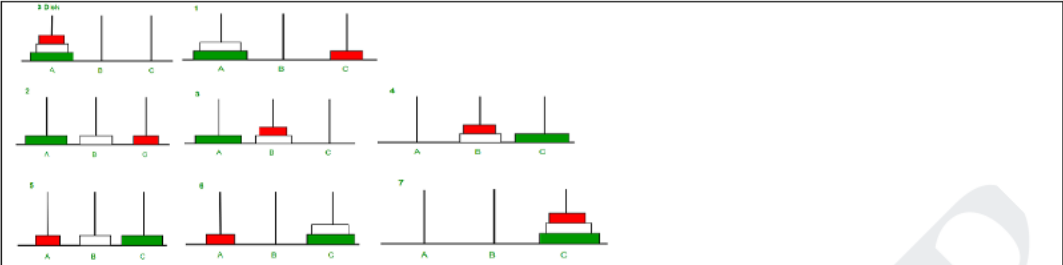


GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING	
UNIT 1- ALGORITHMIC PROBLEM SOLVING	
SYLLABUS	
Algorithms, building blocks of algorithms (statements, state, control flow, functions), notation (pseudo code, flow chart, programming language), algorithmic problem solving, simple strategies for developing 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.	
PART-A	
Q.No	Questions
1	Point out any 5 programming language Java, Python, C, C++,C-Sharp
2	Define an algorithm An algorithm is a finite sequence of <u>well-defined</u> , computer-implementable instructions, typically to solve a class of problems or to perform a computation. Algorithms are <u>unambiguous</u> specifications for performing <u>calculation</u> , <u>data processing</u> , <u>automated reasoning</u> , and other tasks.
3	Distinguish between pseudo code and flowchart. Pseudocode is linear (i.e. a sequence of lines with instructions), a flowchart is not. Flowcharts are a higher abstraction level, used before writing pseudocode or for documentation. Flowcharts have, in my opinion, two strong advantages over pseudocode: Firstly, they are graphical.
4	Define control flow statement with an eg: A program's control flow is the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls. ... Raising and handling exceptions also affects control flow ; Control flow example if x < 0: print "x is negative" elif x % 2: print "x is positive and odd" else: print "x is even and non-negative"
5	Describe recursion. A recursive function is a function defined in terms of itself via self-referential expressions. This means that the function will continue to call itself and repeat its behavior until some condition is met to return a result. def factorial(x): if x==1: return 1 else: return x*factorial(x-1) f=factorial(5) print ("factorial of 5 is ",f) The result is factorial of 5 is 120
	Discover the concept of towers of Hanoi. Tower of Hanoi consists of three pegs or towers with n disks placed one over the other. The

6	<p>objective of the puzzle is to move the stack to another peg following these simple rules. Only one disk can be moved at a time. No disk can be placed on top of the smaller disk.</p> 
7	<p>Explain list The most commonly used data structure in Python is List. Python list is a container like an array that holds an ordered sequence of objects. The object can be anything from a string to a number or the data of any available type.</p>
8	<p>Explain Iteration An iteration statement, which allows a code block to be repeated a certain number of times. Repeated execution of a set of statements is called iteration. Iteration is the repetition of a process in order to generate a sequence of outcomes. The sequence will approach some end point or end value. Each repetition of the process is a single iteration, and the outcome of each iteration is then the starting point of the next iteration.</p>
9	<p>Define simple computational problem The process of computational problem solving involves understanding the problem, designing a solution, and writing the solution (code). Computational thinking is an approach to solving problems using concepts and ideas from computer science, and expressing solutions to those problems so that they can be run on a computer.</p>
10	<p>Assess problem solving method. The problem solving methodology using a computer is a process that evolves on the following steps:</p> <p>Formulate the problem; Formalize the problem; Develop an algorithm that solves the problem; Program the algorithm (i.e., encode the algorithm as a valid program of a programming language available on the computer);</p> <p>Evaluate the Outcome</p> <p>The project implementation now needs to be monitored by the group to ensure their recommendations are followed. Monitoring includes checking:</p> <p>Milestones are met Costs are contained Necessary work is completed</p>

11	<p>What is meant by sorting ? mention its types Sorting is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or by length.</p> <ul style="list-style-type: none"> ➤ <u>Bubble Sort</u> ➤ <u>Selection Sort</u> ➤ <u>Insertion Sort</u> ➤ <u>Merge Sort</u> ➤ <u>Heap Sort</u> ➤ <u>Quick Sort</u>
12	<p>Develop algorithm for Celsius to Fahrenheit and vice versa # Python Program to convert temperature in celsius to fahrenheit</p> <p>Program to Convert Celsius To Fahrenheit</p> <p>In the following program we are taking the input from user, user enters the temperature in Celsius and the program converts the entered value into Fahrenheit using the conversion formula we have seen above.</p> <pre>celsius = float(input("Enter temperature in celsius: ")) fahrenheit = (celsius * 9/5) + 32 print('%0.2f Celsius is: %0.2f Fahrenheit' %(celsius, fahrenheit))</pre> <p>Output:</p> <pre>Enter temperature in celsius: 37 37.00 Celsius is: 98.60 Fahrenheit</pre> <p>Program to Convert Fahrenheit to Celsius</p> <p>In the following program user enters the temperature in Fahrenheit and the program converts the entered value into Celsius using the Fahrenheit to Celsius conversion formula.</p> <pre>fahrenheit = float(input("Enter temperature in fahrenheit: ")) celsius = (fahrenheit - 32) * 5/9 print('%0.2f Fahrenheit is: %0.2f Celsius' %(fahrenheit, celsius))</pre> <p>Output:</p> <pre>Enter temperature in fahrenheit: 99 99.00 Fahrenheit is: 37.22 Celsius</pre>
13	<p>Define programming language A programming language is a formal language, which comprises a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.</p>

14	<p>Identify the function types</p> <p>There are two basic types of functions: built-in functions and user defined functions. The built-in functions are part of the Python language; for instance <code>dir()</code> , <code>len()</code> , or <code>abs()</code> . The user defined functions are functions created with the def keyword.</p>
15	<p>Examine a simple program to print the integer number from 1 to 50</p> <pre># Sum of natural numbers up to 50 num = 50 if num < 0: print("Enter a positive number") else: sum = 0 # use while loop to iterate until zero while(num > 0): sum += num num -= 1 print("The sum is", sum)</pre>
16	<p>Discuss building blocks of algorithm</p> <p>An algorithm is made up of three basic building blocks: sequencing, selection, and iteration.</p> <p>Sequencing: An algorithm is a step-by-step process, and the order of those steps are crucial to ensuring the correctness of an algorithm.</p> <p>Selection: Algorithms can use selection to determine a different set of steps to execute based on a Boolean expression.</p> <p>Iteration: Algorithms often use repetition to execute steps a certain number of times or until a certain condition is met.</p>
17	<p>Discover the steps of simple strategies for developing algorithms.</p> <p>Approach the problem in stages:</p> <p>Think:</p> <ol style="list-style-type: none"> Analyze the problem Restate the problem Write out examples of input and output Break the problem into its component parts Outline a solution in psuedo-code Step through your example data with your psuedo-code <p>Execute</p> <ol style="list-style-type: none"> Code it up Test your solution against your examples
	Differentiate user defined function and predefined function

18	<p><u>User-Defined function</u></p> <ul style="list-style-type: none"> • In Python, a user-defined function's declaration begins with the keyword def and followed by the function name. • The function may take arguments(s) as input within the opening and closing parentheses, just after the function name followed by a colon. • After defining the function name and arguments(s) a block of program statement(s) start at the next line and these statement(s) must be indented. <p><u>Python Built-in Function.</u></p> <p>The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.</p>
19	<p>Analyze the notations used in algorithmic problem solving</p> <p>We usually present algorithms in the form of some pseudo-code, which is normally a mixture of English statements, some mathematical notations, and selected keywords from a programming language.</p> <p>Algorithms may also be represented by diagrams. One popular diagrammatic method is the flowchart, which consists of terminator boxes, process boxes, and decision boxes, with flows of logic indicated by arrows.</p>
20	<p>Describe some example for recursion function</p> <p>Recursive Functions in Python</p> <p>A recursive function is a function defined in terms of itself via self-referential expressions. This means that the function will continue to call itself and repeat its behavior until some condition is met to return a result.</p> <pre>def factorial(n): if n == 1: return 1 else: return n * factorial(n-1)</pre>
PART-B	
1	<p>Explain the algorithm GCD and find LCM</p> <p>Euclid's algorithm is based on repeated application of equality $\text{gcd}(m,n) = \text{gcd}(n, m \bmod n)$ until the second number becomes 0, which makes the problem trivial.</p>

	<p>Example: $\text{gcd}(60,24) = \text{gcd}(24,12) = \text{gcd}(12,0) = 12$</p> <p><u>Pseudocode</u> while $n \neq 0$ do $r \leftarrow m \bmod n$ $m \leftarrow n$ $n \leftarrow r$ return m</p>
2	<p>Discuss with suitable examples i)Find minimum in a list</p> <pre>def smallest_num_in_list(list): min = list[0] for a in list: if a < min: min = a return min print(smallest_num_in_list([1, 2, -8, 0]))</pre> <p>Sample Output: -8</p> <p>ii)Find Maximum in a list</p> <pre>def max_num_in_list(list): max = list[0] for a in list: if a > max: max = a return max print(max_num_in_list([1, 2, -8, 0]))</pre> <p>Sample Output: 2</p>

3	<p>i)Summarize advantage and disadvantage of flow chart</p> <p>Advantages and Disadvantages of Flowchart Flowchart Meaning It is said that a single picture is worth thousands words and flowchart works basically on that concept only as it illustrates solution of complex problems through diagrams and thus helps an individual to understand the concept better, however sometimes it may complicate the solution which in turn will make it even more difficult for an individual to understand the solution of the problem</p> <p>Advantages of Flowchart</p> <ul style="list-style-type: none"> ➤ Short and Simple
---	---








- The biggest advantage of using flowchart is that it is short as well as simple
- **Logical Steps**
It helps them understand the solution of the problem logically.
 - **Effective Communication**
It is one of the effective ways of communicating because flowchart can be made on 1 or 2 pages only as opposed to other methods of communication like written communication which may take many pages, hence if one wants to save time and communicate effectively than flowcharts can be a good option for them.

Disadvantages of Flowchart

- **Not suitable where Solution is long**
When the solution of the problem is short than it is a good method but if the solution is longer than this may not be the ideal method.
- **Complicate Things**
One does not understand the solution even when the solution is right due to the wrong presentation through flowcharts.
- **Difficult to Alter**
Another limitation is that flowcharts are difficult to alter because if there is one mistake than one has to alter the whole flowchart

ii) Summarize the symbol used in flow chart

i)

Symbol	Description	Symbol	Description
	START / STOP		PROCESS
	DECISION		INPUT
	OUTPUT		CONNECTORS
	STORAGE		

4	<p>Describe Build an algorithm for the following</p> <p>i) Prime number or not</p> <p># Program to check if a number is prime or not</p> <pre>num = 111 # To take input from the user #num = int(input("Enter a number: ")) # prime numbers are greater than 1 if num > 1: # check for factors for i in range(2,num): if (num % i) == 0: print(num,"is not a prime number") print(i,"times",num//i,"is",num) break else: print(num,"is a prime number") # if input number is less than # or equal to 1, it is not prime else: print(num,"is not a prime number")</pre> <p>Output</p> <p>111 is not a prime number 3 times 37 is 111</p> <p>ii) Odd or even</p> <p># Python program to check if the input number is odd or even. # A number is even if division by 2 gives a remainder of 0. # If the remainder is 1, it is an odd number.</p> <pre>num = int(input("Enter a number: ")) if (num % 2) == 0: print("{0} is Even".format(num)) else: print("{0} is Odd".format(num))</pre> <p>output</p> <p>Enter a number: 7 7 7 is Odd</p>
---	--

5

Explain the rules for pseudo code and uses of keywords

RULES FOR PSEUDOCODE

1. Write only one stmt per line

Each stmt in your pseudocode should express just one action for the computer.
If the task list is properly drawn, then in most cases each task will correspond to one line of pseudocode.

Eg: TASK LIST:

Read name, hourly rate, hours worked, deduction rate

Perform calculations

gross = hourlyRate * hoursWorked
deduction = grossPay * deductionRate
net pay = grossPay – deduction
Write name, gross, deduction, net pay

PSEUDOCODE:

READ name, hourlyRate, hoursWorked, deductionRate
grossPay = hourlyRate * hoursWorked
deduction = grossPay * deductionRate
netPay = grossPay – deduction
WRITE name, grossPay, deduction, netPay

2. Capitalize initial keyword

In the example above, **READ** and **WRITE** are in caps. There are just a few keywords we will use:

READ, WRITE, IF, ELSE, ENDIF, WHILE, ENDWHILE, REPEAT, UNTIL

3. Indent to show hierarchy

We will use a particular indentation pattern in each of the design structures:

SEQUENCE: keep statements that are “stacked” in sequence all starting in the same column.

SELECTION: indent the statements that fall inside the selection structure, but not the keywords that form the selection

LOOPING: indent the statements that fall inside the loop, but not the keywords that form the loop

Eg: In the example above, employees whose *grossPay* is less than 100 do not have any deduction.

Task List:

Read name, hourly rate, hours worked, deduction rate
Compute gross, deduction, net pay
Is gross \geq 100?
YES: calculate deduction
NO: no deduction
Write name, gross, deduction, net pay

Pseudocode:

READ name, hourlyRate, hoursWorked
grossPay = hourlyRate * hoursWorked
IF grossPay \geq 100
 deduction = grossPay * deductionRate
ELSE
 deduction = 0
ENDIF
netPay = grossPay – deduction
WRITE name, grossPay, deduction, netPay

4. End multiline structures

See how the IF/ELSE/ENDIF is constructed above. The ENDIF (or END

	<p>whatever) always is in line with the IF (or whatever starts the structure).</p> <p>5. Keep stmts language independent</p> <p>Resist the urge to write in whatever language you are most comfortable with. In the long run, you will save time! There may be special features available in the language you plan to eventually write the program in; if you are SURE it will be written in that language, then you can use the features. If not, then avoid using the special features</p> <p>Pseudocode uses Keywords (Reserved Words) to control the structure of a solution. Reserved words are written in capitals. Structural elements come in pairs, eg for every BEGIN there is an END, for every IF there is an ENDIF, etc. Indenting is used to show structure in the algorithm.</p>
6	<p>Explain the following programming language</p> <p>i). Machine language ii). Assembly language iii). High level language</p> <p>i) Machine language</p> <p>Machine code is a computer program written in machine language <u>instructions</u> that can be executed directly by a <u>computer's central processing unit</u> (CPU). Each instruction causes the CPU to perform a very specific task, such as a load, a store, a <u>jump</u>, or an <u>ALU</u> operation on one or more units of data in <u>CPU registers</u> or memory.</p> <p>ii) Assembly language</p> <p>Assembly language (or assembler language), is any <u>low-level programming language</u> in which there is a very strong correspondence between the instructions in the language and the <u>architecture's machine code instructions</u>. Assembly code is converted into executable machine code by an <u>assembler</u>. Assembly language instructions usually consist of an <u>opcode</u> mnemonic followed by a list of data, arguments. These are translated by an <u>assembler</u> into <u>machine language</u> instructions that can be loaded into memory and executed.</p> <p>iii) High level language</p> <p>High-level programming languages mean that languages of writing computer instructions in a way that is easily understandable and close to human language. High-level languages are created by developers so that programmers don't need to know highly difficult low level/machine language. Programmers can easily learn high-level languages as it is very close to human language.</p>
7	<p>Neat sketch explain the following building blocks of alg.</p> <p>i). Statements ii). Control Flow</p> <p>i) Statements</p> <p>A computer program statement is an instruction for the computer program to perform an action. There are many different types of statements that can be given in a computer program in order to direct the actions the program performs. In <u>computer programming</u>, a statement is a syntactic unit of an <u>imperative programming language</u> that expresses some action to be carried out. A program written in such a language is formed</p>

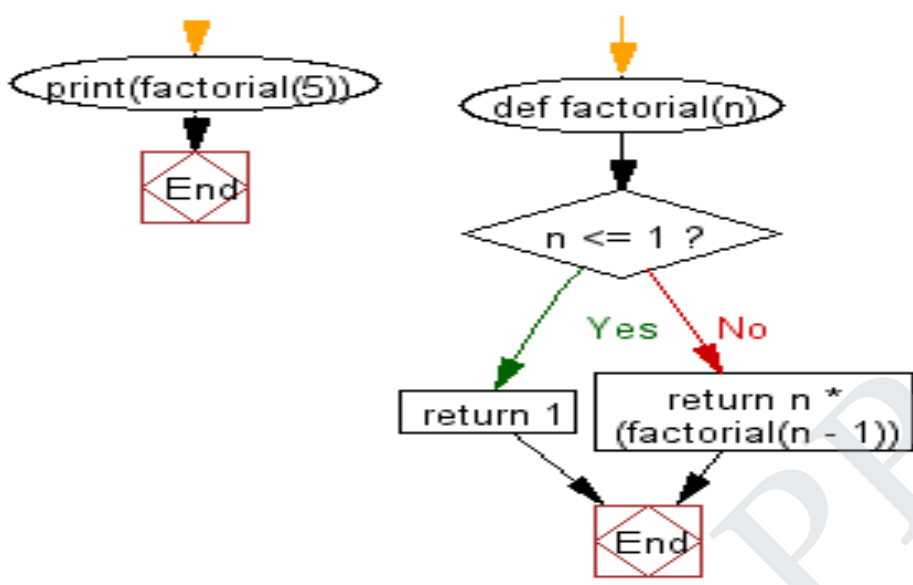
	<p>by a sequence of one or more statements. A statement may have internal components (e.g., <u>expressions</u>).</p> <p>ii) Control flow</p> <p>Control Flow Statements</p> <p>Without control flow statements, the interpreter executes these statements in the order they appear in the file from left to right, top to bottom. You can use <i>control flow statements</i> in your programs to conditionally execute statements, to repeatedly execute a block of statements, and to otherwise change the normal, sequential flow of control.</p>
8	<p>Describe State and function in Building Block and examples.</p> <p>Building blocks of algorithms (statements, state, control flow, functions)</p> <p>Algorithms can be constructed from basic building blocks namely, sequence, selection and iteration.</p> <p><u>Statements:</u></p> <p>Statement is a single action in a computer.</p> <p>In a computer statements might include some of the following actions</p> <ul style="list-style-type: none"> ➤ input data-information given to the program ➤ process data-perform operation on a given input ➤ output data-processed result <p><u>State:</u></p> <p>Transition from one process to another process under specified condition with in a time is called state.</p> <p><u>Control flow:</u></p> <p>The process of executing the individual statements in a given order is called control flow.</p> <p>The control can be executed in three ways</p> <ol style="list-style-type: none"> 1. sequence 2. selection 3. iteration <p><u>Sequence:</u></p> <p>All the instructions are executed one after another is called sequence execution</p> <p><u>Selection:</u></p>

