the Element 2 :5
Enter the Element 3 :2
Enter the Element 4 :8
Enter the Element 5 :1
Original Array :
['3', '5', '2', '8', '1']
Intermediate Steps :
['3', '5', '2', '8', '1']
['2', '3', '5', '8', '1']
['2', '3', '5', '8', '1']
['1', '2', '3', '5', '8']
Sorted Array is:
['1', '2', '3', '5', '8']
```

4.4.4 Merge Sort

This method uses following two concepts:

- If a list is empty or it contains only one element, then the list is already sorted. A list that contains only one element is also called **singleton**.

- It uses old proven technique of ‘divide and conquer’ to recursively divide the list into sub-lists until it is left with either empty or singleton lists.

In fact, this algorithm divides a given list into two almost equal sub-lists. Each sub-list, thus obtained, is recursively divided into further two sub-lists and so on till singletons or empty lists are left as shown in **Figure 4.7**.

Since the singleton and empty list are inherently sorted, the only step left is to merge the singletons into sub-lists containing two elements each (see **figure 4.7**) which are further merged into sub-lists containing four elements each and so on. This merging operation is recursively carried out till a final merged list is obtained as shown in **figure 4.8**.

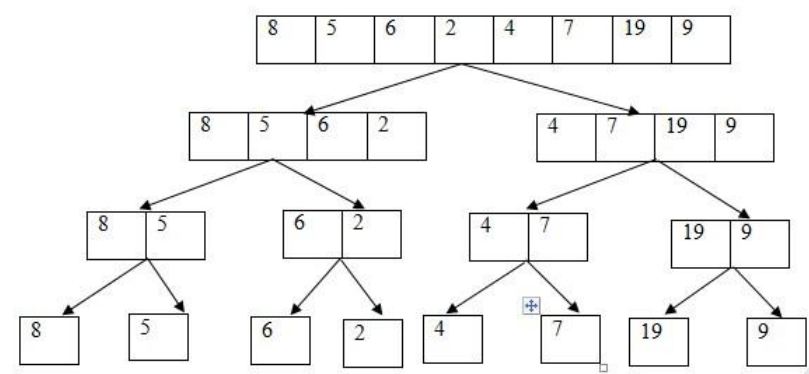


Figure 4.7 First step of merge sort (divide)

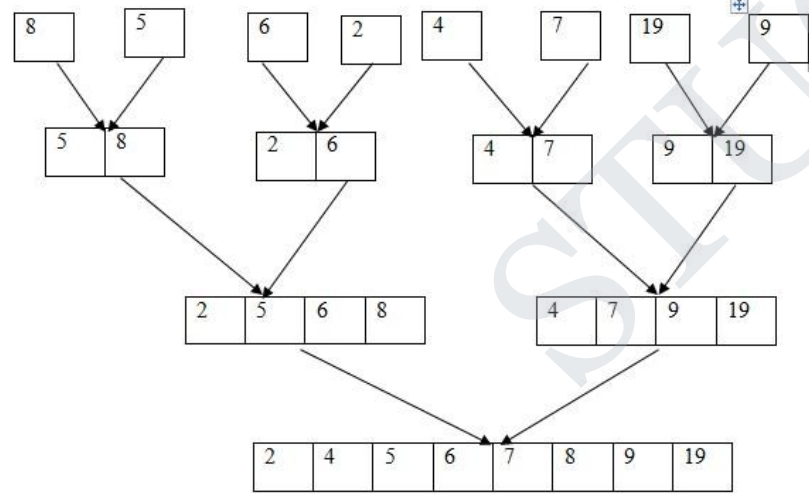


Figure 4.8 Second step of merge sort (merge)

Note: The merge operation is a time consuming and slow operation. The working of merge operation is discussed in the next section.

Merging of lists It is an operation in which two ordered lists are merged into a single ordered list. The merging of two lists PAR1 and PAR2 can be done by examining the elements at the head of two lists and selecting the smaller of the two. The smaller element is then stored into a third list called mergeList. For example, consider the lists PAR1 and PAR2 given the **figure 4.9**. Let Ptr1, Ptr2, and Ptr3 variables point to the first locations of lists PAR1, PAR2 and PAR3, respectively. The comparison of PAR1[Ptr1] and PAR2[Ptr2] shows that the element of PAER1 (i.e., ‘2’) is smaller. Thus, this element will be placed in the mergeList as per the following operation:

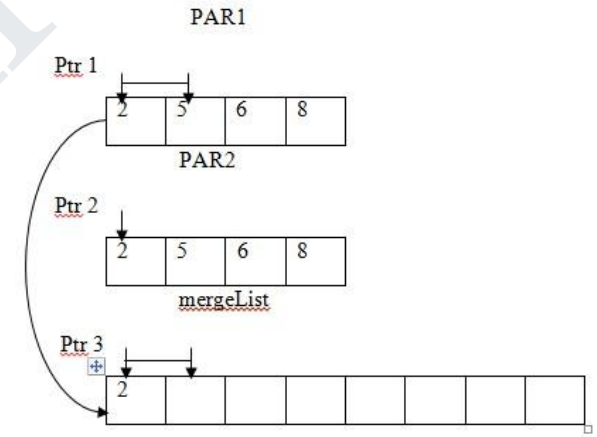


Figure 4.9 Merging of lists (first step)

