KARPAGAM ACADEMY OF HIGHER EDUCATION
Coimbatore-641 021
(For the candidates admitted from 2017 onwards)

DEPARTMENT OF CS, CA & IT
SYLLABUS

SUBJECT CODE: 18CTU304B **SEMESTER: III**
SUBJECT NAME: PROGRAMMING IN PHYTHON **CLASS: II B. Sc. [CT]**

Instruction Hours / week: L: 3 T: 0 P: 0 **Marks:** Internal : **40** External : **60** Total: **100**
End Semester Exam : 3 Hours

Course Objectives

- To Learn Syntax and Semantics and create Functions in Python.
- To Handle Strings and Files in Python.
- To Understand Lists, Dictionaries in Python.
- To Implement Object Oriented Programming concepts in Python
- To Build GUI applications

Course Outcomes (COs)

Upon completion of the course, students will be able to

1. Develop algorithmic solutions to simple computational problems
2. Read, write, execute by hand simple Python programs.
3. Structure simple Python programs for solving problems.
4. Decompose a Python program into functions.
5. Represent compound data using Python lists, tuples, dictionaries.
6. Read and write data from/to files in Python Programs.

Unit I: ALGORITHMIC PROBLEM SOLVING

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

Unit II: DATA, EXPRESSIONS, STATEMENTS

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

Unit III: CONTROL FLOW, FUNCTIONS

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: state, while, for, break, continue, pass; Fruitful functions:

return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Lists as arrays. Illustrative programs: square root, gcd, exponentiation, sum an array of numbers, linear search, binary search.

Unit IV: LISTS, TUPLES, DICTIONARIES

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

Unit V: FILES, MODULES, PACKAGES

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

Suggested Readings

1. Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist‘‘, 2nd edition,
2. Updated for Python 3, Shroff/O‘Reilly Publishers, 2016 (http://greenteapress.com/wp/thinkpython/)
3. Guido van Rossum and Fred L. Drake Jr, ―An Introduction to Python – Revised and updated for Python 3.2, Network Theory Ltd., 2011.
4. John V Guttag, ―Introduction to Computation and Programming Using Python‘‘, Revised and expanded Edition, MIT Press , 2013
5. Robert Sedgewick, Kevin Wayne, Robert Dondero, ―Introduction to Programming in Python: An Inter-disciplinary Approach, Pearson India Education Services Pvt. Ltd., 2016.
6. Timothy A. Budd, ―Exploring Python‖, Mc-Graw Hill Education (India) Private Ltd.,, 2015.
7. Kenneth A. Lambert, ―Fundamentals of Python: First Programs‖, CENGAGE Learning, 2012.
8. Charles Dierbach, ―Introduction to Computer Science using Python: A Computational ProblemSolving Focus, Wiley India Edition, 2013.
9. Paul Gries, Jennifer Campbell and Jason Montojo, ―Practical Programming: An Introduction to Computer Science using Python 3‖, Second edition, Pragmatic Programmers, LLC, 2013.

WEB SITES

1. https://rmd.ac.in/dept/snh/notes/1/PSPP/unit1.pdf
2. https://www.sece.ac.in/pdf/lr-cse/UNIT-II.pdf
3. https://rmd.ac.in/dept/snh/notes/1/PSPP/unit1.pdf
4. https://rmd.ac.in/dept/snh/notes/1/PSPP/unit4.pdf
5. http://wingsofaero.in/wp-content/uploads/2018/12/Files-and-exception.pdf

Karpagam Academy of Higher Education

(Deemed to be University)
(Established under Section 3 of UGC Act 1956)
Coimbatore -21.

Department of CS, CA & IT

SUBJECT NAME**: Programming in Python** SEMESTER**: III**

SUBJECT CODE**: 18CTU304B** CLASS**: II B. Sc. CT**

STAFF**: Mr. K. VEERASAMY** BATCH**: 2018-2021**

S. No.	Lecture Duration (Hr)	Topics	Support Materials
		UNIT -I	
1.	1	Algorithms, Building Block of Algorithms	S1:65
2.	1	Notation(Pseudo Code, Notation), Flow Control	S3:66-70
3.	1	Algorithmic Problem Solving	S3:71-72
4.	1	Simple Strategies for Developing algorithms(Iteration, recursion)	S3:44
5.	1	Illustrative Programs: Find Minimum in a List, Insert a card in a list of sorted cards	W1
6.	1	Guess an integer numbersin a range, Towers of Hanoi	W2
7.	1	Recapitulation and Discussion of important questions	
		Total No. of Periods allotted for Unit – I	**7**
		UNIT-II	
1.	1	Data, Expressions, Statements, Python interpreter and interactive mode, values and types, int, float, Boolean, string and list	S1:13-14,20-22
2.	1	Variables, Expressions, Statements, tuple assignments, precedence of operators, comments	S1:8-9,15-19
3.	1	Modules and functions	S1:177-181
4.	1	Function Definition and use, flow of execution	S1:77-81
5.	1	Parameters and arguments	S1:82-84
6.	1	Illustrative programs: Exchange the values of two variable, circulate the values of n variable, distance between two	W3

		points	
7.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours allotted for Unit – II	**7**
UNIT-III			
1.	1	Conditionals: Boolean, values and Operators conditional lists, alternative(if else), chained conditions(if elif-else)	S1:39-48
2.	1	Iteration: state, white, for, break, continue, pass	S1:49-62
3.	1	Functions: return values, parameters, local & global scope, function comparison, recursion	S1:83-85,91-95,105
4.	1	Strings: string slices immutability	S1:104-109,115
5.	1	String functions and methods, string module list as arrays	S1:114,120-122
6.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours allotted for Unit – III	**6**
UNIT-IV			
1.	1	Lists: List operations, List styles, List Methods	S1:133-136,142-143
2.	1	List loop, list multiplicity, list aliasing, cloning lists, list parameters	S1:68,137, 139-141
3.	1	Tuples: Tuples assignment, tuple as return value	S1:129-131
4.	1	Dictionaries: Operations and methods, advanced list processing – list comprehension	S1:143,151-154
5.	1	Illustrative Programs: Insertion sort, Merge sort, Histogram	W4
6.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours allotted for Unit – IV	**6**
UNIT-V			
1.	1	Text files, reading and writing files, format operator, command line arguments	S1:167-170
2.	1	Errors and Exceptions, Handling expressions	S1:255-258
3.	1	Modules, Packages	S1:171-183
4.	1	Illustrative programs: word count, copy file	W5
5.	1	Exception program(examples)	W5
6.	1	Reading and writing files examples	W5
7.	1	Recapitulation and Discussion of important questions	
8.	1	Discussion of previous ESE Question papers	

9.		Discussion of previous ESE Question papers	
10.		Discussion of previous ESE Question papers	
		Total No. of Hours allotted for Unit – V	**10**

Total No. of Hours: 40

Suggested Readings

Suggested Readings

1. Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist‘‘, 2nd edition,
2. Updated for Python 3, Shroff/O‘Reilly Publishers, 2016 (http://greenteapress.com/wp/thinkpython/)
3. Guido van Rossum and Fred L. Drake Jr, —An Introduction to Python – Revised and updated for Python 3.2, Network Theory Ltd., 2011.
4. John V Guttag, —Introduction to Computation and Programming Using Python‘‘, Revised and expanded Edition, MIT Press , 2013
5. Robert Sedgewick, Kevin Wayne, Robert Dondero, —Introduction to Programming in Python: An Inter-disciplinary Approach, Pearson India Education Services Pvt. Ltd., 2016.
6. Timothy A. Budd, —Exploring Python , Mc-Graw Hill Education (India) Private Ltd.,, 2015.
7. Kenneth A. Lambert, —Fundamentals of Python: First Programs , CENGAGE Learning, 2012.
8. Charles Dierbach, —Introduction to Computer Science using Python: A Computational Problem Solving Focus, Wiley India Edition, 2013.
9. Paul Gries, Jennifer Campbell and Jason Montojo, —Practical Programming: An Introduction to Computer Science using Python 3 , Second edition, Pragmatic Programmers, LLC, 2013.

WEB SITES

1. https://rmd.ac.in/dept/snh/notes/1/PSPP/unit1.pdf
2. https://www.sece.ac.in/pdf/lr-cse/UNIT-II.pdf
3. https://rmd.ac.in/dept/snh/notes/1/PSPP/unit1.pdf
4. https://rmd.ac.in/dept/snh/notes/1/PSPP/unit4.pdf
5. http://wingsofaero.in/wp-content/uploads/2018/12/Files-and-exception.pdf

FACULTY **HOD**

UNIT I ALGORITHMIC PROBLEM SOLVING 9

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

Algorithm

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

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

Building blocks of algorithm

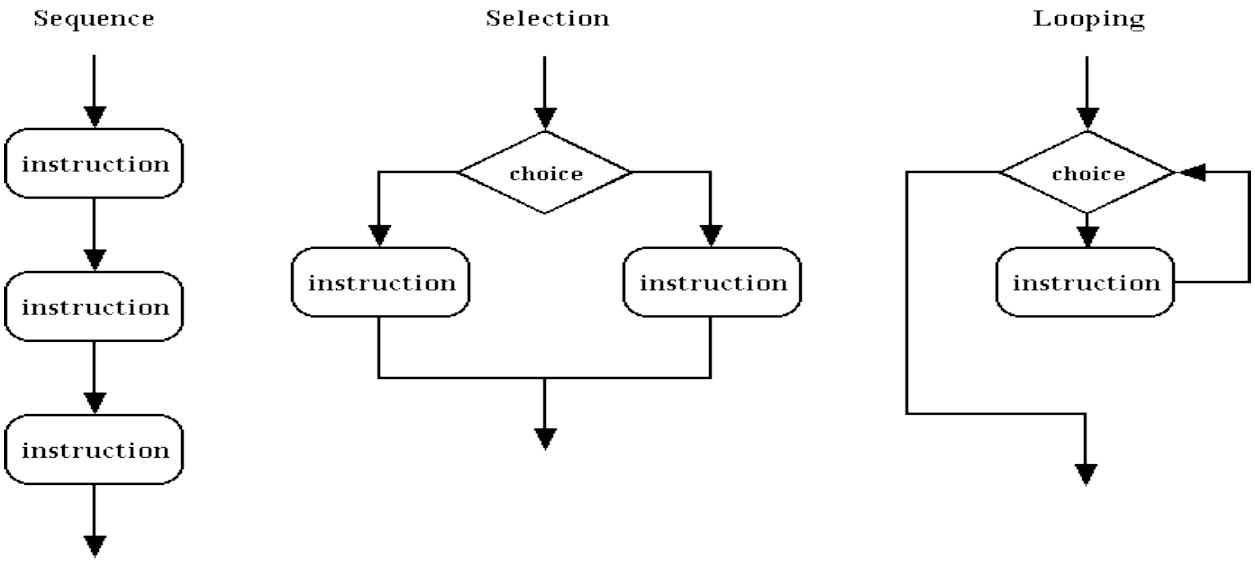
It has been proven that any algorithm can be constructed from just three basic building blocks. These three building blocks are Sequence, Selection, and Iteration.

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

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

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

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



Write an algorithm to add two numbers entered by user.

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

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

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

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

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

Step 3: Calculate discriminant
D←b2-4ac
Step 4: If D≥0
r1←(-b+√D)/2a
r2←(-b-√D)/2a
Display r1 and r2 as roots.
Else
Calculate real part and imaginary part
rp←b/2a
ip←√(-D)/2a
Display rp+j(ip) and rp-j(ip) as roots
Step 5: Stop

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

Step 1: Start
Step 2: Declare variables n,factorial and i.
Step 3: Initialize variables
factorial←1
i←1
Step 4: Read value of n
Step 5: Repeat the steps till i=n
5.1: factorial←factorial*i
5.2: i←i+1
Step 6: Display factorial
Step 7: Stop

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

Step 1: Start
Step 2: Declare variables n,i,flag.
Step 3: Initialize variables
flag←1
i←2
Step 4: Read n from user.
Step 5: Repeat the steps till i<(n/2)
5.1 If remainder of n÷i equals 0
flag←0
Go to step 6
5.2 i←i+1
Step 6: If flag=0
Display n is not prime
else
Display n is prime
Step 7: Stop

Write an algorithm to find the Fibonacci series till term≤1000.

Step 1: Start

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

Pseudo code:

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

Compute the area of a rectangle:

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

Compute the perimeter of a rectangle:

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

Iteration:

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

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

Recursion

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

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

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

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

Disadvantages of Recursion over iteration

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

Advantages of Recursion over iteration

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

Flow Charts

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

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

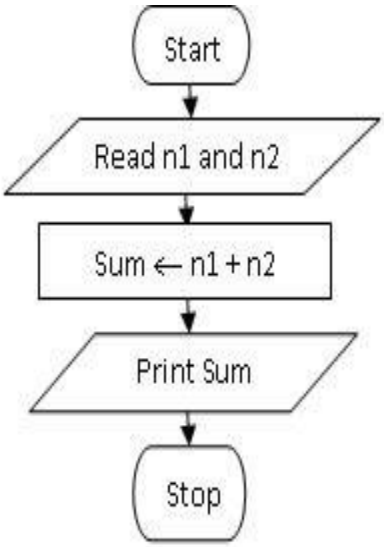
Flow charts tend to consist of four main symbols, linked with arrows that show the direction of flow:

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

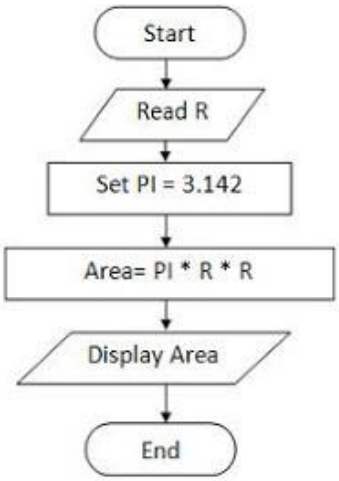
Name	Symbol	Description
Process		Process or action step
Flow line		Direction of process flow
Start/ terminator		Start or end point of process flow
Decision		Represents a decision making point
Connector		Inspection point
Inventory		Raw material storage
Inventory		Finished goods storage
Preparation		Initial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

Flow Chart Examples:

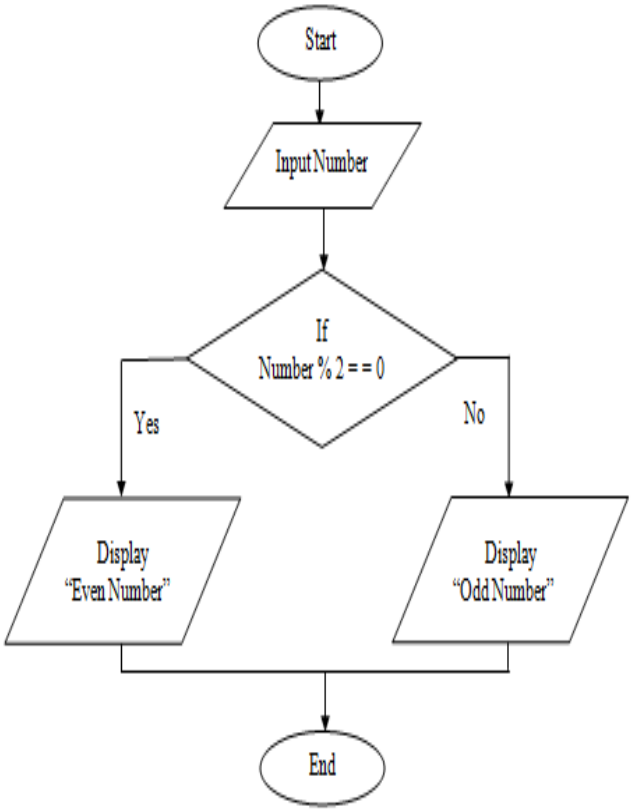
Sequence to find the sum of two number



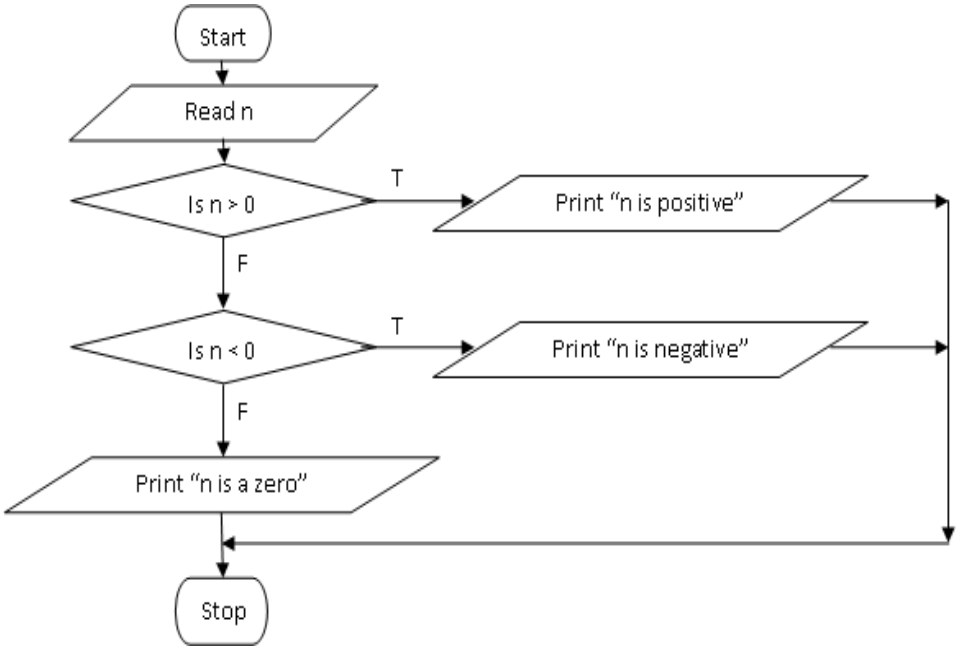
Area of triangle



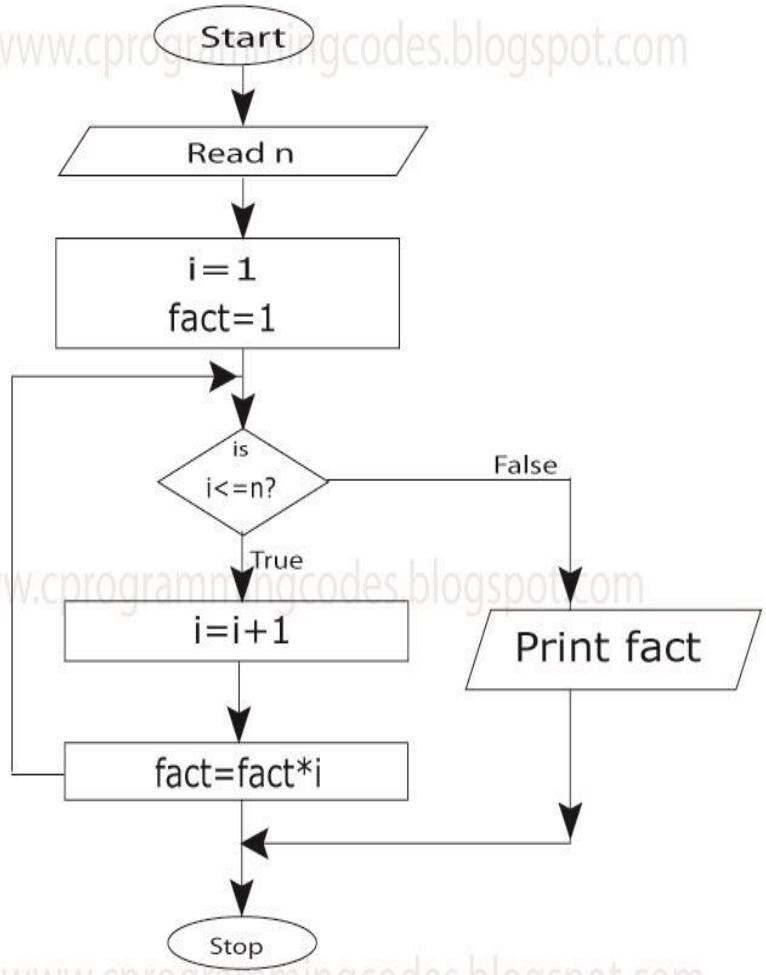
Flowchart to find the number is odd or Even

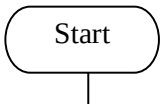


Flow Chart to find Positive or Negative



Factorial of the given Number



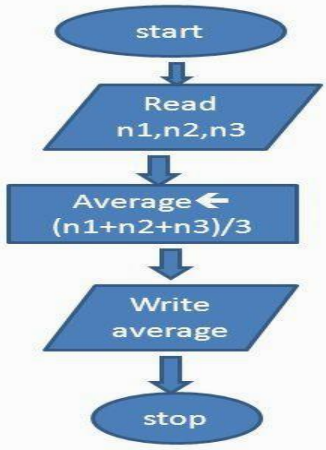


Find the biggest of two numbers

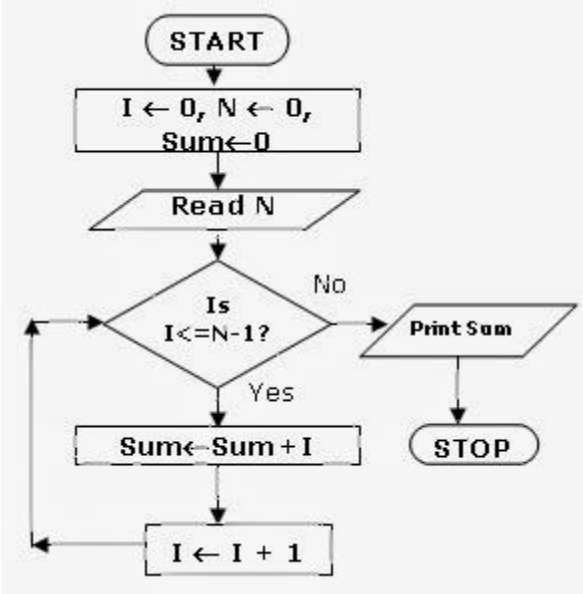
Flowchart algorithm3

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

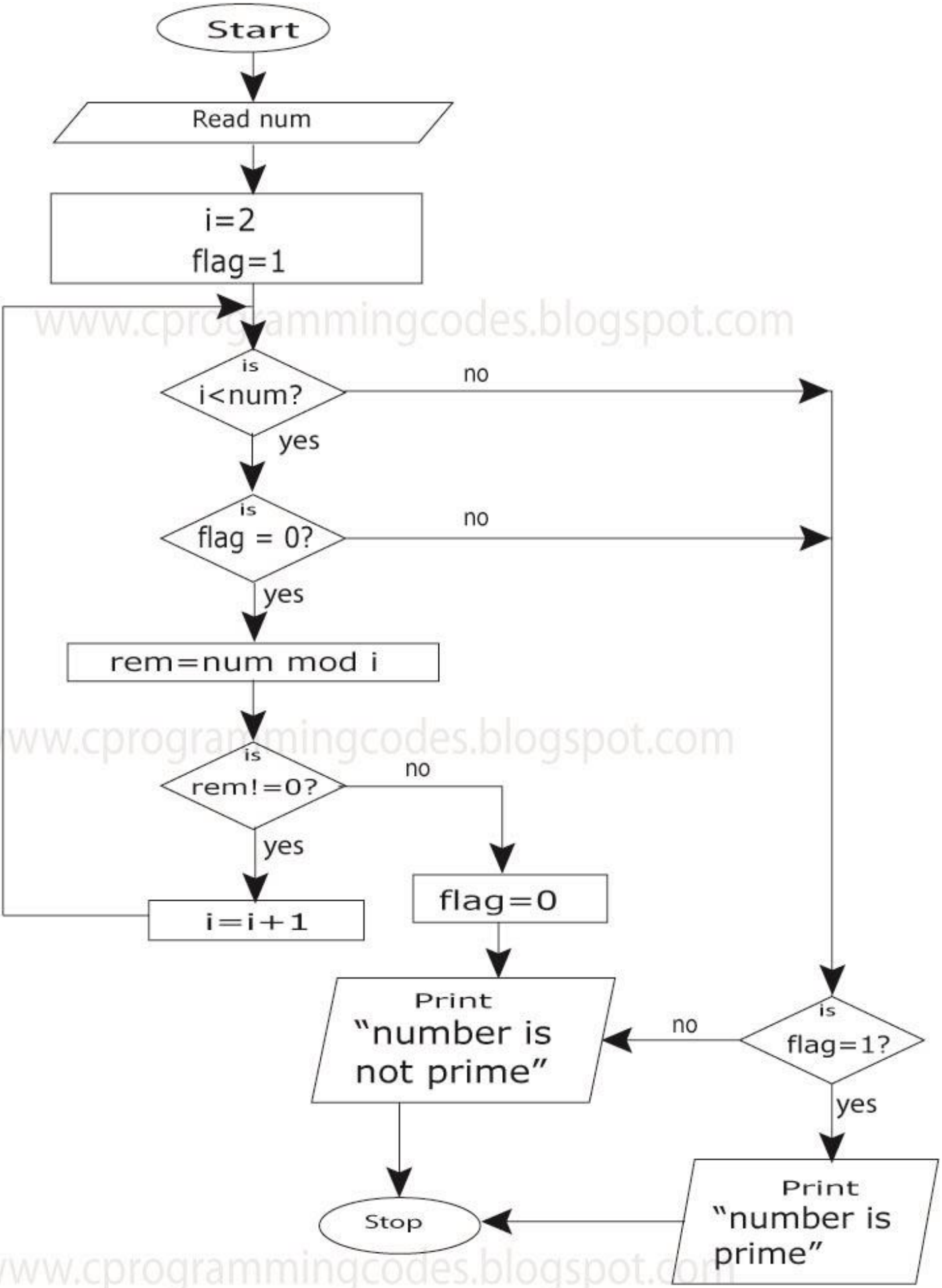
Start
Read n1,n2,n3
Average←(n1+n2+n3)/3
Write average
stop