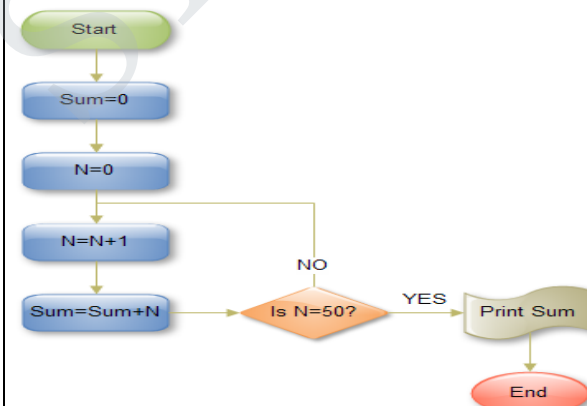
	<p>A selection statement causes the program control to be transferred to a specific part of the program based upon the condition.</p> <p>If the conditional test is true, one part of the program will be executed, otherwise it will execute the other part of the program.</p> <p><u>Iteration:</u></p> <p>In some programs, certain set of statements are executed again and again based upon conditional test. i.e. executed more than one time. This type of execution is called looping or iteration.</p> <p><u>Functions:</u></p> <ul style="list-style-type: none"> ❖ Function is a sub program which consists of block of code(set of instructions) that performs a particular task. ❖ For complex problems, the problem is been divided into smaller and simpler tasks during algorithm design. <p><u>Benefits of Using Functions</u></p> <ul style="list-style-type: none"> ❖ Reduction in line of code ❖ code reuse ❖ Better readability ❖ Information hiding ❖ Easy to debug and test ❖ Improved maintainability <p><u>Example:</u></p> <p><u>Algorithm for addition of two numbers using function</u></p> <p><u>Main function()</u></p> <p>Step 1: Start</p> <p>Step 2: Call the function add()</p> <p>Step 3: Stop</p> <p><u>sub function add()</u></p> <p>Step 1: Function start</p> <p>Step 2: Get a, b Values</p> <p>Step 3: add $c=a+b$</p>
--	--

	<p>Step 4: Print c</p> <p>Step 5: Return</p> <pre> graph TD subgraph HighLevel Start([Start]) --> CallAdd[Call add()] CallAdd --> Stop([Stop]) end subgraph Detailed add([add()]) --> GetAB[/Get a,b/] GetAB --> CeqB[c=a+b] CeqB --> PrintC[/print c/] PrintC --> Return([return]) end </pre>
9	<p>Draw a flow chart print all prime number between to intervals</p> <pre> # Python program to print all # prime number in an interval start = 11 end = 25 for val in range(start, end + 1): # If num is divisible by any number # between 2 and val, it is not prime if val > 1: for n in range(2, val): if (val % n) == 0: break else: print(val) </pre> <p>Output:</p> <pre> 11 13 17 </pre>

	<p>19</p> <p>23</p> <pre> graph TD START([START]) --> INPUT[/INPUT N & M/] INPUT --> WHILE1{WHILE N <= M} WHILE1 -- NO --> NINC[N++] NINC --> STOP([STOP]) WHILE1 -- YES --> I2[I = 2] I2 --> WHILE2{WHILE I <= N} WHILE2 -- NO --> PRINT[/Print NUM/] PRINT --> NINC WHILE2 -- YES --> IF1{IF N % I == 0} IF1 -- NO --> IINC[I++] IF1 -- YES --> IINC IINC --> IF2{IF I == NUM} IF2 -- NO --> PRINT IF2 -- YES --> NINC </pre>
10	<p>i). Describe pseudo code for Fibonacci sequence using</p> <p>ii). Draw a flow chart for factorial given number (3*3)</p> <p># Function for nth Fibonacci number</p> <pre> def Fibonacci(n): if n<0: print("Incorrect input") # First Fibonacci number is 0 elif n==1: return 0 # Second Fibonacci number is 1 </pre>

	<pre>elif n==2: return 1 else: return Fibonacci(n-1)+Fibonacci(n-2) # Driver Program print(Fibonacci(9))</pre> <p>Output:</p> <p>21</p> <p>i) Draw a flowchart for factorial of given Number</p> <p># Python 3 program to find # factorial of given number</p> <pre>def factorial(n): # single line to find factorial return 1 if (n==1 or n==0) else n * factorial(n - 1) # Driver Code num = 5 print ("Factorial of",num,"is", factorial(num))</pre> <p>Output:</p> <p>Factorial of 5 is 120</p>
--	--

	 <pre> graph TD Start([print(factorial(5))]) --> End1{{End}} Def([def factorial(n)]) --> Cond{n <= 1 ?} Cond -- Yes --> Ret1[return 1] Cond -- No --> Ret2[return n * (factorial(n - 1))] Ret1 --> End2{{End}} Ret2 --> End2 </pre>
11	<p>i). Describe the program to insert an element in a sorted list</p> <p>ii). Draw the flow chart sum of n numbers</p> <pre> # Python3 program to insert # an element into sorted list # Function to insert element def insert(list, n): # Searching for the position for i in range(len(list)): if list[i] > n: index = i break # Inserting n in the list list = list[:i] + [n] + list[i:] return list </pre>

	<pre># Driver function list = [1, 2, 4] n = 3 print(insert(list, n))</pre> <p>Output:</p> <pre>[1, 2, 3, 4]</pre> <p>i) Draw the flowchart to find the sum of series $1+2+3+4+\dots+100$</p> <pre># Sum of natural numbers up to num num = 50 if num < 0: print("Enter a positive number") else: sum = 0 # use while loop to iterate until zero while(num > 0): sum += num num -= 1 print("The sum is", sum)</pre> <p>output</p> <pre>The sum is 1275 >>></pre>  <pre>graph TD Start([Start]) --> Sum0[Sum=0] Sum0 --> N0[N=0] N0 --> Nplus1[N=N+1] Nplus1 --> SumplusN[Sum=Sum+N] SumplusN --> IsN50{Is N=50?} IsN50 -- NO --> Nplus1 IsN50 -- YES --> PrintSum[/Print Sum/] PrintSum --> End([End])</pre>
	<p>i). Summarize the difference between algorithm, flow chart and pseudo code</p>

12	<p>ALGORITHM</p> <ul style="list-style-type: none"> • An algorithm is defined as a finite sequence of explicit instructions, which when provided with a set of input values produces an output and then terminates. • To be an algorithm, the steps must be unambiguous and after a finite number of steps, the solution of the problem is achieved. <p>FLOWCHART</p> <ul style="list-style-type: none"> • A flowchart is a pictorial representation of an algorithm in which the steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows. • The boxes represent operations and the arrows represent the sequence in which the operations are implemented. <p>PSEUDOCODE</p> <ul style="list-style-type: none"> • Pseudo code is a generic way of describing an algorithm without using any specific programming language-related notations. • It is an outline of a program, written in a form, which can easily be converted into real programming statements.
13	<p>(i). Explain algorithmic problem solving technique in detail.</p> <p>Algorithms are the solutions to computational problems. They define a method that uses the input to a problem in order to produce the correct output. A computational problem can have many solutions. Efficient algorithms can solve the computational problems more effectively.</p> <p>To harness the power of computers we use <i>programming</i>. Programming is the art of developing a solution to a <i>computational problem</i>, in the form of a set of instructions that a computer can execute. These instructions are what we call <i>code</i>, and the language in which they are written a <i>programming language</i>.</p> <p>The abstract method that such code describes is what we call an <i>algorithm</i>. The aim of <i>algorithmic problem solving</i> is thus to, given a computational problem, devise an algorithm that solves it. One does not necessarily need to complete the full programming process (i.e. write code that implements the algorithm in a programming language) to enjoy solving algorithmic problems. However, it often provides more insight and trains you at finding simpler algorithms to problems.</p>
14	<p>Explain program life cycle</p> <ul style="list-style-type: none"> ➤ Program Development Cycle <ul style="list-style-type: none"> • Development cycle of a program includes: ➤ Analyse/Define the Problem

	<ul style="list-style-type: none"> ➤ Task Analysis ➤ Developing Algorithm ➤ Testing the Algorithm for Accuracy ➤ Coding the Solution ➤ Test and Debug the Program ➤ Documentation ➤ Implementation ➤ Maintenance and Enhancement
PART-C	
1	<p>What is pseudo code? Explain how it can be designed and write benefits and limitations.</p> <p>PSEUDOCODE</p> <ul style="list-style-type: none"> ➤ Pseudo code is a generic way of describing an algorithm without using any specific programming language-related notations. ➤ It is an outline of a program, written in a form, which can easily be converted into real programming statements. <p>Pseudocode is a kind of structured english for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudocode needs to be complete. It describe the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code.</p> <p>In general the vocabulary used in the pseudocode should be the vocabulary of the problem domain, not of the implementation domain. The pseudocode is a narrative for someone who knows the requirements (problem domain) and is trying to learn how the solution is organized.</p> <p>Advantages and benefits of pseudo code:</p> <p>Programming can be a complex process when the requirements of the program are complex in nature. The pseudo code provides a simple method of developing the program logic as it uses every language to prepare a brief set of instructions in the order in which they appear. In the completed program it allows the programmer programmers to focus on</p>

	<p>the steps required to solve a problem rather than on how to use the computer language. Some of the most significant benefits of the Pseudo code are:</p> <ul style="list-style-type: none"> ➤ Since it is a language-independent it can be used by most programmers it allows the developer to express the design in plain and natural language. ➤ It is easier to develop a program from a pseudo code as compared to the flow chart. Programmers do not have to think about syntax, we simply have to concentrate on the underline logic. The focus is on the steps to solve a problem rather than how to use the computer language. ➤ Often it is easy to translate pseudocode into a programming language, a step which can be accomplished by less experienced ➤ The uses of words and phrases in pseudo code, which are in the lines of basic computer operations simplify the translation from the pseudo code algorithm to the specific programming language. ➤ Unlike flow charts, pseudo code is at and does not tend to run over many pages. Its simple structure and readability make it easier to modify. ➤ The pseudocode allows programmers to work in different computer languages to talk to others they can be reviewed by groups easier than the real code. <p>Disadvantages/limitation of Pseudo Code:</p> <p>Although the pseudo code is a very simple mechanism to specify problem-solving logic, it has some of the limitations that are listed below:</p> <ul style="list-style-type: none"> ➤ The main disadvantages are that it does not provide a visual representation of the programming logic. ➤ There are no accepted standards for writing the pseudo code. Programmers use their own styles of writing pseudo code. ➤ The pseudo code cannot be compiled nor executed and there is no real formative of a syntax of rules. It is simply one step, an important one, in producing the final code.
2	<p>Explain guidelines for preparing flowcharts, benefits and limitation of flowcharts and preparing flow chart for quadratic equation</p> <p>Advantages and Disadvantages of Flowchart</p> <p>Flowchart Meaning</p> <p>It is said that a single picture is worth thousands words and flowchart works basically on that concept only as it illustrates solution of complex problems through diagrams and thus helps an individual to understand the concept better, however sometimes it may complicate the solution which in turn will make it even more difficult for an individual to understand the solution of the problem</p> <p>Advantages of Flowchart</p> <p>Short and Simple</p>

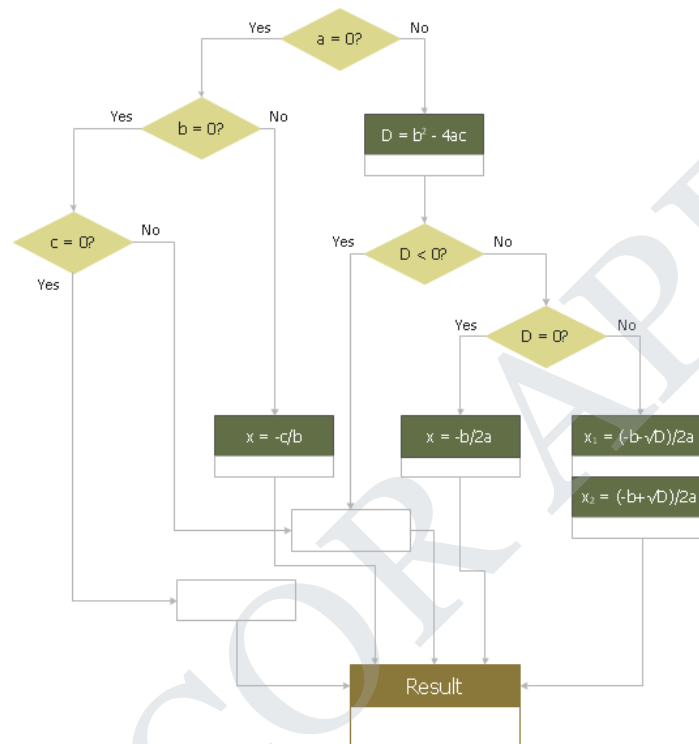
	<p>The biggest advantage of using flowchart is that it is short as well as simple</p> <p>Logical Steps</p> <p>It helps them understand the solution of the problem logically.</p> <p>Effective Communication</p> <p>It is one of the effective ways of communicating because flowchart can be made on 1 or 2 pages only as opposed to other methods of communication like written communication which may take many pages, hence if one wants to save time and communicate effectively than flowcharts can be a good option for them.</p> <p>Disadvantages of Flowchart</p> <p>Not suitable where Solution is long</p> <p>When the solution of the problem is short than it is a good method but if the solution is longer than this may not be the ideal method.</p> <p>Complicate Things</p> <p>One does not understand the solution even when the solution is right due to the wrong presentation through flowcharts.</p> <p>Difficult to Alter</p> <p>Another limitation is that flowcharts are difficult to alter because if there is one mistake than one has to alter the whole flowchart</p>
--	--

Algorithm for Solving Quadratic Equations

$$ax^2 + bx + c = 0$$

Enter the values for a, b, c and see the result

a = b = c =



3

Describe the algorithm for finding sum and average of n numbers.

Sum of natural numbers up to num

num = 50

if num < 0:

 print("Enter a positive number")

else:

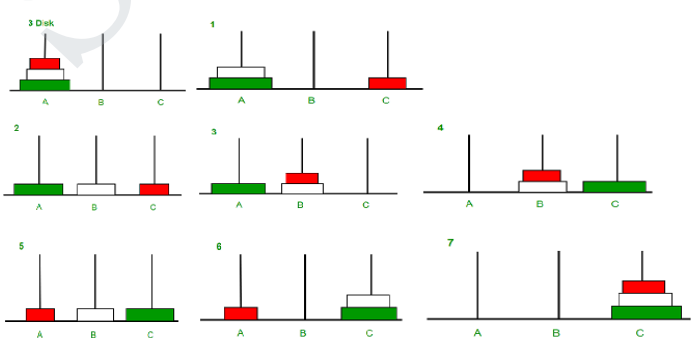
 sum = 0

 # use while loop to iterate until zero

while(num > 0):

 sum += num

	<pre>num -= 1 print("The sum is", sum) output The sum is 1275 >>></pre> <p># Python program to get average of a list</p> <p># Using reduce() and lambda</p> <p># importing reduce()</p> <pre>from functools import reduce</pre> <pre>def Average(lst): return reduce(lambda a, b: a + b, lst) / len(lst)</pre> <p># Driver Code</p> <pre>lst = [15, 9, 55, 41, 35, 20, 62, 49] average = Average(lst)</pre> <p># Printing average of the list</p> <pre>print("Average of the list =", round(average, 2))</pre> <p>Output:</p> <p>Average of the list = 35.75</p> <p># Python program to get average of a list</p> <p># Using mean()</p> <p># importing mean()</p> <pre>from statistics import mean</pre> <pre>def Average(lst): return mean(lst)</pre>
--	---

	<pre># Driver Code lst = [15, 9, 55, 41, 35, 20, 62, 49] average = Average(lst) # Printing average of the list print("Average of the list =", round(average, 2))</pre> <p>Output:</p> <p>Average of the list = 35.75</p> <pre>def cal_average(num): sum_num = 0 for t in num: sum_num = sum_num + t</pre> <p>Also state the properties of a good algorithm</p>
4	<p>Describe the algorithm of towers of Hanoi problem.</p> <p>Tower of Hanoi Problem</p> <p>Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:</p> <ol style="list-style-type: none"> 1) Only one disk can be moved at a time. 2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack. 3) No disk may be placed on top of a smaller disk.  <p># Recursive Python function to solve tower of hanoi</p>

	<pre>def TowerOfHanoi(n , from_rod, to_rod, aux_rod): if n == 1: print "Move disk 1 from rod",from_rod,"to rod",to_rod return TowerOfHanoi(n-1, from_rod, aux_rod, to_rod) print "Move disk",n,"from rod",from_rod,"to rod",to_rod TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)</pre>
--	---