```
mergeList[Ptr3] S= PAR1[Ptr1];  
Ptr1++;  
Ptr3++;
```

Since an element from the list *PAR1* has been taken to *mergeList*, the variable *Ptr1* is accordingly incremented to point to the next location in the list. The variable *Ptr3* is also incremented to point to next vacant location in *mergeList*.

This process of comparing, storing and shifting is repeated till both the lists are merged and stored in *mergeList* as shown in **figure 4.10**.

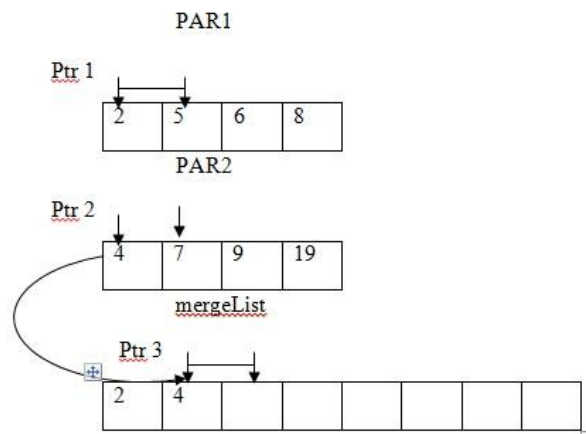


Figure 4.10 Merging of lists (second step)

It may be noted here that during this merging process, a situation may arise when we run out of elements in one of the lists. We must, therefore, stop the merging process and copy rest of the elements from unfinished list into the final list.

The algorithm for merging of lists is given below. In this algorithm, the two sub-lists are part of the same array *List[]*. The first sub-list is located in *List[lb]* to *List[mid]* and the second sub-list is stored in locations *List[mid+1]* to *List[ub]* where *lb* and *ub* mean lower and upper bounds of the array, respectively.

Algorithm merge (List, lb, mid, ub)

```

1. ptr1 = lb           # index of first list
2. ptr2 = mid          # index of second list
3. ptr3 = lb           # index of merged list
4. while ((ptr1 < mid) && ptr2 <= ub)  #merge the lists
    4.1 if (List[ptr1] <= List[ptr2])
        4.1.1 mergeList[ptr3] = List[ptr1]  #element from firstlist is taken
        4.1.2 ptr1++                        #move to next element in the list
        4.1.3 ptr3++
    4.2 else
        4.2.1 mergeList[ptr3] = List[ptr2]  #element from second list is taken
        4.2.2 ptr2++                        #move to next element in the list
        4.2.3 ptr3++
5. while(ptr1 < mid)    #copy remaining first list
    5.1 mergeList [ptr3] = List[ptr1]
```

```

5.2 ptr1++
5.3 ptr3++
}
6. while (ptr2 <= ub)           #copy remaining second list
    6.1 mergeList [ptr3] = List[ptr2]
    6.2 ptr2++
    6.3 ptr3++
7. for(i=lb; i<ptr3; i++)      #copy merged list back into original list
    7.1 List[i] = mergeList[i]
8. Stop
```

It may be noted that an extra temporary array called *mergedList* is required to store the intermediate merged sub-lists. The contents of the *mergeList* are finally copied back into the original list.

The algorithm for the merge sort is given below. In this algorithm, the elements of a list stored in an array called *List[]* are sorted in an ascending order. The algorithm has two parts- *mergeSort* and *merge*. The *merge* algorithm, given above, merges two given sorted lists into a third list, which is also sorted. The *mergeSort* algorithm takes a list and stores into an array called *List[]*. It uses two variables *lb* and *ub* to keep track of lower and upper bounds of list or sub-lists as the case may be. It recursively divides the list into almost equal parts till singleton or empty lists are left. The sub-lists are recursively merged through *merge* algorithm to produce final sorted list.

Algorithm mergeSort (List, lb, ub)

```

1. if (lb<ub)
    1.1 mid = (lb+ub)/2      #divide the list into two sub-lists
    1.2 mergeSort(List, lb, mid)  #sort the left sub-list
    1.3 mergeSort(List, mid+1, ub) #sort the right sub-list
    1.4 merge(List, lb, mid+1, ub) #merge the lists
2. Stop
```

Program to sort a List in ascending order using Merge Sort

```

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 int(lefthalf[i]) < int(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)
data = []
print(Merge Sort :)
n = int(raw_input(Enter Number of Elements in the Array: ))
for i in range(0, n):
    x = raw_input(Enter the Element %d :'%(i+1))
    data.append(x)
print(Original Array :)
print(data)
print(Intermediate Steps :)
mergeSort(data)
print(Sorted Array is:)
print(data)

```

Output