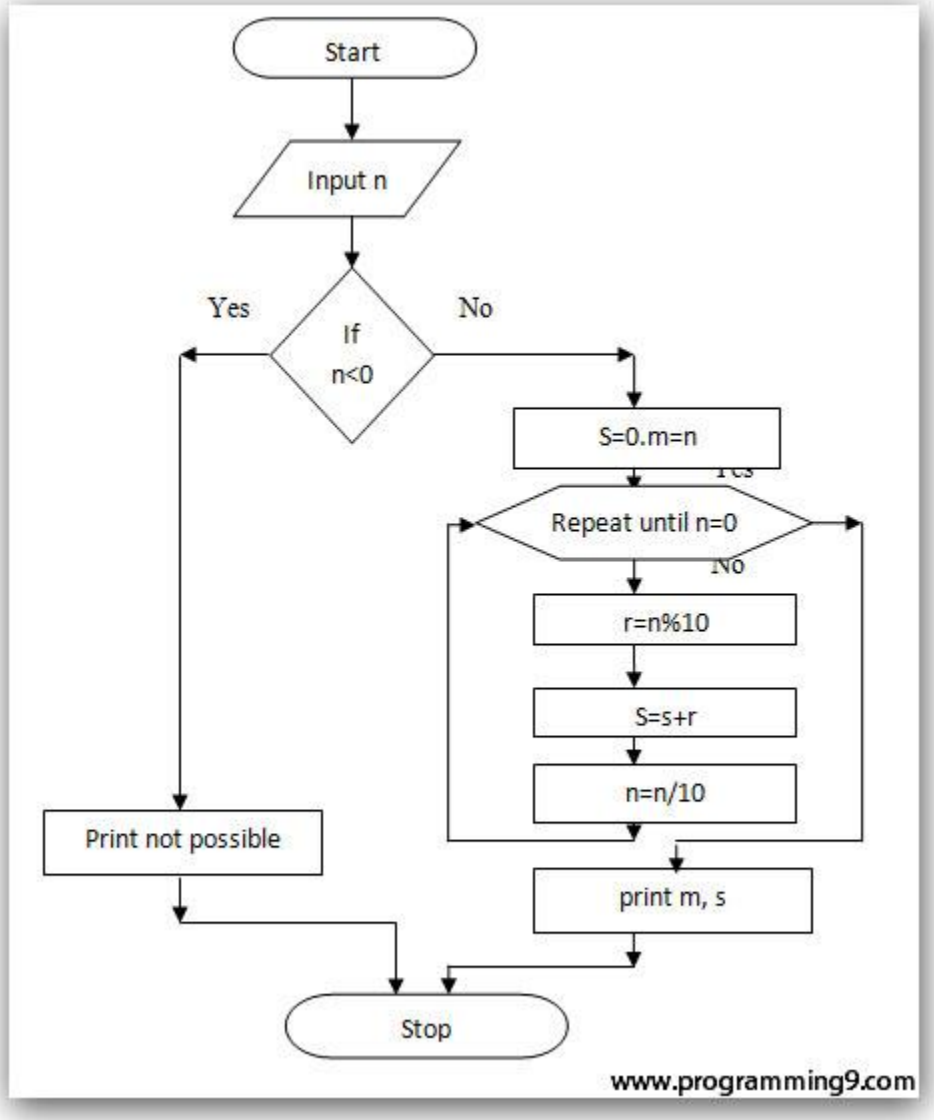
Flow chart to find the sum of n numbers



Flow chart for finding Prime number



Flowchart for finding sum of digits of a given number

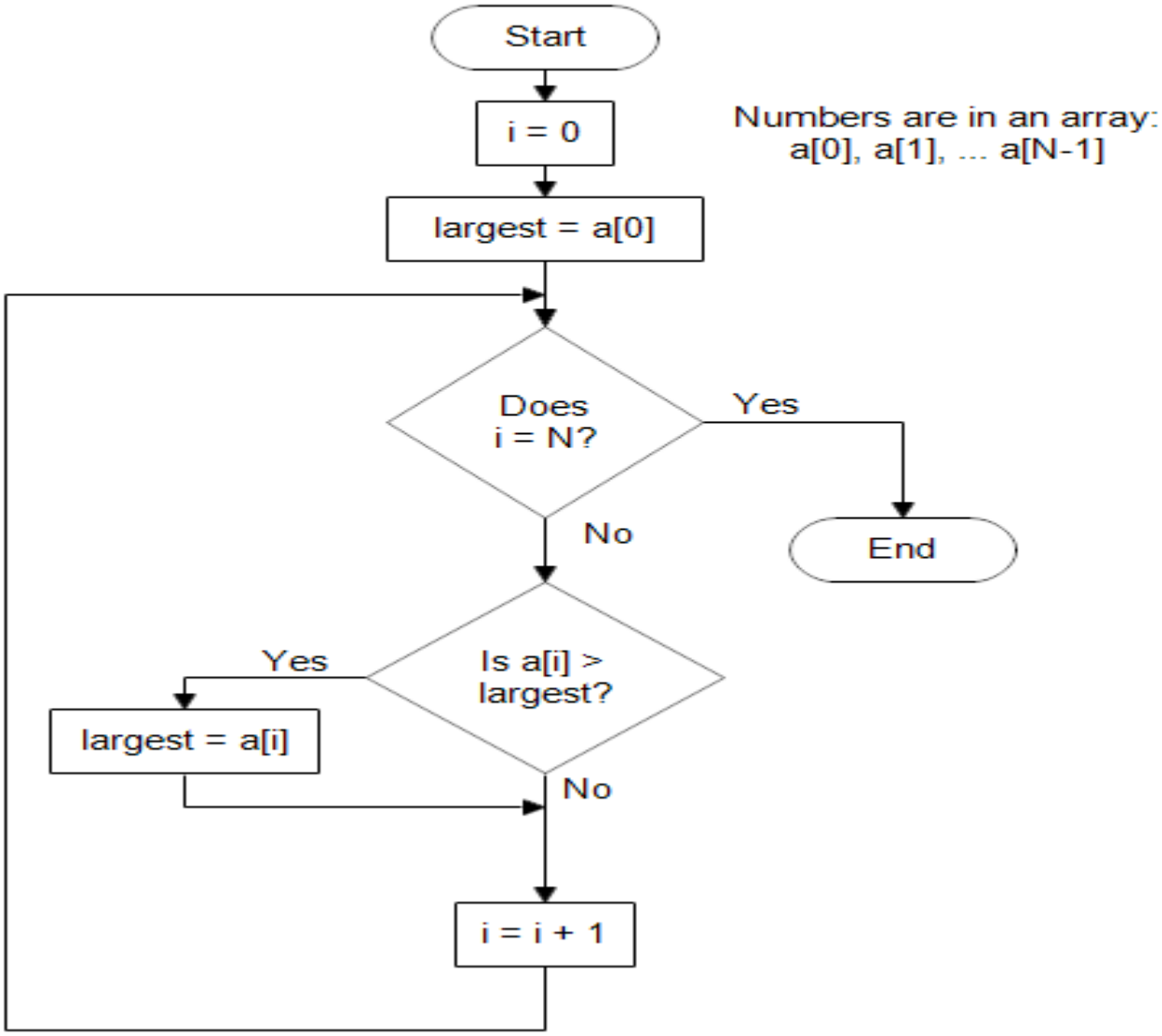


Illustrative problems: find minimum in a list, insert a card in a list of sorted cards, guess an integer number in a range, Towers of Hanoi.

Find minimum in a list:

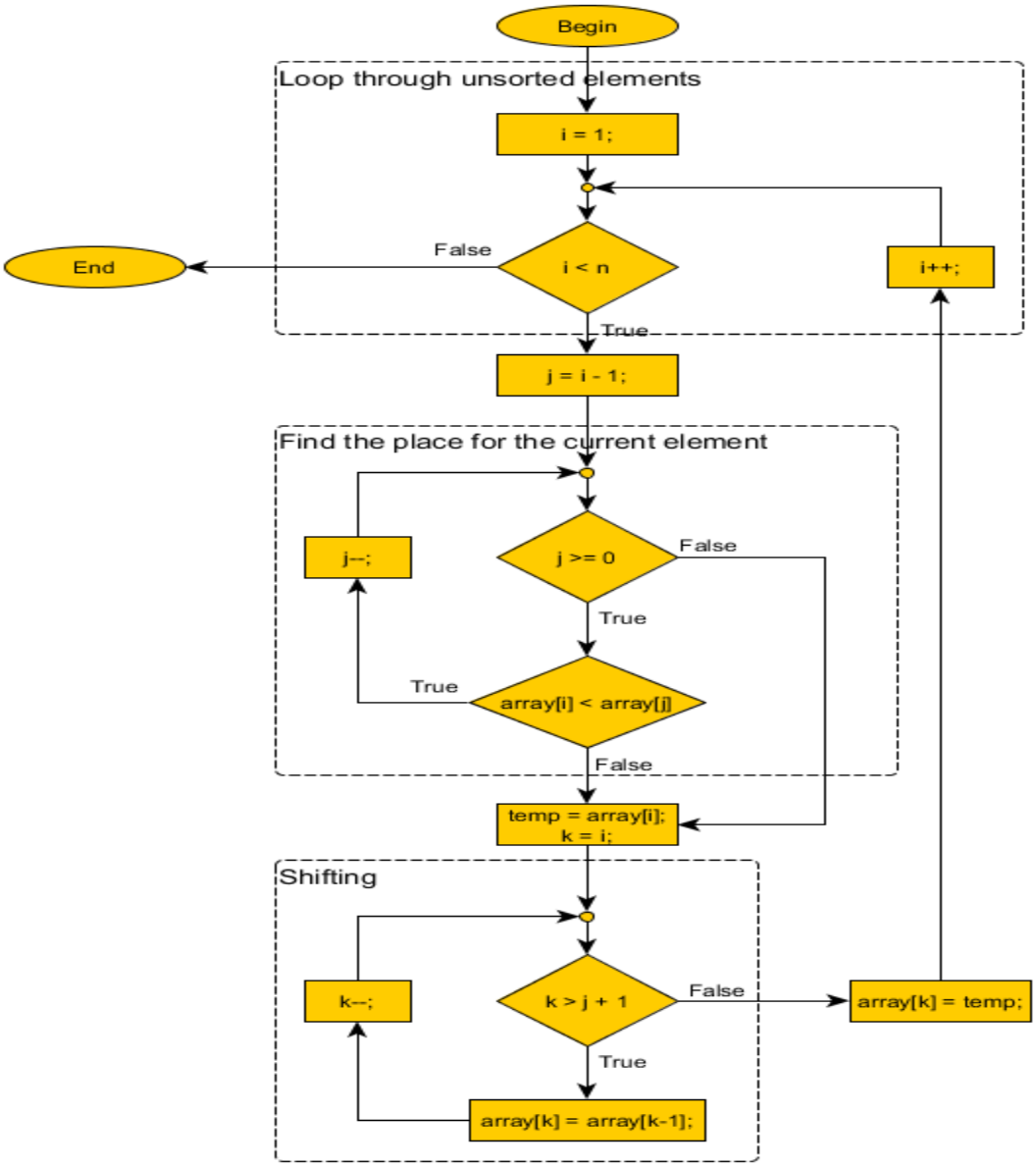
```
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
Max = Min = a [1];
For i = 2 to n do
{
If (a [i] > Max) then Max = a [i];
If (a [i] < Min) then Min = a [i];
}}
```

Finding the Largest Number in a List of Numbers



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

Insertion Sort



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

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

Guess a Number in python

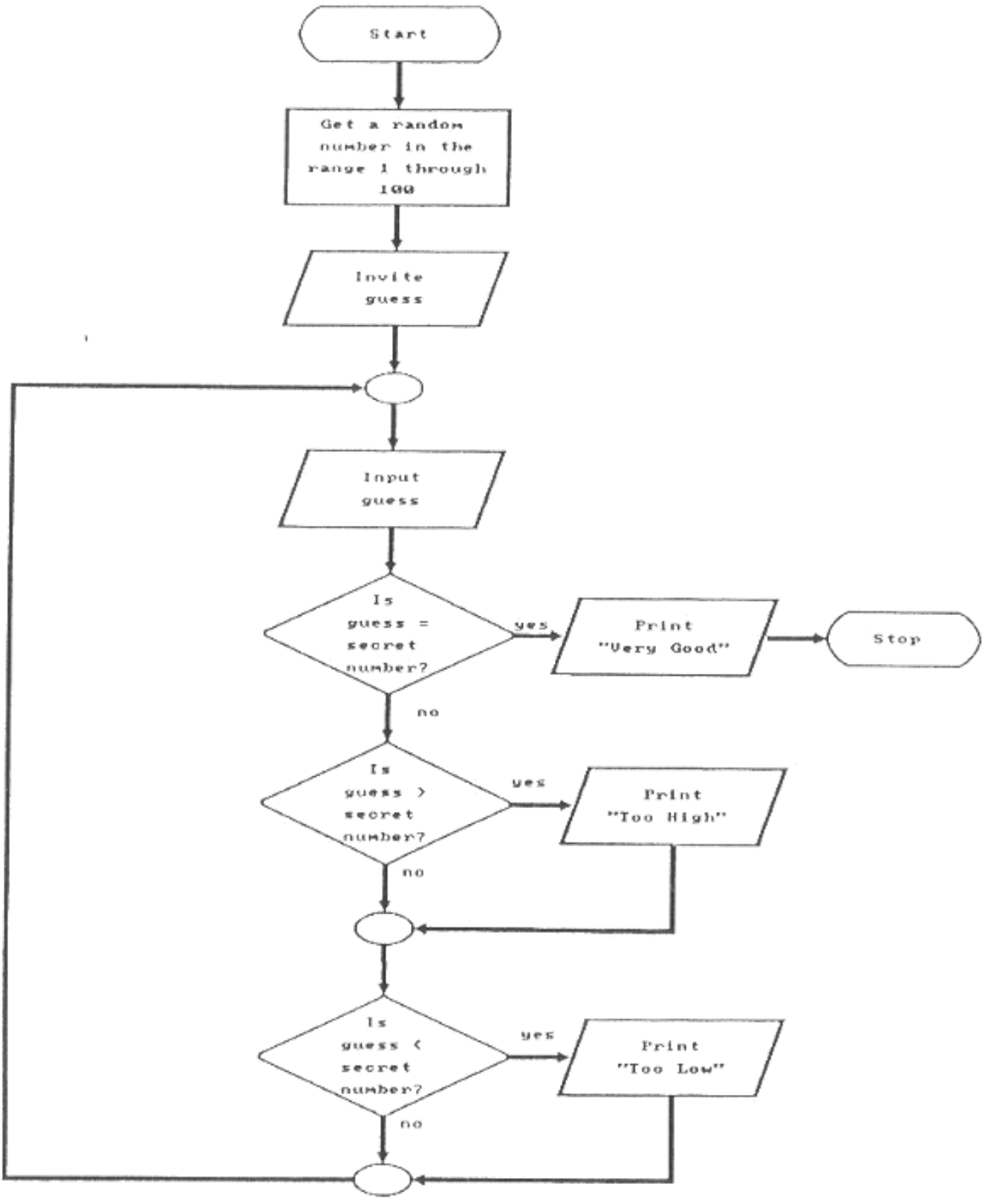
import random

```
x = random.randrange(1, 201)
print(x)
guess = int(input("Please enter your guess: "))
if guess == x:
    print("Hit!")
elif guess < x:
    print("Your guess is too low")
else:
    print ("Your guess is too high")
```

OUTPUT

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

Flow Chart


Start
Get a random number in the range 1 through 100
Invite guess
Input guess
Is guess = secret number?
yes
Print "Very Good"
Stop
no
Is guess > secret number?
yes
Print "Too High"
no
Is guess < secret number?
yes
Print "Too Low"
no

Range function in python

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

range(stop)

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

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

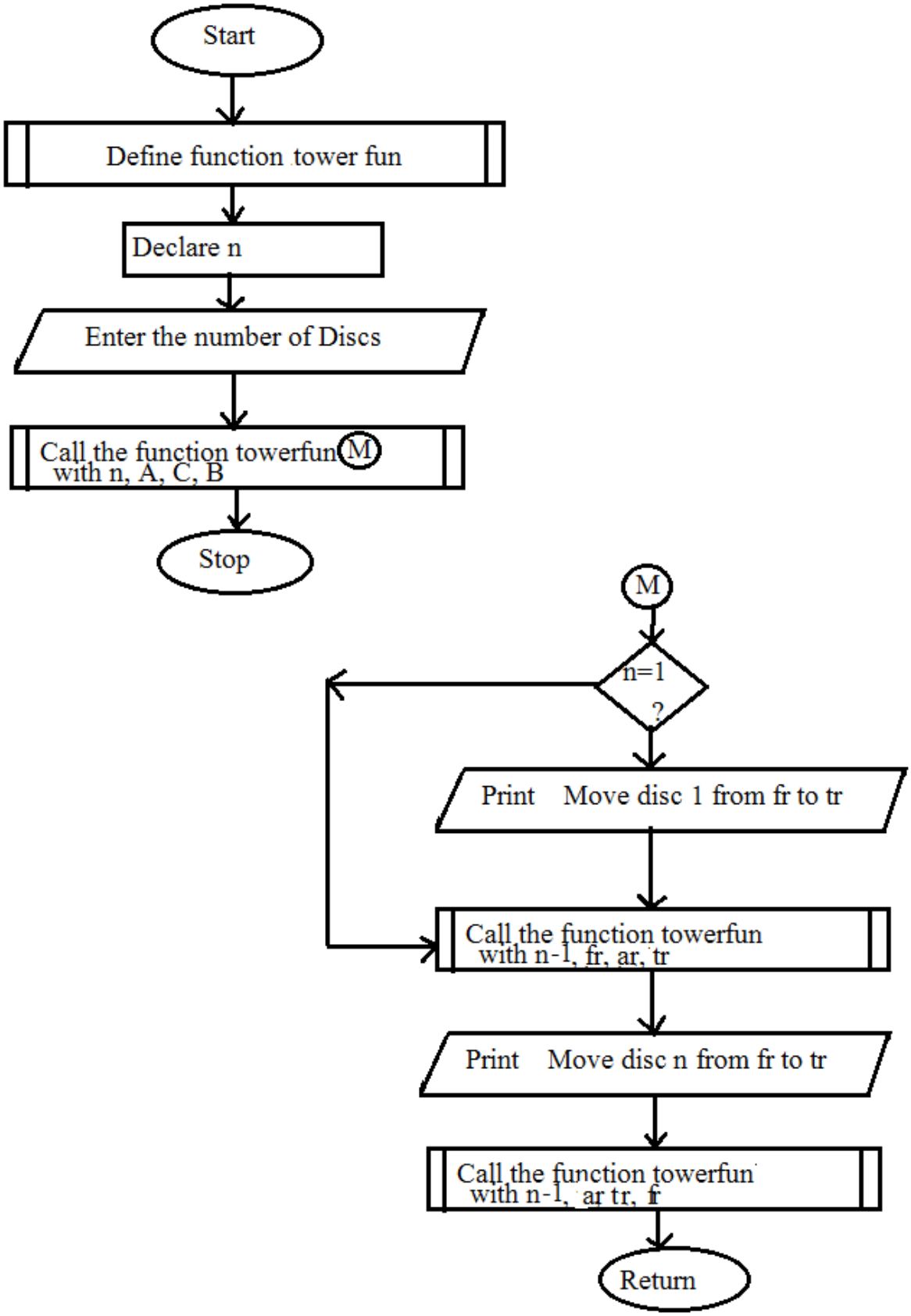
- start: Starting number of the sequence.
- stop: Generate numbers up to, but not including this number.
- step: Difference between each number in the sequence.

Note that:

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

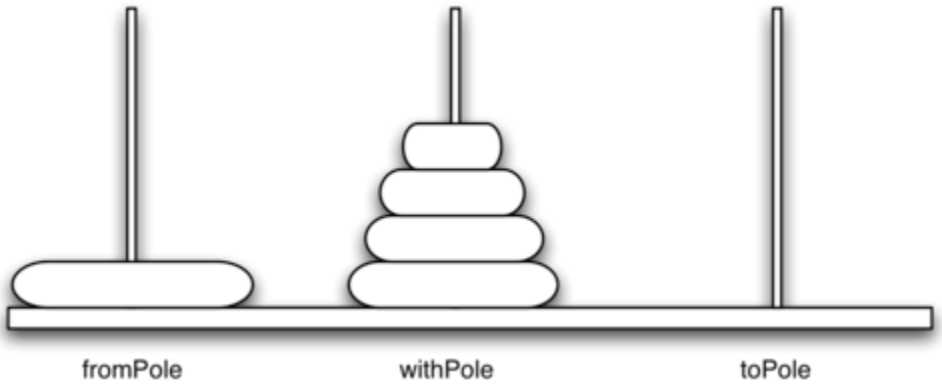
```
>>> # One parameter
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> # Three parameters
>>> for i in range(4, 10, 2):
...     print(i)
...
4
6
8
>>> # Going backwards
>>> for i in range(0, -10, -2):
...     print(i)
...
0
-2
-4
-6
-8
```

Tower of the Hanoi


Start
Define function tower fun
Declare n
Enter the number of Discs
Call the function towerfun M
with n, A, C, B
Stop
M
n=1
?
Print Move disc 1 from fr to tr
Call the function towerfun
with n-1, fr, ar, tr
Print Move disc n from fr to tr
Call the function towerfun
with n-1, ar tr, fr
Return

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

Although the legend is interesting, you need not worry about the world ending any time soon. The number of moves required to correctly move a tower of 64 disks is $264-1=18,446,744,073,709,551,615$. At a rate of one move per second, that is 584,942,417,355584,942,417,355 years! Clearly there is more to this puzzle than meets the eye.
Figure 1 shows an example of a configuration of disks in the middle of a move from the first peg to the third. Notice that, as the rules specify, the disks on each peg are stacked so that smaller disks are always on top of the larger disks. If you have not tried to solve this puzzle before, you should try it now. You do not need fancy disks and poles–a pile of books or pieces of paper will work.



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

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

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

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

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

OUTPUT

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

UNIT II

DATA, EXPRESSIONS, STATEMENTS

PYTHON INTERPRETER AND INTERACTIVE MODE:

Python being a interpreter language has two modes namely: **Interactive mode, Script Mode or Normal mode**

INTERACTIVE MODE:

- Is a command line shell which gives the immediate feedback for each statement
- This interactive python mode can be initiated by selecting python->IDLE from either the task bar or menu bar
- The first three lines are the information about the python version number and Operating System

Eg:

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

IDLE:

- Stands for Integrated Development and Learning Environment

FEATURES:

- Coded in pure python
- Works same on all platforms
- It has a window with colouring of code Input, Output, Error Messages
- The Interactive Interpreter uses primary prompt(>>>) and for continuation of lines we use secondary prompt(…)

SCRIPT MODE:

Python Script ends with .py extension

- **Select Filename from the shell window**
- **Enter the python expressions or statements on separate lines**
- **Save the file using File-> save->Filename.py**
- **Run the code by pressing F5 or selecting RUN Module from Run Menu**

VALUES AND TYPES:

A **value** is one of the basic things a program works with, like a letter or a number. **Type** is a category of values. These values belong to different **types**

1. **Integer -** A type that represents whole numbers
2. **Floating Point -** A type that represents numbers with fractional parts
3. **Strings -** A type that represents sequences of characters.

The Built-in Function to know the type of value is type()

Example:

```
type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

OPERATIONS AND EXPRESSION:

VARIABLES:

A name that refers to a value.

RULES FOR DEFINING A VARIABLE: (2 marks)

- **Variables names must start with a letter or an underscore**, such as:
  - **_underscore**
  - **underscore_**
- **The remainder of your variable name may consist of letters, numbers and underscores.**
  - **password1**
  - **n00b**
  - **un_der_scores**
- **Keywords cannot be a variable.**
  - **Print, input, case, if ,else**
- **Names are case sensitive**.
  - **case_sensitive, CASE_SENSITIVE, and Case_Sensitive are each a different variable**.

EXPRESSION:

An **expression** is a combination of values, variables, and operators.

STATEMENT:

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

ORDER OF OPERATIONS: (4 marks)

When an expression contains more than one operator, the order of evaluation depends on the **order of operations**. For mathematical operators, Python follows mathematical convention.

The acronym **PEMDAS** is a useful way to remember the rules:

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

• Exponentiation has the next highest precedence, so 1 + 2**3 is 9, not 27, and 2 * 3**2 is 18, not 36.

• Multiplication and Division have higher precedence than Addition and Subtraction .
So 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.
• Operators with the same precedence are evaluated from left to right (except exponentiation)

EXERCISES:

a)5 % 10 < 10 and -25 > 1 * 8 // 5
Ans: False
b)7 ** 2 < = 5 // 9 % 3 or 'bye' &'Bye'
Ans: False
c) 15 &22
Ans: 6
d) (7 – 4 * 2) * 10 / 5 ** 2 + 15
Ans:14.6

COMMENTS:

As programs get bigger and more complicated, they get more difficult to read. It is often 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**
Example:

 # compute the percentage of the hour that has elapsed

FUNCTIONS:

Functions are "**self contained**" modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. Once a function is written, it can be used over and over and over again.

STEPS:

1. **Function Definition**
2. **Function call**

Function Definition

'**def**' is the keyword for defining a function. It should be followed by a function name. The rules for defining a variable name are similar for function name also. **function definition** specifies the name of a new function and the sequence of statements that run when the function is called

Syntax for Function Definition:

def function_name():

.

.

The empty parentheses after the name indicate that this function doesn't take any arguments.

Function call:

A statement that runs a function. It consists of the function name followed by an argument list in parentheses

function_name()

Local Variable:

A variable defined **inside a function**. A local variable can only be used inside its function

return value:

It is **the result of a function**. If a function call is used as an expression, the return value is the value of the expression.

Order 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, A **call to a function jumps to the body of the function,** runs the statements there, and then comes back to pick up where it left off.

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!

Parameters and arguments

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

```
def print_twice(bruce):
print(bruce)
print(bruce)
```

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

FUNCTION PROGRAM WITH NO ARGUMENTS:

```
def add(): #no arguments
   print("Enter the value for a")
   a=int(input())
   print("Enter the value for b")
   b=int(input()
   c=a+b
   print(c)

add()
```

OUTPUT:

```
Enter the value for a
10
Enter the value for b
20
30
```

FUNCTION WITH ARGUMENTS BUT NO RETURN VALUE

```
def add(a,b):#arguments are passed
   c=a+b
   print(c)

print("Enter the value for a")
a=int(input())
print("Enter the value for b")
b=int(input()
add(a,b)
```

OUTPUT:

```
Enter the value for a
10
Enter the value for b
20
30
```

FUNCTION WITH ARGUMENTS AND RETURN VALUE

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

```
    c=a+b
    return c
print("Enter the value for a")

a=int(input())
print("Enter the value for b")
b=int(input()
c=add(a,b)
print(c)
```

OUTPUT:

```
Enter the value for a
10
Enter the value for b
20
30
```

*Note: Please refer to the class notes for more example programs

PYTHON PROGRAM TO EXCHANGE TWO VALUES

```
def swa(num1,num2):
  #swapping with temporary variable
   temp=num1
   num1=num2
   num2=temp
   #print the result
   print("Swapped Numbers")
   print(num1)
   print(num2)

print("Enter the first number")  #input first number
n1=int(input())
print("Enter the second number") #input second number
n2=int(input())
swa(n1,n2)    #calling Function
```

OUTPUT:

```
Enter the first number
10
Enter the second number
```

```
20
Swapped Numbers
20
10
```

DISTANCE BETWEEN TWO POINTS

```
def dist(x1,y1,x2,y2):
    d1=(x2-x1)**2
    d2=(y2-y1)**2
    #compute distance
    distance=((d1)+(d2))**2
    print("The distance between two points",distance)

print("Enter the value of x1")       #input First number
x1=int(input())
print("Enter the value of y1")       #input Second number
y1=int(input())
print("Enter the value of x2")       #input third number
x2=int(input())
print("Enter the value of y2")       #input Fourth number
y2=int(input())
dist(x1,y1,x2,y2)       #function call
#circulate the values of n variables,

def rotate(l,order):
    for i in range(0,order):
        j=len(l)-1
        while(j>0):
            #swapping
            t=l[i]
            l[i]=l[j]
            l[j]=t
            j=j-1
        print(i,'rotation',l)
    return
L=[10,20,30,40,50]    #input List
rotate(L,3)           #function call
```

OUTPUT:

```
0 rotation [20, 30, 40, 50, 10]
1 rotation [20, 40, 50, 10, 30]
2 rotation [20, 10, 40, 30, 50]
```

UNIT III CONTROL FLOW, FUNCTIONS 9

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

UNIT 3

Conditionals:

Boolean values:

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

A string in Python can be tested for truth value.

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

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