STUCOR APP

GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING	
UNIT 2 - DATA, EXPRESSIONS, STATEMENTS	
SYLLABUS	
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.	
PART-A	
Q. No.	Q&A
1.	<p>Define the two modes in Python.</p> <p>Python has two basic modes: script and interactive. The normal mode is the mode where the scripted and finished .py files are run in the Python interpreter. Interactive mode is a command line shell which gives immediate feedback for each statement, while running previously fed statements in active memory. As new lines are fed into the interpreter, the fed program is evaluated both in part and in whole. Interactive mode is a good way to play around and try variations on syntax.</p> <p>The basic differences between these two modes are as follows: Interactive mode is used when an user wants to run one single line or one block of code. It runs very quickly and gives the output instantly. Script Mode, on the other hand, is used when the user is working with more than one single code or a block of code</p>
2.	<p>Give the various data types in Python</p> <p>Python Data Types</p> <p>There are different types of python data types. Some built-in python data types are:</p> <p>Python Data Type – Numeric</p> <p>Python numeric data type is used to hold numeric values like;</p> <p>int – holds signed integers of non-limited length.</p> <p>long- holds long integers(exists in Python 2.x, deprecated in Python 3.x).</p> <p>float- holds floating precision numbers and it's accurate upto 15 decimal places.</p> <p>complex- holds complex numbers.</p> <p>In Python we need not to declare datatype while declaring a variable like C or C++. We can simply just assign values in a variable.</p> <p>Python Data Type – String</p> <p>The string is a sequence of characters. Python supports Unicode characters. Generally, strings are represented by either single or double quotes.</p> <p>Python Data Type – List</p> <p>The list is a versatile data type exclusive in Python. In a sense, it is the same as the array in C/C++. But the interesting thing about the list in Python is it can simultaneously hold different types of data. Formally list is an ordered sequence of some data written using square brackets([]) and commas(.).</p> <p>Python Data Type – Tuple</p> <p>Tuple is another data type which is a sequence of data similar to list. But it is immutable. That means data in a tuple is write protected. Data in a tuple is</p> <p>Dictionary</p> <p>Python Dictionary is an unordered sequence of data of key-value pair form. It is similar to the hash table type. Dictionaries are written within curly braces in the form key:value.</p>
3.	<p>Point out the rules to be followed for naming any identifier</p> <p>A Python identifier is a name used to identify a variable, function, class, module or other object.</p> <p>An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).</p>

4.	<p>Assess a program to assign and access variables</p> <p>Python Variables</p> <p><u>Python</u> is not “statically typed”. We do not need to declare variables before using them, or declare their type. A variable is created the moment we first assign a value to it.</p> <p><i>The following is a sample program used to assign and access variables :</i></p> <pre># An integer assignment age = 45 # A floating point salary = 1456.8 # A string name = "John" print(age) print(salary) print(name)</pre> <p>Output:</p> <pre>45 1456.8 John</pre>
5.	<p>Compose the importance of indentation in Python</p> <p>Python Indentation</p> <p>Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.</p> <p>A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.</p> <p>Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.</p> <p>Example code</p> <pre>for i in range(1,11): print(i) if i == 5: break</pre>
6.	<p>Select and assign how an input operation was done in Python.</p> <p>input () : This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python. For example –</p> <p># Python program showing a use of input()</p> <pre>val = input("Enter your value: ") print(val)</pre> <p>Output:</p>

```
Enter your value: 123
123
>>>
```

How the input function works in Python :

- When input() function executes program flow will be stopped until the user has given an input.
- The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
- Whatever you enter as input, input function convert it into a string. if you enter an integer value still input() function convert it into a string. You need to explicitly convert it into an integer in your code using **typecasting**.

7. Demonstrate the various operations in Python

Basic Operators in Python

1. **Arithmetic operators:** Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

OPERATOR	DESCRIPTION	SYNTAX
+	Addition: adds two operands	$x + y$
-	Subtraction: subtracts two operands	$x - y$
*	Multiplication: multiplies two operands	$x * y$
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	$x // y$
%	Modulus: returns the remainder when first operand is divided by the second	$x \% y$

2. **Relational Operators:** Relational operators compares the values. It either returns **True** or **False** according to the condition.

OPERATOR	DESCRIPTION	SYNTAX
>	Greater than: True if left operand is greater than the right	$x > y$
<	Less than: True if left operand is less than the right	$x < y$
==	Equal to: True if both operands are equal	$x == y$
!=	Not equal to - True if operands are not equal	$x != y$
>=	Greater than or equal to: True if left operand is greater than or equal to the right	$x >= y$
<=	Less than or equal to: True if left operand is less than or equal to the right	$x <= y$

3. **Logical operators:** Logical operators perform **Logical AND**, **Logical OR** and **Logical NOT** operations.

OPERATOR	DESCRIPTION	SYNTAX
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

4. **Bitwise operators:** Bitwise operators acts on bits and performs bit by bit operation.

OPERATOR	DESCRIPTION	SYNTAX
&	Bitwise AND	x & y
	Bitwise OR	x y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

5. **Assignment operators:** Assignment operators are used to assign values to the variables.

OPERATOR	DESCRIPTION	SYNTAX
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right side operand with left side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a=b a=a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right	a//=b a=a//b

	operand and then assign the value(floor) to left operand														
=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a=b	a=a**b												
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b	a=a&b												
=	Performs Bitwise OR on operands and assign value to left operand	a =b	a=a b												
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b	a=a^b												
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b	a=a>>b												
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b << b	a= a												
6.	Special operators: There are some special type of operators like- <ul style="list-style-type: none">Identity operators- is and is not are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.is True if the operands are identicalis not True if the operands are not identical														
8.	Discover the difference between logical and bitwise operator. Logical operators Logical operators are the and, or, not operators. <table><tr><th>Operator</th><th>Meaning</th><th>Example</th></tr><tr><td>and</td><td>True if both the operands are true</td><td>x and y</td></tr><tr><td>or</td><td>True if either of the operands is true</td><td>x or y</td></tr><tr><td>not</td><td>True if operand is false (complements the operand)</td><td>not x</td></tr></table> <div>Logical operators in Python</div> Example #3: Logical Operators in Python <pre>1. x = True 2. y = False 3. 4. # Output: x and y is False</pre>			Operator	Meaning	Example	and	True if both the operands are true	x and y	or	True if either of the operands is true	x or y	not	True if operand is false (complements the operand)	not x
Operator	Meaning	Example													
and	True if both the operands are true	x and y													
or	True if either of the operands is true	x or y													
not	True if operand is false (complements the operand)	not x													

```
5. print('x and y is',x and y)
```

```
6.
```

```
7. # Output: x or y is True
```

```
8. print('x or y is',x or y)
```

```
9.
```

```
10. # Output: not x is False
```

```
11. print('not x is',not x)
```

Here is the [truth table](#) for these operators.

Bitwise operators

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

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

In the table below: Let $x = 10$ (0000 1010 in binary) and $y = 4$ (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	$x \& y = 0$ (0000 0000)
	Bitwise OR	$x y = 14$ (0000 1110)
~	Bitwise NOT	$\sim x = -11$ (1111 0101)
^	Bitwise XOR	$x \wedge y = 14$ (0000 1110)
>>	Bitwise right shift	$x >> 2 = 2$ (0000 0010)
<<	Bitwise left shift	$x << 2 = 40$ (0010 1000)
Bitwise operators in Python		

9. **What is a tuple? How literals of type tuple are written? Give examples.**

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example –

```
tup1 = ('physics', 'chemistry', 1997, 2000);
```

```
tup2 = (1, 2, 3, 4, 5 );
```

```
tup3 = "a", "b", "c", "d";
```

10. Give the operator precedence in Python.

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

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

Operator precedence affects how an expression is evaluated.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first multiplies $3*2$ and then adds into 7.

11.	<p>Define the scope and lifetime of a variable in Python.</p> <p>Variable scope and lifetime</p> <p>Not all variables are accessible from all parts of our program, and not all variables exist for the same amount of time. Where a variable is accessible and how long it exists depend on how it is defined. We call the part of a program where a variable is accessible its <i>scope</i>, and the duration for which the variable exists its <i>lifetime</i>.</p> <p>A variable which is defined in the main body of a file is called a <i>global</i> variable. It will be visible throughout the file, and also inside any file which imports that file. Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them. Only objects which are intended to be used globally, like functions and classes, should be put in the global namespace.</p> <p>A variable which is defined inside a function is <i>local</i> to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.</p>
12.	<p>Point out the uses of default arguments in Python.</p> <div data-bbox="154 947 776 1283"> <p>DEFAULT ARGUMENTS</p> <ul style="list-style-type: none"> • Default value to function argument is passed using assignment operator '='. <p>Example:</p> <pre>def greet(name, msg = "Good morning!"): print("Hello, " name + ', ' + msg)</pre> <ul style="list-style-type: none"> • Non-default argument cannot follow default argument <p>Example: <code>def greet(msg = "Good morning!", name):</code> SyntaxError: non-default argument follows default argument</p> </div>
13.	<p>Generalize the uses of Python module.</p> <p>Python Language Modules</p> <ol style="list-style-type: none"> 1) What is Python Module? 2) Purpose of Modules 3) Types of Modules 4) How to use Modules 5) User defined Modules in Python <p>1) What is Python Module?</p> <p>> A module is a file consisting of Python code. It can define functions, classes, and variables, and can also include runnable code. Any Python file can be referenced as a module.</p> <p>> A file containing Python code, for example: test.py, is called a module, and its name would be test..</p> <p>Module vs. Function</p> <p>Function: it's a block of code that you can use/reuse by calling it with a keyword. Eg. print() is a function.</p> <p>Module: it's a .py file that contains a list of functions (it can also contain variables and classes). Eg. in</p>

statistics.mean(a), mean is a function that is found in the statistics module.

2) Purpose of Modules

> As our program grows more in the size we may want to split it into several files for easier maintenance as well as re-usability of the code. The solution to this is Modules.

> We can define your most used functions in a module and import it, instead of copying their definitions into different programs. A module can be imported by another program to make use of its functionality. This is how you can use the Python standard library as well.

3) Types of Modules

> Python provides us with some built-in modules, which can be imported by using the “import” keyword.

> Python also allows us to create your own modules and use them in your programs.

4) How to use Modules

There is a Python Standard Library with dozens of built-in modules. From those, five important modules,

random, statistics, math, datetime, csv

Python math module,

This contains factorial, power, and logarithmic functions, but also some trigonometry and constants.

i) import math

And then:

```
math.factorial(5)
math.pi
math.sqrt(5)
math.log(256, 2)
```

ii) import math as m

And then:

```
m.factorial(5)
m.pi
m.sqrt(5)
m.log(256, 2)
```

5) User defined Modules in Python

i) Create a module

A simple module, calc.py

```
price=1000
```

```
def add(x, y):
    return (x+y)
```

```
def sub(x, y):
    return (x-y)
```

```
def mul(x, y):
    return (x*y)
```

ii) Use Module

```
# importing module calc.py
```

```
import calc
```

```
print calc.add(10, 2)
```

```
# from calc import mul
```

```
print(add(10, 2))
```

14. Demonstrate how a function calls another function.

```
Python10.1.py
1 #define a function
2 def func1():
3     print("I am learning Python Function")
4
5 func1()
6 #print func1()
7 #print func1
8
9
```

Function definition

Function Call

Run Python10.1

"C:\Users\DK\Desktop\Python code\Python Test\Python 10\Python10
10\Python10 Code\Python10.1.py"

I am learning Python Function

Function output

15. List the syntax for function call with and without Arguments

Functions in Python

A function in Python is defined with the **def** keyword. Functions do not have declared return types. A function without an explicit **return** statement returns **None**. In the case of no arguments and no return value, the definition is very simple.

Function call without arguments

Calling the function is performed by using the call operator **()** after the name of the function.

```
>>> def hello_function():
...     print 'Hello World, it\'s me. Function.'
...
>>> hello_function()
Hello World, it's me. Function.
```

Function call with arguments

```
# A simple Python function to check
# whether x is even or odd
def evenOdd( x ):
    if (x % 2 == 0):
        print "even"
    else:
        print "odd"
```

```
# Driver code
evenOdd(2)
evenOdd(3)
```

Output:

```
even
```

```
odd
```

16. Define recursive function

In programming, recursion is when a function calls itself.

```
1. >>> def factorial(n):
2.     if n==1:
3.         return 1
4.         return n*factorial(n-1)
5. >>> factorial(5)
```

```
120
```

	<p>Python Recursion Function – Pros & Cons</p> <p>a. Python Recursion Function Advantages</p> <p>With Python recursion, there are some benefits we observe:</p> <ol style="list-style-type: none"> 1. A recursive code has a cleaner-looking code. 2. Recursion makes it easier to code, as it breaks a task into smaller ones. 3. It is easier to generate a sequence using recursion than by using nested iteration. <p>b. Python Recursion Function Disadvantages</p> <p>The flip side of the coin is easy to quote:</p> <ol style="list-style-type: none"> 1. Although it makes code look cleaner, it may sometimes be hard to follow. 2. They may be simpler, but recursive calls are expensive. They take up a lot of memory and time. 3. Finally, it isn't as easy to debug a recursive function.
17.	<p>Define the syntax for passing arguments</p> <p>The special syntax <code>*args</code> in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.</p> <ul style="list-style-type: none"> • Example for usage of *arg: <pre># Python program to illustrate # *args with first extra argument def myFun(arg1, *argv): print ("First argument :", arg1) for arg in argv: print("Next argument through *argv :", arg) myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')</pre> <p>Output:</p> <pre>First argument : Hello Next argument through *argv : Welcome Next argument through *argv : to Next argument through *argv : GeeksforGeeks</pre>
18.	<p>What are the two parts of function definition. Give its syntax.</p> <pre>"""Function definition and invocation.""" def happyBirthdayEmily(): print("Happy Birthday to you!") print("Happy Birthday to you!") print("Happy Birthday, dear Emily.") print("Happy Birthday to you!") happyBirthdayEmily() happyBirthdayEmily()</pre>

"Function with parameter called in main"

```
def happyBirthday(person):
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear " + person + ".")
    print("Happy Birthday to you!")
```

```
def main():
    happyBirthday('Emily')
    happyBirthday('Andre')
```

```
main()
```

19. Point out the difference between recursive and iterative technique.

Recursive vs. Iterative Algorithms

The following highlights the difference between two types of algorithms: Iterative and Recursive algorithms.

The challenge we will focus on is to define a function that returns the result of $1+2+3+4+\dots+n$ where n is a parameter.

The Iterative Approach

The following code uses a loop – in this case a counting loop, aka a “For Loop”. This is the main characteristic of iterative code: it uses loops.

Iterative Approach

Python

```
1 # iterative Function (Returns the result of: 1 +2+3+4+5+...+n)
2 def iterativeSum(n):
3     total=0
4     for i in range(1,n+1):
5         total += i
6     return total
```

The Recursive Approach

The following code uses a function that calls itself. This is the main characteristic of a recursive approach.

```
1 # Recursive Function (Returns the result of: 1 +2+3+4+5+...+n)
2 def recursiveSum(n):
3     if (n > 1):
4         return n + recursiveSum(n - 1)
5     else:
6         return n
```

OUTPUT

Using an iterative approach

1+2+3+4+...+99+100=

5050

Using a recursive approach

1+2+3+4+...+99+100=

5050

20. **Give the syntax for variable length arguments.**

Python *args

Python has `*args` which allow us to pass the variable number of non keyword arguments to function.

In the function, we should use an asterisk `*` before the parameter name to pass variable length arguments. The arguments are passed as a tuple and these passed arguments make tuple inside the function with same name as the parameter excluding asterisk `*`.

Example : Using *args to pass the variable length arguments to the function

```
1. def adder(*num):
2.     sum = 0
3.
4.     for n in num:
5.         sum = sum + n
6.
7.     print("Sum:",sum)
8.
9. adder(3,5)
10. adder(4,5,6,7)
11. adder(1,2,3,5,6)
```

When we run the above program, the output will be

Sum: 8
Sum: 22
Sum: 17

In the above program, we used `*num` as a parameter which allows us to pass variable length argument list to the `adder()` function. Inside the function, we have a loop which adds the passed argument and prints the result. We passed 3 different tuples with variable length as an argument to the function.

PART-B

1. **i) Illustrate a program to display different data types using variables and literals constants.**

Python Data Types

A Data Type describes the characteristic of a variable.

Python has six standard Data Types:

- Numbers
- String
- List
- Tuple
- Set
- Dictionary

Data Type	Example	Output
#1) Numbers In Numbers, there are mainly 3 types which include Integer, Float, and Complex.	Example: 1 a = 5 2 print(a, "is of type", type(a))	Output: 5 is of type <class 'int'>
	1 b = 2.5 2 print(b, "is of type", type(b))	Output: 2.5 is of type <class 'float'>
	1 c = 6+2j 2 print(c, "is a type", type(c))	Output: (6+2j) is a type <class 'complex'>
#2) String A string is an ordered sequence of characters.	Example: 1 String1 = "Welcome" 2 String2 = "To Python" 3 print(String1+String2)	Output: Welcome To Python
#3) List A list can contain a series of values. List variables are declared by using brackets []. A list is mutable, which means we can modify the list.	Example: 1 List = [2,4,5.5,"Hi"] 2 print("List[2] = ", List[2])	Output: List[2] = 5.5
#4) Tuple A tuple is a sequence of Python objects separated by commas. Tuples are immutable, which means tuples once created cannot be modified. Tuples are defined using parentheses ().	Example: 1 Tuple = (50,15,25.6,"Python") 2 print("Tuple[1] = ", Tuple[1]) 3 print("Tuple[0:3] = ", Tuple[0:3])	Output: Tuple[1] = 15 Output: Tuple[0:3] = (50, 15, 25.6)
#5) Set A set is an unordered collection of items. Set is	Example: 1 Set = {5,1,2.6,"python"}	Output: {'python', 1, 5, 2.6}

defined by values separated by a comma inside braces { }.	2 print(Set)	
<p>#6) Dictionary</p> <p>Dictionaries items are stored and fetched by using the key. Dictionaries are used to store a huge amount of data. To retrieve the value we must know the key. In Python, dictionaries are defined within braces { }.</p> <p>We use the key to retrieve the respective value. But not the other way around.</p>	<p>Syntax: Key:value Example: 1 Dict = {1:'Hi',2:7.5, 3:'Class'} print(Dict)</p> <p>2 Example: 1 print(Dict[2])</p>	<p>Output: {1: 'Hi', 2: 7.5, 3: 'Class'}</p> <p>Output: 7.5</p>

ii) Show how an input and output function is performed in Python with an example.