```

Merge Sort :
Enter Number of Elements in the Array: 5
Enter the Element 1 :4
Enter the Element 2 :8
Enter the Element 3 :2
Enter the Element 4 :9
Enter the Element 5 :1
Original Array :
[4, '8', '2', '9', '1']
Intermediate Steps :
('Splitting ', [4, '8', '2', '9', '1'])
('Splitting ', [4, '8'])
('Splitting ', [4])
('Merging ', [4])
('Splitting ', [8])
('Merging ', [8])

```

```

('Merging ', [4, '8'])
('Splitting ', [2, '9', '1'])
('Splitting ', [2])
('Merging ', [2])
('Splitting ', [9, '1'])
('Splitting ', [9])
('Merging ', [9])
('Splitting ', [1])
('Merging ', [1])
('Merging ', [1, '9'])
('Merging ', [1, '2', '9'])
('Merging ', [1, '2', '4', '8', '9'])
Sorted Array is:
[1, '2', '4', '8', '9']

```

4.4.5 Quick Sort

This method also uses the techniques of 'divide and conquer'. On the basis of a selected element (pivot) from of the list, it partitions the rest of the list into two parts- a sub-list that contains elements less than the pivot and other sub-list containing elements greater than the pivot. The pivot is inserted between the two sub-lists. The algorithm is recursively applied to the sub-lists until the size of each sub-list becomes 1, indicating that the whole list has become sorted.

Consider the list given in figure 4.11. Let the first element (i.e., 8) be the pivot. Now the rest of the list can be divided into two parts- a sub-list that contains elements less '8' and the other sub-list that contains elements greater than '8' as shown in figure 4.11.

Now this process can be recursively applied on the two sub-lists to completely sort the whole list. For instance, '7' becomes the pivot for left sub-lists and '19' becomes pivot for the right sub-lists.

Note: Two sub-lists can be safely joined when every element in the first sub-list is smaller than every element in the second sub-list. Since 'join' is a faster operation as compared to a 'merge' operation, this sort is rightly named as a 'quick sort'.

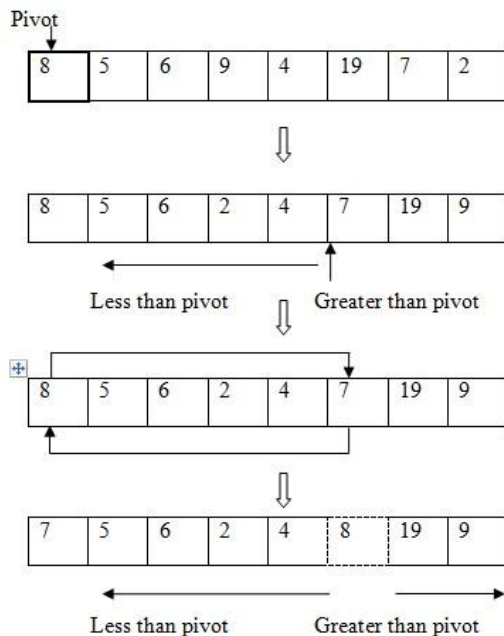


Figure 4.11 Quick Sort

The algorithm for the quick sort is given below:

In this algorithm, the elements of a list, stored in an array called List[], are sorted in an ascending order. The algorithm has two parts- quicksort and partition. The partition algorithm divides the list into two sub-lists around a pivot. The quick sort algorithm takes a list and stores it into an array called List[]. It uses two variables lb and ub to keep track of lower and upper bounds of list or sub-lists as the case may be. It employs partition algorithm to sort the sub-lists.

Algorithm quickSort()

1. Lb 50 #set lower bound
2. ub = N-1 #set upper bound
3. pivot = List[lb]
4. lb++
5. partition (pivot, List, lb, ub)

Algorithm partition (pivot, List, lb, ub)

1. i = lb
2. j=ub
3. while(i<=j)

```
#travel the list from lb till an element greater than the pivot is found
3.1 while (List[i] <= pivot) i++
# travel the list from ub till an element smaller than the pivot is found
3.2 while (List[j]> pivot) j--
3.3 if (i <= j) #exchange the elements
    3.3.1 temp = List[i]
    3.3.2 List[i] = List[j]
    3.3.3 List[j] = temp
4. temp =List[j] #place the pivot at mid of the sub-lists
5. List[j] = List[lb-1]
6. List[lb-1] = temp
7. if (j>lb) quicksort (List, lb, j-1) #sort left sub-list
8. if(j<ub) quicksort (List, j+1, ub) #sort the right sub-lists
```

Program to sort a List in ascending order using

```
def quicksort(myList, start, end):
    if start < end:
        pivot = partition(myList, start, end)
        print(myList)
        quicksort(myList, start, pivot-1)
        quicksort(myList, pivot+1, end)
    return myList

def partition(myList, start, end):
    pivot = int(myList[start])
    left = start+1
    right = end
    done = False
    while not done:
        while left <= right and int(myList[left]) <= pivot:
            left = left + 1
        while int(myList[right]) >= pivot and right >= left:
            right = right - 1
        if right < left:
            done= True
        else:
            temp=myList[left]
            myList[left]=myList[right]
            myList[right]=temp
    temp=myList[start]
    myList[start]=myList[right]
    myList[right]=temp
    return right

data = []
print('Quick Sort :')
n = int(raw_input('Enter Number of Elements in the Array: '))
for i in range(0, n):
    x = raw_input('Enter the Element %d :'%(i+1))
```