```
str="Hello World"

print str.isalnum()          #False     #check if all char are numbers

print str.isalpha()          #False     #check if all char in the string are alphabetic

print str.isdigit()          #False     #test if string contains digits

print str.istitle()          #True      #test if string contains title words

print str.isupper()          #False     #test if string contains upper case

print str.islower()          #False     #test if string contains lower case

print str.isspace()          #False     #test if string contains spaces

print  str.endswith('d') #True      #test if string endswith a d

print str.startswith('H')          #True     #test if string startswith H
```

The if statement

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

if BOOLEAN EXPRESSION:

    STATEMENTS

A few important things to note about if statements:

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

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

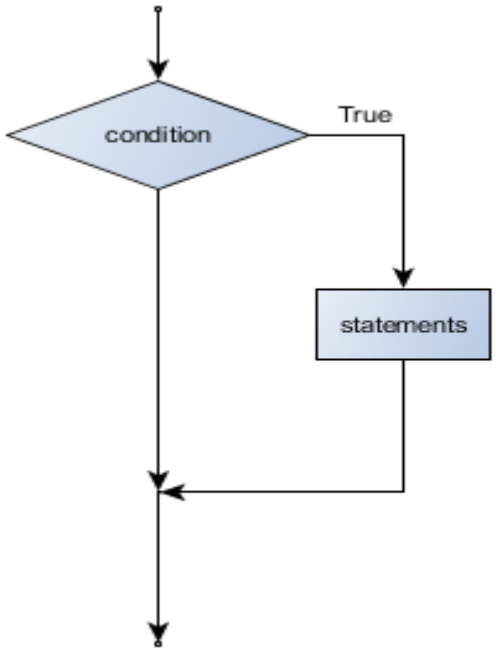
All lines indented the same amount after the colon will be executed whenever the BOOLEAN_EXPRESSION is true.

Here is an example:

If a>0:

 Print (“a is positive”)

Flow Chart:



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

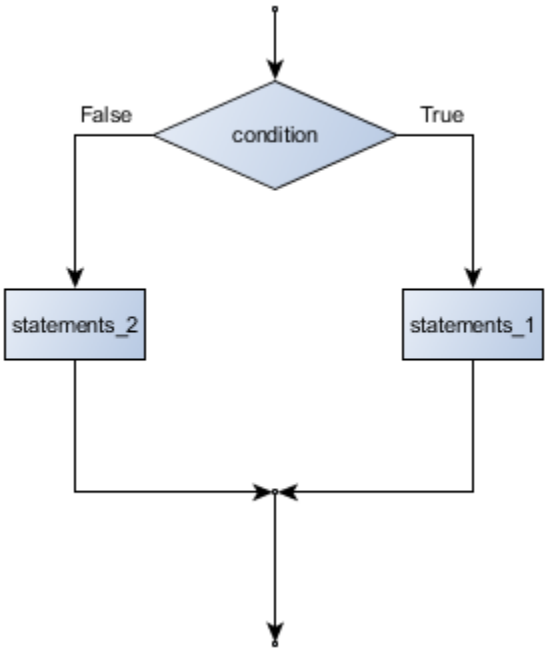
The indented statements that follow are called a block. The first unintended statement marks the end of the block. Each statement inside the block must have the same indentation.

if else statement

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

```
if  a>0:

 print(“ a is positive”)

else:

 print(“ a is negative”)
```

Flow Chart



The syntax for an if else statement looks like this:

```
if BOOLEAN EXPRESSION:
    STATEMENTS_1        # executed if condition evaluates to True
else:
    STATEMENTS_2        # executed if condition evaluates to False
```

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

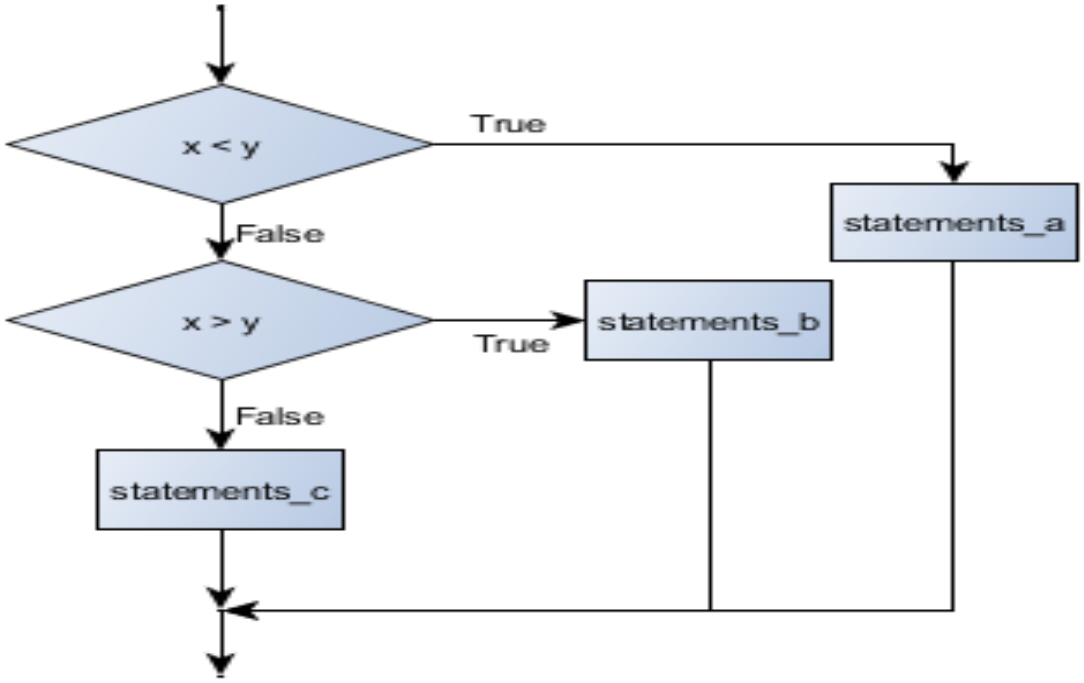
```
if True:          # This is always true
    pass          # so this is always executed, but it does nothing
else:
    pass
```

Chained Conditionals

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

```
if x < y:
    STATEMENTS_A
elif x > y:
    STATEMENTS_B
else:
    STATEMENTS_C
```

Flowchart of this chained conditional



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

```
if choice == '1':
    print("Monday.")
elif choice == '2':
    print(“Tuesday")
elif choice == '3':
    print("Wednesday")
else:
    print("Invalid choice.")
```

Another Example

```
def letterGrade(score):
    if score >= 90:
```

```
        letter = 'A'
    else:   # grade must be B, C, D or F
        if score >= 80:
            letter = 'B'
        else:  # grade must be C, D or F
            if score >= 70:
                letter = 'C'
            else:    # grade must D or F
                if score >= 60:
                    letter = 'D'
                else:
                    letter = 'F'
    return letter
```

Program1:

Largest of two numbers

```
x1 = int(input("Enter two numbers: "))
x2 = int(input("Enter two numbers: "))
if x1>x2:
   print("x1 is big")
else:
   print("x2 is big")
```

Program2:

Largest of three numbers

```
num1 = int(input("enter first number"))
num2 = int(input("enter second number"))
```

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

Program 3: Roots of a quadratic equation

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

Program 4: Odd or even

```
x=int(input("enter a number"))
if x%2==0:
    print("it is Even ")
else:
    print("it is Odd")
```

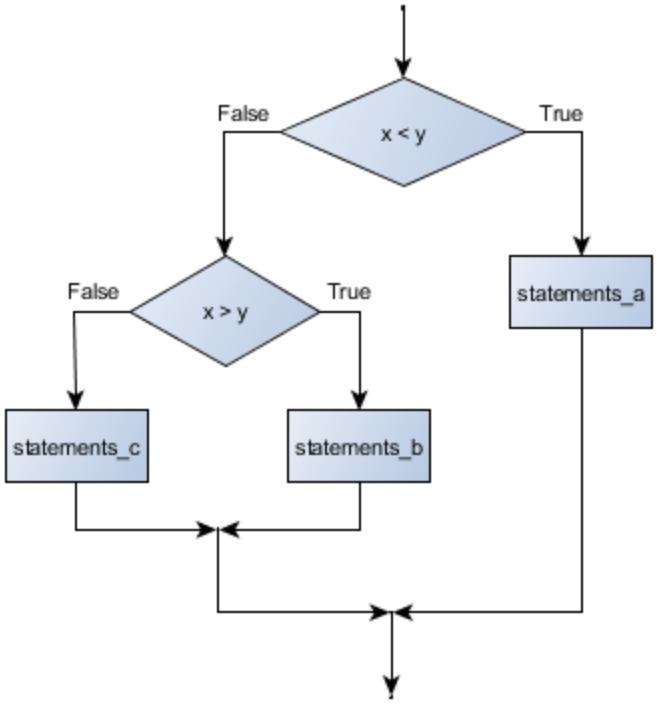
Program 4: Positive or negative

```
x=int(input("enter a number"))
if x>0:
    print("it is positive")
else:
    print("it is negative")
```

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

Nested Conditionals

Flowchart




```
if x < y:
    STATEMENTS_A
else:
    if x > y:
        STATEMENTS_B
    else:
        STATEMENTS_C
```

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

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

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

Iteration

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

While loop

While statement

The general syntax for the while statement looks like this:

While BOOLEAN_EXPRESSION:

    STATEMENTS

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

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

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

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

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

Program for Sum of digits

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

s=0

while(n>0):

    r=n%10

    s=s+r

    n=n/10

print("sum of digits",int(s))
```

OUTPUT

```
enter n 122

sum of digits 5
```

Program to check Armstrong Number

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

Program to check for Number Palindrome

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

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

for loop

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

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

The general form of a for loop is:

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

```
>>> for i in range(5):
...     print('i is now:', i)
i is now 0
i is now 1
i is now 2
i is now 3
i is now 4
>>>
```

Example 2:

```
for x in range(13):   # Generate numbers 0 to 12
    print(x, '\t', 2**x)
```

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

```
0       1
1       2
```

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

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

Program for Converting Decimal to Binary

```
def convert(n):
      if n > 1:
       convert(n//2)
   print(n % 2,end = '')
#  enter decimal number
dec=int(input("enter n"))
```

```
convert(dec)
```

OUPUT

```
enter n  25
11001
```

Program for Converting Decimal to Binary, Octal, HexaDecimal

```
dec = int(input("enter n"))
print("The decimal value of",dec,"is:")
print(bin(dec),"in binary.")
print(oct(dec),"in octal.")
print(hex(dec),"in hexadecimal.")
```

OUTPUT

```
enter n 25
The decimal value of 25 is:
0b11001 in binary.
0o31 in octal.
0x19 in hexadecimal.
```

Program for finding the Factorial

```
n= int(input("enter a number"))
f = 1
for i in range(1,n+1):
   f = f*i
print("The factorial of",n,"is",f)
```

OUTPUT

```
enter a number 7
The factorial of 7 is 5040
```

Prime Number Generation between different intervals:

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

OUTPUT

```
enter start value 0
enter end value25
Prime numbers between 0 and  25 are:
2
3
5
7
11
13
17
19
23
```

Program for swapping two numbers:

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

Program for swapping two numbers without using temporary variable

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

Program for Four Function Calculator

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

```
elif ch == '*':
  res = n1 * n2
  print(n1, "*", n2, "=", res)
elif ch == '/':
  res = n1 / n2
  print(n1, "/", n2, "=", res)
else:
  print("enter a valid operator")
```

Fruitful function

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

```
def cal(r1):
    return 3.14*r1*r1
r=int(input("enter r"))
print("ans=",cal(r))
```

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

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

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

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

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

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

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

Program for Distance Calculation

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

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

Fibonocci using recursion and fruitful function

```
def recur_fibo(n):

   if n <= 1:

      return n

  else:

      return(recur_fibo(n-1) + recur_fibo(n-2))

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

if nterms <= 0:

  print("Plese enter a positive integer")

else:

  print("Fibonacci sequence:")
```

```
for i in range(nterms):
    print(recur_fibo(i))
```

Function Composition in python

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

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

```
def compose2(f, g):
    return lambda x: f(g(x))
```

Example:

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

Syntax

```
pass
```

Example

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

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

Output

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

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

Local and Global Scope variables

Local Variables

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

```
def f():
```

```
    print(s)

s = "I love Paris in the summer!"

f()
```

Output:

I Love Paris in the summer.

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

```
def f():
    s = "I love London!"
    print(s)

s = "I love Paris!"

f()

print(s)
```

Output:

I love London!

I love Paris!

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

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

For Example,

```
def f():
  print(s)
  s = "I love London!"
  print(s)
s = "I love Paris!"
f()
```

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

To avoid such Problem,

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

```
def f():
    global s
    print(s)
    s = "Only in spring, but London is great as well!"
    print(s)


s = "I am looking for a course in Paris!"
f()
print(s)
```

Output:

I am looking for a course in Paris!

Only in spring, but London is great as well!

Only in spring, but London is great as well!

Recursive Functions:

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

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

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

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

Output:

The factorial of a given no is:120

2. **Finding nth Fibonacci Number**

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

Output:

The nth Fibonacci series is 55.

Strings

A string is a sequence of characters.

fruit='banana'

fruit[1]

Out[3]: 'a'

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

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

Len

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

len(fruit)

Out[4]: 6

String Slices:

A segment of a string is called string slices.

Fruit=’banana’

Fruit[1:4]

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

Fruit[:3]

It includes from the first character. It produces the output as ‘ban’.

Fruit[3:]

It includes the end of the string. It returns output as ‘ana’

Fruit[3:3]

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

Fruit[:]

It returns the entire string. Out put is ‘banana’.

Fruit[-1]

It returns the last character in a string. Output ‘a’.

String Immutable

We cant change the value of the strings.

```
fruit[0]='w'
Traceback (most recent call last):
  File "<ipython-input-10-839a456c8838>", line 1, in <module>
    fruit[0]='w'
TypeError: 'str' object does not support item assignment
```

Strings are immutable.

String Methods

Built-in String Methods

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

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

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

6	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.

The string module

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

Example: Using the string module

```
# File: string-example-1.py
import string
text = "Monty Python's Flying Circus"
print "upper", "=>", string.upper(text)
print "lower", "=>", string.lower(text)
print "split", "=>", string.split(text)
print "join", "=>", string.join(string.split(text), "+")
print "replace", "=>", string.replace(text, "Python", "Java")
print "find", "=>", string.find(text, "Python"), string.find(text, "Java")
print "count", "=>", string.count(text, "n")
```

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

Example: Using string methods instead of string module functions

```
# File: string-example-2.py
text = "Monty Python's Flying Circus"
print "upper", "=>", text.upper()
print "lower", "=>", text.lower()
print "split", "=>", text.split()
print "join", "=>", "+".join(text.split())
print "replace", "=>", text.replace("Python", "Perl")
print "find", "=>", text.find("Python"), text.find("Perl")
```

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

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

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

Example: Using the string module to convert strings to numbers

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

print int("4711"),
print string.atoi("4711"),
print string.atoi("11147", 8), # octal
print string.atoi("1267", 16), # hexadecimal
print string.atoi("3mv", 36) # whatever...

print string.atoi("4711", 0),
print string.atoi("04711", 0),
print string.atoi("0x4711", 0)

print float("4711"),
print string.atof("1"),
print string.atof("1.23e5")
```

```
4711 4711 4711 4711 4711
```

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

Programs

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

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

Output:

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

3. **Searching a letter present in word**

```
def find(word,letter):
    index=0
    while index<len(word):
        if word[index]==letter:
            return index
        index=index+1
    return -1
print("the letter present", find('malayalm','l'))
```

Output:

The letter present 2

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

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

Output:

The total count for the given letter 2

3. String Palindrome

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

Output:

Not Palindrome.

5. Write a Program for GCD of two numbers

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

Output:

GCD of two numbers 12

6. Exponent of a number

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

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

Output

25

UNIT IV LISTS, TUPLES, DICTIONARIES 9

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

LISTS, TUPLES, DICTIONARIES

4.1 LISTS:

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

Elements or Items:

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

Examples:

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

Nested List:

A list within another list is called nested list.

Example:

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

Empty List:

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

Assigning List Values to Variables:

Example:

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

Output:

```
[40,12]['x','y']
```

4.1.1 LIST OPERATIONS:

Example 1: The + operator concatenates lists:

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

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

Example 2:

The * operator repeats a list given number of times:

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

Output:

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

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

Output:

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

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

4.1.2 LIST SLICES:

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

Syntax:

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

Example:

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

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

Example:

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

Output:

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

4.1.3 LIST METHODS:

Python provides methods that operate on lists.

Example:

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

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

Output:

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

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

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

Output:

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

t2 will be unmodified.

4.1.4 LIST LOOP:

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

Example 1:

```
>>>mylist=[[1,2],[4,5]]
>>>for x in mylist:
        if len(x)==2:
            print x
```

Output:

```
[1,2]
[4,5]
```

Example 2:

```
>>>for i in range(len(numbers)):
        Numbers[i]=numbers[i]*2
```

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

Example:

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

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

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

Example 3:

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

Output:

```
red
green
blue
purple
```

4.1.5 MUTABILITY:

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

Example for mutable data types are:

List, Set and Dictionary

Example 1:

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

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

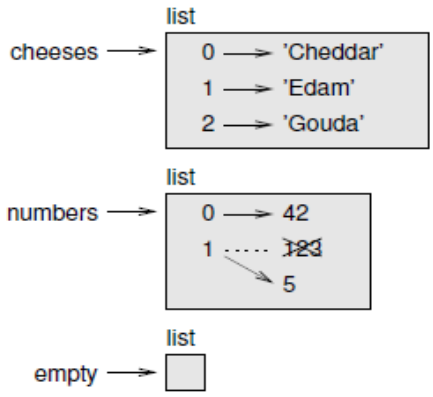


Figure 4.1 shows the state diagram for cheeses, numbers and empty:

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

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

The in operator also works on lists.

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

4.1.6 ALIASING

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

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

The state diagram looks like Figure 4.2.

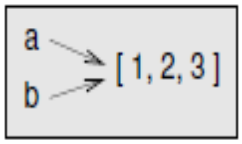


Figure 4.2 State Diagram

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

4.1.7 CLONING LISTS Copy list v Clone list

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

Output:

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

4.1.8 LIST PAPRAMETERS

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

Example:

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

Here's how it is used:

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

Output:

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

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

Example 1:

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

Output:

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

Example 2:

```
>>> t2
```

Output:

None

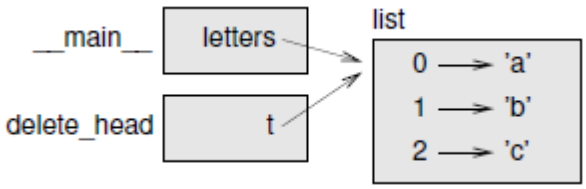


Figure 4.3 Stack Diagram

4.2. TUPLES

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

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

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

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

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

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

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

Output:

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

A value in parentheses is not a tuple:

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

Output:

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

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

Example:

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

Output: ()

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

Example:

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

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

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

Example:

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

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

Example:

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

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

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

TypeError: object doesn't support item assignmentBecause tuples are immutable, we can't modify the elements. But we can replace one tuple with another:

Example:

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

Output:

('A', 'b', 'c', 'd', 'e')This statement makes a new tuple and then makes t refer to it.

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

Example 1:

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

Output:

True

Example 2:

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

Output:

True

4.2.1 TUPLE ASSIGNMENT

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

Example:

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

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

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

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

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

ValueError: too many values to unpack

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

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

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

```
>>> uname
'monty'
>>> domain
'python.org'
```

4.2.2 TUPLE AS RETURN VALUES

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

Example:

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

Ouput:

```
(2, 1)
```

Or use tuple assignment to store the elements separately:

Example:

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

Output:

```
2
```

Example:

```
>>> rem
```

Output:

```
1
```

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

```
def min_max(t):
        return min(t), max(t)
```

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

4.3 DICTIONARIES

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

4.3.1 OPERATIONS AND METHODS

Example:

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
```

Output:

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

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

Example:

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

Output:

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

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

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

Example:

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

Output:

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

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

Example:

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

Output:

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

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

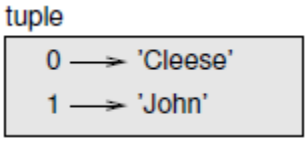
The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary. For last, first in directory:

print(first, last, directory[last,first])

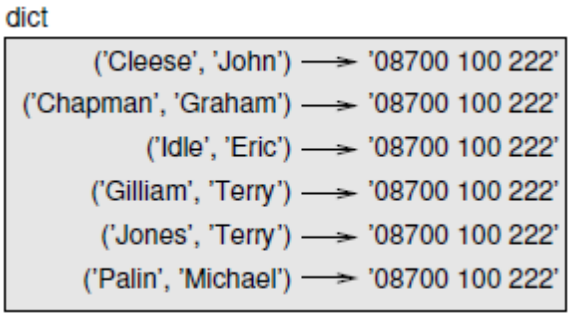
This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

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

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



4.4 State Diagram



4.5 State Diagram

4.4 ADVANCED LIST PROCESSING

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

4.4.1 LIST COMPREHENSION

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

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

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

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

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

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

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

We can rewrite it using a list comprehension

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

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

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

4.5 ILLUSTRATIVE PROGRAMS SELECTION SORT

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

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

ALGORITHM:

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

PROGRAM:

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

Output

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

4.5.2 INSERTION SORT

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

ALGORITHM:

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

PROGRAM:

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

```
        #swap
        a[j-1],a[j] = a[j],a[j-1]
    else:
        #in order
        break
    j = j-1
print (a)
```

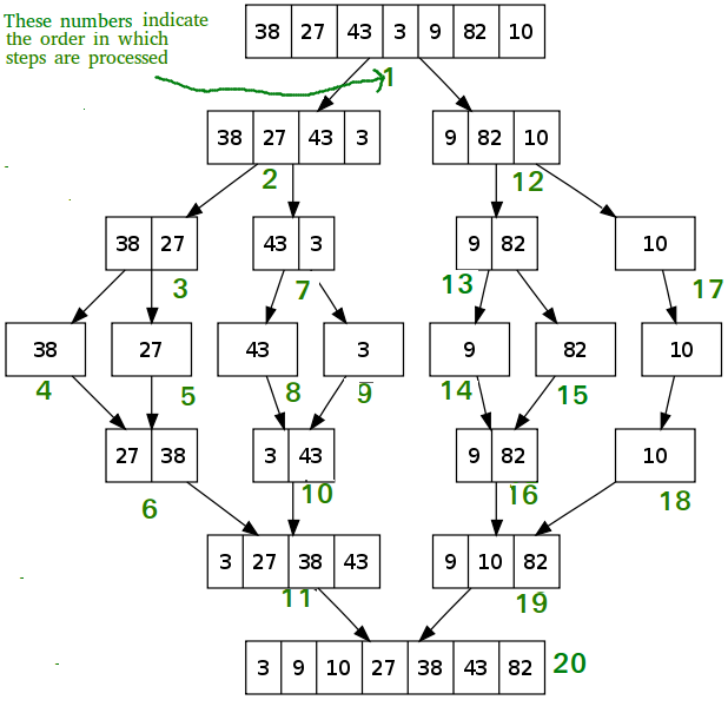
Output

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

4.5.3 MERGE SORT

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

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



ALGORITHM:

MergeSort(arr[], l, r)
If r > l

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

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

PROGRAM:

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

Input:

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

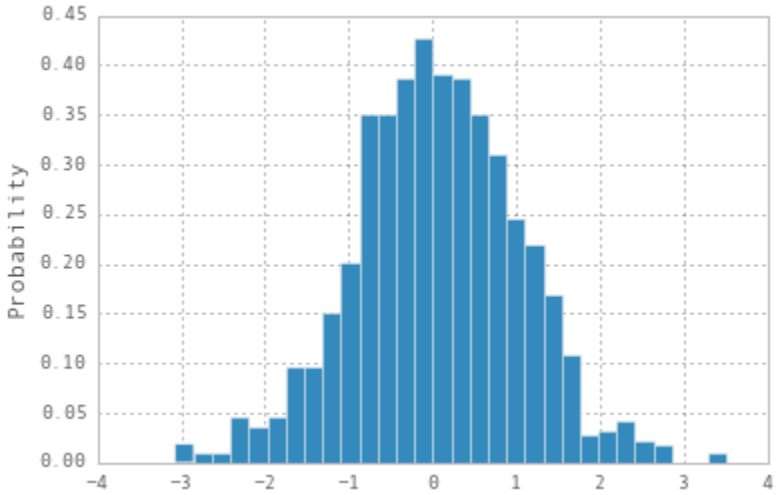
Output:

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

4.5.5 HISTOGRAM

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

Example:




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

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

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

Example:

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

Output:

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

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

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

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

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

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

Here's what the output looks like:

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

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

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

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

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

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

TWO MARKS

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

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

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

2. What is a nested list? Give example.

   A list within another list is called nested list.
   Example:

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

3. What is a empty list?

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

4. What is list slicing? Write its syntax.

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

   Listname [start:end:step]

5. What is mutability? What is its use?

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

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

Example for mutable data types are:

List, Set and Dictionary

Example 1:

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

Output: [42,5]

7. What is aliasing? Give example.

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

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

Example:

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

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

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

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

9. Give example for copying and cloning lists.

Example:

```
veggies=["potatoes","carrots","pepper","parsnips","swedes","onion","min
ehead"]
veggies[1]="beetroot"
Copying a list
        daniel=veggies
CLONES a list
        david=veggies[:]
daniel[6]="emu"
```

10. Define Dictionaries. Give example.

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

Example:

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
```

Output:

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

11. What is list processing?

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

UNIT 5
FILES, MODULES AND PACKAGES

Files: text files, reading and writing files, format operator, command line arguments, **Errors and Exceptions**: handling exceptions, **Modules, Packages**; Illustrative programs: word count, copy file.

File:

- File is a named location on disk to store related information. It is used to permanently store data in a memory (e.g. hard disk).

File Types:

1. Text file
2. Binary file

Text File	Binary file
Text file is a sequence of characters that can be sequentially processed by a computer in forward direction.	A binary files store the data in the binary format (i.e. 0's and 1's)
Each line is terminated with a special character, called the EOL or End of Line character	It contains any type of data (PDF , images , Word doc ,Spreadsheet, Zip files,etc)

Operations on Files:

In Python, a file operation takes place in the following order,

1. Opening a file
2. Reading / Writing file
3. Closing the file

How to open a file:

Syntax:	Example:
file_object=open("file_name.txt","mode")	f=open("sample.txt","w")

How to create a file:

Syntax:	Example:
file_object=open("file_name.txt","mode") file_object.write(string) file_object.close()	f=open("sample.txt","w") f.write("hello") f.close()

Modes in file:

modes	description
r	read only mode
w	write only
a	appending mode
r+	read and write mode
w+	write and read mode

Differentiate write and append mode:

write mode	append mode
It is use to write a string into a file.	It is used to append (add) a string into a file.
If file is not exist it creates a new file.	If file is not exist it creates a new file.
If file is exist in the specified name, the existing content will overwrite in a file by the given string.	It will add the string at the end of the old file.

File operations and methods:

S.No	Syntax	Example	Description
1	f.write(string)	f.write("hello")	Writing a string into a file.
2	f.writelines(sequence)	f.writelines("1st line \n second line")	Writes a sequence of strings to the file.
3	f.read(size)	f.read( ) #read entire file f.read(4) #read the first 4 charecter	To read the content of a file.
4	f.readline( )	f.readline( )	Reads one line at a time.
5	f.readlines( )	f.readlines( )	Reads the entire file and returns a list of lines.
6	f.seek(offset,whence) whence value is optional.	f.seek(0)	Move the file pointer to the appropriate position. It sets the file pointer to the starting of the file.
	whence =0 from begining	f.seek(3,0)	Move three character from the beginning.
	whence =1 from current position	f.seek(3,1)	Move three character ahead from the current position.
	whence =2 from last position	f.seek(-1,2)	Move to the first character from end of the file
7	f.tell( )	f.tell( )	Get the current file pointer position.
8	f.flush( )	f.flush( )	To flush the data before closing any file.
9	f.close( )	f.close( )	Close an open file.
10	f.name	f.name o/p: 1.txt	Return the name of the file.

11	f.mode	f.mode o/p: w	Return the Mode of file.
12	os.rename(old name,new name)	import os os.rename("1.txt","2.txt")	Renames the file or directory.
13	os.remove(file name)	import os os.remove("2.txt")	Remove the file.

Format operator

The argument of write( ) has to be a string, so if we want to put other values along with the string in a file, we have to convert them to strings.

Convert no into string:	**output**
>>> x = 52 >>> f.write(str(x))	"52"
Convert to strings using format operator, %	**Example:**
print ("format string"%(tuple of values)) file.write("format string"%(tuple of values)	>>>age=13 >>>print("The age is %d"%age) The age is 13
Program to write even number in a file using format operator	**OutPut**

f=open("t.txt","w")
n=eval(input("enter n:"))
for i in range(n):
  a=int(input("enter number:"))
  if(a%2==0):
  f.write(a)
f.close()
	enter n:4
	enter number:3
	enter number:4
	enter number:6
	enter number:8
	result in file t.txt
	4
	6
	8

- The first operand is the format string, which specifies how the second operand is formatted.
- The result is a string. For example, the format sequence '%d' means that the second operand should be formatted as an integer (d stands for "decimal"):

Format character	**Description**
%c	Character
%s	String formatting
%d	Decimal integer
%f	Floating point real number

Command line argument:

- ❖ The command line argument is used to pass input from the command line to your program when they are started to execute.
- ❖ Handling command line arguments with Python need sys module.
- ❖ sys module provides information about constants, functions and methods of the pyhton interpretor.

argv[ ] is used to access the command line argument. The argument list starts from 0.
sys.argv[0]= gives file name
sys.argv[1]=provides access to the first input

Example 1	**output**
import sys print("the file name is %s" %(sys.argv[0]))	python demo.py the file name is demo.py
addition of two num	**output**
import sys a= sys.argv[1] b= sys.argv[2] sum=int(a)+int(b) print("sum is",sum)	sam@sam~$ python sum.py 2 3 sum is 5
Word count using comment line arg:	**Output**

from sys import argv
a = argv[1].split()
dict = {}
for i in a:
  if i in dict:
  dict[i]=dict[i]+1
  else:
  dict[i] = 1
print(dict)
print(len(a))
	C:\Python34>python word.py
	"python is awesome lets program in python"
	{'lets': 1, 'awesome': 1, 'in': 1, 'python': 2, 'program': 1, 'is': 1}
	7

Errors and exception:

Errors

Errors are the mistakes in the program also referred as bugs. They are almost always the fault of the programmer. The process of finding and eliminating errors is called debugging. Errors can be classified into three major groups:

1. Syntax errors
2. Runtime errors
3. Logical errors

Syntax errors

- Syntax errors are the errors which are displayed when the programmer do mistakes when writing a program.
- When a program has syntax errors it will not get executed.
- Common Python syntax errors include:
    1. leaving out a keyword
    2. putting a keyword in the wrong place
    3. leaving out a symbol, such as a colon, comma or brackets
    4. misspelling a keyword
    5. incorrect indentation
    6. empty block

Runtime errors:

- If a program is syntactically correct – that is, free of syntax errors – it will be run by the Python interpreter.
- However, the program may exit unexpectedly during execution if it encounters a runtime error.
- When a program has runtime error I will get executed but it will not produce output.
- Common Python runtime errors include:
    1. division by zero
    2. performing an operation on incompatible types
    3. using an identifier which has not been defined
    4. accessing a list element, dictionary value or object attribute which doesn't exist
    5. trying to access a file which doesn't exist

Logical errors:

- Logical errors are the most difficult to fix.
- They occur when the program runs without crashing, but produces an incorrect result.
- Common Python logical errors include:
    1. using the wrong variable name
    2. indenting a block to the wrong level
    3. using integer division instead of floating-point division
    4. getting operator precedence wrong
    5. making a mistake in a boolean expression

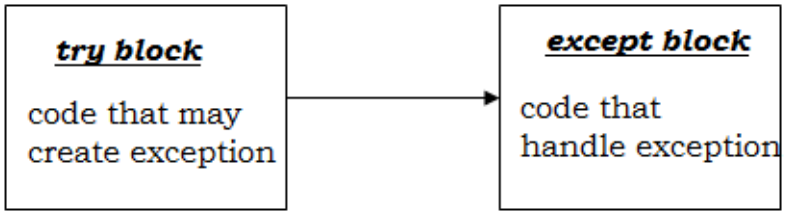
Exceptions:

- An exception(runtime time error)is an error, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates or quit.

S.No.	Exception Name	Description
1	FloatingPointError	Raised when a floating point calculation fails.
2	ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
3	AttributeError	Raised in case of failure of attribute reference or assignment.
4	ImportError	Raised when an import statement fails.
5	KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
6	IndexError	Raised when an index is not found in a sequence
7	KeyError	Raised when the specified key is not found in the dictionary.
8	NameError	Raised when an identifier is not found in the local or global name space
9	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.
10	SyntaxError	Raised when there is an error in Python syntax.
11	IndentationError	Raised when indentation is not specified properly.
12	SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
13	SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
14	TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
15	ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
16	RuntimeError	Raised when a generated error does not fall into any category.

Exception Handling:

- Exception handling is done by try and catch block.
- Suspicious code that may raise an exception, this kind of code will be placed in try block.
- A block of code which handles the problem is placed in except block.



Catching Exceptions:

1. **try…except**
2. **try…except…inbuilt exception**
3. **try… except…else**
4. **try…except…else….finally**
5. **try.. except..except..**
6. **try…raise..except..**

try … except

- In Python, exceptions can be handled using a try statement.
- A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.
- It is up to us, what operations we perform once we have caught the exception. Here is a simple example.

Syntax

```
try:
    code that create exception
except:
    exception handling statement
```

Example:	Output

try:
  age=int(input("enter age:"))
  print("ur age is:",age)
except:
  print("enter a valid age")
	enter age:8
	ur age is: 8
	enter age:f
	enter a valid age

try…except…inbuilt exception

Syntax

```
try:
    code that create exception
except inbuilt exception:
    exception handling statement
```

Example:	Output

try:
  age=int(input("enter age:"))
  print("ur age is:",age)
except ValueError:
  print("enter a valid age")
	enter age:d
	enter a valid age

try ... except ... else clause

- Else part will be executed only if the try block doesn't raise an exception.
- Python will try to process all the statements inside try block. If value error occurs, the flow of control will immediately pass to the except block and remaining statement in try block will be skipped.

Syntax

```
try:
   code that create exception
except:
   exception handling statement
else:
    statements
```

Example program	Output

try:
  age=int(input("enter your age:"))
except ValueError:
  print("entered value is not a number")
else:
  print("your age :",age)
	enter your age: six
	entered value is not a number
	enter your age:6
	your age is 6

try ... except...finally

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.

Syntax

```
try:
   code that create exception
except:
   exception handling statement
else:
  statements
finally:
  statements
```

Example program	Output

try:
  age=int(input("enter your age:"))
except ValueError:
  print("entered value is not a number")
else:
  print("your age :",age)
finally:
  print("Thank you")
	enter your age: six
	entered value is not a number
	Thank you
	enter your age:5
	your age is 5
	Thank you

try...multiple exception:

Syntax

try:
  code that create exception
except:
  exception handling statement
except:
  statements

Example	Output:

a=eval(input("enter a:"))
b=eval(input("enter b:"))
try:
  c=a/b
  print(c)
except ZeroDivisionError:
  print("cant divide by zero")
except ValueError:
  print("its not a number")
	enter a:2
	enter b:0
	cant divide by zero
	enter a:2
	enter b: h
	its not a number

Raising Exceptions

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword raise.

Syntax:

>>> raise error name

Example:	Output:

try:
  age=int(input("enter your age:"))
  if (age<0):
  raise ValueError("Age can't be negative")
except ValueError:
  print("you have entered incorrect age")
else:
  print("your age is:",age)
	enter your age:-7
	Age can't be negative

MODULES:

- A module is a file containing Python definitions ,functions, statements and instructions.
- Standard library of Python is extended as modules.
- To use modules in a program, programmer needs to import the module.
- To get information about the functions and variables supported module you can use the built-in function help(Module name), Eg: help("math").
- The dir() function is used to list the variables and functions defined inside a module. If an argument, i.e. a module name is passed to the dir, it returns that modules variables and function names else it returns the details of the current module. Eg: dir(math)

OS module

- The OS module in python provides functions for interacting with the operating system
- To access the OS module have to import the OS module in our program

import os

method	example	description
name	os.name 'nt'	This function gives the name of the operating system.
getcwd()	os.getcwd() 'C:\\Python34'	returns the Current Working Directory(CWD) of the file used to execute the code.
mkdir(folder)	os.mkdir("python")	Create a directory(folder) with the given name.
rename(oldname, new name)	os.rename("python","pspp")	Rename the directory or folder
remove("folder")	os.remove("pspp")	Remove (delete) the directory or folder.

getuid( )	os.getuid( )	Return the current process's user id.
environ	os.environ	Get the users environment

Sys module

- ❖ Sys module provides information about constants, functions and methods.
- ❖ It provides access to some variables used or maintained by the interpreter.

import sys

method	example	description
sys.argv	sys.argv	Provides The list of command line arguments passed to a Python script
	sys.argv[0]	Provides to access the file name
	sys.argv[1]	Provides to access the first input
sys.path	sys.path	It provides the search path for modules
sys.path.append()	sys.path.append()	Provide the access to specific path to our program
sys.platform	sys.platform 'win32'	Provides information about the operating system platform
sys.exit	sys.exit <built-in function exit>	Exit from python

Steps to create the own module

Here we are going to create calc module: our modules contains four functions (i.e) add(),sub(),mul(),div()

Program for calculator module	output

Module Name: calc.py
def add(a,b):
  print(a+b)
def sub(a,b):
  print(a-b)
def mul(a,b):
  print(a*b)
def div(a,b):
  print(a/b)
	import calculator
	calculator.add(2,3)
	
	Output
	>>> 5

Package:

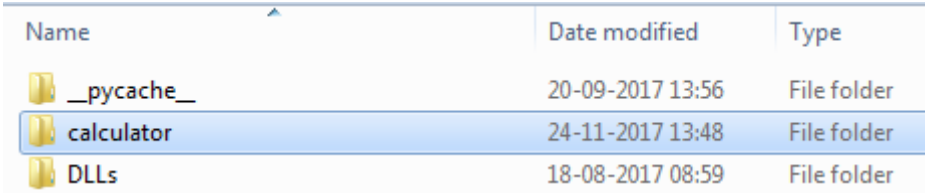
- A package is a collection of Python modules. Module is a single Python file containing function definitions; a package is a directory (folder) of Python modules containing an additional _ _init_ _.py file, to differentiate a package from a directory.
- Packages can be nested to any depth, provided that the corresponding directories contain their own _ _init_ _.py file.
- _ _init_ _.py file is a directory indicates to the python interpreter that the directory should be treated like a python package._ _init_ _.py is used to initialize the python package.

Steps to create a package

Step 1: Create the Package Directory

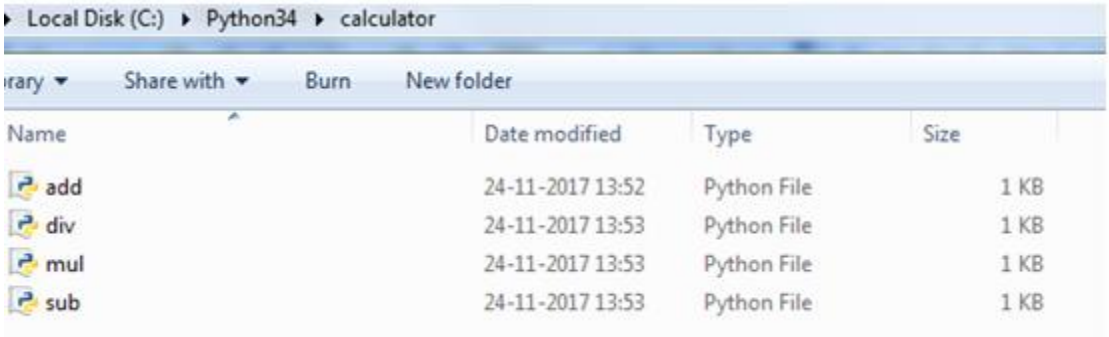
Create a directory(folder) and give it your package's name.

Here the package name is calculator.



Step 2: write Modules for calculator directory add save the modules in calculator directory.

Here four modules have created for calculator directory.

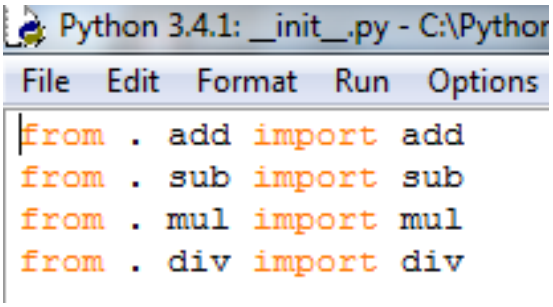


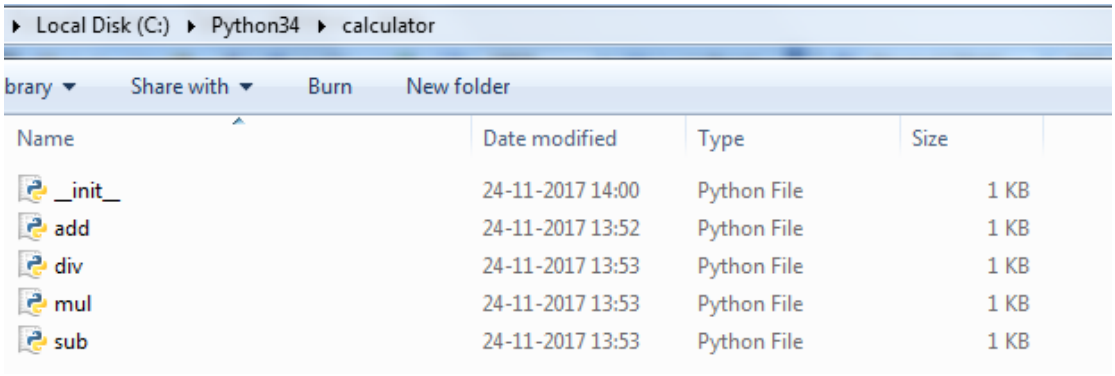
add.py	sub.py	mul.py	div.py
def add(a,b): print(a+b)	def sub(a,b): print(a-b)	def mul(a,b): print(a*b)	def div(a,b): print(a/b)

Step 3: Add the _ _init_ _.py File in the calculator directory

A directory must contain a file named _ _init_ _.py in order for Python to consider it as a package.

Add the following code in the __init__.py file

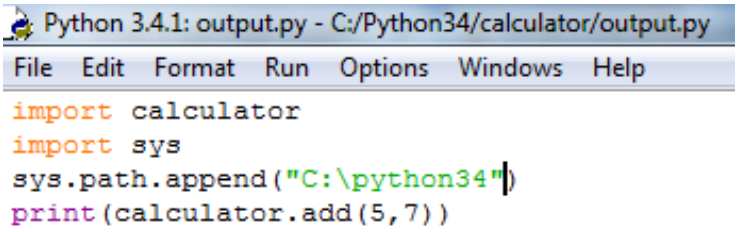




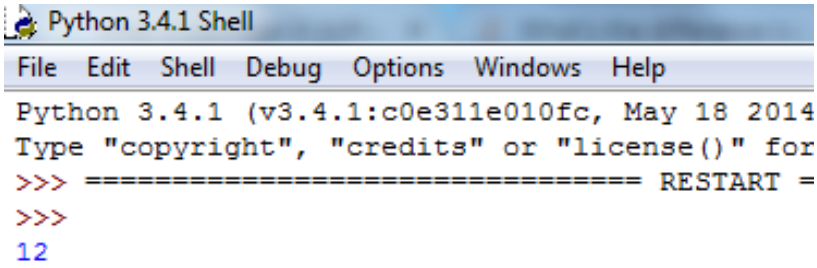
Step 4:To test your package.

Import calculator package in your program and add the path of your package in your program by using sys.path.append().

Here the path is "C:\python34"



OUTPUT:



Illustrative Programs in unit 5:

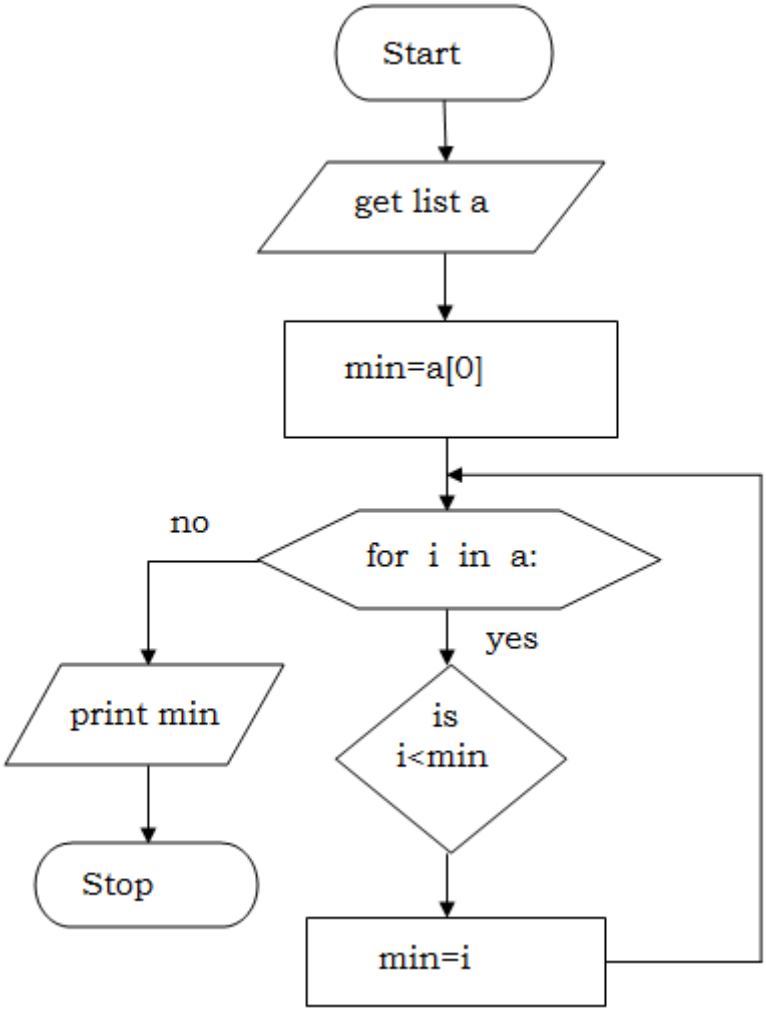
Word count	**Output**

from sys import argv
a = argv[1].split()
dict = {}
for i in a:
  if i in dict:
  dict[i]=dict[i]+1
  else:
  dict[i] = 1
print(dict)
print(len(a))
	C:\Python34>python word.py
	"python is awesome lets program in python"
	{'lets': 1, 'awesome': 1, 'in': 1, 'python': 2, 'program': 1, 'is': 1}
	7

Copy a file	**Output**
f1=open("1.txt","r") f2=open("2.txt","w") for i in f1: f2.write(i) f1.close() f2.close()	no output internally the content in f1 will be copied to f2
copy and display contents	**Output**
f1=open("1.txt","r") f2=open("2.txt","w+") for i in f1: f2.write(i) f2.seek(0) print(f2.read()) f1.close() f2.close()	hello welcome to python programming (content in f1 and f2)

Illustrative problems in Unit 1:

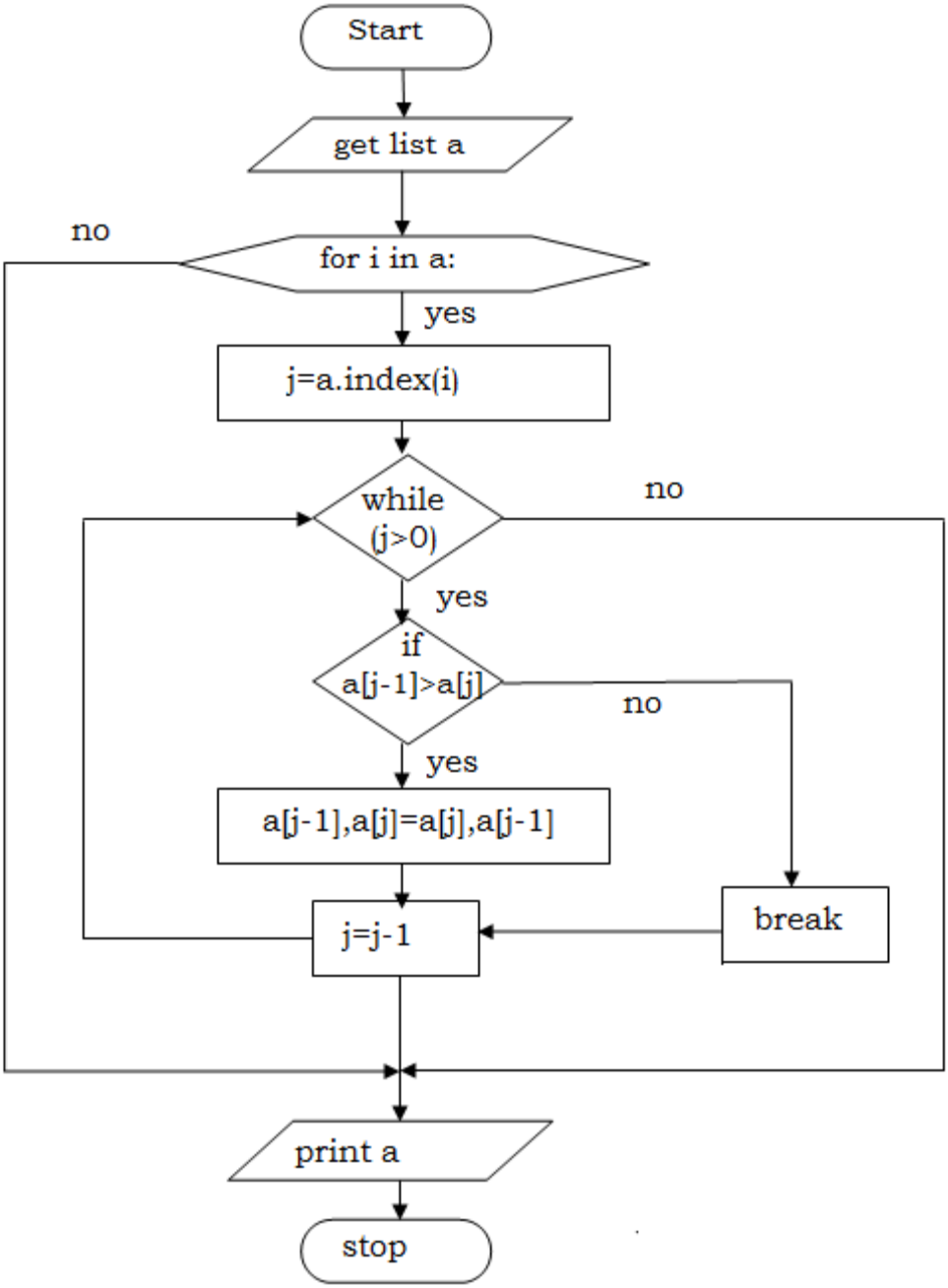
Minimum element in a list
Step 1: Start Step 2: Get a list “a” step 3: assign the first element of the list as min Step 4: for each element(i) in “a” goto step5 else goto step 6 Step 5: if(i<min) then min=i else goto step 4 step 6: print min value Step 8: Stop



Insert a card in a list of sorted cards

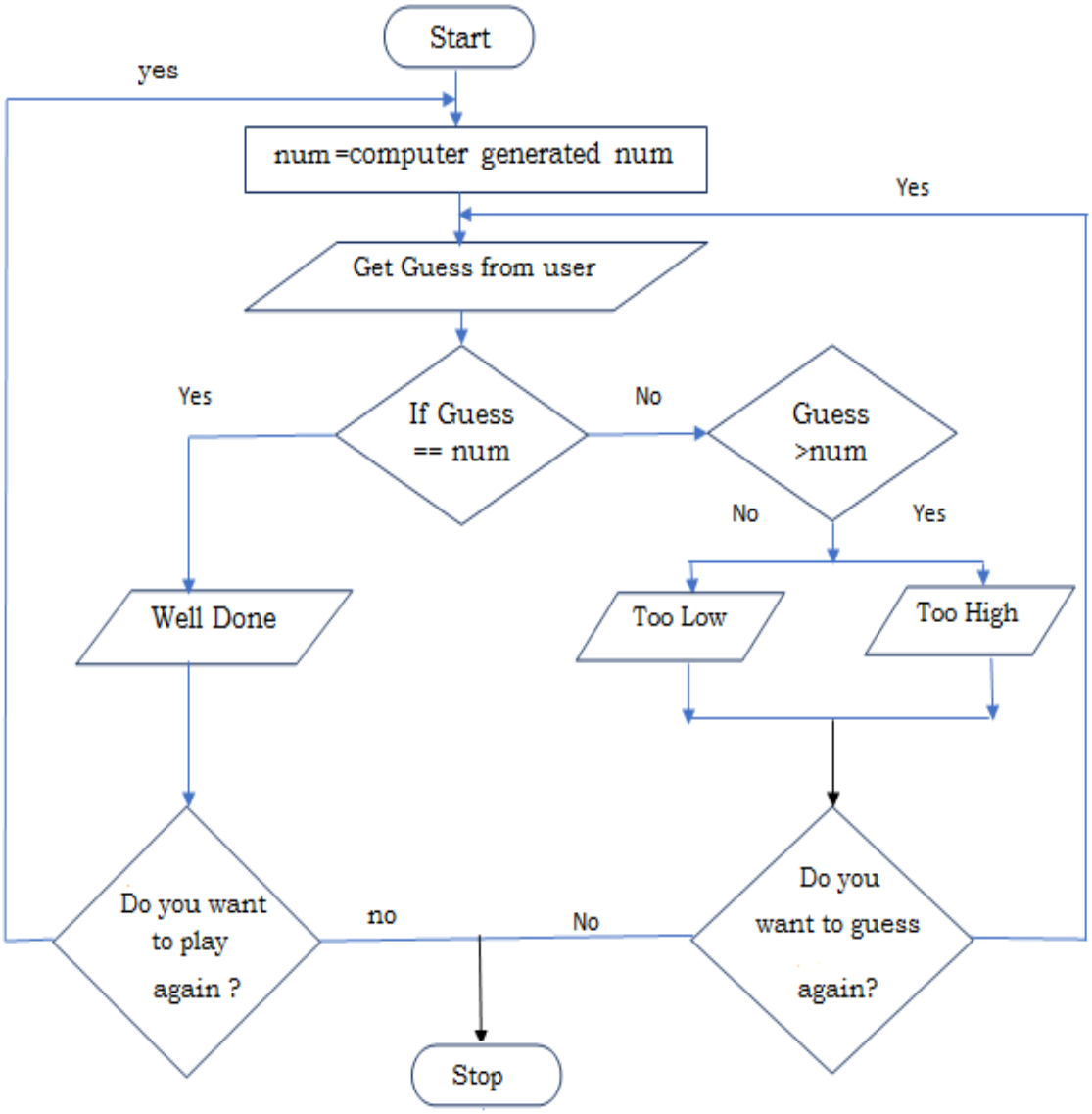
step 1: Start
Step 2: get a list a
Step 3: for each element (i) in a goto step4 else goto step 8
step 4: get the index of i assign it to j and goto step 5
Step 5: while (j>0) got to step6 else goto step3
Step 6: if(a[j-1]>a[j]) then swap a[j],a[j-1] and goto step7 else goto step7
step 7:decrement j value and goto step 5
Step 8: Print the list a
Step 9: Stop

Flowchart ("*same as insertion sort flow chart*")



Guess a integer number in a range

step 1: Start
Step 2: Assign computer generated number as num
Step 3: Get guess value from user
Step 4: if(guess==num) then print "weldone" and goto step 8 else goto step 5
Step 5: if(guess>num) then print "too high" and goto step 7
step 6: else print "too low" and goto step 7
Step 7: if you want to guess again goto step 3 else goto step 9
Step 8: if you want to play again goto step 2 else goto step 9
Step 9: stop



Towers of Hanoi

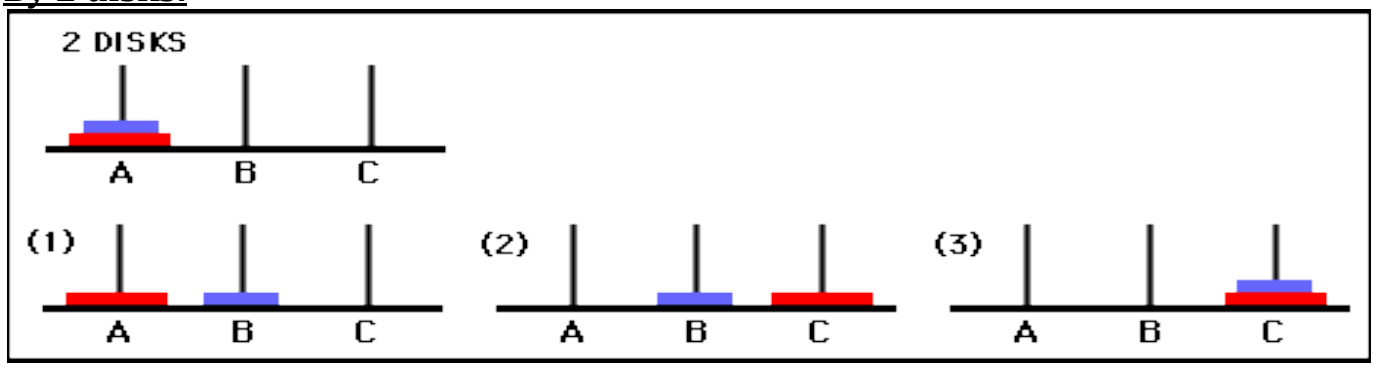
A tower of Hanoi is a mathematical puzzle with three rods and n number of discs.
The mission is to move all the disks to some another tower without violating the sequence of arrangement.

A few rules to be followed for Tower of Hanoi are –

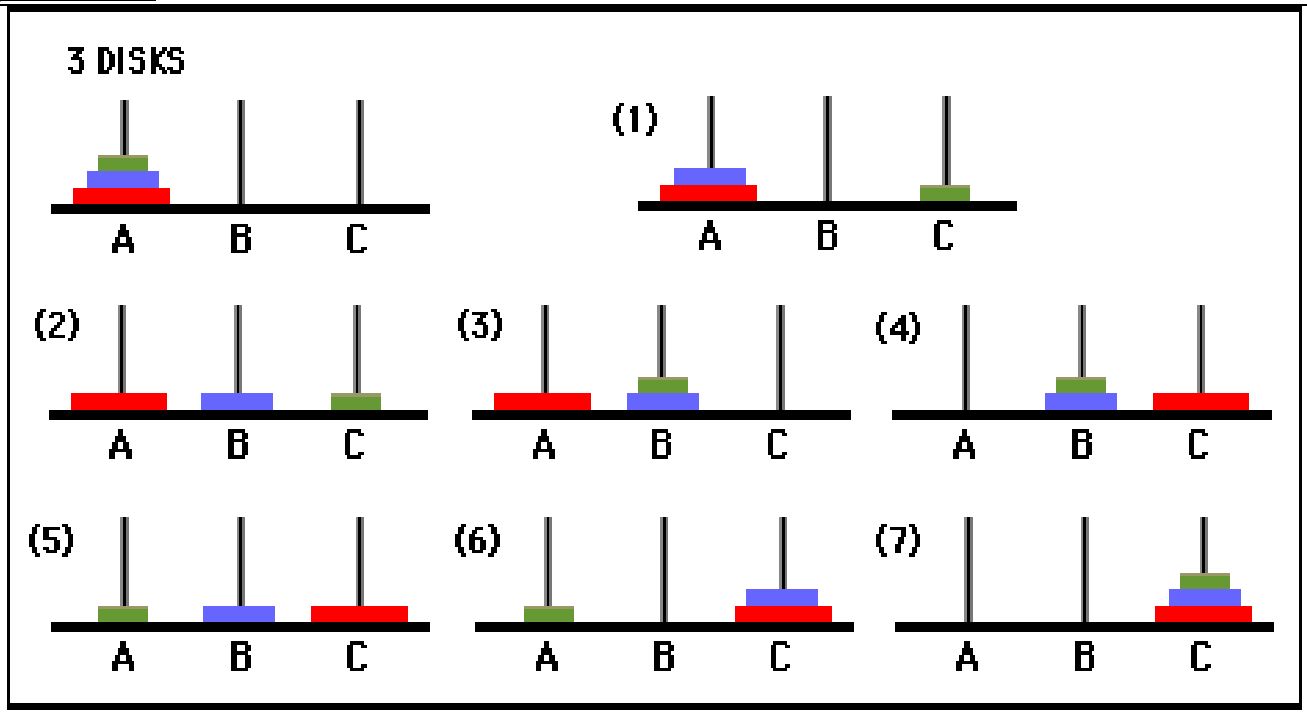
- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

No of efficient moves = $2^n - 1$

By 2 disks:



By 3 Disks:



Steps for algorithm:

Step 1 – Move n-1 disks from source to aux

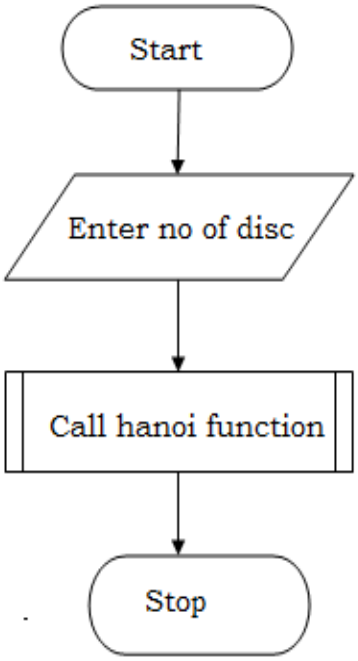
Step 2 – Move n^{th} disk from source to dest

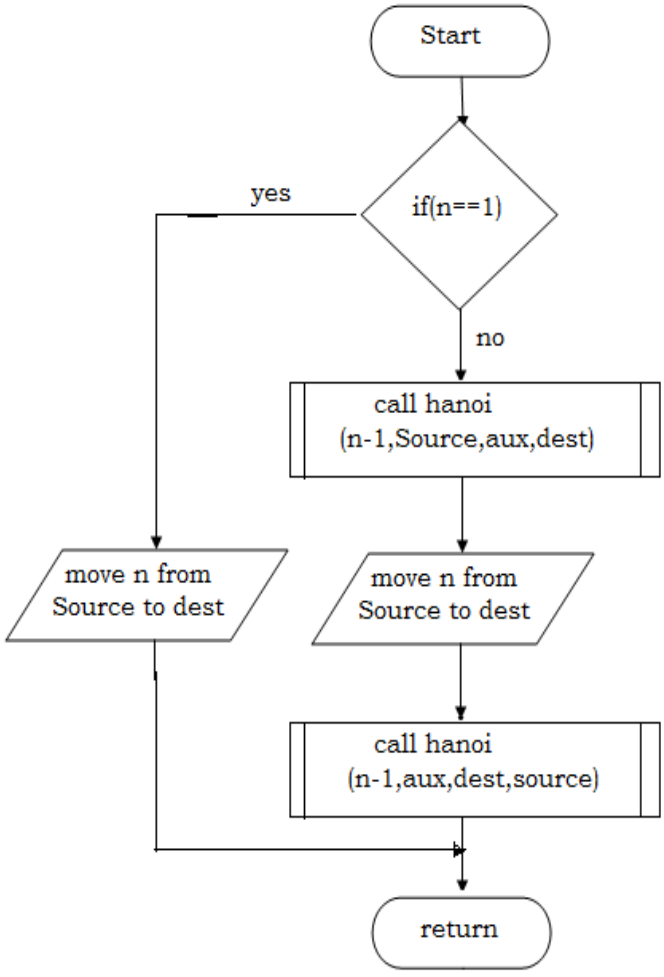
Step 3 – Move n-1 disks from aux to dest

Pseudocode for tower of hanoi

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

Flowchart for tower of hanoi





PART - A

1. Point out different modes of file opening
2. Differentiate text file and binary file.
3. Distinguish between files and modules
4. List down the operations on file.
5. list down some inbuilt exception.
6. Define read and write file.
7. Differentiate write and append mode.
8. Describe renaming and remove
9. Discover the format operator available in files.
10. Explain with example the need for exceptions
11. Explain built in exceptions
12. Difference between built in exceptions and handling exception
13. Write a program to write a data in a file for both write and append modes.
14. How to import statements?
15. Express about namespace and scoping.
16. Difference between global and local.
17. Identify what are the packages in python
18. Examine buffering.
19. Discover except Clause with Multiple Exceptions
20. Differentiate mutable.

Part-B

1. Write a Python program to demonstrate the file I/O operations
2. Discuss with suitable examples
    i) Close a File.
    ii) Writing to a File.
3. Write a program to catch a Divide by zero exception. Add a finally block too.
4. Explain format operator in detail.
5. Describe in detail about Exception with Arguments
6. Describe in detail about user – defined Exceptions
7. Explain with example of closing a file
8. Discover syntax for reading from a file
9. Structure Renaming a file
10. Explain about the Files Related Methods
11. Describe the import Statements
12. Describe the from…import statements
13. Describe in detail locating modules
14. Identify the various methods used to delete the elements from the dictionary
15. Describe in detail exception handling with program
16. Write a program to find the one's complement of binary number using file

Register Number: _____

[18CTU304B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

Coimbatore-641021.

(For the candidates admitted from 2018 onwards)

COMPUTER TECHNOLOGY

FIRST INTERNAL EXAMINATION- JULY 2019

First Semester

PROGRAMMING IN PYTHON

Date & Session: 23.07.2019 & AN **Duration : 2 Hours**

Class: II B.Sc. CT **Maximum : 50 Marks**

PART-A (20 X 1 = 20 Marks)

Answer all the Questions

1. _____ is the process of formulating a problem, finding a solution, and expressing the solution.

   a. Problem solving b. Recover c. Format d. Retrieve

2. _____programming language is designedto be easy for humans to read and write.

   a. low level language **b. high level language** c. machine language d. assembly language

3. A programming language that is designed to be easy for a computer to execute; also called

   a. low level language b. high level language **c. machine language** d. assembly language

4. _____ is designed to be easy for humans to read and write.

   a. Machine language b. assembly language c. Both a& b **d. high level language**

5. _____ is to execute a program in a high-level language by translating it one line at a time.

   a. Compile **b. Interpret** c. Portability d. assembly

6. _____ is to translate a program written in a high-level language into a low level language all at once, in preparation for later execution.

   a. Compile b. Interpret c. Portability d. assembly

7. _____ Program in a high-level language before being compiled.

   a. object code **b. source code** c. Executable d. script

8. _____ is the output of the compiler after it translates the program.

   a. object code b. source code c. Executable d. script

9. _____ is a set of instructions that specifies a computation.

   a. Algorithm **b. Program** c. Object d. Source

10. _____ is the structure of a program.

    a. Algorithm b. Program c. Syntax d. Source

11. _____ is an error in a program that makes it impossible to parse (and therefore impossible to interpret).

    a. Syntax error b. runtime error c. Exception d. Semantic error

12. _____ is the meaning of a program.

    a. Algorithm b. Program c. Syntax **d. Semantics**

13. _____ is an error in a program that makes it do something other than what the programmer intended.

    a. Syntax error b. runtime error c. Exception d. Semantic error

14. Python was designed by _____

    a. John Chamber b. Robert Gentleman **c. Guido van Rossum** d. Ritchie

15. Which of the following data types is not supported in python?

    a. Number b. String c. List **d. Slice**

16. _____ are formal languages that have been designed to express computations.

    a. Programming Languages b. Natural Languages

    c. Script Languages **d. Machine Languages**

17. _____ are languages that are designed by people for specific applications

    a. Programming Languages b. Natural Languages

    c. Script Languages d. Machine Languages

18. _____ are the languages that people speak, such as English, Spanish, and French.

    a. Programming Languages **b. Natural Languages**

    c. Script Languages d. Machine Languages

19. A _____ is a name that refers to a value.

    a. variable b. data type c. Keyword d. Syntax

20. Python is an _____ language.

a. Logical **b. Interpreted** c. Procedural d. Structural

Part –B (3 x 2 = 6 Marks)

Answer all the Questions

21. What is a Debugging?

Answer:

Programming is a complex process, and because it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called bugs and the process of tracking them down and correcting them is called debugging.

22. List the types of errors

Answer:

Three kinds of errors can occur in a program:

- Syntax errors
- Runtime errors
- Semantic errors

23. Define Algorithm

Answer:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Part –C (3 x 10 = 30 Marks)

Answer all the Questions

24. a. Explain the types of Errors in Programming

Answer:

Three kinds of errors can occur in a program:

- Syntax errors
- Runtime errors
- Semantic errors

Syntax errors:

- Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. Syntax refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. This sentence contains a syntax error. So does this one for most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will print an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.
- Here are some ways to avoid the most common syntax errors:
    - 1. Make sure you are not using a Python keyword for a variable name.
    - 2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.
    - 3. Check that indentation is consistent. You may indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount.
    - 4. Make sure that any strings in the code have matching quotation marks.
    - 5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
    - 6. An unclosed bracket—(, {, or [—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.

- 7. Check for the classic = instead of == inside a conditional. If nothing works, move on to the next section...

Runtime errors:

- The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.
- **My program does absolutely nothing.**
  - This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions. If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the "Flow of Execution" section below.

Semantic errors:

- The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.
- Try them out by writing simple test cases and checking the results.
  - In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.
  - The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component

independently. Once you find the discrepancy between your model and reality, you can solve the problem.

- Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

(OR)

b. Explain the Concept of Problem Solving

Answer:

CONCEPT OF PROBLEM SOLVING

- Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem solving skills.

PROBLEM DEFINITION

- The problem is *'I want a program which creates a backup of all my important files'*.
- Although, this is a simple problem, there is not enough information for us to get started with the solution. A little more **analysis** is required. For example, how do we specify which files are to be backed up? Where is the backup stored? How are they stored in the backup?
- After analyzing the problem properly, we **design** our program. We make a list of things about how our program should work. In this case, I have created the following list on how *I* want it to work. If you do the design, you may not come up with the same kind of problem - every person has their own way of doing things, this is ok.

1. The files and directories to be backed up are specified in a list.
2. The backup must be stored in a main backup directory.
3. The files are backed up into a zip file.
4. The name of the zip archive is the current date and time.
5. We use the standard **zip** command available by default in any standard Linux/Unix distribution. Windows users can use the Info-Zip program. Note that

you can use any archiving command you want as long as it has a command line interface so that we can pass arguments to it from our script.

THE SOLUTION

- As the design of our program is now stable, we can write the code which is an **implementation** of our solution.

FIRST VERSION

EXAMPLE: 10.1. A BACKUP SCRIPT - THE FIRST VERSION