Python Input, Output and Import

Python provides functions `input()` and `print()` that are used for standard input and output operations respectively.

Python Output Using `print()` function

We use the `print()` function to output data to the standard output device (screen).

```
print('This sentence is output to the screen')
# Output: This sentence is output to the screen
a = 5
print('The value of a is', a)
# Output: The value of a is 5
```

```
print(1,2,3,4)
# Output: 1 2 3 4
```

```
print(1,2,3,4,sep='*')
# Output: 1*2*3*4
```

```
print(1,2,3,4,sep='#',end='&')
# Output: 1#2#3#4&
```

Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the `str.format()` method. This method is visible to any string object.

```
1. >>> x = 5; y = 10
2. >>> print('The value of x is {} and y is {}'.format(x,y))
3. The value of x is 5 and y is 10
```

Here the curly braces { } are used as placeholders. We can specify the order in which it is printed by

```
print('I love {0} and {1}'.format('bread','butter'))
# Output: I love bread and butter
print('I love {1} and {0}'.format('bread','butter'))
# Output: I love butter and bread
```

Python Input

To allow flexibility we might want to take the input from the user.

In Python, we have the input() function to allow this. The syntax for input() is

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

```
1. >>> num = input('Enter a number: ')
2. Enter a number: 10
3. >>> num
4. '10'
```

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

```
1. >>> int('10')
2. 10
3. >>> float('10')
4. 10.0
```

Python Import

When our program grows bigger, it is a good idea to break it into different modules.

A module is a file containing Python definitions and statements. [Python modules](#) have a filename and end with the extension .py.

Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.

For example, we can import the math module by typing in import math.

```
import math
print(math.pi)
```

Now all the definitions inside math module are available in our scope. We can also import some specific attributes and functions only, using the from keyword. For example:

```
1. >>> from math import pi
2. >>> pi
3. 3.141592653589793
```

2. Explain in detail about the various operators in python with suitable examples.

Operator in Python

#1: Arithmetic operators in Python

#2: Comparison operators in Python

#3: Logical Operators in Python

<pre> 1. x = 15 2. y = 4 3. 4. # Output: x + y = 19 5. print('x + y =',x+y) 6. 7. # Output: x - y = 11 8. print('x - y =',x-y) 9. 10. # Output: x * y = 60 11. print('x * y =',x*y) 12. 13. # Output: x / y = 3.75 14. print('x / y =',x/y) 15. 16. # Output: x // y = 3 17. print('x // y =',x//y) 18. 19. # Output: x ** y = 50625 20. print('x ** y =',x**y) </pre>	<pre> 1. x = 10 2. y = 12 3. 4. # Output: x > y is False 5. print('x > y is',x>y) 6. 7. # Output: x < y is True 8. print('x < y is',x<y) 9. 10. # Output: x == y is False 11. print('x == y is',x==y) 12. 13. # Output: x != y is True 14. print('x != y is',x!=y) 15. 16. # Output: x >= y is False 17. print('x >= y is',x>=y) 18. 19. # Output: x <= y is True 20. print('x <= y is',x<=y) </pre>	<pre> 1. x = True 2. y = False 3. 4. # Output: x and y is False 5. print('x and y is',x and y) 6. 7. # Output: x or y is True 8. print('x or y is',x or y) 9. 10. # Output: not x is False 11. print('not x is',not x) </pre>																														
<p>Bitwise operators</p> <p>Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.</p> <p>For example, 2 is 10 in binary and 7 is 111.</p> <p>In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)</p> <table border="1"> <thead> <tr> <th>Operator</th><th>Meaning</th><th>Example</th></tr> </thead> <tbody> <tr> <td>&</td><td>Bitwise AND</td><td>x & y = 0 (0000 0000)</td></tr> <tr> <td> </td><td>Bitwise OR</td><td>x y = 14 (0000 1110)</td></tr> <tr> <td>~</td><td>Bitwise NOT</td><td>~x = -11 (1111 0101)</td></tr> </tbody> </table>	Operator	Meaning	Example	&	Bitwise AND	x & y = 0 (0000 0000)		Bitwise OR	x y = 14 (0000 1110)	~	Bitwise NOT	~x = -11 (1111 0101)	<p>Assignment operators are used in Python to assign values to variables.</p> <p>a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.</p> <p>There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.</p> <table border="1"> <thead> <tr> <th>Operator</th><th>Example</th><th>Equivalent to</th></tr> </thead> <tbody> <tr> <td>=</td><td>x = 5</td><td>x = 5</td></tr> <tr> <td>+=</td><td>x += 5</td><td>x = x + 5</td></tr> <tr> <td>-=</td><td>x -= 5</td><td>x = x - 5</td></tr> <tr> <td>*=</td><td>x *= 5</td><td>x = x * 5</td></tr> <tr> <td>/=</td><td>x /= 5</td><td>x = x / 5</td></tr> </tbody> </table>	Operator	Example	Equivalent to	=	x = 5	x = 5	+=	x += 5	x = x + 5	-=	x -= 5	x = x - 5	*=	x *= 5	x = x * 5	/=	x /= 5	x = x / 5	<p>#5: Membership operators in Python</p> <pre> 1. x = 'Hello world' 2. y = {1:'a',2:'b'} 3. 4. # Output: True 5. print('H' in x) 6. 7. # Output: True 8. print('hello' not in x) 9. 10. # Output: True 11. print(1 in y) 12. 13. # Output: False 14. print('a' in y) </pre>
Operator	Meaning	Example																														
&	Bitwise AND	x & y = 0 (0000 0000)																														
	Bitwise OR	x y = 14 (0000 1110)																														
~	Bitwise NOT	~x = -11 (1111 0101)																														
Operator	Example	Equivalent to																														
=	x = 5	x = 5																														
+=	x += 5	x = x + 5																														
-=	x -= 5	x = x - 5																														
*=	x *= 5	x = x * 5																														
/=	x /= 5	x = x / 5																														

	^	Bitwise XOR	x ^ y = 14 (0000 1110)	%=	x %= 5	x = x % 5			
				//=	x //= 5	x = x // 5			
	>>	Bitwise right shift	x>> 2 = 2 (0000 0010)	**=	x **= 5	x = x ** 5			
				&=	x &= 5	x = x & 5			
	<<	Bitwise left shift	x<< 2 = 40 (0010 1000)	=	x = 5	x = x 5			
				^=	x ^= 5	x = x ^ 5			
				>>=	x >>= 5	x = x >> 5			
				<<=	x <<= 5	x = x << 5			

3.

Python List

List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

Declaring a list

Items separated by commas are enclosed within brackets [].

```
1. >>> a = [1, 2.2, 'python']
```

We can use the slicing operator [] to extract an item or a range of items from a list. Index starts form 0 in Python.

```
a = [5,10,15,20,25,30,35,40]
# a[2] = 15
print("a[2] = ", a[2])
# a[0:3] = [5, 10, 15]
print("a[0:3] = ", a[0:3])
# a[5:] = [30, 35, 40]
print("a[5:] = ", a[5:])
```

Lists are mutable, meaning, value of elements of a list can be altered.

```
1. >>> a = [1,2,3]
2. >>> a[2]=4
3. >>> a
4. [1, 2, 4]
```

ii) Discuss the various operation that can be performed on a tuple and lists (minimum 5) with an example program	
LIST OPERATIONS	
List Operations	Examples
Adding and Appending <ul style="list-style-type: none"> append(): Used for appending and adding elements to List. It is used to add elements to the last position of List. Syntax: list.append (element) 	<pre># Adds List Element as value of List. List = ['Mathematics', 'chemistry', 1997, 2000] List.append(20544) print(List) Output: ['Mathematics', 'chemistry', 1997, 2000, 20544]</pre>
<ul style="list-style-type: none"> insert(): Inserts an elements at specified position. Syntax: list.insert(<position, element) 	<pre>List = ['Mathematics', 'chemistry', 1997, 2000] # Insert at index 2 value 10087 List.insert(2,10087) print(List) Output: ['Mathematics', 'chemistry', 10087, 1997, 2000, 20544]</pre>
<ul style="list-style-type: none"> extend(): Adds contents to List2 to the end of List1. Syntax: List1.extend(List2) 	<pre>List1 = [1, 2, 3] List2 = [2, 3, 4, 5] # Add List2 to List1 List1.extend(List2) print(List1) #Add List1 to List2 now List2.extend(List1) print(List2) Output: [1, 2, 3, 2, 3, 4, 5] [2, 3, 4, 5, 1, 2, 3, 2, 3, 4, 5]</pre>
<ul style="list-style-type: none"> sum() : Calculates sum of all the elements of List. Syntax: sum(List) 	<pre>List = [1, 2, 3, 4, 5] print(sum(List)) Output: 15</pre>
<ul style="list-style-type: none"> count(): Calculates total occurrence of given element of List. Syntax: List.count(element) 	<pre>List = [1, 2, 3, 1, 2, 1, 2, 3, 2, 1] print(List.count(1)) Output: 4</pre>
<ul style="list-style-type: none"> length: Calculates total length of List. Syntax: 	<pre>List = [1, 2, 3, 1, 2, 1, 2, 3, 2, 1] print(len(List))</pre>

	len(list_name)	Output: 10	
	<ul style="list-style-type: none"> index(): Returns the index of first occurrence. Start and End index are not necessary parameters. Syntax: List.index(element[,start[,end]]) 	List = [1, 2, 3, 1, 2, 1, 2, 3, 2, 1] print(List.index(2)) Output: 1	
	<ul style="list-style-type: none"> min() : Calculates minimum of all the elements of List. Syntax: min(List) 	List = [2.3, 4.445, 3, 5.33, 1.054, 2.5] print(min(List)) Output: 1.054	
	<ul style="list-style-type: none"> max(): Calculates maximum of all the elements of List. Syntax: max(List) 	List = [2.3, 4.445, 3, 5.33, 1.054, 2.5] print(max(List)) Output: 5.33	
	<ul style="list-style-type: none"> reverse(): Sort the given data structure (both tuple and list) in ascending order. Key and reverse_flag are not necessary parameter and reverse_flag is set to False, if nothing is passed through sorted(). Syntax: sorted([list[,key[,Reverse_Flag]]]) list.sort([key[,Reverse_flag]]) 	List = [2.3, 4.445, 3, 5.33, 1.054, 2.5] #Reverse flag is set True List.sort(reverse=True) #List.sort().reverse(), reverses the sorted list print(List) Output: [5.33, 4.445, 3, 2.5, 2.3, 1.054] List = [2.3, 4.445, 3, 5.33, 1.054, 2.5] sorted(List) print(List) Output: [1.054, 2.3, 2.5, 3, 4.445, 5.33]	
	Deletion of List Elements To Delete one or more elements, i.e. remove an element, many built in functions can be used, such as pop() & remove() and keywords such as del.	<ul style="list-style-type: none"> pop(): Index is not a necessary parameter, if not mentioned takes the last index. Syntax: list.pop([index]) List = [2.3, 4.445, 3, 5.33, 1.054, 2.5] print(List.pop()) 	

		Output: 2.5	
<ul style="list-style-type: none"> del() : Element to be deleted is mentioned using list name and index. Syntax: del list.[index] remove(): Element to be deleted is mentioned using list name and element. Syntax: list.remove(element) 		List = [2.3, 4.445, 3, 5.33, 1.054, 2.5] del List[0] print(List) Output: [4.445, 3, 5.33, 1.054, 2.5] List = [2.3, 4.445, 3, 5.33, 1.054, 2.5] List.remove(3) print(List) Output: [2.3, 4.445, 5.33, 1.054, 2.5]	
Tuples in Python <p>A Tuple is a collection of Python objects separated by commas. In someways a tuple is similar to a list in terms of indexing, nested objects and repetition but a tuple is immutable unlike lists which are mutable.</p>			
Tuple Operation			
Creating Tuples <pre># An empty tuple empty_tuple = () print (empty_tuple) Output: () # Creating non-empty tuples # One way of creation tup = 'python', 'geeks' print(tup) # Another for doing the same tup = ('python', 'geeks') print(tup) Output ('python', 'geeks') ('python', 'geeks')</pre>			
Concatenation of Tuples <pre># Code for concatenating 2 tuples</pre>			

<pre> tuple1 = (0, 1, 2, 3) tuple2 = ('python', 'geek') # Concatenating above two print(tuple1 + tuple2) Output: (0, 1, 2, 3, 'python', 'geek') </pre>	
<p>Nesting of Tuples</p> <pre> # Code for creating nested tuples tuple1 = (0, 1, 2, 3) tuple2 = ('python', 'geek') tuple3 = (tuple1, tuple2) print(tuple3) Output : ((0, 1, 2, 3), ('python', 'geek')) </pre> <p>Repetition in Tuples</p> <pre> # Code to create a tuple with repetition tuple3 = ('python',)*3 print(tuple3) Output ('python', 'python', 'python') </pre>	
<p>Using cmp(), max() , min()</p> <pre> # A python program to demonstrate the use of # cmp(), max(), min() tuple1 = ('python', 'geek') tuple2 = ('coder', 1) if (cmp(tuple1, tuple2) != 0): # cmp() returns 0 if matched, 1 when not tuple1 # is longer and -1 when tuple1 is shorter print('Not the same') else: print('Same') print ('Maximum element in tuples 1,2: ' + str(max(tuple1)) + ',' + str(max(tuple2))) print ('Minimum element in tuples 1,2: ' + str(min(tuple1)) + ',' + str(min(tuple2))) Output </pre>	

	<div>Not the same</div> <div>Maximum element in tuples 1,2: python,coder</div> <div>Minimum element in tuples 1,2: geek,1</div>																		
4.	<div><div>i) What is a numeric literal? Give examples.</div><div><div>Numeric Literals</div><div>You can refer to numeric values using integers, floating point numbers, scientific notation, hexadecimal notation, octal, and complex numbers: Python integers can be any size. Integers larger than 2147483647 are called "long" integers because they can't be stored in 32 bits.</div><div>123 # an integer</div><div>1.23 # a floating point number</div><div>-1.23 # a negative floating point number</div><div>1.23E45; # scientific notation</div><div>0x7b; # hexadecimal notation (decimal 123)</div><div>0173; # octal notation (decimal 123)</div><div>12+3*j; # complex number 12 + 3i (Note that Python uses "j"!) </div><div>2147483648L # a long integer</div></div><div><div>ii) Describe the arithmetic operators in Python with an example.</div><div><div>1. Arithmetic operators: Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.</div><table><tr><th>OPERATOR</th><th>DESCRIPTION</th><th>SYNTAX</th></tr><tr><td>+</td><td>Addition: adds two operands</td><td>x + y</td></tr><tr><td>-</td><td>Subtraction: subtracts two operands</td><td>x - y</td></tr><tr><td>*</td><td>Multiplication: multiplies two operands</td><td>x * y</td></tr><tr><td>/</td><td>Division (float): divides the first operand by the second</td><td>x / y</td></tr><tr><td>//</td><td>Division (floor): divides the first operand by the second</td><td>x // y</td></tr></table></div></div></div>	OPERATOR	DESCRIPTION	SYNTAX	+	Addition: adds two operands	x + y	-	Subtraction: subtracts two operands	x - y	*	Multiplication: multiplies two operands	x * y	/	Division (float): divides the first operand by the second	x / y	//	Division (floor): divides the first operand by the second	x // y
OPERATOR	DESCRIPTION	SYNTAX																	
+	Addition: adds two operands	x + y																	
-	Subtraction: subtracts two operands	x - y																	
*	Multiplication: multiplies two operands	x * y																	
/	Division (float): divides the first operand by the second	x / y																	
//	Division (floor): divides the first operand by the second	x // y																	

	%	Modulus: returns the remainder when first operand is divided by the second	x % y	
	# Examples of Arithmetic Operator a = 9 b = 4 # Addition of numbers add = a + b # Subtraction of numbers sub = a - b # Multiplication of number mul = a * b # Division(float) of number div1 = a / b # Division(floor) of number div2 = a // b # Modulo of both number mod = a % b	# print results print(add) print(sub) print(mul) print(div1) print(div2) print(mod)	Output: 13 5 36 2.25 2 1	
5.	<p>Demonstrate the various expressions in Python with suitable examples. An expression is an instruction that <i>combines values and operators</i> and <i>always evaluates down to a single value</i>.</p> <p>Statements and expressions</p> <p>A statement is an instruction that the Python interpreter can execute. Examples of statements are , the assignment statement ,</p> <p>the import statement. S, if statements, while statements, and for statements.</p> <p>When you type a statement on the command line, Python executes it.</p> <p>An expression is a combination of values, variables, operators, and calls to functions. If you type an expression at the Python prompt,</p> <p>the interpreter evaluates it and displays the result, which is always a <i>value</i>:</p> <p>Python expressions</p> <pre>length = 5 breadth = 2 area = length * breadth print('Area is', area) print('Perimeter is', 2 * (length + breadth))</pre>			