```
data.append(x)
print('Original Array :')
print(data)
print('Intermediate Steps :')
quicksort(data,0,n-1)
print('Sorted Array is:')
print(data)
```

Output

```
Quick Sort :
Enter Number of Elements in the Array: 8
Enter the Element 1 :8
Enter the Element 2 :5
Enter the Element 3 :6
Enter the Element 4 :9
Enter the Element 5 :4
Enter the Element 6 :19
Enter the Element 7 :7
Enter the Element 8 :2
Original Array :
['8', '5', '6', '9', '4', '19', '7', '2']
Intermediate Steps :
['7', '5', '6', '2', '4', '8', '19', '9']
['4', '5', '6', '2', '7', '8', '19', '9']
['2', '4', '6', '5', '7', '8', '19', '9']
['2', '4', '5', '6', '7', '8', '19', '9']
['2', '4', '5', '6', '7', '8', '9', '19']
Sorted Array is:
['2', '4', '5', '6', '7', '8', '9', '19']
```

GE8151 - PROBLEM SOLVING AND PYTHON
PROGRAMMING

REGULATIONS – 2017

UNIT – V

Prepared By :

Mr. Vinu S, ME,
Assistant Professor,
St. Joseph's College of Engineering,
Chennai -600119.

UNIT V

FILES, MODULES, PACKAGES

5.1 FILES

Most of the programs we have seen so far **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.

One of the simplest ways for programs to maintain their data is by reading and writing text files. An alternative is to store the state of the program in a database.

5.1.1 Text Files

A **text file** is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. Text file contain only text, and has no special formatting such as bold text, italic text, images, etc. Text files are identified with the .txt file extension.

5.1.2 Reading and Writing to Text Files

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language, 0s and 1s).

- **Text files:** In this type of file, each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

In order to perform some operations on files we have to follow below steps

- Opening
- Reading or writing
- Closing

Here we are going to discuss about opening, closing, reading and writing data in a text file.

5.1.2.1 File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the **File Handle** in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

- **Read Only ('r') :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exist, raises I/O error. This is also the default mode in which file is opened.

- **Read and Write ('r+')** : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
- **Write Only ('w')** : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.
- **Write and Read ('w+')** : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
- **Append Only ('a')** : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
- **Append and Read ('a+')** : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

5.1.3 Opening a File

It is done using the open() function. No module is required to be imported for this function.

Syntax:

```
File_object = open(r"File_Name","Access_Mode")
```

The file should exist in the same directory as the python program file else, full address (path will be discussed in later section of this unit) of the file should be written on place of filename. Note: The **r** is placed before filename to prevent the characters in filename string to be treated as special character. For example, if there is \temp in the file address, then \t is treated as the tab character and error is raised of invalid address. The **r** makes the string raw, that is, it tells that the string is without any special characters. The **r** can be ignored if the file is in same directory and address is not being placed.

Example:

```
>>> f1 = open("sample.txt","a")
>>> f2 = open(r"G:\class\python\sample3.txt","w+")
```

Here, f1 is created as object for sample.txt and f3 as object for sample3.txt (available in G:\class\python directory)

5.1.4 Closing a File

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

Syntax:

```
File_object.close()
```

Example:

```
>>> f1 = open("smapl.txt","a")
>>> f1.close()
```

After closing a file we can't perform any operation on that file. If want to do so, we have to open the file again.

5.1.5 Reading from a File

To read the content of a file, we must open the file in reading mode.

There are three ways to read data from a text file.

1. **read()** : Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.
File_object.read([n])
2. **readline()** : Reads a line of the file and returns in form of a string. For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.
File_object.readline([n])
3. **readlines()** : Reads all the lines and return them as each line a string element in a list.
File_object.readlines()

Example:

Consider the content of file sample.txt that is present in location G:\class\python\code\ as

```
Read Only
Read and Write
Write OnlyWrite and Read
Append Only
Append and Read
```

Now execute the following file reading script.

```
>>> f1=open("G:\class\python\code\sample.txt","r")
>>> f1.read()
'Read Only\nRead and Write\nWrite Only\nWrite and Read\nAppend Only\nAppend and Read'
```

Here \n denotes next line character. If you again run the same script, you will get empty string. Because during the first read statement itself file handler reach the end of the file. If you read again it will return empty string

```
>>> f1.read()
"
```

So in order to take back the file handler to the beginning of the file you have to open the file again or use seek() function. We will discuss about seek function in upcoming section.