```
#!/usr/bin/python
# Filename: backup_ver1.py

import os
import time

# 1. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or
something like that

# 2. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
target = target_dir + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))
```

```
# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'
```

OUTPUT

```
$ python backup_ver1.py
Successful backup to /mnt/e/backup/20041208073244.zip
```

- Now, we are in the **testing** phase where we test that our program works properly. If it doesn't behave as expected, then we have to **debug** our program i.e. remove the *bugs* (errors) from the program.

How It Works

- You will notice how we have converted our *design* into *code* in a step-by-step manner.
- We make use of the os and time modules and so we import them. Then, we specify the files and directories to be backed up in the source list. The target directory is where store all the backup files and this is specified in the target_dir variable. The name of the zip archive that we are going to create is the current date and time which we fetch using the time.strftime() function. It will also have the .zipextension and will be stored in the target_dir directory.
- The time.strftime() function takes a specification such as the one we have used in the above program. The %Y specification will be replaced by the year without the cetury.

The %m specification will be replaced by the month as a decimal number between 01 and 12 and so on. The complete list of such specifications can be found in the [Python Reference Manual] that comes with your Python distribution. Notice that this is similar to (but not same as) the specification used in print statement (using the % followed by tuple).

- We create the name of the target zip file using the addition operator which *concatenates* the strings i.e. it joins the two strings together and returns a new one. Then, we create a string zip_command which contains the command that we are going to execute. You can check if this command works by running it on the shell (Linux terminal or DOS prompt).
- The **zip** command that we are using has some options and parameters passed. The -q option is used to indicate that the zip command should work **q**uietly. The -r option specifies that the zip command should work **r**ecursively for directories i.e. it should include subdirectories and files within the subdirectories as well. The two options are combined and specified in a shorter way as -qr. The options are followed by the name of the zip archive to create followed by the list of files and directories to backup. We convert the source list into a string using the join method of strings which we have already seen how to use.
- Then, we finally *run* the command using the os.system function which runs the command as if it was run from the *system* i.e. in the shell - it returns 0 if the command was successfully, else it returns an error number.
- Depending on the outcome of the command, we print the appropriate message that the backup has failed or succeeded and that's it, we have created a script to take a backup of our important files!

Note to Windows Users

You can set the source list and target directory to any file and directory names but you have to be a little careful in Windows. The problem is that Windows uses the backslash (\) as the directory separator character but Python uses backslashes to represent escape sequences!

So, you have to represent a backslash itself using an escape sequence or you have to use raw strings. For example, use 'C:\\Documents' or r'C:\Documents' but do **not** use'C:\Documents' - you are using an unknown escape sequence \D !

- Now that we have a working backup script, we can use it whenever we want to take a backup of the files. Linux/Unix users are advised to use the executable method as discussed earlier so that they can run the backup script anytime anywhere. This is called the **operation** phase or the **deployment** phase of the software.
- The above program works properly, but (usually) first programs do not work exactly as you expect. For example, there might be problems if you have not designed the program properly or if you have made a mistake in typing the code, etc. Appropriately, you will have to go back to the design phase or you will have to debug your program

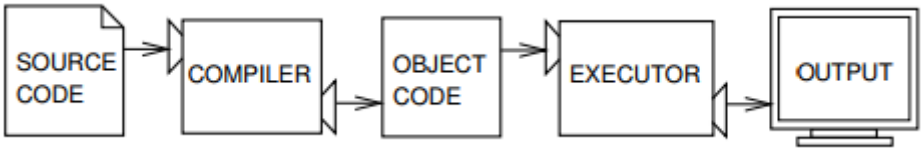
25. a. Discuss about Python programming Language

Answer:

The Python programming Language

- The programming language you will be learning is Python. Python is an example of a high-level language; other high-level languages you might have heard of are C, C++, Perl, and Java.
- As you might infer from the name "high-level language," there are also lowlevel languages, sometimes referred to as "machine languages" or "assembly 2 The way of the program languages." Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.
- But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are portable, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.
- Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications. Two kinds of programs process high-level languages into low-level languages: interpreters and compilers. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

- A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.



- Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: command line mode and script mode. In command-line mode, you type Python programs and the interpreter prints the result:

```
$ python
Python 2.4.1 (#1, Apr 29 2005, 00:28:56)
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1 + 1
2
```

- The first line of this example is the command that starts the Python interpreter. The next two lines are messages from the interpreter. The third line starts with >>>, which is the prompt the interpreter uses to indicate that it is ready. We typed print 1 + 1, and the interpreter replied 2. Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a script. For example, we used a text editor to create a file named latoya.py with the following contents:

```
print 1 + 1
```

- By convention, files that contain Python programs have names that end with .py.
- To execute the program, we have to tell the interpreter the name of the script:

```
$ python latoya.py 2
```

- In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.
- Most of the examples in this book are executed on the command line. Working on the command line is convenient for program development and testing, because you can type

programs and execute them immediately. Once you have a working program, you should store it in a script so you can execute or modify it in the future.

(OR)

b. Discuss in detail about the Algorithms

Answer:

ALGORITHMS

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language. From the data structure point of view, following are some important categories of algorithms –
    - **Search** − Algorithm to search an item in a data structure.
    - **Sort** − Algorithm to sort items in a certain order.
    - **Insert** − Algorithm to insert item in a data structure.
    - **Update** − Algorithm to update an existing item in a data structure.
    - **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

- Not all procedures can be called an algorithm. An algorithm should have the following characteristics –
    - **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps , and their inputs/outputs should be clear and must lead to only one meaning.
    - **Input** − An algorithm should have 0 or more well-defined inputs.
    - **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
    - **Finiteness** − Algorithms must terminate after a finite number of steps.
    - **Feasibility** − Should be feasible with the available resources.
    - **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

- As we know that all programming languages share basic code constructs like loops , flow-control , etc. These common constructs can be used to write an algorithm.
- We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.
- Example Let's try to learn algorithm-writing by using an example.
  - **Problem − Design an algorithm to add two numbers and display the result.**

    Step 1 − START

    Step 2 − declare three integers a, b & c

    Step 3 − define values of a & b

    Step 4 − add values of a & b

    Step 5 − store output of step 4 to c

    Step 6 − print c

    Step 7 − STOP

  - Algorithms tell the programmers how to code the program.
- Alternatively, the algorithm can be written as –

  Step 1 − START ADD

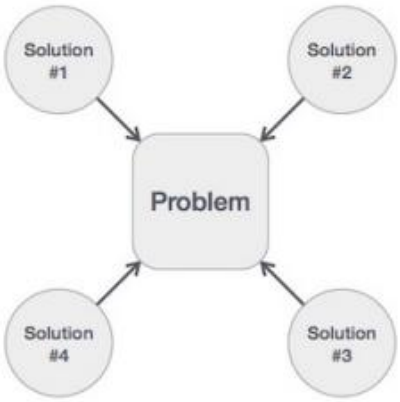
  Step 2 − get values of a & b

  Step 3 − c ← a + b

  Step 4 − display c

  Step 5 − STOP

- In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.
- Writing step numbers, is optional.
- We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



- Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

26. a. Explain about Decision table

Answer:

DECISION TABLE

- A decision table is used to represent conditional logic by creating a list of tasks depicting business level rules. Decision tables can be used when there is a consistent number of a condition that must be evaluated and assigned a specific set of actions to be used when the conditions are finally met.
- Decision tables are fairly similar to decision trees except for the fact that decision tables will always have the same number of conditions that need to be evaluated and actions that must be performed even if the set of branches being analyzed is resolved to true. A decision tree, on the other hand, can have one branch with more conditions that need to be evaluated than other branches on the tree.
- Decision tables are a concise visual representation for specifying which actions to perform depending on given conditions. They are algorithms whose output is a set of actions. The information expressed in decision tables could also be represented as decision trees or in a programming language as a series of if-then-else and switch-case statements.

Example

- The limited-entry decision table is the simplest to describe. The condition alternatives are simple Boolean values, and the action entries are check-marks, representing which of the actions in a given column are to be performed.

- A technical support company writes a decision table to diagnose printer problems based upon symptoms described to them over the phone from their clients.
- The following is a balanced decision table (created by Systems Made Simple).

Printer troubleshooter									
		Rules							
Conditions	Printer prints	No	No	No	No	Yes	Yes	Yes	Yes
	A red light is flashing	Yes	Yes	No	No	Yes	Yes	No	No
	Printer is recognized by computer	No	Yes	No	Yes	No	Yes	No	Yes
Actions	Check the power cable			✓					—
	Check the printer-computer cable	✓		✓					—
	Ensure printer software is installed	✓		✓		✓		✓	—
	Check/replace ink	✓	✓				✓		—
	Check for paper jam		✓		✓				—

- Of course, this is just a simple example (and it does not necessarily correspond to the reality of printer troubleshooting), but even so, it demonstrates how decision tables can scale to several conditions with many possibilities.

(OR)

b. Explain about Flowchart with an example

Answer:

FLOWCHARTING

Introduction to planning

- It is important to be able to plan code with the use of flowcharts.
- Even though you can code without a plan, it is not good practice to start coding without a guide to follow.

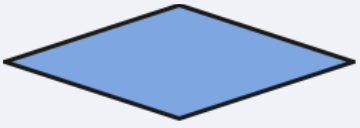
A good plan:

- creates a guide you can follow
- helps you plan efficient structure
- helps communicate to others what your code will do

- Poor planning can result in inefficient, unstructured code known as 'spaghetti code'.

FLOWCHARTS

- A standard way to plan code is with flowcharts.
- Specific parts of the flowchart represent specific parts of your code.

Symbol	Name	What it does in the code
	Start/End	Ovals show a start point or end point in the code
	Connection	Arrows show connections between different parts of the code
	Process	Rectangles show processes e.g. calculations (most things the computer does that does not involve an input, output or decision)
	Input/Output	Parallelograms show inputs and outputs (remember print is normally an input)
	Conditional/ Decision	Diamonds show a decision/conditional (this is normally if, else if/elif, while

and for)

revision

adding text to flowcharts

- There is no one right or wrong way to label flowcharts; you are presenting the structure of your code in a way that humans can understand. Only add extra details to parts of the flowchart when it is not obvious what they do.

Creating Flowcharts

There are many tools you can use to create flow charts.

Pseudo Code

- Pseudo code is an ordered version of your code written for human understanding rather than machine understanding.
- There is no one set way to write pseudo code.
- Good pseudo code should:
    - not be in a specific coding language
    - draft the structure of your code
    - be understandable to humans

e.g. pseudo code

```
if number <= 10 then
  ouput small number sentence
```

python code

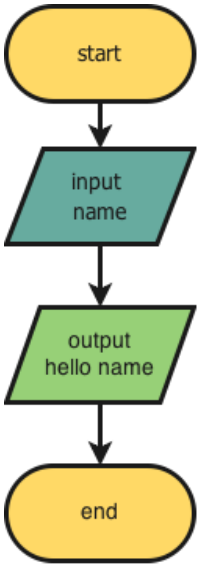
```
if number <=10:
  print("That's a small number!")
```

Note: Pseudo code may seem unnecessary but it is very useful to draft bits of code without worrying about the specifics of making it understandable to a computer.

Example 1

- This program asks the user their name then says "Hello [Name]":

flowchart for Example 1



Note: This flow chart only shows **ovals** and **parallelograms** because the code only has a **start** and **end** and one **input** and one **output**.

pseudo code for Example 1

```
input username
  output "Hello username"
```

Python code for Example 1

```
name= input("What is your name?")
  print("Hello "+ name)
```

Example 2

Description for Example 2

- This program asks the user to "Pick a number:" then prints 1 of 3 different outputs based on how big the number is.

Reg.No_____

[18CTU304B]

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act. 1956)
Coimbatore–641021

COMPUTERTECHNOLOGY

Third Semester
SECOND INTERNAL EXAMINATION – AUGUST 2019

PROGRAMMING IN PYTHON

Class:IIB.Sc.CT **Duration:** 2 Hours
Date&Session:.8.2019 **Maximum:** 50Marks

PART-A(20*1=20Marks)
AnswerALLtheQuestions

1. Python allows string slicing. What is the output of below code: s='cppbuzzchicago' print(s[3:5])
   a) pbuzz b) buzzc **c) bu** d) buz
2. How do you insert COMMENTS in Python code?
   a) /*…*/ b)// **c) #** d) !
3. What is a correct syntax to output "Hello World" in Python?
   a) echo("Hello World") b)echo "Hello World" c)p("Hello World") **d)print("Hello World")**
4. Which one is NOT a legal variable name?
   a) _myvar b) Myvar **c) my_var** d) my-var
5. print(chr(ord('b')+1))
   a) b b) syntax error **c) c** d) b+1
6. Which of the following operators is used to get accurate result (i.e fraction part also) in case of division ?
   a) // b) % c) / d) \
7. What is the associatively of Operators with the same precedence?
   a) Depends on operators **b) Left to Right**
   c) Right to Left d) Depends on Python Compiler
8. Which of the following has more precedence?
   a) + b) / c) - **d) ()**
9. First step in process of problem solving is to _____.
   a) design a solution **b) define a problem** c) practicing the solution
   d) Organizing the data
10. Python was created by
    a) James Gosling b) Bill Gates c) Steve Jobs **d) Guido van Rossum**
11. What is answer of this expression, 22 % 3 is?
    a) 7 **b) 1** c) 0 d) 5
12. What is the output of this expression, 3*1**3?
    a) 27 b) 9 **c) 3** d) 1
13. What data type is the object below?
    a) list b) dictionary c) array d) tuple

14. Which predefined Python function is used to find length of string?
    a) length() **b) len()** c) strlen() d) stringlength()
15. Syntax of constructor in Python?
    a) def __init__() b) def _init_() c)_init_() d) All of these
16. How to find the last element of list in Python? Assume `bikes` is the name of list.
    a) bikes[0] b) bikes[-1] c) bikes[lpos] **d) bikes[:-1]**
17. If a='cpp', b='buzz' then which of the following operation would show 'cppbuzz' asoutput?
    a) a+b b) a+"+b c) a+""+b **d) All the options**
18. What is the output of the code shown below? t=32.00 [round((x-32)*5/9) for x in t]
    a) [0] b) 0 c) [0.00] **d) Error**
19. What is the output of the following? print([i.lower() for i in "HELLO"])
    a) [‘h’, ‘e’, ‘l’, ‘l’, ‘o’] b) ‘hello’ c) [‘hello’] d) hello
20. The format function returns _____.
    a) an int b) a float **c) a str** d) a chat

PART-B(3*2=6Marks)
(AnswerALLtheQuestions)

21. What are the different types of operators?
    - Python Arithmetic Operator
    - Python Relational Operator
    - Python Assignment Operator
    - Python Logical Operator
    - Python Membership Operator
    - Python Identity Operator
    - Python Bitwise Operator

22. Write a python program print students Grade.