6.	<p>i) What is membership and identity operators. <u>Python Membership(in, not in) & Identity Operators (is, is not)</u></p> <p>Membership Operators Membership operators are operators used to validate the membership of a value. It test for membership in a sequence, such as strings, lists, or tuples.</p>		
	<table border="1"> <tr> <td data-bbox="284 427 820 1151"> <p>1. <u>in operator</u> : The 'in' operator is used to check if a value exists in a sequence or not. Evaluates to true if it finds a variable in the specified sequence and false otherwise. # Python program to illustrate # Finding common member in list # using 'in' operator list1=[1,2,3,4,5] list2=[6,7,8,9] for item in list1: if item in list2: print("overlapping") else: print("not overlapping") Output: not overlapping</p> </td><td data-bbox="820 427 1508 1151"> <p>1. <u>'not in' operator</u>- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. # Python program to illustrate # not 'in' operator x = 24 y = 20 list = [10, 20, 30, 40, 50];</p> <p>if (x not in list): print "x is NOT present in given list" else: print "x is present in given list"</p> <p>if (y in list): print "y is present in given list" else: print "y is NOT present in given list"</p> </td></tr> </table>	<p>1. <u>in operator</u> : The 'in' operator is used to check if a value exists in a sequence or not. Evaluates to true if it finds a variable in the specified sequence and false otherwise. # Python program to illustrate # Finding common member in list # using 'in' operator list1=[1,2,3,4,5] list2=[6,7,8,9] for item in list1: if item in list2: print("overlapping") else: print("not overlapping") Output: not overlapping</p>	<p>1. <u>'not in' operator</u>- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. # Python program to illustrate # not 'in' operator x = 24 y = 20 list = [10, 20, 30, 40, 50];</p> <p>if (x not in list): print "x is NOT present in given list" else: print "x is present in given list"</p> <p>if (y in list): print "y is present in given list" else: print "y is NOT present in given list"</p>
<p>1. <u>in operator</u> : The 'in' operator is used to check if a value exists in a sequence or not. Evaluates to true if it finds a variable in the specified sequence and false otherwise. # Python program to illustrate # Finding common member in list # using 'in' operator list1=[1,2,3,4,5] list2=[6,7,8,9] for item in list1: if item in list2: print("overlapping") else: print("not overlapping") Output: not overlapping</p>	<p>1. <u>'not in' operator</u>- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. # Python program to illustrate # not 'in' operator x = 24 y = 20 list = [10, 20, 30, 40, 50];</p> <p>if (x not in list): print "x is NOT present in given list" else: print "x is present in given list"</p> <p>if (y in list): print "y is present in given list" else: print "y is NOT present in given list"</p>		
	<p>Identity operators In Python are used to determine whether a value is of a certain class or type. They are usually used to determine the type of data a certain variable contains. There are different identity operators such as</p>		
	<table border="1"> <tr> <td data-bbox="284 1328 820 1861"> <p>1. <u>'is' operator</u> – Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. # Python program to illustrate the use # of 'is' identity operator x = 5 if (type(x) is int): print ("true") else: print ("false") Output: true</p> </td><td data-bbox="820 1328 1508 1861"> <p>1. <u>'is not' operator</u> – Evaluates to false 2. if the variables on either side of the operator 3. point to the same object and true otherwise. # Python program to illustrate the # use of 'is not' identity operator x = 5.2 if (type(x) is not int): print ("true") else: print ("false") Output: true</p> </td></tr> </table>	<p>1. <u>'is' operator</u> – Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. # Python program to illustrate the use # of 'is' identity operator x = 5 if (type(x) is int): print ("true") else: print ("false") Output: true</p>	<p>1. <u>'is not' operator</u> – Evaluates to false 2. if the variables on either side of the operator 3. point to the same object and true otherwise. # Python program to illustrate the # use of 'is not' identity operator x = 5.2 if (type(x) is not int): print ("true") else: print ("false") Output: true</p>
<p>1. <u>'is' operator</u> – Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. # Python program to illustrate the use # of 'is' identity operator x = 5 if (type(x) is int): print ("true") else: print ("false") Output: true</p>	<p>1. <u>'is not' operator</u> – Evaluates to false 2. if the variables on either side of the operator 3. point to the same object and true otherwise. # Python program to illustrate the # use of 'is not' identity operator x = 5.2 if (type(x) is not int): print ("true") else: print ("false") Output: true</p>		
	<p>ii) Write a program to perform addition, subtraction, multiplication, integer division, floor division and modulo division on two integer and float. # Examples of Arithmetic Operator</p>		

```
a = 9
b = 4

# Addition of numbers
add = a + b
# Subtraction of numbers
sub = a - b
# Multiplication of number
mul = a * b
# Division(float) of number
div1 = a / b
# Division(floor) of number
div2 = a // b
# Modulo of both number
mod = a % b

# print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
```

13

5

36

2.25

2

1

Output:

13

5

36

2.25

2

1

1. **Relational Operators:** Relational operators compares the values. It either returns **True** or **False** according t

7.	<p>i) Formulate the difference between type casting and type coercion with suitable example.</p> <p>Casting is when you convert a variable value from one type to another. This is, in Python, done with functions such as <code>int()</code> or <code>float()</code> or <code>str()</code>. A very common pattern is that you convert a number, currently as a string into a proper number.</p> <p>Python code to demonstrate Type conversion</p> <pre># using int(), float() # initializing string s = "10010" # printing string converting to int base 2 c = int(s,2) print ("After converting to integer base 2 : ", end="") print (c) # printing string converting to float e = float(s) print ("After converting to float : ", end="") print (e) Output: After converting to integer base 2 : 18 After converting to float : 10010.0</pre> <p>ii) Write a program to print the digit at ones place and hundreds place of a number.</p> <p>iii) Write a program to convert temperature in degree Fahrenheit to Celsius.</p>
8.	<p>i) Discuss the need and importance of function in Python.</p> <p>Functions are an essential part of the Python programming language. Many important functions are built-in in the Python language. However, as a Data Scientist, developers constantly need to write their own functions to solve problems that their data poses.</p> <p>Functions in Python</p> <p>You use functions in programming to bundle a set of instructions that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub-program and called when needed. That means that a function is a piece of code written to carry out a specified task. To carry out that specific task, the function might or might not need multiple inputs. When the task is carried out, the function can or can not return one or more values.</p> <p>There are three types of functions in Python:</p> <ul style="list-style-type: none"> Built-in functions, such as <code>help()</code> to ask for help, <code>min()</code> to get the minimum value, <code>print()</code> to print an object to the terminal,... You can find an overview with more of these functions here. User-Defined Functions (UDFs), which are functions that users create to help them out; And Anonymous functions, which are also called lambda functions because they are not declared with the standard <code>def</code> keyword. <pre>def my_function(country = "Norway"): print("I am from " + country)</pre>

	<pre> my_function("Sweden") my_function("India") my_function() my_function("Brazil") </pre> <p>ii) Illustrate a program to exchange the value of two variables with temporary variables</p> <p># Program published on https://beginnersbook.com</p> <p># Python program to swap two variables</p> <pre> num1 = input('Enter First Number: ') num2 = input('Enter Second Number: ') print("Value of num1 before swapping: ", num1) print("Value of num2 before swapping: ", num2) # swapping two numbers using temporary variable temp = num1 num1 = num2 num2 = temp print("Value of num1 after swapping: ", num1) print("Value of num2 after swapping: ", num2) </pre> <p>Output:</p> <pre> Enter First Number: 101 Enter Second Number: 99 Value of num1 before swapping: 101 Value of num2 before swapping: 99 Value of num1 after swapping: 99 Value of num2 after swapping: 101 </pre> <p>iii)</p>
9.	<p>Briefly discuss in detail about function prototyping in python with suitable example program</p> <p>What is the purpose of a function prototype?</p> <p>The Function prototype serves the following purposes –</p> <ol style="list-style-type: none"> 1) It tells the return type of the data that the function will return. 2) It tells the number of arguments passed to the function. 3) It tells the data types of the each of the passed arguments. 4) Also it tells the order in which the arguments are passed to the function. <p>Therefore essentially, function prototype specifies the input/output interlace to the function i.e. what to give to the function and what to expect from the function.</p> <p>Prototype of a function is also called signature of the function.</p> <p>.</p>
10.	<p>i) Analyze the difference between local and global variables.</p> <p>Example-1</p> <pre> # This function uses global variable s def f(): print s </pre>

```
# Global scope
s = "I am Global variable"
f()
```

Output:

```
I am Global variable
```

```
a = 1
```

Example-2

```
# Uses global because there is no local 'a'
```

```
def f():
    print 'Inside f() : ', a
```

```
# Variable 'a' is redefined as a local
```

```
def g():
    a = 2
    print 'Inside g() : ', a
```

```
# Uses global keyword to modify global 'a'
```

```
def h():
    global a
    a = 3
    print 'Inside h() : ', a
```

```
# Global scope
```

```
print 'global : ', a
f()
print 'global : ', a
g()
print 'global : ', a
h()
print 'global : ', a
```

Output:

```
global : 1
```

```
Inside f() : 1
```

```
global : 1
```

```
Inside g() : 2
```

```
global : 1
```

```
Inside h() : 3
```

```
global : 3
```

ii) **Explain with an example program to circulate the values of n variables.**

#circulate the values of n variables

```
# Python program to right rotate a list by n
```

```
# Returns the rotated list
def rightRotate(lists, num):
    output_list = []
```

	<pre> # Will add values from n to the new list for item in range(len(lists) - num, len(lists)): output_list.append(lists[item]) # Will add the values before # n to the end of new list for item in range(0, len(lists) - num): output_list.append(lists[item]) return output_list # Driver Code rotate_num = 3 list_1 = [1, 2, 3, 4, 5, 6] print(rightRotate(list_1, rotate_num)) Output : [4, 5, 6, 1, 2, 3] </pre>
11.	<p>i) Describe in detail about lambda functions or anonymous function.</p> <p>What are lambda functions in Python?</p> <p>In Python, anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.</p> <p>Hence, anonymous functions are also called lambda functions.</p> <p>How to use lambda Functions in Python?</p> <p>A lambda function in python has the following syntax.</p> <p>Syntax of Lambda Function in python</p> <pre>lambda arguments: expression</pre> <p>Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.</p> <p>Example of Lambda Function in python</p> <p>Here is an example of lambda function that doubles the input value.</p> <pre> # Program to show the use of lambda functions double = lambda x: x * 2 # Output: 10 print(double(5)) </pre>

	<p>In the above program, <code>lambda x: x * 2</code> is the lambda function. Here <code>x</code> is the argument and <code>x * 2</code> is the expression that gets evaluated and returned.</p> <p>ii) Describe in detail about the rules to be followed while using lambda function. <i>Python lambda (Anonymous Functions)</i></p> <p>In Python, anonymous function means that a function is without a name. As we already know that <code>def</code> keyword is used to define the normal functions and the <i>lambda</i> keyword is used to create anonymous functions. It has the following syntax:</p> <p>lambda arguments: expression</p> <ul style="list-style-type: none"> • This function can have any number of arguments but only one expression, which is evaluated and returned. • One is free to use lambda functions wherever function objects are required. • You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression. • It has various uses in particular fields of programming besides other types of expressions in functions.
12.	<p>i) Explain with an example program to return the average of given number passed as argument to a function.</p> <p># Python program to get average of a list # Using reduce() and lambda</p> <pre># importing reduce() from functools import reduce def Average(lst): return reduce(lambda a, b: a + b, lst) / len(lst) # Driver Code lst = [15, 9, 55, 41, 35, 20, 62, 49] average = Average(lst) # Printing average of the list print("Average of the list =", round(average, 2)) Output: Average of the list = 35.75</pre> <p># Python program to get average of a list # Using mean()</p> <pre># importing mean() from statistics import mean def Average(lst): return mean(lst) # Driver Code lst = [15, 9, 55, 41, 35, 20, 62, 49] average = Average(lst) # Printing average of the list print("Average of the list =", round(average, 2))</pre>

Output:

Average of the list = 35.75

```
def cal_average(num):
    sum_num = 0
    for t in num:
        sum_num = sum_num + t

    avg = sum_num / len(num)
    return avg

print("The average is", cal_average([18,25,3,41,5]))
```

OUTPUT

The average is 18.4

- ii) Explain the various features of functions in Python

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

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

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

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

Syntax

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

13. i) **Describe the syntax and rules involved in the return statement in Python.**

Python return statement

A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned.

Syntax:

```
def fun():
    statements
    .
    .
    return [expression]
```

Example:

Python program to
demonstrate return statement

```
def add(a, b):
    # returning sum of a and b
    return a + b

def is_true(a):
    # returning boolean of a
    return bool(a)

# calling function
res = add(2, 3)
print("Result of add function is {}".format(res))

res = is_true(2<5)
print("\nResult of is_true function is {}".format(res))
```

Output:

Result of add function is 5
Result of is_true function is True

- ii) Write a program to demonstrate the flow of control after the return statement in Python.

A function that returns a list of the numbers of the Fibonacci series:

```
>>>
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
```

```

... a, b = 0, 1
... while a < n:
...     result.append(a) # see below
...     a, b = b, a+b
... return result
...
>>> f100 = fib2(100) # call it
>>> f100             # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

```

14. i) **Explain the operator precedence of arithmetic operators in Python.**

Precedence of Python Operators

The combination of values, [variables](#), [operators](#) and [function](#) calls is termed as an expression. Python interpreter can evaluate a valid expression.

For example:

```

1.
2. >>> 5 - 7
3. -2

```

Here 5 - 7 is an expression.

There can be more than one operator in an expression. To evaluate these type of expressions there is a rule of precedence in Python. It guides the order in which operation are carried out.

For example, multiplication has higher precedence than subtraction.

```

# Multiplication has higher precedence
# than subtraction
# Output: 2
10 - 4 * 2

# Parentheses () has higher precedence
# Output: 12
(10 - 4) * 2

```

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

Operator precedence rule in Python	
Operators	Meaning
()	Parentheses
**	Exponent

+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR
Operator precedence rule in Python	
Associativity of Python Operators Associativity is the order in which an expression is evaluated that has multiple operator of the same precedence. Almost all the operators have left-to-right associativity. For example, multiplication and floor division have the same precedence. Hence, if <pre># Left-right associativity # Output: 3 print(5 * 2 // 3) # Shows left-right associativity # Output: 0 print(5 * (2 // 3))</pre> Run Powered by DataCamp Exponent operator ** has right-to-left associativity in Python. <pre># Right-left associativity of ** exponent operator # Output: 512 print(2 ** 3 ** 2) # Shows the right-left associativity # of ** # Output: 64 print((2 ** 3) ** 2)</pre>	

ii) Write a Python program to exchange the value of two variables

```

1. # Python program to swap two variables
2.
3. x = 5
4. y = 10
5.
6. # To take inputs from the user
7. #x = input('Enter value of x: ')
8. #y = input('Enter value of y: ')
9.
10. # create a temporary variable and swap the values
11. temp = x
12. x = y
13. y = temp
14.
15. print('The value of x after swapping: {}'.format(x))
16. print('The value of y after swapping: {}'.format(y))

```

iii) Write a Python program using function to find the sum of first 'n' even numbers and print the result.

sum of Even numbers in python

Python program to get input n and calculate the sum of even numbers till n

Solution

```

n=int(input("Enter n value:"))
sum=0
for i in range(2,n+1,2):
    sum+=i
print(sum)

```

GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING	
UNIT 3 - CONTROL FLOW, FUNCTIONS	
SYLLABUS	
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.	
PART-A	
Q. No.	Q&A
1.	<p>Analyze different ways to manipulate strings in Python.</p> <p>Slicing</p> <p>In Python slice operator is used to slice a part of a string. The syntax uses start and end index with a ":" in between as shown in the following example:</p> <pre>>>> str = "Python is great"</pre> <pre>>>> first_six = str[0:6] >>> first_six</pre> <p>OUTPUT : Python</p>
2.	<p>Write the syntax of if and if-else statements.</p> <p>Python if Statement Syntax</p> <pre>if test expression: statement(s)</pre> <p>Python if Statement Flowchart</p> <pre>graph TD Start(()) --> Test{Test Expression} Test -- True --> Body[Body of if] Test -- False --> Join(()) Body --> Join Join --> End(())</pre> <p>Fig: Operation of if statement</p> <p># If the number is positive, we print an appropriate message</p> <pre>num = 3 if num > 0: print(num, "is a positive number.") print("This is always printed.")</pre>

OUTPUT

3 is a positive number.

This is always printed.

Python if...else Statement

Syntax of if...else

if test expression:

Body of if

else:

Body of else

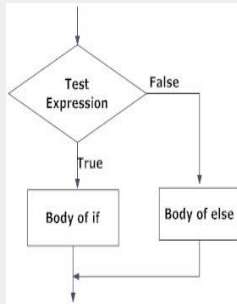


Fig: Operation of if...else statement

```
num = 3
```

```
if num >= 0:
```

```
    print("Positive or Zero")
```

```
else:
```

```
    print("Negative number")
```

OUTPUT

'Positive or zero'

3. List out the applications of arrays.

Arrays

Arrays are used to store multiple values in one single variable:

```
cars = ["Ford", "Volvo", "BMW"]
```

```
print(cars)
```

OUTPUT

```
['Ford', 'Volvo', 'BMW']
```

4. Discuss about continue and pass statements.

```
for i in 'hello':
```

```
    if(i == 'e'):
```

```
        print('pass executed')
```

```
    pass
```

```
    print(i)
```

```
print('----')
```

```
for i in 'hello':
```

```
    if(i == 'e'):
```

```
        print('continue executed')
```

```
    continue
```

```
    print(i)
```


	<p>Output :-</p> <pre> h pass executed e l l o ---- h continue executed l l o </pre>
5.	<p>What will be the output of <code>print(str[2:5])</code> if <code>str='hello world!'</code>?</p> <pre> str='hello world!' print(str[2:5]) output llo </pre>
6.	<p>Give the use of <code>return()</code> statement with a suitable example.</p> <p>A function in Python is defined by a <code>def</code> statement. The general syntax looks like this:</p> <pre> def function-name(Parameter list): statements, i.e. the function body </pre> <p>The parameter list consists of none or more parameters. Parameters are called arguments, if the function is called. The function body consists of indented statements. The function body gets executed every time the function is called. Parameter can be mandatory or optional.</p> <p>Function bodies can contain one or more return statement. They can be situated anywhere in the function body. A return statement ends the execution of the function call and "returns" the result, i.e. the value of the expression following the return keyword, to the caller. Example:</p> <pre> def fahrenheit(T_in_celsius): """ returns the temperature in degrees Fahrenheit """ return (T_in_celsius * 9 / 5) + 32 for t in (22.6, 25.8, 27.3, 29.8): print(t, ":", fahrenheit(t)) </pre> <p>The output of this script looks like this:</p> <pre> 22.6 : 72.68 25.8 : 78.44 27.3 : 81.14 29.8 : 85.64 </pre>
7.	<p>Write a program to iterate a range using <code>continue</code> statement.</p>

The *continue* Statement:

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

The **continue** statement can be used in both *while* and *for* loops.