```
>>> f1=open("G:\class\python\code\sample.txt","r")
>>> f1.read(10)
'Read Only\n'
>>> f1=open("G:\class\python\code\sample.txt","r")
>>> f1.readline()
'Read Only\n'
>>> f1=open("G:\class\python\code\sample.txt","r")
>>> f1.readlines()
['Read Only\n', 'Read and Write\n', 'Write Only\n', 'Write and Read\n', 'Append Only\n', 'Append and Read']
```

5.1.6 File Positions

tell(): The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

seek(): The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position. If the second argument is omitted, it also means use the beginning of the file as the reference position.

Example:

```
>>> f1=open("G:\class\python\code\sample.txt","r")
>>> f1.tell()
0L
>>> f1.readline()
'Read Only\n'
>>> f1.tell()
11L
>>> f1.seek(0)
>>> f1.tell()
0L
>>> f1.seek(5)
>>> f1.tell()
5L
>>> f1.readline()
'Only\n'
```

5.1.7 Writing to a File

In order to write into a file we need to open it in write 'w' or append 'a'. We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.

There are two ways to write in a file.

1. **write()** : Inserts the string *str1* in a single line in the text file.

```
File_object.write(str1)
```
2. **writelines()** : For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

```
File_object.writelines(L) for L = [str1, str2, str3]
```

Example:

```
>>> f4=open("fruit.txt","w")
```

```
>>> fruit_list=['Apple\n','Orange\n','Pineapple\n']
>>> f4.writelines(fruit_list)
>>> f4.write('Strawberry\n')
>>> f4.close()
>>> f4=open("fruit.txt","r")
>>> f4.read()
'Apple\nOrange\nPineapple\nStrawberry\n'
```

5.1.8 Appending to a File

Adding content at the end of a file is known as append. In order to do appending operation, we have to open the file with append mode.

Example:

```
>>> f4=open('fruit.txt','a')
>>> f4.write('Banana')
>>> f4.close()
>>> f4=open('fruit.txt','r')
>>> f4.read()
'Apple\nOrange\nPineapple\nStrawberry\nBanana\n'
```

5.1.9 The File Object Attributes

Once a file is opened and you have one file object, you can get various information related to that file. Here is a list of all attributes related to file object:

Attribute	Description
File_object.closed	Returns true if file is closed, false otherwise.
File_object.mode	Returns access mode with which file was opened.
File_object.name	Returns name of the file.
File_object.softspace	Returns false if space explicitly required with print, true otherwise.

5.1.10 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:

```
>>> f5=open('stringsample.txt','w')
>>> f5.write(5)
TypeError: expected a string or other character buffer object
>>> f5.write(str(5))
```

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 decimal value is converted to string.

```
>>> run=8
>>> '%d'%run
```

```
'8'
```

The result is the string '8', which is not to be confused with the integer value 8. Some other format strings are.

Conversion	Meaning
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
u	Unsigned decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).
r	String (converts any python object using repr()).
s	String (converts any python object using str()).
%	No argument is converted, results in a "%" character in the result.

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

```
>>> 'India need %d runs'%3
'India need 3 runs'
```

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.

```
>>> 'India need %d runs in %d balls'%(3,5)
'India need 3 runs in 5 balls'
```

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

```
>>> '%d %s price is %g rupees'%(5,'apple',180.50)
'5 apple price is 180.500000 rupees'
```

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' % 'apple'
```

```
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

5.1.11 Filenames and Paths

Files are organized into **directories** (also called “folders”). Every running program has a “current directory”, which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The os module provides functions for working with files and directories (“os” stands for “operating system”). os.getcwd returns the name of the current directory:

```
>>> import os
>>> os.getcwd()
'C:\\Python27'
```

cwd stands for “current working directory”. A string like 'C:\\Python27' that identifies a file or directory is called a **path**.

A simple filename, like 'stringsample.txt' is also considered a path, but it is a **relative path** because it relates to the current directory.

If the current directory 'C:\\Python27', the filename 'stringsample.txt' would refer to 'C:\\Python27\\stringsample.txt'.

A path that begins with drive letter does not depend on the current directory; it is called an **absolute path**. To find the absolute path to a file, you can use os.path.abspath:

```
>>> os.path.abspath('stringsample.txt')
'C:\\Python27\\stringsample.txt'
```

os.path provides other functions for working with filenames and paths. For example, os.path.exists checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, os.path.isdir checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('C:\\Python27')
True
```

Similarly, os.path.isfile checks whether it's a file.

os.listdir returns a list of the files (and other directories) in the given directory:

```
>>> cwd=os.getcwd()
>>> os.listdir(cwd)
['DLLs', 'Doc', 'include', 'inifitLoop.py', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt',
'parameter.py', 'python.exe', 'pythonw.exe', 'README.txt', 'sample.txt', 'sample2.txt',
'Scripts', 'stringsample.txt', 'swapwith third.py', 'tcl', 'Tools', 'w9xpopen.exe', 'wc.py', 'wc.pyc']
```

To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
>>> def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)
        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
>>> cwd=os.getcwd()
>>> walk(cwd)
```

Output:

```
C:\\Python27\\DLLs\\bz2.pyd
C:\\Python27\\DLLs\\py.ico
C:\\Python27\\DLLs\\pyc.ico
C:\\Python27\\DLLs\\pyexpat.pyd
C:\\Python27\\DLLs\\select.pyd
```

```
C:\Python27\DLLs\sqlite3.dll
C:\Python27\DLLs\tcl85.dll
C:\Python27\include\abstract.h
C:\Python27\include\asdl.h
C:\Python27\include\ast.h
```

os.path.join takes a directory and a file name and joins them into a complete path.