```
sub1=int(input("Enter marks of the first subject: "))
sub2=int(input("Enter marks of the second subject: "))
sub3=int(input("Enter marks of the third subject: "))
sub4=int(input("Enter marks of the fourth subject: "))
sub5=int(input("Enter marks of the fifth subject: "))
avg=(sub1+sub2+sub3+sub4+sub4)/5
if(avg>=90):
    print("Grade: A")
elif(avg>=80&avg<90):
    print("Grade: B")
elif(avg>=70&avg<80):
    print("Grade: C")
elif(avg>=60&avg<70):
    print("Grade: D")
else:
    print("Grade: F")
```

23. Define Boolean data type.
    A Boolean value can be True or False.

```
1. >>> a=2>1
2. >>> type(a)
```

<class ‘bool’>

Read: Python Functions with Syntax and Examples

PART-C(3*8=24Marks)
(AnswerALLtheQuestions)

24. a) Explain the role of function call and function definition with example.

Python function in any programming language is a sequence of statements in a certain order, given a name. When called, those statements are executed. So we don't have to write the code again and again for each [type of] data that we want to apply it to. This is called code re-usability.

User-Defined Functions in Python

For simplicity purposes, we will divide this lesson into two parts. First, we will talk about user-defined functions in Python. Python lets us group a sequence of statements into a single entity, called a function. A Python function may or may not have a name. We'll look at functions without a name later in this tutorial.

Advantages of User-defined Functions in Python

1. This Python Function help divide a program into modules. This makes the code easier to manage, debug, and scale.
2. It implements code reuse. Every time you need to execute a sequence of statements, all you need to do is to call the function.
3. This Python Function allow us to change functionality easily, and different programmers can work on different functions.

Defining a Function in Python

To define your own Python function, you use the 'def' keyword before its name. And its name is to be followed by parentheses, before a colon(:).

```
>>> def hello():
print("Hello")
```

The contents inside the body of the function must be equally indented.

As we had discussed in our article on **Python syntax**, you may use a docstring right under the first line of a function declaration. This is a documentation string, and it explains what the function does.

```
>>> def hello():
"""
This Python function simply prints hello to the screen
"""
print("Hello")
```

You can access this docstring using the __doc__ attribute of the function.

```
>>> def func1():
"""
This is the docstring
"""
print("Hello")
```

```
>>> func1.__doc__
```

```
2. '\n\tThis is the docstring\n\t'
```

However, if you apply the attribute to a function without a docstring, this happens.

```
1. >>> sum.__doc__
2. >>> type(sum.__doc__)
3. <class 'NoneType'>
4. >>> bool(sum.__doc__)
```

False

If you don't yet know what to put in the function, then you should put the pass statement in its body. If you leave its body empty, you get an error "Expected an indented block".

```
1. >>> def hello1():
2. pass
3. >>> hello1()
```

You can even reassign a function by defining it again.

Learn Methods in Python – Classes, Objects and Functions in Python

Python Built-In Functions with Syntax and Examples

abs()

The abs() is one of the most popular Python built-in functions, which returns the absolute value of a number. A negative value's absolute is that value is positive.

```
1. >>> abs(-7)
```

7

```
1. >>> abs(7)
```

7

```
1. >>> abs(0)
```

all()

The all() function takes a container as an argument. This Built in Functions returns True if all values in a python iterable have a Boolean value of True. An empty value has a Boolean value of False.

```
1. >>> all({'*',''})
```

False

```
1. >>> all([' ',' ',' '])
```

True

any()

Like all(), it takes one argument and returns True if, even one value in the iterable has a Boolean value of True.

```
1. >>> any((1,0,0))
```

True

1. >>> **any**((0,0,0))

False

4. ascii()

It is important Python built-in functions, returns a printable representation of a **python object** (like a string or a **Python list**). Let's take a Romanian character.

1. >>> **ascii**('ș')

"'\\u0219'"

Since this was a non-ASCII character in python, the interpreter added a backslash (\) and escaped it using another backslash.

1. >>> **ascii**('ușor')

"'u\\u0219or'"

Let's apply it to a list.

1. >>> **ascii**(['s','ș'])

"['s', '\\u0219']"

5. bin()

bin() converts an integer to a binary string. We have seen this and other functions in our article on **Python Numbers**.

1. >>> **bin**(7)

'0b111'

We can't apply it on floats, though.

1. >>> **bin**(7.0)

Traceback (most recent call last):

File "<pyshell#20>", line 1, in <module>

bin(7.0)

TypeError: 'float' object cannot be interpreted as an integer

6. bool()

bool() converts a value to Boolean.

1. >>> **bool**(0.5)

True

1. >>> **bool**('')

False

```
1. >>> bool(True)
```
True

7. bytearray()

bytearray() returns a python array of a given byte size.

```
1. >>> a=bytearray(4)
2. >>> a
```
bytearray(b'\x00\x00\x00\x00′)

```
1. >>> a.append(1)
2. >>> a
```
bytearray(b'\x00\x00\x00\x00\x01′)

```
1. >>> a[0]=1
2. >>> a
```
bytearray(b'\x01\x00\x00\x00\x01′)

```
1. >>> a[0]
```
1

Let's do this on a list.

```
1. >>> bytearray([1,2,3,4])
```
bytearray(b'\x01\x02\x03\x04′)

8. bytes()

bytes() returns an immutable bytes object.

```
1. >>> bytes(5)
```
b'\x00\x00\x00\x00\x00′

```
1. >>> bytes([1,2,3,4,5])
```
b'\x01\x02\x03\x04\x05′

```
1. >>> bytes('hello','utf-8')
```
b'hello'

Here, utf-8 is the encoding.

Both bytes() and bytearray() deal with raw data, but bytearray() is mutable, while bytes() is immutable.

```
1. >>> a=bytes([1,2,3,4,5])
2. >>> a
```
b'\x01\x02\x03\x04\x05′

```
1. >>> a[4]=
3
Traceback (most recent call last):
File "<pyshell#46>", line 1, in <module>
a[4]=3
TypeError: 'bytes' object does not support item assignment
```

Let's try this on bytearray().

```
1. >>> a=bytearray([1,2,3,4,5])
2. >>> a
bytearray(b'\x01\x02\x03\x04\x05′)
1. >>> a[4]=3
2. >>> a
bytearray(b'\x01\x02\x03\x04\x03′)
```

9. callable()

callable() tells us if an object can be called.

```
1. >>> callable([1,2,3])
False
1. >>> callable(callable)
True
1. >>> callable(False)
False
1. >>> callable(list)
True
```

A function is callable, a list is not. Even the callable() python Built In function is callable.

10. chr()

chr() Built In function returns the character in python for an ASCII value.

```
1. >>> chr(65)
'A'
1. >>> chr(97)
'a'
```

```
>>> chr(9)
```
‘\t’

```
>>> chr(48)
```
‘0’

11. classmethod()

classmethod() returns a class method for a given method.

```
>>> class fruit:
def sayhi(self):
print("Hi, I'm a fruit")

>>> fruit.sayhi=classmethod(fruit.sayhi)
>>> fruit.sayhi()
```
Hi, I’m a fruit

When we pass the method sayhi() as an argument to classmethod(), it converts it into a python class method one that belongs to the class. Then, we call it like we would call any static **method in python** without an object.

12. compile()

compile() returns a Python code object. We use Python in built function to convert a string code into object code.

```
>>> exec(compile('a=5\nb=7\nprint(a+b)','','exec'))
```
12

Here, ‘exec’ is the mode. The parameter before that is the filename for the file form which the code is read. Finally, we execute it using exec().

13. complex()

complex() function creates a complex number. We have seen this is our article on **Python Numbers**.

```
>>> complex(3)
```
(3+0j)

```
>>> complex(3.5)
```
(3.5+0j)

```
>>> complex(3+5j)
```
(3+5j)

14. delattr()

delattr() takes two arguments- a class, and an attribute in it. It deletes the attribute.

```
>>> class fruit:
size=7
```

```
3.
4. >>> orange=fruit()
5. >>> orange.size
```

7

```
1. >>> delattr(fruit,'size')
2. >>> orange.size
```

Traceback (most recent call last):

File "<pyshell#95>", line 1, in <module>

orange.size

AttributeError: 'fruit' object has no attribute 'size'

(OR)

b) How do you make use of math functions in Python?

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

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

```
# Python code to demonstrate the working of
# ceil() and floor()

# importing "math" for mathematical operations
import math

a = 2.3

# returning the ceil of 2.3
print ("The ceil of 2.3 is : ", end="")
print (math.ceil(a))

# returning the floor of 2.3
print ("The floor of 2.3 is : ", end="")
print (math.floor(a))
```

Output:

```
The ceil of 2.3 is : 3

The floor of 2.3 is : 2
```

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

filter_none
edit
play_arrow
brightness_4

```
# Python code to demonstrate the working of
```

```
# fabs() and factorial()

# importing "math" for mathematical operations
import math

a = -10

b= 5

# returning the absolute value.
print ("The absolute value of -10 is : ", end="")
print (math.fabs(a))

# returning the factorial of 5
print ("The factorial of 5 is : ", end="")
print (math.factorial(b))
```

Output:

```
The absolute value of -10 is : 10.0

The factorial of 5 is : 120
```

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

6. gcd() :- This function is used to compute the **greatest common divisor of 2 numbers** mentioned in its arguments. This function works in python 3.5 and above.

filter_none
edit
play_arrow
brightness_4

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

# importing "math" for mathematical operations
import math

a = -10
b = 5.5
c = 15
d = 5

# returning the copysigned value.
print ("The copysigned value of -10 and 5.5 is : ", end="")
print (math.copysign(5.5, -10))

# returning the gcd of 15 and 5
print ("The gcd of 5 and 15 is : ", end="")
print (math.gcd(5,15))
```

Output:

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

The gcd of 5 and 15 is : 5
```

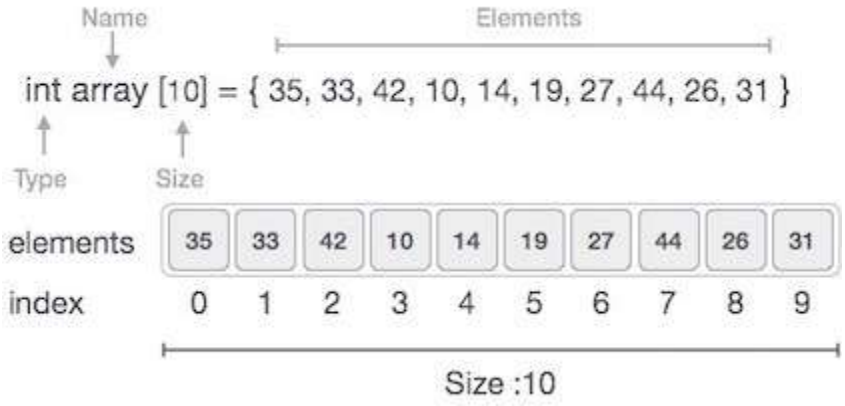
25. a) Explain list as arrays with example program.

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element**− Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

Array is created in Python by importing array module to the python program. Then the array is declared as shown eblow.

```
from array import *

arrayName = array(typecode, [Initializers])
```

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are:

Typecode	Value
b	Represents signed integer of size 1 byte/td>
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Before lookign at various array operations lets create and print an array using python.

The below code creates an array named array1.

```
from array import *

array1 = array('i', [10,20,30,40,50])

for x in array1:
 print(x)
```

When we compile and execute the above program, it produces the following result −

Output

```
10
20
30
40
50
```

Accessing Array Element

We can access each element of an array using the index of the element. The below code shows how

```
from array import *
```

```
array1 = array('i', [10,20,30,40,50])

print (array1[0])

print (array1[2])
```

When we compile and execute the above program, it produces the following result − which shows the element is inserted at index position 1.

Output

```
10
30
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1.insert(1,60)

for x in array1:
 print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is inserted at index position 1.

Output

```
10
60
20
30
40
50
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1.remove(40)

for x in array1:
 print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is removed form the array.

Output

```
10
20
30
50
```

Search Operation

You can perform a search for an array element based on its value or its index.

Here, we search a data element using the python in-built index() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

print (array1.index(40))
```

When we compile and execute the above program, it produces the following result which shows the index of the element. If the value is not present in the array then th eprogram returns an error.

Output

```
3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Here, we simply reassign a new value to the desired index we want to update.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1[2] = 80

for x in array1:
 print(x)
```

When we compile and execute the above program, it produces the following result which shows the new value at the index position 2.

Output

```
10
20
80
40
50
```

(OR)

b) Write a Python program to find GCD of two numbers and explain with flow chart.

Problem Solution

1. Take two numbers from the user.
2. Pass the two numbers as arguments to a recursive function.
3. When the second number becomes 0, return the first number.
4. Else recursively call the function with the arguments as the second number and the remainder when the first number is divided by the second number.
5. Return the first number which is the GCD of the two numbers.
6. Print the GCD.
7. Exit.

Program/Source Code

Here is source code of the Python Program to find the GCD of two numbers using recursion. The program output is also shown below.

```
def gcd(a,b):
    if(b==0):
        return a
    else:
        return gcd(b,a%b)
a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
GCD=gcd(a,b)
print("GCD is: ")
print(GCD)
```

Program Explanation

1. User must enter two numbers.
2. The two numbers are passed as arguments to a recursive function.
3. When the second number becomes 0, the first number is returned.
4. Else the function is recursively called with the arguments as the second number and the remainder when the first number is divided by the second number.

5. The first number is then returned which is the GCD of the two numbers.

6. The GCD is then printed.

Runtime Test Cases

```
Case 1:
Enter first number:5
Enter second number:15
GCD is:
5

Case 2:
Enter first number:30
Enter second number:12
GCD is:
6
```

26. a)Explain the conditional statements with example program.

Conditional Statement in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

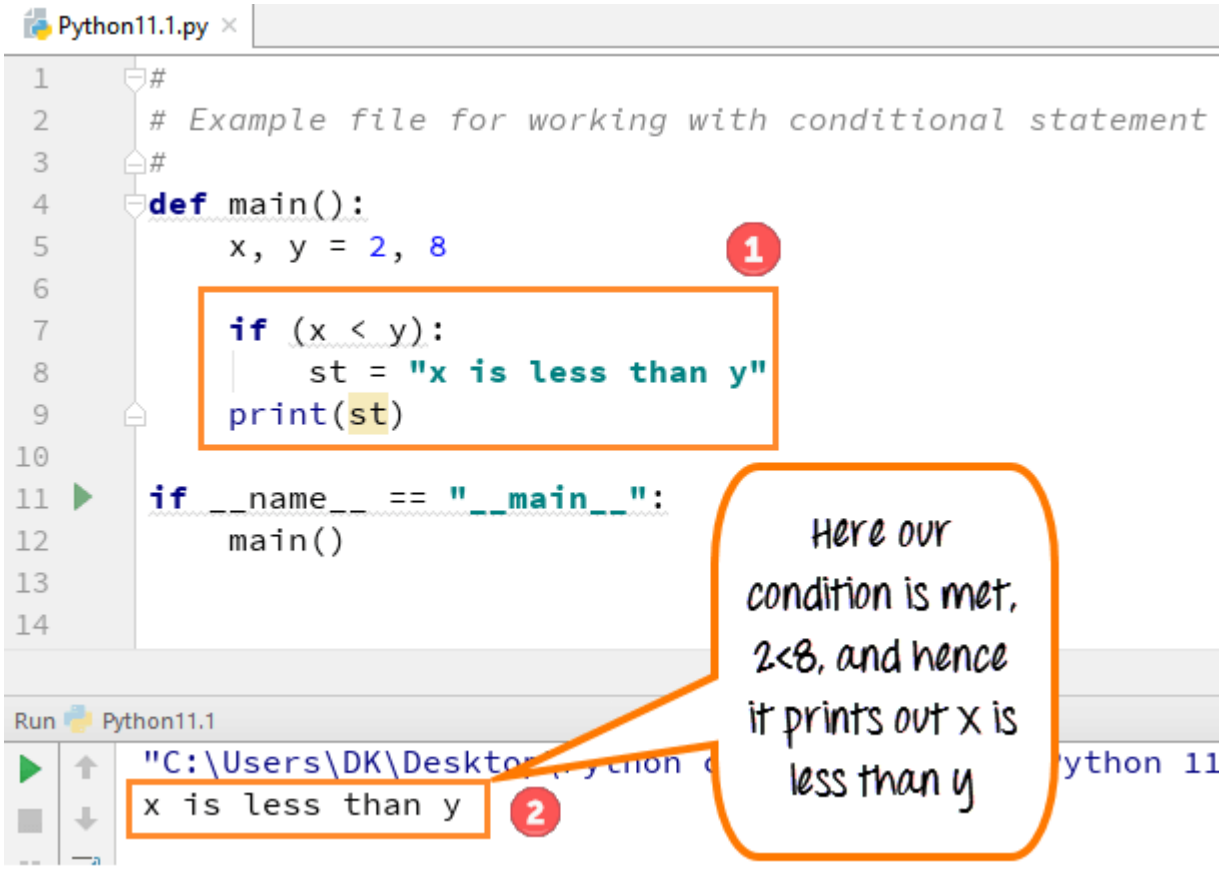
In Python, If Statement is used for decision making. It will run the body of code only when IF statement is true.

When you want to justify one condition while the other condition is not true, then you use "if statement".

Syntax:

```
if expression
 Statement
else
 Statement
```

Let see an **example**-

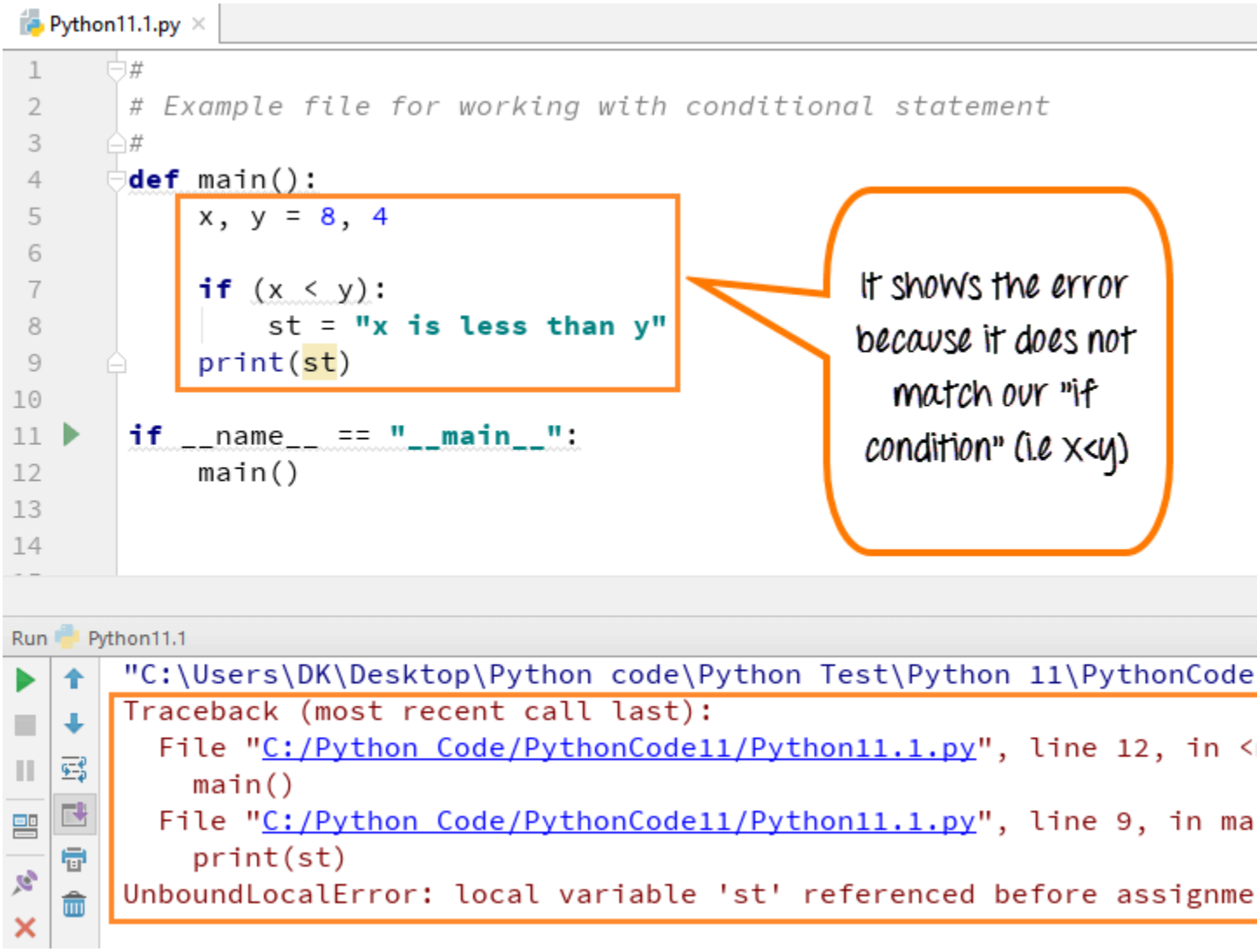



```
#
#Example file for working with conditional statement
#
def main():
        x,y =2,8

        if(x < y):
                st= "x is less than y"
        print(st)

if __name__ == "__main__":
        main()
```

- Code Line 5: We define two variables x, y = 2, 8
- Code Line 7: The if Statement checks for condition x<y which is **True** in this case
- Code Line 8: The variable st is set to "x is less than y."
- Code Line 9: The line print st will output the value of variable st which is "x is less than y",

What happen when "if condition" does not meet

In this step, we will see what happens when your "if condition" does not meet.

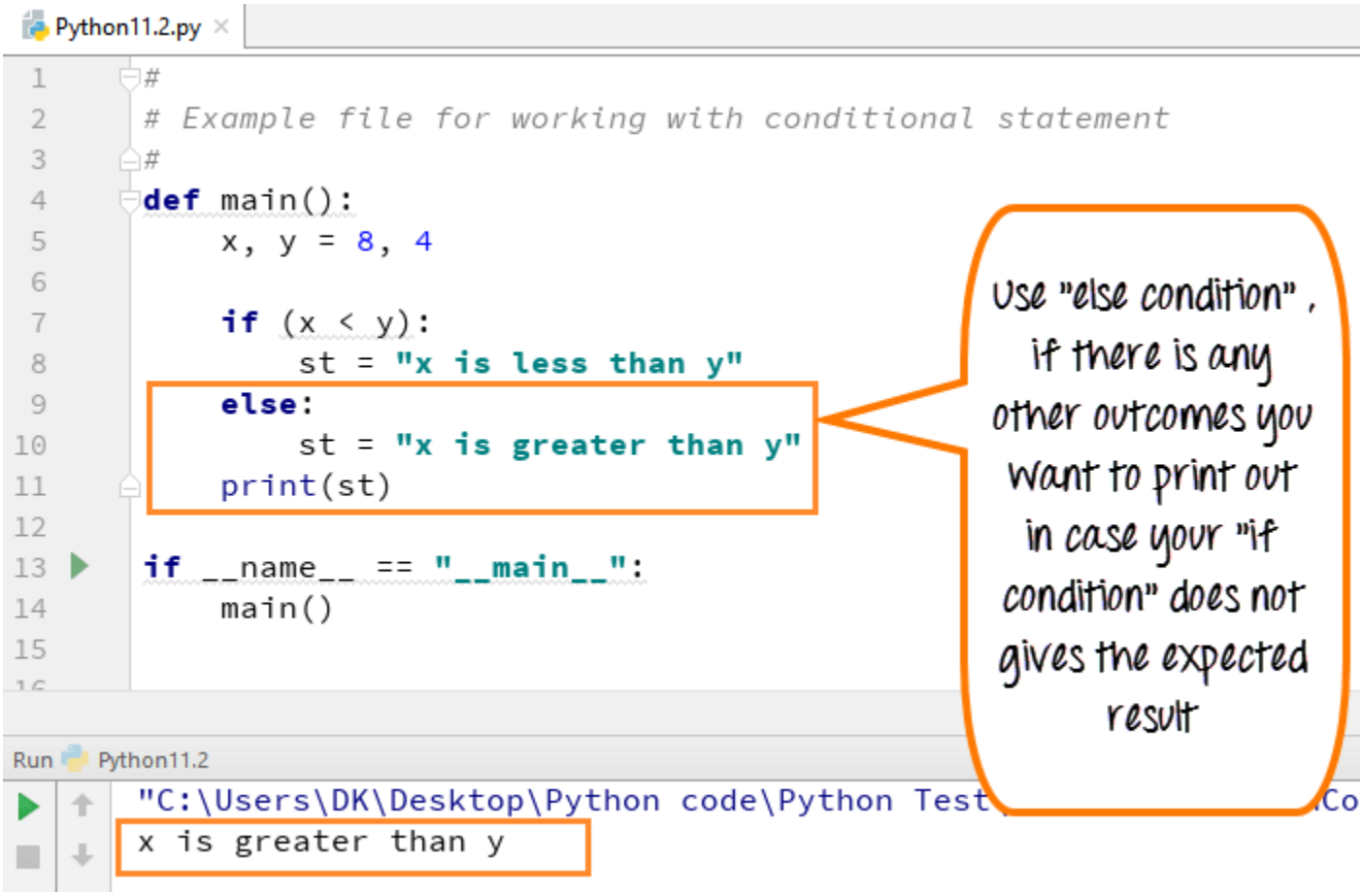


- Code Line 5: We define two variables x, y = 8, 4
- Code Line 7: The if Statement checks for condition x<y which is **False** in this case
- Code Line 8: The variable st is **NOT** set to "x is less than y."
- Code Line 9: The line print st - is trying to print the value of a variable that was never declared. Hence, we get an error.

How to use "else condition"

The "else condition" is usually used when you have to judge one statement on the basis of other. If one condition goes wrong, then there should be another condition that should justify the statement or logic.

Example:




```
#
#Example file for working with conditional statement
#
def main():
        x,y =8,4

        if(x < y):
                st= "x is less than y"
        else:
                st= "x is greater than y"
        print (st)

if __name__ == "__main__":
        main()
```

- Code Line 5: We define two variables x, y = 8, 4
- Code Line 7: The if Statement checks for condition x<y which is **False** in this case
- Code Line 9: The flow of program control goes to else condition
- Code Line 10: The variable st is set to "x is **greater** than y."
- Code Line 11: The line print st will output the value of variable st which is "x is greater than y",

When "else condition" does not work

There might be many instances when your "else condition" won't give you the desired result. It will print out the wrong result as there is a mistake in program logic. In most cases, this happens when you have to justify more than two statement or condition in a program.