In the following script , when we have encountered a spam item, continue prevents us from eating spam!

```
edibles = ["ham", "spam", "eggs", "nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        continue
    print("Great, delicious " + food)
    # here can be the code for enjoying our food :-)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

OUTPUT:

```
$ python for.py
Great, delicious ham
No more spam please!
Great, delicious eggs
Great, delicious nuts
I am so glad: No spam!
Finally, I finished stuffing myself
```

8. Name the type of Boolean operators.

Logical operators

Logical operators are the and, or, not operators.

The *logical operators* **and**, **or** and **not** are also referred to as *boolean operators*.

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

	Boolean and operator returns true if both operands return true.	Boolean or operator returns true if any one operand is true	The not operator returns true if its operand is a false expression and returns false if it is true.
	<pre>>>> a=50 >>> b=25 >>> a>40 and b>40 False >>> a>100 and b<50 False >>> a==0 and b==0 False >>> a>0 and b>0 True</pre>	<pre>>>> a=50 >>> b=25 >>> a>40 or b>40 True >>> a>100 or b<50 True >>> a==0 or b==0 False >>> a>0 or b>0 True</pre>	<pre>>>> a=10 >>> a>10 False >>> not(a>10) True</pre>

9. Explain about break statement with an example.

The *break* Statement:

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

Example:

```
for letter in 'Python': # First Example
    if letter == 't':
        break
    print 'Current Letter :', letter
```

```
var = 10 # Second Example
while var > 0:
    print 'Current variable value :', var
    var = var -1
    if var == 8:
        break
```

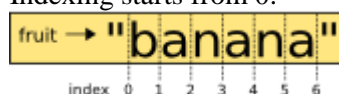
```
print "Good bye!"
```

OUTPUT:

```
Current Letter : P
Current Letter : y
Current variable value : 10
Current variable value : 9
Good bye!
```

10. Where does indexing starts in Python?

Indexing starts from 0.



```
fruit → "banana"
index  0  1  2  3  4  5  6
```

```
>>> fruit = "banana"
```

```
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

11. Illustrate the flowchart of if-elif-else statements.

Python if...elif...else Statement

Syntax of if...elif...else

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

The elif is short for else if. It allows us to check for multiple expressions.

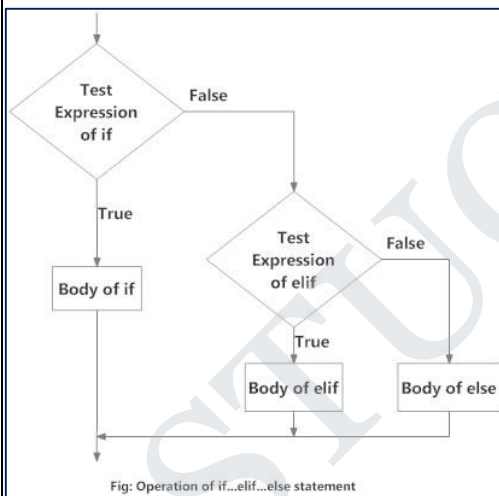
If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.

Flowchart of if...elif...else



Example of if...elif...else

In this program, we check if the number is positive or negative or zero
and display an appropriate message

```
num = 3.4
```

```
if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

When variable num is positive, Positive number is printed.
 If num is equal to 0, Zero is printed.
 If num is negative, Negative number is printed

12. Describe any 4 methods used on a string.

String Methods

Python provides lots of built-in methods which we can use on strings.
 Here are the list of string methods available in Python 3.

Method	Description	Examples	
Count(sub, [start], [end])	Returns the number of non-overlapping occurrences of substring (sub) in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.	<pre>>>> mystr = "Hello Python" >>> print(mystr.count("o")) 2 >>> print(mystr.count("th")) 1 >>> print(mystr.count("l")) 2 >>> print(mystr.count("h")) 1 >>> print(mystr.count("H")) 1 >>> print(mystr.count("hH")) 0</pre>	
Index(sub, [start], [end])	Searches the string for a specified value and returns the position of where it was found	<pre>>>> mystr = "HelloPython" >>> print(mystr.index("P")) 5 >>> print(mystr.index("hon")) 8 >>> print(mystr.index("o")) 4</pre>	
replace(old, new[,count])	Returns a string where a specified value is replaced with a specified value	<pre>>>> mystr = "Hello Python. Hello Java. Hello C++." >>> print(mystr.replace("Hello", "Bye")) Bye Python. Bye Java. Bye C++. >>> print(mystr.replace("Hello", "Hell", 2)) Hell Python. Hell Java. Hello C++.</pre>	
split(sep=None, maxsplit=-1)	Splits the string at the specified separator, and returns a list	<pre>>>> mystr = "Hello Python" >>> print(mystr.split()) ['Hello', 'Python'] >>> mystr1 = "Hello,,Python"</pre>	

			<pre>>>> print(mystr1.split(", ")) ['Hello', ' ', 'Python']</pre>	
	strip([chars])	Returns a trimmed version of the string	<pre>>>> mystr = " Hello Python " >>> print(mystr.strip(), "!") Hello Python ! >>> print(mystr.strip(), " ") Hello Python</pre>	
	upper()	Converts a string into upper case	<pre>>>> mystr = "hello Python" >>> print(mystr.upper()) HELLO PYTHON</pre>	

13. What are the advantages and disadvantages of recursion function?

Advantages of Recursion

Recursive functions make the code look clean and elegant.

A complex task can be broken down into simpler sub-problems using recursion.

Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

Sometimes the logic behind recursion is hard to follow through.

Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

Recursive functions are hard to debug.

14. Explain the significance of for loop with else in an example.

For Loops

For loop is a programming language statement, i.e. an iteration statement, which allows a code block to be repeated a certain number of times.

Syntax of the For Loop

The Python for loop is an iterator based for loop. It steps through the items of lists, tuples, strings, the keys of dictionaries and other iterables. The Python for loop starts with the keyword "for" followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through. The general syntax looks like this:

```
for <variable> in <sequence>:
```

```
    <statements>
```

```
else:
```

```
    <statements>
```

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

Example of a simple for loop in Python:

```

>>> languages = ["C", "C++", "Perl", "Python"]
>>> for x in languages:
...     print(x)
...
C
C++
Perl
Python
>>>

```

15. Define array with an example.

Arrays

Arrays are used to store multiple values in one single variable:

Example

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
cars = ["Ford", "Volvo", "BMW"]
```

```
x = cars[0]
```

```
print(x)
```

OUTPUT

```
Ford
```

16. Differentiate for loop and while loop.

The **for loop** is a programming language statement, i.e. an iteration statement, which allows a code block to be repeated a certain number of times.

```
for <variable> in <sequence>:
```

```
    <statements>
```

```
else:
```

	<div data-bbox="159 197 1420 235" data-label="Text"> <p><statements></p> </div> <div data-bbox="159 264 628 293" data-label="Text"> <p>Example of a simple for loop in Python:</p> </div> <div data-bbox="159 315 1420 797" data-label="Code-Block"> <pre>>>> languages = ["C", "C++", "Perl", "Python"] >>> for x in languages: ... print(x) ... C C++ Perl Python >>></pre> </div> <div data-bbox="159 819 1420 887" data-label="Text"> <p>A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.</p> </div> <div data-bbox="159 925 244 958" data-label="Section-Header"> <h3>Syntax</h3> </div> <div data-bbox="159 996 933 1028" data-label="Text"> <p>The syntax of a while loop in Python programming language is –</p> </div> <div data-bbox="159 1048 362 1079" data-label="Text"> <p>while expression:</p> </div> <div data-bbox="159 1081 323 1113" data-label="Text"> <p> statement(s)</p> </div> <div data-bbox="159 1115 248 1146" data-label="Text"> <p>n = 100</p> </div> <div data-bbox="159 1182 217 1214" data-label="Text"> <p>s = 0</p> </div> <div data-bbox="159 1216 288 1247" data-label="Text"> <p>counter = 1</p> </div> <div data-bbox="159 1249 387 1281" data-label="Text"> <p>while counter <= n:</p> </div> <div data-bbox="159 1283 362 1314" data-label="Text"> <p> s = s + counter</p> </div> <div data-bbox="159 1317 335 1348" data-label="Text"> <p> counter += 1</p> </div> <div data-bbox="159 1384 619 1415" data-label="Text"> <p>print("Sum of 1 until %d: %d" % (n,s))</p> </div> <div data-bbox="159 1451 280 1482" data-label="Section-Header"> <h3>OUTPUT</h3> </div> <div data-bbox="159 1485 426 1514" data-label="Text"> <p>Sum of 1 until 100: 5050</p> </div>
17.	<div data-bbox="159 1554 681 1583" data-label="Text"> <p>Classify global variable with local variable.</p> </div> <div data-bbox="159 1615 368 1644" data-label="Section-Header"> <h3>Local Variables</h3> </div> <div data-bbox="159 1646 1420 1776" data-label="Text"> <p>When you define variables inside a function definition, they are local to this function by default. This means that anything you 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.</p> </div> <div data-bbox="159 1812 798 1848" data-label="Section-Header"> <h3>Global and local Variables in Functions</h3> </div> <div data-bbox="159 1886 1367 1948" data-label="Text"> <p>The following example, demonstrates, how global values can be used inside the body of a function:</p> </div> <div data-bbox="159 1971 1420 2024" data-label="Code-Block"> <pre>def f():</pre> </div>

	<pre>print(s) #s is global variable s = "I love Paris in the summer!" f()</pre> <p>Local Variable</p> <pre>def f(): #Here is id local variable s = "I love London!" print(s) s = "I love Paris!" f() print(s)</pre> <p>The output looks like this:</p> <pre>I love London! I love Paris!</pre>
18.	<p>Write a Python program to accept two numbers, multiply them and print the result.</p> <pre>a = int(input("enter first number: ")) b = int(input("enter second number: ")) result = a * b. print("result :", result)</pre> <p>OUTPUT</p> <pre>enter first number: 4 enter second number: 5 result : 20</pre>
19.	<p>Justify the effects of slicing operation on an array.</p> <p>How to slice arrays?</p> <p>We can access a range of items in an array by using the slicing operator :.</p> <ol style="list-style-type: none"> 1. import array as arr 2. 3. numbers_list = [2, 5, 62, 5, 42, 52, 48, 5] 4. numbers_array = arr.array('i', numbers_list) 5. 6. print(numbers_array[2:5]) # 3rd to 5th 7. print(numbers_array[:5]) # beginning to 4th 8. print(numbers_array[5:]) # 6th to end 9. print(numbers_array[:]) # beginning to end

	<p>When you run the program, the output will be:</p> <pre>array('i', [62, 5, 42]) array('i', [2, 5, 62]) array('i', [52, 48, 5]) array('i', [2, 5, 62, 5, 42, 52, 48, 5])</pre>
20.	<p>How to access the elements of an array using index.</p> <p>How to access array elements?</p> <p>We use indices to access elements of an array:</p> <pre>1. import array as arr 2. a = arr.array('i', [2, 4, 6, 8]) 3. 4. print("First element:", a[0]) 5. print("Second element:", a[1]) 5. print("Last element:", a[-1])</pre> <p>OUTPUT First element: 2 Second element: 4 Last element: 8</p>
PART-B	
1.	<p>i. Write a Python program to find the sum of N natural numbers.</p> <pre># Program to add natural numbers upto n # sum = 1+2+3+...+n # To take input from the user, # n = int(input("Enter n: ")) n = 100 # initialize sum and counter sum = 0 i = 1 while i <= n: sum = sum + i i = i+1 # update counter # print the sum print("The sum is", sum) OUTPUT The sum is 5050</pre> <p>ii. What is the use of pass statement? Illustrate with an example.</p> <p>The <i>pass</i> Statement:</p> <p>The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.</p>

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

Example:

```
#!/usr/bin/python

for letter in 'Python':
    if letter == 'h':
        pass
    print 'This is pass block'
    print 'Current Letter :', letter

print "Good bye!"
```

OUTPUT

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

2. Define methods in a string with an example program using at least 5 methods.

Python String Methods

Python provides lots of built-in methods which we can use on strings.

Method	Description	Examples	
capitalize()	Returns a copy of the string with its first character capitalized and the rest lowercased.	>>> print(capitalize("Hello Python")) Hello python	
Casefold()	Returns a casefolded copy of the string. Casefolded strings may be used for caseless matching.	>>> mystring = "hello PYTHON" >>> print(mystring.casefold()) hello python	
Center(width, [fillchar])	Returns the string centered in a string of length width. Padding can be done using the specified fillchar (the default padding uses an ASCII space). The original string is returned if width is less than or equal to len(s)	>>> mystring = "Hello" >>> x = mystring.center(12, "-") >>> print(x) ---Hello---	
Count(sub, [start], [end])	Returns the number of non-overlapping occurrences of substring (sub) in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.	>>> mystr = "Hello Python" >>> print(mystr.count("o")) 2 >>> print(mystr.count("th")) 1 >>> print(mystr.count("l")) 2 >>> print(mystr.count("h"))	

			<pre> 1 >>> print(mystr.count("H")) 1 >>> print(mystr.count("hH")) 0 </pre>	
	endswith(suffix, [start], [end])	Returns True if the string ends with the specified suffix, otherwise it returns False.	<pre> >>> mystr = "Python" >>> print(mystr.endswith("y")) False >>> print(mystr.endswith("hon")) True </pre>	
<p>How to access characters of a string?</p> <p>Individual characters in a string can be accessed by specifying the string name followed by a number in square brackets (<code>[]</code>). String indexing in Python is zero-based: the first character in the string has index 0 , the next has index 1 , and so on.</p>				
3.	<p>Write a program for binary search using Arrays.</p> <p>Python Program for Binary Search (Recursive and Iterative)</p> <p>We basically ignore half of the elements just after one comparison.</p> <ol style="list-style-type: none"> 1. Compare x with the middle element. 2. If x matches with middle element, we return the mid index. 3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half. 4. Else (x is smaller) recur for the left half. <p>Iterative:</p> <pre> # Iterative Binary Search Function # It returns location of x in given array arr if present, # else returns -1 def binarySearch(arr, l, r, x): while l <= r: mid = l + (r - l) / 2; # Check if x is present at mid if arr[mid] == x: return mid # If x is greater, ignore left half elif arr[mid] < x: l = mid + 1 # If x is smaller, ignore right half else: r = mid - 1 # If we reach here, then the element was not present return -1 # Test array arr = [2, 3, 4, 10, 40] </pre>			

	<pre>x = 10 # Function call result = binarySearch(arr, 0, len(arr)-1, x) if result != -1: print "Element is present at index %d" % result else: print "Element is not present in array"</pre> <p>Output: Element is present at index 3</p>
4.	<p>What is call by value and call by reference and explain it with suitable example</p> <p>call-by-value :</p> <pre>>def plus_1(x) : > x=x+1 > >x=5 >plus_1(x) >print x 5</pre> <p>Here, x was passed by value - local changes within the function didn't echo back to the calling scope.</p> <p>However, if we use a list, elements are <i>passed by reference</i>. So that here,</p> <pre>>def plus_1(x) : > x[0]=x[0]+1 > >x=[5] >plus_1(x) >print x[0] 6</pre>
5.	<p>i) Write a python program to find the given number is odd or even</p> <pre>1. # Python program to check if the input number is odd or even. 2. # A number is even if division by 2 gives a remainder of 0. 3. # If the remainder is 1, it is an odd number. 4. 5. num = int(input("Enter a number: ")) 6. if (num % 2) == 0: 7. print("{0} is Even".format(num)) 8. else: 9. print("{0} is Odd".format(num))</pre> <p>Enter a number: 77 77 is Odd</p>
6.	<p>Write a Python program to count the number of vowels in a string provided by the user.</p> <p>Counting vowels: String Way</p> <p>In this method, we will store all the vowels in a string and then pick every character from the enquired string and check whether it is in the vowel string or not. The vowel string consists of all the vowels with both cases since we are not ignoring the cases here. If the vowel is encountered then count gets</p>

incremented and stored in a list and finally printed.

Python code to count and display number of vowels

```
# Simply using for and comparing it with a
# string containg all vowels
def Check_Vow(string, vowels):
    final = [each for each in string if each in vowels]
    print(len(final))
    print(final)
```

Driver Code

```
string = "I wandered lonely as a cloud"
```

```
vowels = "AaeEeliOoUu"
```

```
Check_Vow(string, vowels);
```

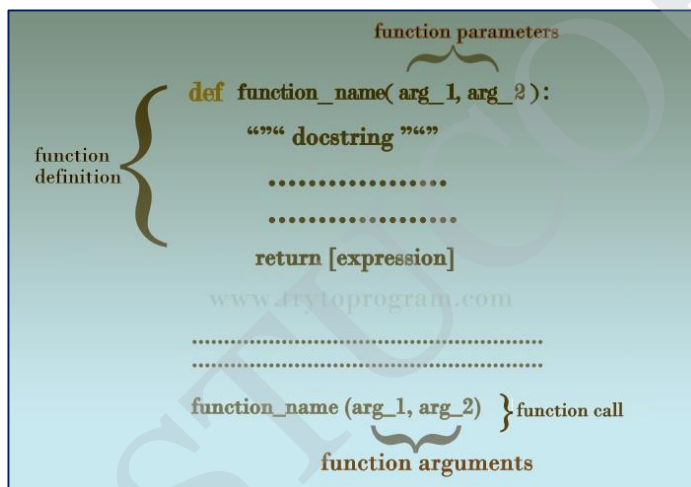
Output:

```
10
```

```
['I', 'a', 'e', 'e', 'o', 'e', 'a', 'a', 'o', 'u']
```

Explain the types of function arguments in Python

Function Arguments



There are three types of Python function arguments using which we can call a function.