```
>>> os.path.join(cwd, 'stringsample.txt')
'C:\\Python27\\stringsample.txt'
```

5.2 COMMAND LINE ARGUMENTS

It is possible to pass some values from the command line to your python programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using **sys** module. We can access command-line arguments via the **sys.argv**. This serves two purposes –

- **sys.argv** is the list of command-line arguments.
- **len(sys.argv)** is the number of command-line arguments.

Here **sys.argv[0]** is the program name ie. script name.

Example 1

Consider the following script **command_line.py**

```
import sys
print 'There are %d arguments'%len(sys.argv)
print 'Argument are', str(sys.argv)
print 'File Name is: ', sys.argv[0]
```

Now run above script as follows – in Command prompt:

```
C:\Python27>python.exe command_line.py vinu ranjith
```

This produce following result –

```
There are 3 arguments
Argument are ['command_line.py', 'vinu', 'ranjith']
File Name is: command_line.py
```

NOTE: As mentioned above, first argument is always script name and it is also being counted in number of arguments. Here 'vinu' and 'ranjith' are extra inputs passed to program through command line argument method while running python program **command_line.py**.

Example 2

This is a Python Program to copy the contents of one file into another. Source and destination file names are given through command line argument while running the program.

In order to do this we have to follow the following steps

- 1) Open file name with command line argument one as read mode (input file).
- 2) Open file name with command line argument two as write mode (output file).
- 3) Read each line from the input file and write it into the output file until the input file data gets over.
- 4) Exit.

Program

```
import sys
source=open(sys.argv[1],'r')
destination=open(sys.argv[2],'w')
while(True):
    new_line=source.readline()
    if new_line=="":
        break
    destination.write(new_line)
source.close()
destination.close()
```

Now run above script as follows – in Command prompt:

```
C:\Python27>python.exe copy_file.py input_file.txt output_file.txt
```

5.3 ERRORS AND EXCEPTIONS

There are two distinguishable kinds of errors: **syntax errors** and **exceptions**.

5.3.1 Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python. **Syntax error** is an error in the syntax of a sequence of characters or tokens that is intended to be written in python. For compiled languages, syntax errors are detected at compile-time. A program will not compile until all syntax errors are corrected. For interpreted languages, however, a syntax error may be detected during program execution, and an interpreter's error messages might not differentiate syntax errors from errors of other kinds.

5.3.2 Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called **exceptions**. You will soon learn how

to handle them in Python programs. Most exceptions are not handled by programs, however, and **result in error messages as shown here:**

```
>>> 55+(5/0)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    55+(5/0)
ZeroDivisionError: integer division or modulo by zero

>>> 5+ repeat*2
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    5+ repeat*2
NameError: name 'repeat' is not defined

>>> '5'+5
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    '5'+5
TypeError: cannot concatenate 'str' and 'int' objects
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are **ZeroDivisionError**, **NameError** and **TypeError**. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Python's built-in exceptions lists and their meanings.

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.

EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
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.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

5.4 HANDLING EXCEPTIONS

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them.

➤ **Exception Handling:**

➤ **Assertions:**

5.4.1 Exception Handling

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include

an **except**: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax:

Here is simple syntax of *try....except...else* blocks –

```
try:
    You do your operations here;
    .....
except ExceptionI:
    If there is ExceptionI, then execute this block.
except ExceptionII:
    If there is ExceptionII, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax –

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example:

This example opens a file with write mode, writes content in the file and comes out gracefully because there is no problem at all

```
try:
    fp = open("test_exception.txt", "w")
    fp.write("Exception handling")
except IOError:
    print "Error: File don't have read permission"
else:
    print "Written successfully"
    fp.close()
```

This produces the following result:

Written successfully

Example:

This example opens a file with read mode, and tries to write the file where you do not have write permission, so it raises an exception

```
try:
    fp = open("test_exception.txt", "r")
    fp.write("Exception handling")
except IOError:
    print "Error: File don't have read permission"
else:
    print "Written successfully"
    fp.close()
```

This produces the following result

Error: File don't have read permission

5.4.2 The except Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows –

```
try:
    You do your operations here;
    .....
except:
    If there is any exception, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

5.4.3 The except Clause with Multiple Exceptions

You can also use the same except statement to handle multiple exceptions as follows –

```
try:
    You do your operations here;
    .....
except(Exception1[, Exception2[,...ExceptionN]]):
    If there is any exception from the given exception list, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

5.4.4 The try-finally Clause

You can use a **finally**: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this –

```
try:
    You do your operations here;
    .....
    Due to any exception, this may be skipped.
finally:
    This would always be executed.
    .....
```

You cannot use *else* clause as well along with a finally clause.

Example

This example opens a file with write mode, writes content in the file and comes out gracefully because there is no problem at all