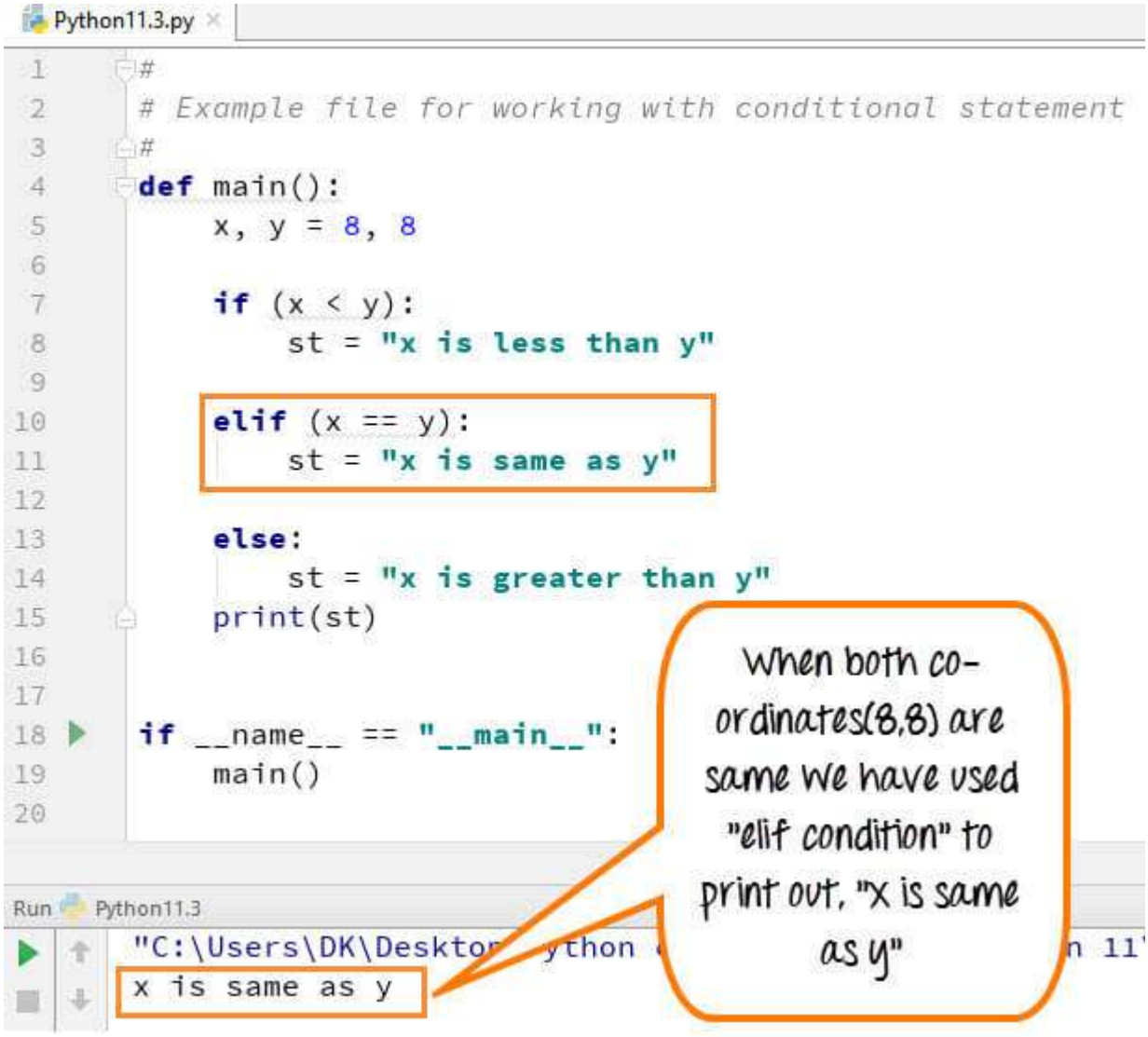
An **example** will better help you to understand this concept.

Here both the variables are same (8,8) and the program output is **"x is greater than y",** which is **WRONG**. This is because it checks the first condition (if condition), and if it fails, then it prints out the second condition (else condition) as default. In next step, we will see how we can correct this error.

How to use "elif" condition

To correct the previous error made by "else condition", we can use **"elif"** statement. By using "**elif**" condition, you are telling the program to print out the third condition or possibility when the other condition goes wrong or incorrect.

Example




```
#
#Example file for working with conditional statement
#
def main():
        x,y =8,8

        if(x < y):
                st= "x is less than y"

        elif (x == y):
                st= "x is same as y"

        else:
                st="x is greater than y"
        print(st)

if __name__ == "__main__":
```

```
    main()
```

- Code Line 5: We define two variables x, y = 8, 8
- Code Line 7: The if Statement checks for condition x<y which is **False** in this case
- Code Line 10: The flow of program control goes to the elseif condition. It checks whether x==y which is true
- Code Line 11: The variable st is set to "x is **same as** y."
- Code Line 15: The **flow of program control exits the if Statement (it will not get to the else Statement).** And print the variable st. The output is "x is same as y" which is correct

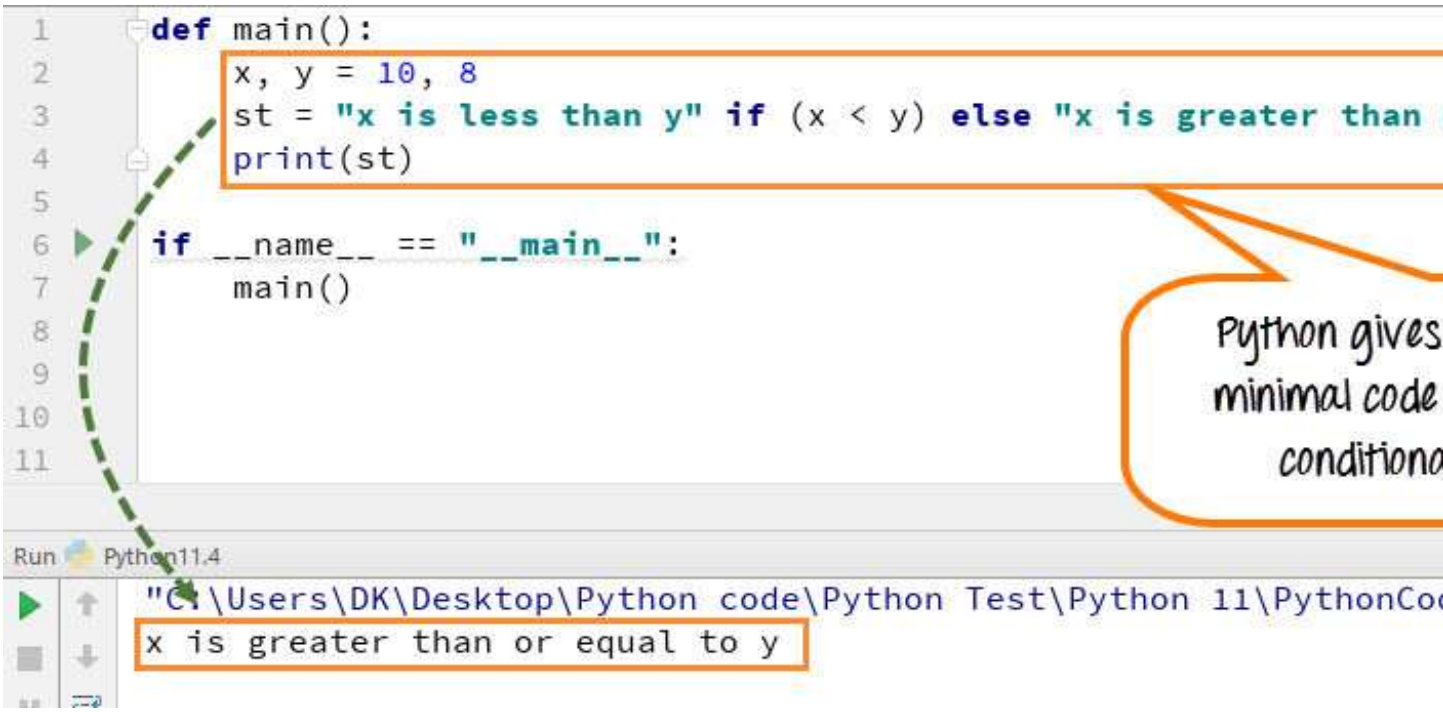
How to execute conditional statement with minimal code

In this step, we will see how we can condense out the conditional statement. Instead of executing code for each condition separately, we can use them with a single code.

Syntax

```
    A If B else C
```

Example:



```
def main():
        x,y = 10,8
        st = "x is less than y" if (x < y) else "x is greater than or equal to y"
        print(st)

if __name__ == "__main__":
        main()
```

- Code Line 2: We define two variables x, y = 10, 8
- Code Line 3: Variable st is set to "x is less than y "if x<y or else it is set to "x is greater than or equal to y". In this x>y variable st is set to **"x is greater than or equal to y."**
- Code Line 4: Prints the value of st and gives the correct output

- Instead of writing long code for conditional statements, Python gives you the freedom to write code in a short and concise way.

Nested IF Statement

Following example demonstrates nested if Statement

```
total = 100
#country = "US"
country = "AU"
if country == "US":
    if total <= 50:
        print("Shipping Cost is  $50")
elif total <= 100:
        print("Shipping Cost is $25")
elif total <= 150:
            print("Shipping Costs $5")
else:
        print("FREE")
if country == "AU":
          if total <= 50:
            print("Shipping Cost is  $100")
else:
            print("FREE")
```

Uncomment Line 2 in above code and comment Line 3 and run the code again

(OR)

b) Enlighten types of operators with example program.

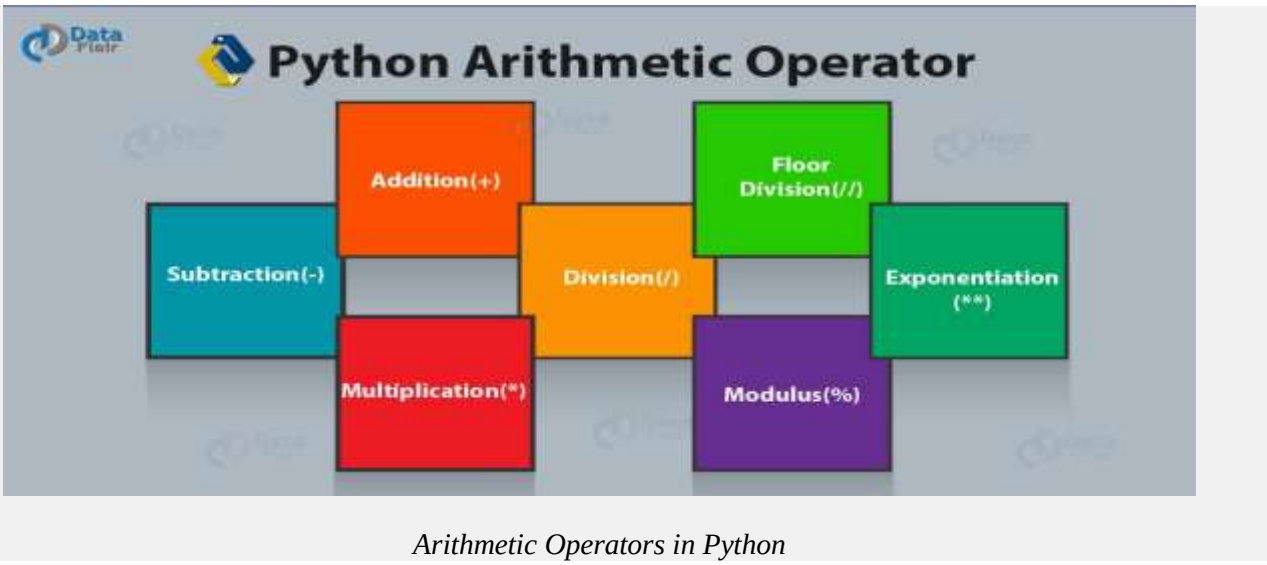
Python Operator falls into 7 categories:

- Python Arithmetic Operator
- Python Relational Operator
- Python Assignment Operator
- Python Logical Operator
- Python Membership Operator
- Python Identity Operator
- Python Bitwise Operator

It is recommended to check the Python master guide before we start with the operator in Python.

Python Arithmetic Operator

These Python arithmetic operators include Python operators for basic mathematical operations.



Arithmetic Operators in Python

a. Addition(+)

Adds the values on either side of the operator.

```
1. >>> 3+4
```

Output: 7

CHECK YOUR KNOWLEDGE – How to use + Operator for concatenation?

Comment, if you know the answer, else check the article – Frequently asked Python Interview Questions

b. Subtraction(-)

Subtracts the value on the right from the one on the left.

```
1. >>> 3-4
```

Output: -1

c. Multiplication(*)

Multiplies the values on either side of the operator.

```
1. >>> 3*4
```

Output: 12

d. Division(/)

Divides the value on the left by the one on the right. Notice that division results in a floating-point value.

```
1. >>> 3/4
```

Output: 0.75

e. Exponentiation(**)
Raises the first number to the power of the second.

```
1. >>> 3**4
```

Output: 81

f. Floor Division(//)
Divides and returns the integer value of the quotient. It dumps the digits after the decimal.

```
1. >>> 3//4
2. >>> 4//3
```

Output: 1

```
1. >>> 10//3
```

Output: 3

g. Modulus(%)
Divides and returns the value of the remainder.

```
1. >>> 3%4
```

Output: 3

```
1. >>> 4%3
```

Output: 1

```
1. >>> 10%3
```

Output: 1

```
1. >>> 10.5%3
```

Output: 1.5

If you face any query in Python Operator with examples, ask us in the comment.

Python Relational Operator

Let's see **Python Relational Operator**.

Relational Operators in Python

Relational Python Operator carries out the comparison between operands. They tell us whether an operand is greater than the other, lesser, equal, or a combination of those.

a. Less than(<)

This operator checks if the value on the left of the operator is lesser than the one on the right.

```
1. >>> 3<4
```

Output: True

b. Greater than(>)

It checks if the value on the left of the operator is greater than the one on the right.

```
1. >>> 3>4
```

Output: False

c. Less than or equal to(<=)

It checks if the value on the left of the operator is lesser than or equal to the one on the right.

```
1. >>> 7<=7
```

Output: True

d. Greater than or equal to(>=)

It checks if the value on the left of the operator is greater than or equal to the one on the right.

```
1. >>> 0>=0
```

Output: True

e. Equal to(= =)

This operator checks if the value on the left of the operator is equal to the one on the right. 1 is equal to the Boolean value True, but 2 isn't. Also, 0 is equal to False.

```
1. >>> 3==3.0
```

Output: True

```
1. >>> 1==True
```

Output: True

1. >>> 7==True

Output: False

1. >>> 0==False

Output: True

1. >>> 0.5==True

Output: False

***DO YOU KNOW* – "Google has declared Python as one of the official programming languages it uses." What are you waiting for? Start learning Python now with DataFlair's self-paced Python training**

f. Not equal to(!=)

It checks if the value on the left of the operator is not equal to the one on the right. The Python operator <> does the same job, but has been abandoned in Python 3.

When the condition for a relative operator is fulfilled, it returns True. Otherwise, it returns False. You can use this return value in a further statement or expression.

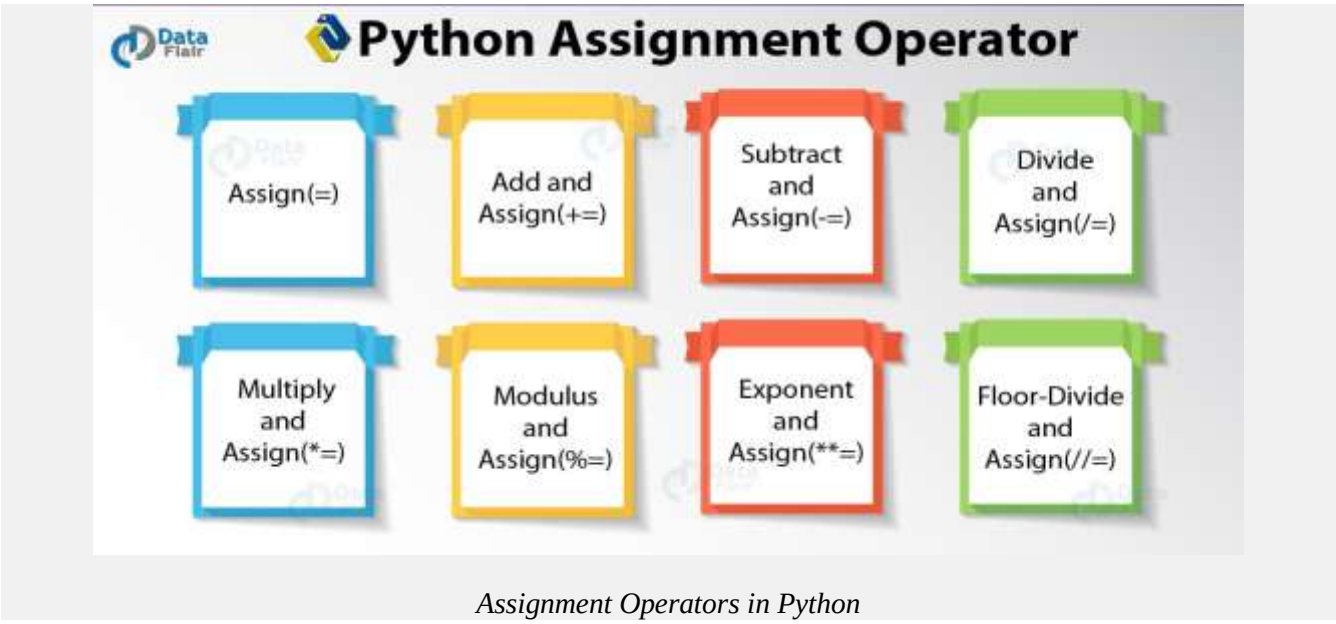
1. >>> 1!=1.0

Output: False

1. >>> -1<>-1.0

#This causes a syntax error

Python Assignment Operator



Assignment Operators in Python

Assignment Python Operator explained –

An assignment operator assigns a value to a variable. It may manipulate the value by a factor before assigning it. We have 8 assignment operators- one plain, and seven for the 7 arithmetic python operators.

a. Assign(=)

Assigns a value to the expression on the left. Notice that = = is used for comparing, but = is used for assigning.

```
1. >>> a=7
2. >>> print(a)
```

Output: 7

b. Add and Assign(+=)
Adds the values on either side and assigns it to the expression on the left. a+=10 is the same as a=a+10.

The same goes for all the next assignment operators.

```
1. >>> a+=2
2. >>> print(a)
```

Output: 9

c. Subtract and Assign(-=)
Subtracts the value on the right from the value on the left. Then it assigns it to the expression on the left.

```
1. >>> a-=2
2. >>> print(a)
```

Output: 7

d. Divide and Assign(/=)
Divides the value on the left by the one on the right. Then it assigns it to the expression on the left.

```
1. >>> a/=7
2. >>> print(a)
```

Output: 1.0

e. Multiply and Assign(*=)
Multiplies the values on either sides. Then it assigns it to the expression on the left.

```
1. >>> a*=8
2. >>> print(a)
```

Output: 8.0

f. Modulus and Assign(%=)
Performs modulus on the values on either side. Then it assigns it to the expression on the left.

```
1. >>> a%=3
2. >>> print(a)
```

Output: 2.0

g. Exponent and Assign(=)**
Performs exponentiation on the values on either side. Then assigns it to the expression on the left.

```
1. >>> a**=5
2. >>> print(a)
```

Output: 32.0

h. Floor-Divide and Assign(//=)
Performs floor-division on the values on either side. Then assigns it to the expression on the left.

```
1. >>> a//=3
2. >>> print(a)
```

Output: 10.0

This is one of the important Python Operator.

Know more about Python Namespace and Variable Scope

Python Logical Operator

These are conjunctions that you can use to combine more than one condition. We have three Python logical operator – and, or, and not that come under python operators.

Logical Operators in Python

a. and
If the conditions on both the sides of the operator are true, then the expression as a whole is true.

```
1. >>> a=7>7 and 2>-1
2. >>> print(a)
```

Output: False

b. or
The expression is false only if both the statements around the operator are false. Otherwise, it is true.

```
1. >>> a=7>7 or 2>-1
2. >>> print(a)
```

Output: True

'and' returns the first False value or the last value; 'or' returns the first True value or the last value

```
1. >>> 7 and 0 or 5
```

Output: 5

c. not
This inverts the **Boolean value** of an expression. It converts True to False, and False to True. As you can see below, the Boolean value for 0 is False. So, not inverts it to True.

```
1. >>> a=not(0)
2. >>> print(a)
```

Output: True

Membership Python Operator

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

a. in

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

```
1. >>> pets=['dog','cat','ferret']
2. >>> 'fox' in pets
```

Output: False

```
1. >>> 'cat' in pets
```

Output: True

```
1. >>> 'me' in 'disappointment'
```

Output: True

b. not in

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

```
1. >>> 'pot' not in 'disappointment'
```

Output: True

In doubt yet in any Python operator with examples? Please comment.

Don't you know about the trending Python Project at DataFlair? Here it is – Gender and Age Detection with Python

Python Identity Operator

Let us proceed towards identity Python Operator.

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

a. is

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

```
1. >>> 2 is 20
```

Output: False

```
1. >>> '2' is "2"
```

Output: True

b. is not
2 is a number, and '2' is a string. So, it returns a True to that.

```
1. >>> 2 is not '2'
```

Output: True

Python Bitwise Operator

Let us now look at Bitwise Python Operator.

Bitwise Operators in Python

On the operands, these operate bit by bit.

a. Binary AND(&)
It performs bit by bit AND operation on the two values. Here, binary for 2 is 10, and that for 3 is 11. &-ing them results in 10, which is binary for 2. Similarly, &-ing 011(3) and 100(4) results in 000(0).

```
1. >>> 2&3
```

Output: 2

```
1. >>> 3&4
```

Output: 0

b. Binary OR(|)
It performs bit by bit OR on the two values. Here, OR-ing 10(2) and 11(3) results in 11(3).

```
1. >>> 2|3
```

Output: 3

ENJOYING NETFLIX??? Python developers are the reason for its popularity.
Check out how Python is used at Netflix?

c. Binary XOR(^)
It performs bit by bit XOR(exclusive-OR) on the two values. Here, XOR-ing 10(2) and 11(3) results in 01(1).

```
1. >>> 2^3
```

Output: 1

d. Binary One's Complement(~)
It returns the one's complement of a number's binary. It flips the bits. Binary for 2 is 00000010. Its one's complement is 11111101. This is binary for -3. So, this results in -3. Similarly, ~1 results in -2.

```
1. >>>~-3
```

Output: 2

Again, one's complement of -3 is 2.

e. Binary Left-Shift(<<)
It shifts the value of the left operand the number of places to the left that the right operand specifies. Here, binary of 2 is 10. 2<<2 shifts it two places to the left. This results in 1000, which is binary for 8.

```
1. >>> 2<<2
```

Output: 8

f. Binary Right-Shift(>>)
It shifts the value of the left operand the number of places to the right that the right operand specifies. Here, binary of 3 is 11. 3>>2 shifts it two places to the right. This results in 00, which is binary for 0. Similarly, 3>>1 shifts it one place to the right. This results in 01, which is binary for 1.

```
1. >>> 3>>2
2. >>> 3>>1
```

Output: 1

Reg.No_____[18CTU304B]

KARPAGAMACADEMYOFHIGHEREDUCATION
(DeemedtobeUniversity)
(EstablishedUnderSection3ofUGCAct1956)
Coimbatore–641021

INFORMATIONTECHNOLOGY

ThirdSemester
THIRDINTERNALEXAMINATION–OCTOBER2019

DATAMINING

Class II B.Sc. CT **Duration :2 Hours**
Date &Session : 15.10.2018 & AN **Maximum: 50 Marks**

PART-A(20*1=20Marks)
AnswerALLtheQuestions

1. Which of the following lines of code will result in an error?
   a) s={abs} b) s={4, 'abc', (1,2)} c) s={2, 2.2, 3, 'xyz'} **d) s={san}**
2. "hat is the output of the line of code shown below, if s1= {1, 2, 3}? s1.issubset(s1)"
a) TRUE b) Error c) No output d) FALSE
3. "What is the output of the code shown below? s=set([1, 2, 3]) s.union([4, 5]) s|([4, 5])"
   a) "{1, 2, 3, 4, 5}{1, 2,3, 4, 5}" b) Error{1, 2, 3, 4, 5} **c) {1, 2, 3, 4, 5}Error** d) ErrorError
4. Which of the following function capitalizes first letter of string?
a) shuffle(lst) **b) capitalize()** c) isalnum() d) isdigit()
5. Which of the following function checks in a string that all characters are digits?
a) shuffle(lst) b) capitalize() c) isalnum() **d) isdigit()**
6. Which of the following function convert an integer to octal string in python?
   a) unichr(x) b) ord(x) c) hex() **d) oct(x)**
7. What is the name of data type for character in python ?
   a) char **b) python do not have any data type for characters** c) charcter d) chr
8. In python 3 what does // operator do ?
   a) Float division **b) Integer division** c) returns remainder d) same as a**b
9. What is "Programming is fun"[4: 6]?
   a) ram **b) ra** c) r d) pr
10. What is "Programming is fun"[-1]?
   a) pr b) ram c) ra **d) n**
11. The keyword_____ is required to define a class.
   a) def b) return **c) class** d) all
12. _____ terminates the process normally.
   a) abort() **b) exit()** c) assert() d) all
13. The _____ statement terminates the loop containing it.

a) break b) continue c) pass d) stop

14. The _____ statement is used to skip the rest of the code inside a loop for the current iteration only

a) break **b)continue** c) pass d) stop

15. Which of the following keyword is a valid placeholder for body of the function ?

a) breakb) continue **c) pass** d) body

16. "Let a = [1,2,3,4,5] then which of the following is correct ?"

a) print(a[:]) => [1,2,3,4] b) "print(a[0:]) =>[2,3,4,5]" **c) "print(a[:100]) =>[1,2,3,4,5]"**

d) print(a[-1:]) => [1,2]

17. "In python which is the correct method to load a module?"

a) include math **b) import math** c) #include<math.h> d) using math

18. The format function, when applied on a string returns :

a) list b) bool c) int **d) str**

19. print(chr(ord('b')+1))

a) b b) syntax error **c) c** d) b+1

20. What is the maximum possible length of an identifier?

a) 32 characters b) 63 characters **d) 79 characters** d) 89 characters

PART-B(3*2=6 Marks)
AnswerALLtheQuestions

21. List any two list operations.

a) Indexing

b) Slicing

22. Give example for list concatenation

```
list_a = list_a + list_b
print list_a
Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

23. Write file types in python.

- **jpg** - image/jpeg
- **jpx** - image/jpx
- **png** - image/png
- **gif** - image/gif
- **webp** - image/webp
- **cr2** - image/x-canon-cr2
- **tif** - image/tiff
- **bmp** - image/bmp
- **jxr** - image/vnd.ms-photo
- **psd** - image/vnd.adobe.photoshop
- **ico** - image/x-icon
- **heic** - image/heic

PART-C(3*8=24Marks)

AnswerALLtheQuestions

24. a) Enlighten the List operations with example for each.
To create python list of items, you need to mention the items, separated by commas, in square brackets. This is the python syntax you need to follow. Then assign it to a variable. Remember once again, you don't need to declare the data type, because Python is dynamically-typed.

```
>>> colors=['red','green','blue']
A Python list may hold different types of values.
>>> days=['Monday','Tuesday','Wednesday',4,5,6,7.0]
A list may have python list.
>>> languages=[['English'],['Gujarati'],['Hindi'],'Romanian','Spanish']
>>> languages
[[‘English’], [‘Gujarati’], [‘Hindi’], ‘Romanian’, ‘Spanish’]
>>>type(languages[0])
<class ‘list’>
```

A list may also contain tuples or so.

```
>>> languages=[('English','Albanian'),'Gujarati','Hindi','Romanian','Spanish']
>>> languages[0]
(‘English’, ‘Romanian’)
>>>type(languages[0])
<class ‘tuple’>
>>> languages[0][0]='Albanian'
Traceback (most recent call last):
File “<pyshell#24>”, line 1, in <module>
languages[0][0]=’Albanian’
TypeError: ‘tuple’ object does not support item assignment
```

4. How to Access Python List?
To access a Python list as a whole, all you need is its name.

```
>>> days
[‘Monday’, ‘Tuesday’, ‘Wednesday’, 4, 5, 6, 7.0]
```

Or, you can put it in a print statement.