Default **Arguments**.

Keyword **Arguments**.

Variable-length **Arguments**.

Function call with variable arguments

```
def display(*name, **address):
```

```
    for items in name:
```

```
        print (items)
```

```
    for items in address.items():
```

```
        print (items)
```

#Calling the function

```
display('john','Mary','Nina',John='LA',Mary='NY',Nina='DC')
```

```
john
```

```
Mary
```

```
Nina
```

```
('John', 'LA')
```

```
('Mary', 'NY')
```

```
('Nina', 'DC')
```

#Function with Keyword arguments

```
def print_name(name1, name2):
```

```
    """ This function prints the name """
```

```
    print (name2 + " and " + name1 + " are friends")
```

#calling the function

```
print_name(name2 = 'John',name1 = 'Gary')
```

```
John and Gary are friends
```

```
def sum(a=4, b=2): #2 is supplied as default argument
```

```
    """ This function will print sum of two numbers
```

```
    if the arguments are not supplied
```

```
    it will add the default value """
```

```
    print (a+b)
```

```
sum(1,2) #calling with arguments
```

```
sum( ) #calling without arguments
```

```
3
```

```
6
```

7. Explain the syntax and flowchart of the following loop statements

- i) for loop

For Loops**Introduction**

For loop is a programming language statement, i.e. an iteration statement, which allows a code block to be repeated a certain number of times. The Python for loop is an iterator based for loop. It steps through the items of lists, tuples, strings, the keys of dictionaries and other iterables. The Python for loop starts with the keyword "for" followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through. The general syntax looks like this:

```
for <variable> in <sequence>:
```

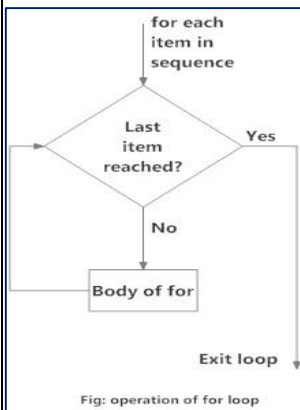
```
    <statements>
```

```
else:
```

```
    <statements>
```

The items of the sequence object are assigned one after the other to the loop variable; to be precise the

variable points to the items. For each item the loop body is executed.



Example of a simple for loop in Python:

```

>>> languages = ["C", "C++", "Perl", "Python"]
>>> for x in languages:
...     print(x)
...
C
C++
Perl
Python
>>>
  
```

ii) while loop

What is while loop in Python?

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax of while Loop in Python

```
while test_expression:
```

```
    Body of while
```

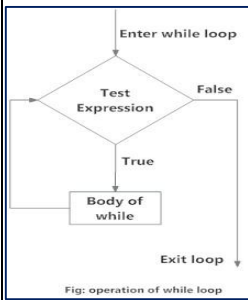
In while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

In Python, the body of the while loop is determined through indentation.

Body starts with indentation and the first unindented line marks the end.

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

Flowchart of while Loop



Example: Python while Loop

Program to add natural numbers upto n

sum = 1+2+3+...+n

To take input from the user,

n = int(input("Enter n: "))

n = 10

initialize sum and counter

sum = 0

i = 1

while i <= n:

 sum = sum + i

 i = i+1 # update counter

print the sum

print("The sum is", sum)

OUTPUT

Enter n: 10

The sum is 55

8. Illustrate with an example nested if and elif header in Python.

Python Nested if statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

Python Nested if Example

```

1. # In this program, we input a number
2. # check if the number is positive or
3. # negative or zero and display
4. # an appropriate message
5.
6. num = float(input("Enter a number: "))
7. if num >= 0:
8.     if num == 0:
9.         print("Zero")
10.    else:
11.        print("Positive number")
12. else:
13.    print("Negative number")
  
```

Output 1

	<p>Enter a number: 5 Positive number</p> <p>Output 2</p> <p>Enter a number: -1 Negative number</p> <p>Output 3</p> <p>Enter a number: 0 Zero</p> <p>Develop a program to find the largest among three numbers.</p>
9.	<p>Explain recursive function. How do recursive function works?</p> <p>Recursive Functions in Python</p> <p>A recursive function is a function defined in terms of itself via self-referential expressions. This means that the function will continue to call itself and repeat its behavior until some condition is met to return a result. The Fibonacci numbers are the numbers of the following sequence of integer values:</p> <p>0,1,1,2,3,5,8,13,21,34,55,89, ...</p> <p>The Fibonacci numbers are defined by:</p> $F_n = F_{n-1} + F_{n-2}$ <p>with $F_0 = 0$ and $F_1 = 1$</p> <pre>def fib(n): if n == 0: return 0 elif n == 1: return 1 else: return fib(n-1) + fib(n-2)</pre>
10.	<p>Create a Python program to find the given year is leap year or not.</p> <p><i># Python program to check if year is a leap year or not.</i></p> <p><i># To get year (integer input) from the user.</i></p> <pre>year = int(input("Enter a year: ")) if (year % 4) == 0: if (year % 100) == 0: if (year % 400) == 0: print("{0} is a leap year".format(year)) else: print("{0} is NOT a leap year".format(year)) else: print("{0} is NOT a leap year".format(year)) else: print("{0} is NOT a leap year".format(year))</pre>

OUTPUT

Enter a year: 2019
 2019 is NOT a leap year
 Enter a year: 2000
 2000 is a leap year

Investigate on mutability and immutability in Python.

Python: Mutable vs. Immutable

Everything in Python is an **object**. You have to understand that **Python** represents all its data as objects. An object's mutability is determined by its type. Some of these objects like lists and dictionaries are **mutable**, meaning you can change their content without changing their identity. Other objects like integers, floats, strings and tuples are objects that can not be changed.

Strings are Immutable	List is mutable	Tuple is immutable
Strings are immutable in Python, which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original.	Having mutable variables means that calling the same method with the same variables may not guarantee the same output, because the variable can be mutated at any time by another method or perhaps, another thread, and that is where you start to go crazy debugging.	
Example <pre>message = "strings immutable" message[0] = 'p' print(message)</pre> output <p>Instead of producing the output "strings immutable", this code produces the runtime error:</p> <p>TypeError: 'str' object does not support item assignment</p> Why are Python strings immutable? <p>Which means a string value cannot be updated. Immutability is a clean and efficient solution to concurrent access. Having immutable variables means that no matter how many times the method is called with the same</p>	Mutable example <pre>my_list = [10, 20, 30] print(my_list)</pre> Output <p>[10, 20, 30]</p> <p>continue...</p> <pre>my_list = [10, 20, 30] my_list[0] = 40 print(my_list)</pre> Output <p>[40, 20, 30]</p>	Immutable example <pre>my_yuple = (10, 20, 30) print(my_yuple)</pre> Output <p>(10, 20, 30)</p> <pre>my_yuple = (10, 20, 30) my_yuple[0] = 40 print(my_yuple)</pre> output <p>Traceback (most recent call last): File "test.py", line 3, in <module> my_yuple[0] = 40 TypeError: 'tuple' object does not support item assignment</p>

	variable/value, the output will always be the same.		
11.	<p>Explain the different types of the function prototype with an example.</p> <p>Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function.</p> <p>Examine a Python program to generate first 'N' Fibonacci numbers.</p> <pre>def fib(n): # return Fibonacci series up to n result = [] a, b = 0, 1 while b < n: result.append(b) a, b = b, a + b return result print(fib(100)) #[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]</pre> <p>Note: The Fibonacci numbers are 0,1,1,2,3,5,8,..... where each number is the sum of preceding two.</p>		
12.	<p>Generate a program that uses lambda function to multiply two numbers.</p> <pre>multi = lambda x, y : x * y print(multi(5, 20)) print("\nResult from a multi Function") def multi_func(x, y): return x * y print(multi_func(5, 20)) OUTPUT 100 Result from a multiply Function 100</pre> <p>Discuss the methods to manipulate the arrays in Python.</p> <p>Python Arrays</p> <p>In programming, an array is a collection of elements of the same type. Arrays are popular in most programming languages like Java, C/C++, JavaScript and so on. However, in Python, they are not that common.</p> <p>Python Lists Vs array Module as Arrays</p>		

We can treat lists as arrays. However, we cannot constrain the type of elements stored in a list.

How to create arrays?

We need to import array module to create arrays. For example:

```
1. import array as arr
2. a = arr.array('d', [1.1, 3.5, 4.5])
3. print(a)
```

Here, we created an array of float type. The letter 'd' is a type code. This determines the type of the array during creation.

How to access array elements?

We use indices to access elements of an array:

```
1. import array as arr
2. a = arr.array('i', [2, 4, 6, 8])
3.
4. print("First element:", a[0])
5. print("Second element:", a[1])
6. print("Last element:", a[-1])
```

Remember, the index starts from 0 (not 1) similar to lists.

How to slice arrays?

We can access a range of items in an array by using the slicing operator :.

```
1. import array as arr
2.
3. numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
4. numbers_array = arr.array('i', numbers_list)
5.
6. print(numbers_array[2:5]) # 3rd to 5th
7. print(numbers_array[:5]) # beginning to 4th
8. print(numbers_array[5:]) # 6th to end
9. print(numbers_array[:]) # beginning to end
```

When you run the program, the output will be:

```
array('i', [62, 5, 42])
array('i', [2, 5, 62])
array('i', [52, 48, 5])
array('i', [2, 5, 62, 5, 42, 52, 48, 5])
```

How to change or add elements?

Arrays are mutable; their elements can be changed in a similar way like lists.

```
1. import array as arr
2.
3. numbers = arr.array('i', [1, 2, 3, 5, 7, 10])
4.
5. # changing first element
6. numbers[0] = 0
7. print(numbers) # Output: array('i', [0, 2, 3, 5, 7, 10])
```

We can concatenate two arrays using + operator.

```
1. import array as arr
2.
3. odd = arr.array('i', [1, 3, 5])
4. even = arr.array('i', [2, 4, 6])
5.
6. numbers = arr.array('i') # creating empty array of integer
7. numbers = odd + even
8.
9. print(numbers)
```

How to remove/delete elements?

We can delete one or more items from an array using [Python's del statement](#).

```
1. import array as arr
2.
3. number = arr.array('i', [1, 2, 3, 3, 4])
4.
5. del number[2] # removing third element
6. print(number) # Output: array('i', [1, 2, 3, 4])
```

13. Explain the significance of xrange() function in for loop with a help of a program.

```
# Python code to demonstrate range() vs xrange() on basis of memory

import sys
from past.builtins import xrange

# initializing a with range()
a = range(1,10000)

# initializing a with xrange()
x = xrange(1,10000)

# testing the size of a
# range() takes more memory

print ("The size allotted using range() is : ")
print (sys.getsizeof(a))

# testing the size of a
# range() takes less memory

print ("The size allotted using xrange() is : ")
```

	<pre>print (sys.getsizeof(x))</pre> <p>Output:</p> <p>The size allotted using range() is :</p> <p>80064</p> <p>The size allotted using xrange() is :</p> <p>40</p>
14.	<p>Create a program to find the factorial of given number without recursion and with recursion.</p> <p># Factorial of a number using recursion</p> <pre>def recur_factorial(n): if n == 1: return n else: return n*recur_factorial(n-1) num = 6 # check if the number is negative if num < 0: print("Sorry, factorial does not exist for negative numbers") elif num == 0: print("The factorial of 0 is 1") else: print("The factorial of", num, "is", recur_factorial(num))</pre> <p>Illustrate the concept of local and global variables.</p> <pre>n=int(input("Enter number:")) fact=1 while(n>0): fact=fact*n n=n-1 print("Factorial of the number is: ") print(fact) Enter number: 6</pre> <p>OUTPUT</p> <p>Factorial of the number is:</p> <p>720</p>

GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING																			
UNIT 4 - LISTS, TUPLES, DICTIONARIES																			
SYLLABUS																			
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.																			
Part A																			
Q. No.	Q&A																		
1.	<p>Define Python list. How lists differ from Tuples.</p> <p>List A list is a collection which is ordered and changeable. In Python lists are written with square brackets. The main difference between lists and a tuples is the fact that lists are mutable whereas tuples are immutable.</p>																		
2.	<p>What are the list operations?</p> <p>Basic List Operations</p> <p>Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.</p> <p>In fact, lists respond to all of the general sequence operations we used on strings.</p> <table><tr><th>Python Expression</th><th>Results</th><th>Description</th></tr><tr><td>len([1, 2, 3])</td><td>3</td><td>Length</td></tr><tr><td>[1, 2, 3] + [4, 5, 6]</td><td>[1, 2, 3, 4, 5, 6]</td><td>Concatenation</td></tr><tr><td>['Hi!'] * 4</td><td>['Hi!', 'Hi!', 'Hi!', 'Hi!']</td><td>Repetition</td></tr><tr><td>3 in [1, 2, 3]</td><td>True</td><td>Membership</td></tr><tr><td>for x in [1, 2, 3]: print x,</td><td>1 2 3</td><td>Iteration</td></tr></table>	Python Expression	Results	Description	len([1, 2, 3])	3	Length	[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation	['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition	3 in [1, 2, 3]	True	Membership	for x in [1, 2, 3]: print x,	1 2 3	Iteration
Python Expression	Results	Description																	
len([1, 2, 3])	3	Length																	
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation																	
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition																	
3 in [1, 2, 3]	True	Membership																	
for x in [1, 2, 3]: print x,	1 2 3	Iteration																	
3.	<p>What are the different ways to create a list?</p> <p>Create a List:</p> <pre>thislist = ["apple", "banana", "cherry"] print(thislist)</pre>																		

	OUTPUT ['apple', 'banana', 'cherry']
4.	<p>Illustrate negative indexing in list with an example.</p> <p>Negative Indexing</p> <p>Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.</p> <p>Example</p> <p>Print the last item of the list:</p> <pre>thislist = ["apple", "banana", "cherry"] print(thislist[-1])</pre> <p>OUTPUT cherry</p>
5.	<p>How to slice a list in Python?</p> <p>Slicing Python Lists</p> <p>Python has an amazing feature known as <i>slicing</i>. Slicing can not only be used for lists, tuples or arrays, but custom data structures as well, with the <i>slice</i> object.</p> <p>Slicing Python Lists/Arrays and Tuples Syntax</p> <p>Let's consider the list a shown below :</p> <pre>1 >>> a = [1, 2, 3, 4, 5, 6, 7, 8]</pre> <p>Slicing operation is done on a normal list a = [1, 2, 3, 4, 5, 6, 7, 8] and sub-elements 2, 3, and 4 returned in a new list as a result.</p> <p>The following example illustrates the slicing operation on lists:</p> <pre>1 >>> a[1:4] 2 [2, 3, 4]</pre>
6.	<p>List out the methods that are available with list object in Python programming.</p> <p>Python List Methods</p> <p>Python has some list methods that you can use to perform frequently occurring task (related to list) with ease. For example, if you want to add element to a list, you can use <code>append()</code> method.</p> <p>The table below contains all methods of list objects. Also, the table includes built-in functions that can take list as a parameter and perform some task. For example, <code>all()</code> function returns True if all elements of an list (iterable) is true. If not, it returns False.</p>

Method	Description
Python List append()	Add a single element to the end of the list
Python List extend()	Add Elements of a List to Another List
Python List insert()	Inserts Element to The List
Python List remove()	Removes item from the list
Python List index()	returns smallest index of element in list
Python List count()	returns occurrences of element in a list
Python List pop()	Removes element at the given index
Python List reverse()	Reverses a List
Python List sort()	sorts elements of a list
Python List copy()	Returns Shallow Copy of a List
Python List clear()	Removes all Items from the List

7. Show the membership operators used in list.

Membership Operators in Python

Membership Operators are the operators, which are used to check whether a value/variable exists in the sequence like string, list, tuples, sets, dictionary or not.

These operator returns either **True** or **False**, if a value/variable found in the list, its returns **True** otherwise it returns **False**.