```
try:
    fp = open("test_exception.txt", "w")
    fp.write("Exception handling")
except IOError:
    print "Error: File don't have read permission"
else:
    print "Written successfully"
finally:
    print "Closing file"
    fp.close()
```

This produces the following result

```
Written successfully
Closing file
```

Example

This example opens a file with read mode, and tries to write the file where you do not have write permission, so it raises an exception

```
try:
    fp = open("test_exception.txt", "r")
    fp.write("Exception handling")
except IOError:
    print "Error: File don't have read permission"
else:
    print "Written successfully"
finally:
```

```
print "Closing file"
fp.close()
```

This produces the following result

```
Error: File don't have read permission
Closing file
```

In the above two examples, one script didn't raise exception and another script raise exception. But we can see that in both cases finally block gets executed.

5.4.5 Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```
try:
    You do your operations here;
    .....
except ExceptionType, Argument:
    You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception

```
def temp_convert(var):
    try:
        return int(var)
    except ValueError, Argument:
        print "The argument is not a numbers\n", Argument
temp_convert("abc")
```

This produces the following result

```
The argument is not a numbers
invalid literal for int() with base 10: 'abc'
```

5.4.6 Hierarchical Exceptions Handle

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
def this_fails():
    x = 1/0
try:
    this_fails()
except ZeroDivisionError, detail:
    print 'Handling run-time error:', detail
```

This produces the following result

Handling run-time error: integer division or modulo by zero

In this example exception is raised in this_fails() function. But, because of this_fails() function don't have except block exception is thrown to the caller function. As there is a except block, it will handle the exception.

5.4.7 Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, *Exception* is the type of exception (for example, `NameError`) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is `None`.

The final argument, `traceback`, is also optional (and rarely used in practice), and if present, is the `traceback` object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows –

```
def functionName( level ):
    if level < 1:
        raise "Invalid level!", level
# if we raise the exception, code below to this not executed
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows

```
try:
    Business Logic here...
except "Invalid level!":
    Exception handling here...
else:
    Rest of the code here...
```

5.4.8 Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a **raise-if-not** statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the `assert` statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The *assert* Statement

When it encounters an `assert` statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an `AssertionError` exception.

The syntax for *assert* is

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses `ArgumentExpression` as the argument for the `AssertionError`. `AssertionError` exceptions can be caught and handled like any other exception using the `try-except` statement, but if not handled, they will terminate the program and produce a `traceback`.

Example

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature

```
def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0), "Colder than absolute zero!"
    return ((Temperature-273)*1.8)+32
print KelvinToFahrenheit(275)
print int(KelvinToFahrenheit(509.25))
print KelvinToFahrenheit(-7)
```

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

```
35.6
457
```

Traceback (most recent call last):

```
File "G:/class/python/code/assertion.py", line 6, in <module>
    print KelvinToFahrenheit(-7)
File "G:/class/python/code/assertion.py", line 2, in KelvinToFahrenheit
    assert (Temperature >= 0), "Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

5.5 MODULES

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a file that contains a collection of related functions. Python has lot of built-in modules; math module is one of them. math module provides most of the familiar mathematical functions.

Before we can use the functions in a module, we have to import it with an import statement:

```
>>> import math
```

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

```
>>> 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.

```
>>> math.log10(200)
2.3010299956639813
```

```
>>> math.sqrt(10)
3.1622776601683795
```

Math module have functions like log(), sqrt(), etc... In order to know what are the functions available in particular module, we can use **dir()** function after importing particular module. Similarly if we want to know detail description about a particular module or function or variable means we can use help() function.

Example

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> help(pow)
```

Help on built-in function pow in module __builtin__:

```
pow(...)
pow(x, y[, z]) -> number
```

With two arguments, equivalent to $x^{**}y$. With three arguments, equivalent to $(x^{**}y) \% z$, but may be more efficient (e.g. for longs).

5.5.1 Writing Modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named addModule.py with the following code:

```
def add(a, b):
    result = a + b
    print(result)
add(10,20)
```

If you run this program, it will add 10 and 20 and print 30. We can import it like this:

```
>>> import addModule
30
```

Now you have a module object addModule

```
>>> addModule
```

```
<module 'addModule' from 'G:/class/python/code/addModule.py'>
```

The module object provides add():

```
>>> addModule.add(120,150)
270
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    add(10,20)
```

__name__ is a built-in variable that is set when the program starts. If the program is running as a script, __name__ has the value '__main__'; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped. Modify addModule.py file as given below.

```
def add(a, b):
    result = a + b
    print(result)
if __name__ == '__main__':
    add(10,20)
```

Now while importing addModule test case is not running

```
>>> import addModule
```

__name__ has module name as its value when it is imported. Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed. If you want to reload a module, you can use the built-in function reload, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

5.6 PACKAGES

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

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.

If we create a directory called **sample_package**, which marks the package name, we can then create a module inside that package called **sample_module**. We also must not forget to add the `__init__.py` file inside the **sample_package** directory.