```
>>> languages=[['English'],['Gujarati'],['Hindi'],'Romanian','Spanish']
>>>print(languages)
[[‘English’], [‘Gujarati’], [‘Hindi’], ‘Romanian’, ‘Spanish’]
```

To access a single element, use its index in square brackets after the list's name. Indexing begins at 0.

```
>>> languages[0]
[‘English’]
```

An index cannot be a float value.

```
>>> languages[1.0]
Traceback (most recent call last):
File “<pyshell#70>”, line 1, in <module>
languages[1.0]
TypeError: list indices must be integers or slices, not float
```

5. Slicing a Python List
When you want only a part of a Python list, you can use the slicing operator [].

```
>>> indices=['zero','one','two','three','four','five']
>>> indices[2:4]
[‘two’, ‘three’]
```

This returns items from index 2 to index 4-1 (i.e., 3)

```
>>> indices[:4]
[‘zero’, ‘one’, ‘two’, ‘three’]
```

This returns items from the beginning of the list to index 3.
>>> indices[4:]
[‘four’, ‘five’]
It returns items from index 4 to the end of the list in Python.
>>> indices[:]
[‘zero’, ‘one’, ‘two’, ‘three’, ‘four’, ‘five’]
This returns the whole list.
Negative indices- The indices we mention can be negative as well. A negative index means traversal from the end of the list.
>>> indices[:-2]
[‘zero’, ‘one’, ‘two’, ‘three’]
This returns item from the list’s beginning to two items from the end.
>>> indices[1:-2]
[‘one’, ‘two’, ‘three’]
It returns items from the item at index 1 to two items from the end.
>>> indices[-2:-1]
[‘four’]
This returns items from two from the end to one from the end.
>>> indices[-1:-2]
[] This returns an empty Python list, because the start is ahead of the stop for the traversal.
6. Reassigning a Python List (Mutable)
Python Lists are mutable. This means that you can reassign its items, or you can reassign it as a whole. Let’s take a new list.
>>> colors=['red','green','blue']
a. Reassigning the whole Python list
You can reassign a Python list by assigning it like a new list.
>>> colors=['caramel','gold','silver','occur']
>>> colors
[‘caramel’, ‘gold’, ‘silver’, ‘occur’]
b. Reassigning a few elements
You can also reassign a slice of a list in Python.
>>> colors[2:]=['bronze','silver']
>>> colors
[‘caramel’, ‘gold’, ‘bronze’, ‘silver’]
If we had instead put two values to a single one in the left, see what would’ve happened.
>>> colors=['caramel','gold','silver','occur']
>>> colors[2:3]=['bronze','silver']
>>> colors
[‘caramel’, ‘gold’, ‘bronze’, ‘silver’, ‘occur’]
colors[2:3] reassigns the element at index 2, which is the third element.
2:2 works too.
>>> colors[2:2]=['occur']
>>> colors
[‘caramel’, ‘gold’, ‘occur’, ‘bronze’, ‘silver’]
c. Reassigning a single element
You can reassign individual elements too.
>>> colors=['caramel','gold','silver','occur']
>>> colors[3]='bronze'
>>> colors
[‘caramel’, ‘gold’, ‘silver’, ‘bronze’]
Now if you want to add another item ‘holographic’ to the list, we cannot do it the conventional way.

>>> colors[4]='holographic'
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
colors[4]='holographic'
IndexError: list assignment index out of range
So, you need to reassign the whole list for the same.
>>> colors=['caramel','gold','silver','bronze','holographic']
>>> colors
['caramel', 'gold', 'silver', 'bronze', 'holographic']
7. How can we Delete a Python List?
You can delete a Python list, some of its elements, or a single element.
a. Deleting the entire Python list
Use the del keyword for the same.
>>> del colors
>>> colors
Traceback (most recent call last):
File "<pyshell#51>", line 1, in <module>
colors
NameError: name 'colors' is not defined
b. Deleting a few elements
Use the slicing operator in python to delete a slice.
>>> colors=['caramel','gold','silver','bronze','holographic']
>>> del colors[2:4]
>>> colors
['caramel', 'gold', 'holographic']
>>> colors[2]
'holographic'
Now, 'holographic' is at position 2.
c. Deleting a single element
To delete a single element from a Python list, use its index.
>>> del colors[0]
>>> colors
['gold', 'holographic']

(Or)

b) Classify the tuple functions and methods with example program.

Python function in any programming language is a sequence of statements in a certain order, given a name. When called, those statements are executed. So we don't have to write the code again and again for each [type of] data that we want to apply it to. This is called code re-usability.

User-Defined Functions in Python

For simplicity purposes, we will divide this lesson into two parts. First, we will talk about user-defined functions in Python. Python lets us group a sequence of statements into a single entity, called a function. A Python function may or may not have a name. We'll look at functions without a name later in this tutorial.

Advantages of User-defined Functions in Python

1. This Python Function help divide a program into modules. This makes the code easier to manage, debug, and scale.
2. It implements code reuse. Every time you need to execute a sequence of statements, all you need to do is to call the function.

3. This Python Function allow us to change functionality easily, and different programmers can work on different functions.

Defining a Function in Python

To define your own Python function, you use the 'def' keyword before its name. And its name is to be followed by parentheses, before a colon(:).

```
>>> def hello():
print("Hello")
```

The contents inside the body of the function must be equally indented.

As we had discussed in our article on **Python syntax**, you may use a docstring right under the first line of a function declaration. This is a documentation string, and it explains what the function does.

```
>>> def hello():
"""
This Python function simply prints hello to the screen
"""
print("Hello")
```

You can access this docstring using the __doc__ attribute of the function.

```
>>> def func1():
"""
This is the docstring
"""
print("Hello")
```

```
>>> func1.__doc__
'\n\tThis is the docstring\n\t'
```

However, if you apply the attribute to a function without a docstring, this happens.

```
>>> sum.__doc__
>>> type(sum.__doc__)
<class 'NoneType'>
>>> bool(sum.__doc__)
```

False

If you don't yet know what to put in the function, then you should put the pass statement in its body. If you leave its body empty, you get an error "Expected an indented block".

```
>>> def hello1():
pass
>>> hello1()
```

You can even reassign a function by defining it again.

Learn Methods in Python – Classes, Objects and Functions in Python

Python Built-In Functions with Syntax and Examples

abs()

The abs() is one of the most popular Python built-in functions, which returns the absolute value of a number. A negative value's absolute is that value is positive.

```
1. >>> abs(-7)
```

7

```
1. >>> abs(7)
```

7

```
1. >>> abs(0)
```

all()

The all() function takes a container as an argument. This Built in Functions returns True if all values in a python iterable have a Boolean value of True. An empty value has a Boolean value of False.

```
1. >>> all({'*','','0'})
```

False

```
1. >>> all([' ',' ',' '])
```

True

any()

Like all(), it takes one argument and returns True if, even one value in the iterable has a Boolean value of True.

```
1. >>> any((1,0,0))
```

True

```
1. >>> any((0,0,0))
```

False

4. ascii()

It is important Python built-in functions, returns a printable representation of a **python object** (like a string or a **Python list**). Let's take a Romanian character.

```
1. >>> ascii('ș')
```

"'\\u0219"

Since this was a non-ASCII character in python, the interpreter added a backslash (\) and escaped it using another backslash.

```
1. >>> ascii('uşor')
```
“‘u\\u0219or”’

Let’s apply it to a list.

```
1. >>> ascii(['s','ș'])
```
“[‘s’, ‘\\u0219’]”

5. bin()

bin() converts an integer to a binary string. We have seen this and other functions in our article on **Python Numbers**.

```
1. >>> bin(7)
```
‘0b111’

We can’t apply it on floats, though.

```
1. >>> bin(7.0)
```
Traceback (most recent call last):

File “<pyshell#20>”, line 1, in <module>

bin(7.0)

TypeError: ‘float’ object cannot be interpreted as an integer

6. bool()

bool() converts a value to Boolean.

```
1. >>> bool(0.5)
```
True

```
1. >>> bool('')
```
False

```
1. >>> bool(True)
```
True

7. bytearray()

bytearray() returns a python array of a given byte size.

```
1. >>> a=bytearray(4)
2. >>> a
```
bytearray(b’\x00\x00\x00\x00′)

```
1. >>> a.append(1)
2. >>> a
```
bytearray(b’\x00\x00\x00\x00\x01′)

```
1. >>> a[0]=1
2. >>> a
```
bytearray(b’\x01\x00\x00\x00\x01′)

```
1. >>> a[0]
```
1

Let’s do this on a list.

```
1. >>> bytearray([1,2,3,4])
```
bytearray(b’\x01\x02\x03\x04′)

8. bytes()

bytes() returns an immutable bytes object.

```
1. >>> bytes(5)
```
b’\x00\x00\x00\x00\x00′

```
1. >>> bytes([1,2,3,4,5])
```
b’\x01\x02\x03\x04\x05′

```
1. >>> bytes('hello','utf-8')
```
b’hello’

Here, utf-8 is the encoding.

Both bytes() and bytearray() deal with raw data, but bytearray() is mutable, while bytes() is immutable.

```
1. >>> a=bytes([1,2,3,4,5])
2. >>> a
```
b’\x01\x02\x03\x04\x05′

```
1. >>> a[4]=
```
3

Traceback (most recent call last):

File “<pyshell#46>”, line 1, in <module>

a[4]=3

TypeError: ‘bytes’ object does not support item assignment

Let’s try this on bytearray().

```
1. >>> a=bytearray([1,2,3,4,5])
2. >>> a
bytearray(b’\x01\x02\x03\x04\x05′)
```

```
1. >>> a[4]=3
2. >>> a
bytearray(b’\x01\x02\x03\x04\x03′)
```

9. callable()

callable() tells us if an object can be called.

```
1. >>> callable([1,2,3])
False
```

```
1. >>> callable(callable)
True
```

```
1. >>> callable(False)
False
```

```
1. >>> callable(list)
True
```

A function is callable, a list is not. Even the callable() python Built In function is callable.

10. chr()

chr() Built In function returns the character in python for an ASCII value.

```
1. >>> chr(65)
‘A’
```

```
1. >>> chr(97)
‘a’
```

```
1. >>> chr(9)
‘\t’
```

```
1. >>> chr(48)
```

‘0’

11. classmethod()

classmethod() returns a class method for a given method.

```
>>> class fruit:
def sayhi(self):
print("Hi, I'm a fruit")

>>> fruit.sayhi=classmethod(fruit.sayhi)
>>> fruit.sayhi()
```

Hi, I’m a fruit

When we pass the method sayhi() as an argument to classmethod(), it converts it into a python class method one that belongs to the class. Then, we call it like we would call any static **method in python** without an object.

12. compile()

compile() returns a Python code object. We use Python in built function to convert a string code into object code.

```
>>> exec(compile('a=5\nb=7\nprint(a+b)','','exec'))
```

12

Here, ‘exec’ is the mode. The parameter before that is the filename for the file form which the code is read.
Finally, we execute it using exec().

13. complex()

complex() function creates a complex number. We have seen this is our article on **Python Numbers**.

```
>>> complex(3)
```

(3+0j)

```
>>> complex(3.5)
```

(3.5+0j)

```
>>> complex(3+5j)
```

(3+5j)

14. delattr()

delattr() takes two arguments- a class, and an attribute in it. It deletes the attribute.

```
>>> class fruit:
size=7
```

```
3.
4. >>> orange=fruit()
5. >>> orange.size
```
7

```
1. >>> delattr(fruit,'size')
2. >>> orange.size
```
Traceback (most recent call last):

File "<pyshell#95>", line 1, in <module>

orange.size

AttributeError: 'fruit' object has no attribute 'size'

25. a) Explicate the dictionary in python with example program for each function and methods.
A real-life dictionary holds words and their meanings. As you can imagine, likewise, a Python dictionary holds key-value pairs. Let's look at how to create one.
Creating a Python Dictionary is easy as pie. Separate keys from values with a colon(:), and a pair from another by a comma(,). Finally, put it all in curly braces.

```
1. >>> {'PB&J':'Peanut Butter and Jelly','PJ':'Pajamas'}
```
{'PB&J': 'Peanut Butter and Jelly', 'PJ': 'Pajamas'}

Optionally, you can put the dictionary in a variable. If you want to use it later in the program, you must do this.

```
1. >>> lingo={'PB&J':'Peanut Butter and Jelly','PJ':'Pajamas'}
```
To create an empty dictionary, simply use curly braces and then assign it to a variable

```
1. >>> dict2={}
2. >>> dict2
```
{}

```
1. >>> type(dict2)
```
<class 'dict'>

a. Python Dictionary Comprehension

You can also create a Python dict using comprehension. This is the same thing that you've learned in your math class. To do this, follow an expression pair with a for-statement loop in python. Finally, put it all in curly braces.

```
1. >>> mydict={x*x:x for x in range(8)}
2. >>> mydict
```

{0: 0, 1: 1, 4: 2, 9: 3, 16: 4, 25: 5, 36: 6, 49: 7}

In the above code, we created a Python dictionary to hold squares of numbers from 0 to 7 as keys, and numbers 0-1 as values.

b. Dictionaries with mixed keys

It isn't necessary to use the same kind of keys (or values) for a dictionary in Python.

```
>>> dict3={1:'carrots','two':[1,2,3]}
>>> dict3
```

{1: 'carrots', 'two': [1, 2, 3]}

As you can see here, a key or a value can be anything from an integer to a list.

c. dict()

Using the dict() function, you can convert a compatible combination of constructs into a Python dictionary.

```
>>> dict(([1,2],[2,4],[3,6]))
```

{1: 2, 2: 4, 3: 6}

However, this function takes only one argument. So if you pass it three lists, you must pass them inside a list or a tuple. Otherwise, it throws an error.

```
>>> dict([1,2],[2,4],[3,6])
```

Traceback (most recent call last):

File "<pyshell#121>", line 1, in <module>

dict([1,2],[2,4],[3,6])

TypeError: dict expected at most 1 arguments, got 3

d. Declaring one key more than once

Now, let's try declaring one key more than once and see what happens.

```
>>> mydict2={1:2,1:3,1:4,2:4}
>>> mydict2
```

{1: 4, 2: 4}

As you can see, 1:2 was replaced by 1:3, which was then replaced by 1:4. This shows us that a dictionary cannot contain the same key twice.

e. Declaring an empty dictionary and adding elements later

When you don't know what key-value pairs go in your Python dictionary, you can just create an empty Python dict, and add pairs later.

```
>>> animals={}
>>> type(animals)
```

<class 'dict'>

```
>>> animals[1]='dog'
>>> animals[2]='cat'
>>> animals[3]='ferret'
>>> animals
```

{1: 'dog', 2: 'cat', 3: 'ferret'}

Any query on Python Dictionay yet? Leave a comment.

4. How to Access a Python Dictionary?

a. Accessing the entire Python dictionary

To get the entire dictionary at once, type its name in the shell.

```
>>> dict3
```

{1: 'carrots', 'two': [1, 2, 3]}

b. Accessing a value

To access an item from a list or a tuple, we use its index in square brackets. This is the python syntax to be followed. However, a Python dictionary is unordered. So to get a value from it, you need to put its key in square brackets.
To get the square root of 36 from the above dictionary, we write the following code.

```
>>> mydict[36]
```

6

c. get()

The Python dictionary get() function takes a key as an argument and returns the corresponding value.

```
>>> mydict.get(49)
```

7

d. When the Python dictionary keys doesn't exist

If the key you're searching for doesn't exist, let's see what happens.

```
>>> mydict[48]
```

Traceback (most recent call last):

File “<pyshell#125>”, line 1, in <module>

mydict[48]

KeyError: 48

Using the key in square brackets gives us a KeyError. Now let’s see what the get() method returns in such a situation.

```
>>> mydict.get(48)
>>>
```

As you can see, this didn’t print anything. Let’s put it in the print statement to find out what’s going on.

```
>>> print(mydict.get(48))
```

None

So we see, when the key doesn’t exist, the get() function returns the value None. We discussed it earlier, and know that it indicates an absence of value.

5. Reassigning a Python Dictionary

The python dictionary is mutable. This means that we can change it or add new items without having to reassign all of it.

a. Updating the Value of an Existing Key

If the key already exists in the Python dictionary, you can reassign its value using square brackets.

Let’s take a new Python dictionary for this.

```
>>> dict4={1:1,2:2,3:3}
```

Now, let’s try updating the value for the key 2.

```
>>> dict4[2]=4
>>> dict4
```

{1: 1, 2: 4, 3: 3}

b. Adding a new key

However, if the key doesn’t already exist in the dictionary, then it adds a new one.

```
>>> dict4[4]=6
>>> dict4
```

{1: 1, 2: 4, 3: 3, 4: 6}

Python dictionary cannot be sliced.

(Or)

b) Explain the modules and its usages in python with example program.

In this Python Modules tutorial, we will discuss what is a module in Python programming language? Moreover, we will talk about how to create python modules and import modules in python. Along with this, we will learn how can we execute python module as a script, standard Python modules, and Python dir functions.

Exiting the interpreter destroys all functions and variables we created. But when we want a longer program, we create a script. With Python, we can put such definitions in a file, and use them in a script, or in an interactive instance of the interpreter. Such a file is a module. If you face any difficulty in article on Python modules, ask us in comments.

In essence, a module is a file that contains Python statements and definitions. A Python modules looks like this:

calc.py

Let's create a Python modules 'calc'.

Microsoft Windows [Version 10.0.16299.309]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\lifei>cd Desktop

C:\Users\lifei\Desktop>mkdir calc

C:\Users\lifei\Desktop>cd calc

C:\Users\lifei\Desktop\calc>echo >__init__.py

C:\Users\lifei\Desktop\calc>echo >calc.py

C:\Users\lifei\Desktop\calc>

And this is what we put inside the module calc.py:

```
1. def add(a,b):
2. return a+b
3. def sub(a,b):
4. return a-b
5. def mul(a,b):
6. return a*b
7. def div(a,b):
8. return a/b
9. def exp(a,b):
10. return a**b
11. def floordiv(a,b):
```

```
12. return a//b
```

Also, calc is a package we create, and we place __init__.py inside it (Refer to **Python Packages**).

4. How can we Import Modules in Python?

Now, to import Python modules, we first get to the Desktop in Python.

```
1. >>> import os
2. >>> os.chdir('C:\\Users\\lifei\\Desktop\\calc')
3. >>> import calc
4. >>>
```

To find the name of this module, we can use the __name__ attribute.

```
1. >>> calc.__name__
```

'calc'

We can now use functions from this module:

We can also assign one of these functions a name:

```
1. >>> fd=calc.floordiv
2. >>> fd(5.5,4)
```

1.0

```
1. >>> fd(9,4)
```

2

```
1. >>> type(fd(9,4))
```

<class 'int'>

```
1. >>> type(fd(5.5,4))
```

<class 'float'>

Read: Python Module vs Package

How to Execute Python Modules as Scripts?

Look at the following code:

```
1. def add(a,b):
2. print(a+b)
3. def sub(a,b):
4. print(a-b)
5. def mul(a,b):
6. print(a*b)
7. def div(a,b):
8. print(a/b)
9. def exp(a,b):
```

```
print(a**b)
def floordiv(a,b):
print(a//b)
if __name__ == "__main__":
import sys
if int(sys.argv[1])==1:
add(int(sys.argv[2]),int(sys.argv[3]))
elif int(sys.argv[1])==2:
sub(int(sys.argv[2]),int(sys.argv[3]))
```

These last few lines let us use the sys module to deal with command line arguments. To execute subtraction, this is what we type in the command prompt:

C:\Users\lifei\Desktop\calc>python calc.py 2 3 4

-1

This way, you can complete the code for other operations as well. Hence, we've created a script. But we can also import this normally like a module:

```
>>> import calc
>>>
```

We may want to run a module as a script for testing purposes.

Any Doubt yet in Python Modules? Please Comment.

26. a) Write a program for file operations in python.

A file is a location on disk that stores related information and has a name. A hard-disk is non-volatile, and we use files to organize our data in different directories on a hard-disk. The RAM (Random Access Memory) is volatile; it holds data only as long as it is up. So, we use files to store data permanently. To read from or write to a file, we must first open it. And then when we're done with it, we should close it to free up the resources it holds (Open, Read/Write, Close).
To start Python file i/o, we deal with files and have a few **in-built functions** and **methods in Python**. To open a file in Python, we use the read() method.
But first, let's get to the desktop, and choose a file to work with.

```
>>> import os
>>> os.getcwd()
```

'C:\\Users\\lifei\\AppData\\Local\\Programs\\Python\\Python36-32'

```
>>> os.chdir('C:\\Users\\lifei\\Desktop')
>>> os.listdir()
```

['Adobe Photoshop CS2.lnk', 'Atom.lnk', 'Backup iPhone7+ 20-1-18', 'Burn Book.txt', 'ch', 'desktop.ini', 'dmkidnap.png', 'Documents', 'Eclipse Cpp Oxygen.lnk', 'Eclipse Java Oxygen.lnk', 'Eclipse Jee Oxygen.lnk', 'gifts.jpg', 'Items for trip.txt', 'Major temp', structure', 'office temp.jpg', 'Papers', 'Remember to remember.txt', 'To do.txt', 'Today.txt'] If this seems new to you, be sure to check out **Python Directory**.
Now, let's open Python file 'To do.txt'.

```
1. >>> open('To do.txt')
```

<_io.TextIOWrapper name='To do.txt' mode='r' encoding='cp1252'>

But to work with this, we must store it into a **Python variable**. Let's do this.

```
1. >>> todo=open('To do.txt')
2. >>> todo
```

<_io.TextIOWrapper name='To do.txt' mode='r' encoding='cp1252'>

We wouldn't have to change directory if we just passed the full path of Python file to open(). But let's work with this for now.

Follow this link to know more about **Python Functions**

Python File Modes

While opening Python file, we can declare our intentions by choosing a mode. We have the following modes:

Mode	Description
r	To read a file (default)
w	To write a file; Creates a new file if it doesn't exist, truncates if it does
x	Exclusive creation; fails if a file already exists
a	To append at the end of the file; create if doesn't exist
t	Text mode (default)
b	Binary mode
+	To open a file for updating (reading or writing)

Let's take a couple of examples.

```
1. >>> todo=open('To do.txt','r+b') #To read and write in binary mode
2. >>> todo=open('To do.txt','a')
```

b. Choosing Your Encoding

Also, it is good practice to specify what encoding we want, because different systems use a different encoding. While Windows uses 'cp1252', Linux uses 'utf-8'.

```
1. >>> todo=open('To do.txt',mode='r',encoding='utf-8')
```

c. When Python File Doesn't Exist

Finally, if you try opening Python file that doesn't exist, the interpreter will throw a FileNotFoundError.

```
1. >>> todo=open('abc.txt')
```

Traceback (most recent call last):

File "<pyshell#27>", line 1, in <module>

todo=open('abc.txt')

FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'

Tell us how do you like the Python Open file Explanation.

4. Python Close File

When we tried to manually go rewrite the Python file, it threw this error dialog when we attempted to save it.
So, remember, always close what you open:

```
1. >>> todo.close()
```

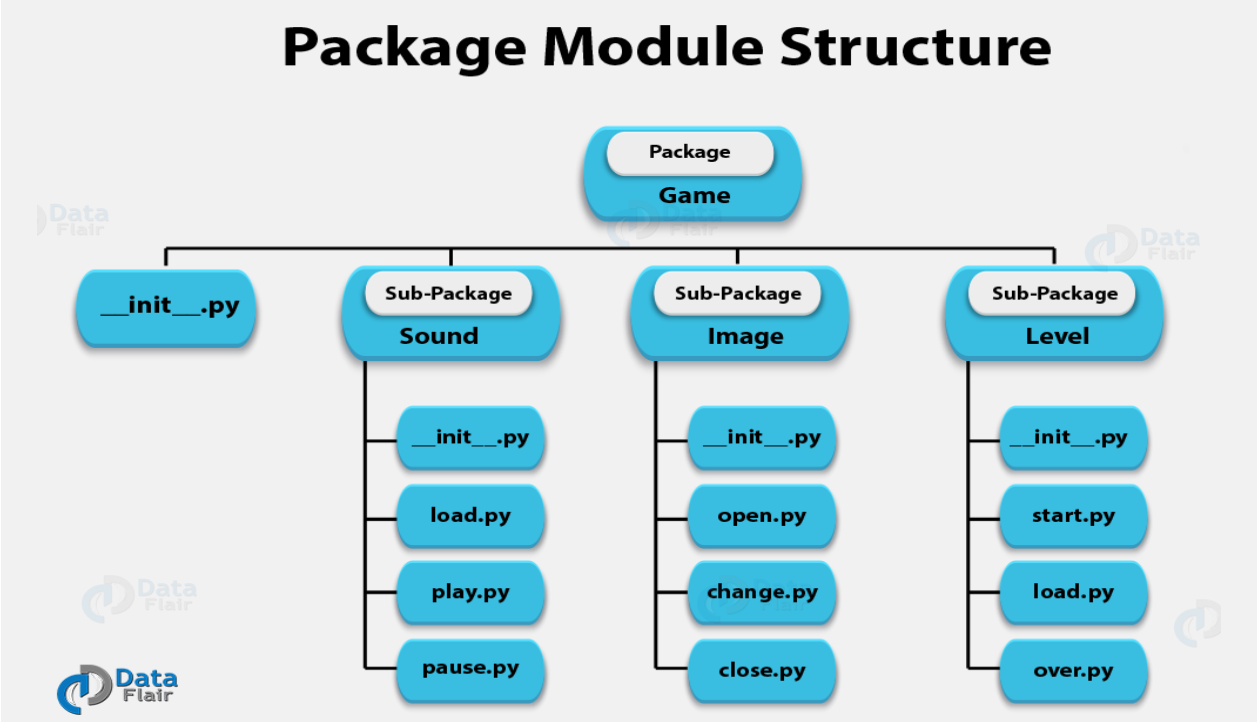
(Or)

b) What do you meant by packages in python? Give example.

In our computer systems, we store our files in organized hierarchies. We don't store them all in one location. Likewise, when our programs grow, we divide it into packages. In real-life projects, programs are much larger than what we deal with in our journey of learning Python. A package lets us hold similar modules in one place.
Like a directory may contain subdirectories and files, a package may contain sub-packages and modules. We have been using modules a lot in the previous lessons. Remember math, os, and collections? Those were all modules that ship with Python officially. We will discuss the difference between a module and a package in our next lesson. But for now, let's dive into the world of Python packages.

Structure of Python Packages

As we discussed, a package may hold other Python packages and modules. But what distinguishes a package from a regular directory? Well, a Python package must have an __init__.py file in the directory. You may leave it empty, or you may store initialization code in it. But if your directory does not have an __init__.py file, it isn't a package; it is just a directory with a bunch of Python scripts. Leaving __init__.py empty is indeed good practice.
Take a look at the following structure for a game:



Python Packages Module Structure

Here, the root package is Game. It has sub packages Sound, Image, and Level, and file __init__.py. Sound further has modules load, play, and pause, apart from file __init__.py. Image has modules open, change, and close, apart from __init__.py. Finally, Level has modules start, load, and over, apart from __init__.py.

4. How to Import Modules from Packages in Python?

A Python package may contain several modules. To import one of these into your program, you must use the dot operator(.)

In the above example, if you want to import the load module from subpackage sound, we type the following at the top of our Python file:

```
import Game.Sound.load
```

Note that we don't type the extension, because that isn't what we refer to the module as. The subpackage Level has a module named load too, but there is no clash here. This is because we refer to the module by its fully qualified name.

To escape having to type so much every time we needed to use the module, we could also import it under an alias:

```
import Game.Sound.load as loadgame
```

(If you're working the interpreter, you may also do the following:
loadgame=Game.Sound.load

This works equally fine.)

Alternatively, you could do:

from Game.Sound import load

Now, if the Sound subpackage has a function volume_up(), we call it this way:
loadgame.volume_up(7)

If we imported this way:

from Game.Sound.load import volume_up() as volup

We could call the function simply, without needing to use a full qualifier:
volup(7)

But this isn't recommended, as this may cause names in a namespace to clash.