To use the module **sample_module**, we can import it in two ways:

```
>>> import sample_package.sample_module
```

or:

```
>>> from sample_package import sample_module
```

In the first method, we must use the **sample_package** prefix whenever we access the module **sample_module**. In the second method, we don't, because we import the module to our module's namespace.

The `__init__.py` file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the `__all__` variable, like so:

```
__init__.py:
__all__ = ["sample_module"]
```

5.7 WORD COUNT

Example.

Following program print each word in the specified file occurs how many times.

```
import sys
def word_count(file_name):
    try:
        file=open(file_name,"r")
        wordcount={}
        entier_words=file.read().split()
        for word in entier_words:
            if word not in wordcount:
                wordcount[word] = 1
            else:
                wordcount[word] += 1
        file.close();
        print ("% -30s %s " % ('Words in the File' , 'Count'))
        for key in wordcount.keys():
            print ("% -30s %d " % (key , wordcount[key]))
    except IOError:
        print ("No file found with name %s" %file_name)
        fname=raw_input("Enter New File Name:")
        word_count(fname)

try:
    word_count(sys.argv[1])
except IndexError:
    print("No file name passed as command line Argument")
    fname=raw_input("Enter File Name:")
    word_count(fname)
```

Content of a sample file word_count_input.txt is:

word count is the program which count each word in file appears how many times.

This produces the following result

```
c:\Python27>python wordcount1.py
No file name passed as command line Argument
Enter File Name:word_count_input
No file found with name word_count_input
Enter New File Name:word_count_input.txt
Words in the File      Count
count                  2
word                   2
file                   1
```

many	1
is	1
in	1
times	1
how	1
program	1
which	1
each	1
the	1
appears	1

Above program shows the file handling with exception handling and command line argument. While running, if you give command line like below, it will read the text file with the name, that is specifies by command line argument one and calculate the count of each word and print it.

```
c:\Python27>python wordcount1.py word_count_input.txt
```

While running, if you give command line like below(ie, no command line argument), On behalf of exception handling it will ask for file name and then do the same operation on the newly entered file.

```
c:\Python27>python wordcount1.py
```

Program also handle file not found exception, if wrong file name is entered. It will ask for new file name to enter and proceed.

Example.

Following program counts number of words in the given file.

```
import sys
def word_count(file_name):
    count=0
    try:
        file=open(file_name,"r")
        entier_words=file.read().split()
        for word in entier_words:
            count=count+1
        file.close();
        print ("%s File have %d words" %(file_name,count))
    except IOError:
        print ("No file found with name %s" %file_name)
        fname=raw_input("Enter New File Name:")
        word_count(fname)

try:
    word_count(sys.argv[1])
except IndexError:
    print("No file name passed as command line Argument")
```

```
fname=raw_input("Enter File Name:")
word_count(fname)
```

This produces the following result

```
c:\Python27>python wordcount2.py
No file name passed as command line Argument
Enter File Name:word_count_input
No file found with name word_count_input
Enter New File Name:word_count_input.txt
word_count_input.txt File have 15 words
```

Above program also shows the file handling with exception handling and command line argument. While running, if you give command line like below, it will read the text file with the name, that is specifies by command line argument one and count number of word present in it and print it.

```
c:\Python27>python wordcount2.py word_count_input.txt
```

While running, if you give command line like below (ie, no command line argument), On behalf of exception handling it will ask for file name and then do the same word counting operation on the newly entered file.

```
c:\Python27>python wordcount2.py
```

Program also handle file not found exception, if wrong file name is entered. It will ask for new file name to enter and proceed.

5.8 COPY FILE

This is a Python Program to copy the contents of one file into another. In order to perform the copying operation we need to follow the following steps.

1. Open source file in read mode.
2. Open destination file in write mode.
3. Read each line from the input file and write it into the output file.
4. Exit.

Also we include command line argument for passing source and destination file names to program. Also exception handling is used to handle exception that occurs when dealing with files.

```
import sys
def copy(src,dest):
    try:
        source=open(src,'r')
        destination=open(dest,'w')
        while(True):
            new_line=source.readline()
            if new_line=="":
                break
            destination.write(new_line)
```

```
        source.close()
        destination.close()
    except IOError:
        print ("Problem with Source or Destination File Name ")
        source_name=raw_input("Enter New Source File Name:")
        destination_name=raw_input("Enter New Destination File Name:")
        copy(source_name,destination_name)

try:
    copy(sys.argv[1],sys.argv[2])
except IndexError:
    print("Insufficient Command line argument!")
    source_name=raw_input("Enter Source File Name:")
    destination_name=raw_input("Enter Destination File Name:")
    copy(source_name,destination_name)

finally:
    print("Copying Done.....")
```

This produces the following result

```
C:\Python27>python copy_file_exception.py input_file.txt
Insufficient Command line argument!
Enter Source File Name:input_file.tx
Enter Destination File Name:output_file.txt
Problem with Source or Destination File Name
Enter New Source File Name:input_file.txt
Enter New Destination File Name:output_file.txt
Copying Done.....
```