Python Membership Operators

Operator	Description	Example
in	It returns True , if a variable/value found in the sequence.	10 in list1
not in	It returns True , if a variable/value does not found in the sequence.	10 not in list1

Example:

```
# Python example of "in" and "not in" Operators
```

```

# declare a list and a string
str1 = "Hello world"
list1 = [10, 20, 30, 40, 50]

# Check 'w' (capital exists in the str1 or not
if 'w' in str1:
    print "Yes! w found in ", str1
else:
    print "No! w does not found in ", str1

# check 'X' (capital) exists in the str1 or not
if 'X' not in str1:
    print "yes! X does not exist in ", str1
else:
    print "No! X exists in ", str1

# check 30 exists in the list1 or not
if 30 in list1:
    print "Yes! 30 found in ", list1
else:
    print "No! 30 does not found in ", list1

# check 90 exists in the list1 or not
if 90 not in list1:
    print "Yes! 90 does not exist in ", list1
else:
    print "No! 90 exists in ", list1

```

Output

```

Yes! w found in Hello world
yes! X does not exist in Hello world
Yes! 30 found in [10, 20, 30, 40, 50]
Yes! 90 does not exist in [10, 20, 30, 40, 50]

```

8.	<p>Define Python Tuple.</p> <p>A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.</p> <p>A tuple is a collection which is ordered and unchangeable.</p> <p>Example</p> <pre>tup1 = ('physics', 'chemistry', 1997, 2000); tup2 = (1, 2, 3, 4, 5);</pre>
9.	<p>What are the advantages of Tuple over list?</p> <p>Tuples are faster than lists.</p> <p>If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list. It makes your code safer if you “write-protect” data that does not need to be changed.</p>
10.	Classify the Python accessing elements in a Tuples.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print "tup1[0]: ", tup1[0];
print "tup2[1:5]: ", tup2[1:5];
```

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

```
tup1[0]: physics
tup2[1:5]: [2, 3, 4, 5]
```

11. Point out the methods used in Tuples

Python Tuple Methods

Method	Description
Python slice()	creates a slice object specified by range()
Python sorted()	returns sorted list from a given iterable
Python sum()	Add items of an Iterable
Python tuple() Function	Creates a Tuple

12. How a Tuple is iterated? Explain with an example.

How to iterate through a tuple

There are different ways to iterate through a tuple object. The for statement in Python has a variant which traverses a tuple till it is exhausted. It is equivalent to foreach statement in Java. Its syntax is –

```
for var in tuple:
```

```
    stmt1
```

```
    stmt2
```

Following script will print all items in the list

```
T = (10,20,30,40,50)
```

```
for var in T:
```

```
    print (T.index(var),var)
```

	<p>The output generated is –</p> <pre>0 10 1 20 2 30 3 40 4 50</pre> <p>Another approach is to iterate over range upto length of tuple, and use it as index of item in tuple</p> <pre>for var in range(len(T)): print (var,T[var])</pre> <p>You can also obtain enumerate object from the tuple and iterate through it. Following code too gives same output.</p> <pre>for var in enumerate(T): print (var)</pre>
13.	<p>Explain how Tuples are used as return values?</p> <p>Tuples as Return Values. Functions can return tuples as return values. ... In each case, a function (which can only return a single value), can create a single tuple holding multiple elements. For example, we could write a function that returns both the area and the circumference of a circle of radius r.</p> <p>Tuples as Return Values</p> <p>Functions can return tuples as return values. This is very useful — we often want to know some batsman's highest and lowest score, or we want to find the mean and the standard deviation, or we want to know the year, the month, and the day. In each case, a function (which can only return a single value), can create a single tuple holding multiple elements.</p> <p>For example, we could write a function that returns both the area and the circumference of a circle of radius</p> <p>Example</p> <pre>def circleInfo(r): """ Return (circumference, area) of a circle of radius r """ c = 2 * 3.14159 * r a = 3.14159 * r * r return (c, a) print(circleInfo(10))</pre> <p>OUTPUT</p> <pre>(62.8318, 314.159)</pre>
14.	<p>Define dictionary with an example.</p> <p>Dictionary</p> <p>A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written</p>

with **curly brackets**, and they have *keys and values*.

Example

Create and print a dictionary:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(thisdict)

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Example-2

As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the keys are strings and the values will also be strings.

One way to create a dictionary is to start with the empty dictionary and add **key-value pairs**. The empty dictionary is denoted `{}`

```
#dictionary creation
#English to Spanish translation
#Key Value pairs
```

```
engtosp = {}
```

```
engtosp['one'] = 'uno'
engtosp['two'] = 'dos'
engtosp['three'] = 'tres'
print(engtosp)
```

OUTPUT

```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}

eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
print(eng2sp)
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

15. What are the properties of dictionary keys?

Properties of Dictionary Keys

First, a given key can appear in a dictionary only once. Duplicate keys are not allowed. A dictionary maps each key to a corresponding value, so it doesn't make sense to map a particular key more than once.

Secondly, a dictionary key must be of a type that is immutable. You have already seen

	<p>examples where several of the immutable types you are familiar with—integer, float, string, and Boolean—have served as dictionary keys.</p> <p>No Restrictions on Dictionary Values</p> <p>By contrast, there are no restrictions on dictionary values. Literally none at all. A dictionary value can be any type of object Python supports, including mutable types like lists and dictionaries, and user-defined objects</p> <pre>dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'} print "dict['Name']: ", dict['Name']</pre> <p>When the above code is executed, it produces the following result –</p> <pre>dict['Name']: Manni</pre> <p>(b) Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.</p>
16.	<p>Give examples of dictionary methods</p> <p>#Dictionary Methods</p> <pre>month = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'June': 6} for key, value in month.items(): #Iteration of dictionary using For Loop print (key, value) # print Key-Value pair print(month.keys()) # print KEYs alone print(month.values()) # print values alone print(month.items()) # print dictionary items</pre> <p>OUTPUT</p> <pre>Jan 1 Feb 2 Mar 3 Apr 4 May 5 June 6</pre> <pre>dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June']) dict_values([1, 2, 3, 4, 5, 6]) dict_items([('Jan', 1), ('Feb', 2), ('Mar', 3), ('Apr', 4), ('May', 5), ('June', 6)])</pre>
17.	<p>Perform the bubble sort on the elements 23,78,45,8,32,56</p> <pre>def bubbleSort(arr): n = len(arr) # Traverse through all array elements for i in range(n): # Last i elements are already in place</pre>

	<pre> for j in range(0, n-i-1): <i># traverse the array from 0 to n-i-1</i> <i># Swap if the element found is greater</i> <i># than the next element</i> if arr[j] > arr[j+1] : arr[j], arr[j+1] = arr[j+1], arr[j] <i># Driver code to test above</i> arr = [23,78,45,8,32,56] bubbleSort(arr) print ("Sorted array is:") print(arr) OUTPUT Sorted array is: [8, 23, 32, 45, 56, 78] </pre>
18.	<p>Compose an example on insertion sort.</p> <p><i># Python program for implementation of Insertion Sort</i></p> <p><i># Function to do insertion sort</i></p> <pre> def insertionSort(arr): <i># Traverse through 1 to len(arr)</i> for i in range(1, len(arr)): key = arr[i] <i># Move elements of arr[0..i-1], that are</i> <i># greater than key, to one position ahead</i> <i># of their current position</i> j = i-1 while j >=0 and key < arr[j] : arr[j+1] = arr[j] j -= 1 arr[j+1] = key <i># Driver code to test above</i> arr = [12, 11, 13, 5, 6] insertionSort(arr) print ("Sorted array is:") for i in range(len(arr)): print ("%d" %arr[i]) Sorted array is: 5 6 11 12 13 </pre>

19. What is the use of `all()`, `any()`, `cmp()` and `sorted()` in dictionary?

Built-in Functions with Dictionary

Built-in functions like `all()`, `any()`, `len()`, `cmp()`, `sorted()` etc. are commonly used with dictionary to perform different tasks.

Function	Description
<code>all()</code>	Return <code>True</code> if all keys of the dictionary are true (or if the dictionary is empty).
<code>any()</code>	Return <code>True</code> if any key of the dictionary is true. If the dictionary is empty, return <code>False</code> .
<code>len()</code>	Return the length (the number of items) in the dictionary.
<code>cmp()</code>	Compares items of two dictionaries.
<code>sorted()</code>	Return a new sorted list of keys in the dictionary.

EXAMPLES

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

```
# Output: 5
print(len(squares))
```

```
# Output: [1, 3, 5, 7, 9]
print(sorted(squares))
```

20. Differentiate between Tuples and dictionaries

List and **tuple** is an ordered collection of items. **Dictionary** is unordered collection. List and **dictionary** objects are mutable i.e. it is possible to add new item or delete an item from it. **Tuple** is an immutable object.

List vs tuple vs dictionary in Python

List and Tuple objects are sequences. A dictionary is a hash table of key-value pairs. List and tuple is an ordered collection of items. Dictionary is unordered collection.

List and dictionary objects are mutable i.e. it is possible to add new item or delete an item from it. Tuple is an immutable object. Addition or deletion operations are not possible on tuple object.

Each of them is a collection of comma-separated items. List items are enclosed in square brackets [], tuple items in round brackets or parentheses (), and dictionary items in curly brackets { }

PART-B

1. i) What is Python List? Describe the list usage with suitable examples.

Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

How to create a list?

In Python programming, a list is created by placing all the items (elements) inside a square bracket [], separated by commas.

```
1. # empty list
2. my_list = []
3.
4. # list of integers
5. my_list = [1, 2, 3]
6.
7. # list with mixed datatypes
8. my_list = [1, "Hello", 3.4]
```

How to access elements from a list?

There are various ways in which we can access the elements of a list.

List Index

We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

How to slice lists in Python?

We can access a range of items in a list by using the slicing operator (colon).

```
1. my_list = ['p','r','o','g','r','a','m','i','z']
2. # elements 3rd to 5th
3. print(my_list[2:5])
```

Iterating Through a List

Using a for loop we can iterate through each item in a list.

```
1. for fruit in ['apple','banana','mango']:
2.     print("I like",fruit)
```

OUTPUT

```
I like apple
I like banana
I like mango
>>>
```

ii) Write a program to illustrate the heterogeneous list.

```
a = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
```

```
a += '123456789'
```

```
b = ['10', '11', '12']
```

```
a = a+b
```

```
print(a)
```

OUTPUT

```
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12']
```

#Hetrogenuous Python List

```
a = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
```

```
a += '123456789'
```

```
b = ['10', '11', '12']
```

```
c = a+b
```

```
for mon in c :
```

```
    print(mon, '-2019')
```

OUTPUT

```
Jan -2019
```

```
Feb -2019
```

```
Mar -2019
```

```
Apr -2019
```

```
May -2019
```

```
Jun -2019
```

```
Jul -2019
```

```
Aug -2019
```

```
Sep -2019
```

```
Oct -2019
```

```
Nov -2019
```

```
Dec -2019
```

```
1 -2019
```

```
2 -2019
```

```
3 -2019
```

```
4 -2019
```

```
5 -2019
```

```
6 -2019
```

```
7 -2019
```

```
8 -2019
```

```
9 -2019
```

```
10 -2019
```

```
11 -2019
```

```
12 -2019
```

2.	<p>Describe the following</p> <p>i) Creating the list</p> <p>The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.</p> <p>Creating a list is as simple as putting different comma-separated values between square brackets. For example –</p> <pre>list1 = ['physics', 'chemistry', 1997, 2000]; list2 = [1, 2, 3, 4, 5]; list3 = ["a", "b", "c", "d"]</pre> <p>Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.</p> <p>ii) Accessing values in the lists</p> <p>Accessing Values in Lists</p> <p>To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –</p> <pre>list1 = ['physics', 'chemistry', 1997, 2000]; list2 = [1, 2, 3, 4, 5, 6, 7]; print "list1[0]: ", list1[0] print "list2[1:5]: ", list2[1:5]</pre> <p>When the above code is executed, it produces the following result –</p> <pre>list1[0]: physics list2[1:5]: [2, 3, 4, 5]</pre> <p>iii) Updating the list</p> <p>Updating Lists</p> <p>You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example –</p> <pre>list = ['physics', 'chemistry', 1997, 2000]; print "Value available at index 2 : " print list[2] list[2] = 2001; print "New value available at index 2 : " print list[2]</pre> <p>When the above code is executed, it produces the following result –</p> <pre>Value available at index 2 : 1997 New value available at index 2 : 2001</pre> <p>iv) Deleting the list elements</p>
----	--

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

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

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

3.

Explain the basic list operations in detail with necessary programs.

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Basic List Operation	Python Expression	Results
Get Length	len([1, 2, 3])	3
Concatenation	[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]
Repetition	['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']
Membership	3 in [1, 2, 3]	True
Iteration	for x in [1, 2, 3]: print x,	1 2 3

Write a Python program to multiply two matrices.

```
1. # Program to multiply two matrices using nested loops
2.
3. # 3x3 matrix
4. X = [[12,7,3],
5.      [4,5,6],
6.      [7,8,9]]
7. # 3x4 matrix
8. Y = [[5,8,1,2],
9.      [6,7,3,0],
```

```

0.     [4,5,9,1]]
1. # result is 3x4
2. result = [[0,0,0,0],
3.           [0,0,0,0],
4.           [0,0,0,0]]
5.
6. # iterate through rows of X
7. for i in range(len(X)):
8.     # iterate through columns of Y
9.     for j in range(len(Y[0])):
10.        # iterate through rows of Y
11.        for k in range(len(Y)):
12.            result[i][j] += X[i][k] * Y[k][j]
13.
14. for r in result:
15.     print(r)

```

Output

```

[114, 160, 60, 27]
[74, 97, 73, 14]
[119, 157, 112, 23]

```

4. i. Discuss the Python list methods with examples.

PYTHON LIST METHODS

Operations	Python List Methods		
Name	Insert()	Append()	Extend()
Description	The insert() method inserts an element to the list at a given index.	The append() method adds an item to the end of the list.	The extend() extends the list by adding all items of a list (passed as an argument) to the end.
Syntax	list.insert(index, element)	list.append(item)	list1.extend(list2)
Parameters	The insert() function takes two parameters: ❖ index - position where an element needs to be inserted ❖ element - this is the element to be inserted in the list	❖ The method takes a single argument ❖ item - an item to be added at the end of the list ❖ The item can be numbers, strings, dictionaries, another list, and so on.	❖ The extend() method takes a single argument (a list) and adds it to the end.
Return Value	returns None.	Returns none	None
Example	# Inserting Element to List	1. # animals list	# language list

	<pre># vowel list vowel = ['a', 'e', 'i', 'u'] # inserting element to list at 4th position vowel.insert(3, 'o') print('Updated List: ', vowel)</pre>	<pre>2. animals = ['cat', 'dog', 'rabbit'] 3. 4. # 'guinea pig' is appended to the animals list 5. animals.append('guinea pig') 6. 7. # Updated animals list 8. print('Updated animals list: ', animals)</pre>	<pre>language = ['French', 'English', 'German'] # another list of language language1 = ['Spanish', 'Portuguese'] language.extend(language1) # Extended List print('Language List: ', language)</pre>
OUTPUT	<pre>Updated List: ['a', 'e', 'i', 'o', 'u']</pre>	<pre>Updated animals list: ['cat', 'dog', 'rabbit', 'guinea pig']</pre>	<pre>Language List: ['French', 'English', 'German', 'Spanish', 'Portuguese']</pre>

- ii. Why it is necessary to have both the functions append and extend? What is the result of the following expression that uses append where it probably intended to use extend?

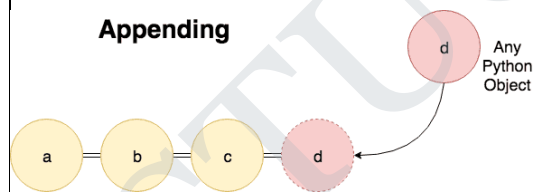
```
>>>lst=[1,2,3]
>>>lst.append([4,5,6])
```

What is the difference between the list methods append and extend?

append adds its argument as a single element to the end of a list. The length of the list itself will increase by one.

extend iterates over its argument adding each element to the list, extending the list.

Example-1: Append



The **append** method is used to add an object to a list.

This object can be of **any data type**, a string, an integer, a boolean, or even another list.

The following code is used to **append** an item to a list **L** that initially has 4 elements

```
>>> L = [1, 2, 3, 4]
>>> L.append(5)
>>> L
[1, 2, 3, 4, 5]
```

The append method adds the new item 5 to the list.

Now, the length of the list has increased by one because the append method adds **only one object** to the list.

This is an important distinction of append method when compared to the case with **extend**.

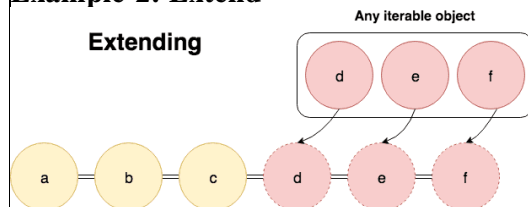
Now let's try to append a **list** to our list.

```
>>> L = [1, 2, 3, 4]
```

```
>>> L.append([5, 6, 7])
>>> L
[1, 2, 3, 4, [5, 6, 7]]
```

Now we appended **one object** (which happens to be of type list) to our list **L**. Again, after the modification the list length grew by only one. Now let's take a look at a similar, yet different, method.

Example-2: Extend



extend is another very common list method.

Unlike **append** that can take an object of **any type** as an argument, **extend** can only take an iterable object as an argument.

An iterable object is an object that you can iterate through like strings, lists, tuples, dicts, or any object with the `__iter__()` method.

What **extend** does is very straightforward, it iterates through the **iterable** object one item at a time and appends each item to the list.

For example, let's try to extend a list by another list.

```
>>> L = [1, 2, 3, 4]
>>> L.extend([5, 6, 7])
>>> L
[1, 2, 3, 4, 5, 6, 7]
```

As you can see in the example above, **extend** takes a list (which is an iterable) as an argument and appends each item of the list to **L**.

Three integer objects were appended to the list and the list size grew by three.

This behavior is obviously different from that of the **append** method.

5. i) Illustrate List comprehension with suitable examples

List comprehension is an elegant way to define and create list in Python. These lists have often the qualities of sets, but are not in all cases sets.

List comprehension is a complete substitute for the lambda function as well as the functions `map()`, `filter()` and `reduce()`. For most people the syntax of list comprehension is easier to be grasped.

A `map()` function to convert Celsius values into Fahrenheit and vice versa - list comprehension:

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = [ ((float(9)/5)*x + 32) for x in Celsius ]
>>> print Fahrenheit
[102.56, 97.70, 99.14, 100.04]
>>>
```

The following list comprehension creates the Pythagorean triples:

```
>>> [(x,y,z) for x in range(1,30) for y in range(x,30) for z in range(y,30) if x**2 +
```



```

y**2 == z**2]
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8,
15, 17), (9, 12, 15), (10, 24, 26), (12, 16, 20), (15,
20, 25), (20, 21, 29)]
>>>
Cross product of two sets:
>>> colours = [ "red", "green", "yellow", "blue" ]
>>> things = [ "house", "car", "tree" ]
>>> coloured_things = [ (x,y) for x in colours for y in things ]
>>> print coloured_things
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'house'), ('green', 'car'), ('green',
'tree'), ('yellow', 'house'), ('yellow', 'car'), ('yellow', 'tree'), ('blue', 'house'), ('blue',
'car'), ('blue', 'tree')]
>>>
iii)    Write a Python program to concatenate two lists.
        # Python 3 code to demonstrate list
        # concatenation using + operator

        # Initializing lists
        test_list3 = [1, 4, 5, 6, 5]
        test_list4 = [3, 5, 7, 2, 5]

        # using + operator to concat
        test_list3 = test_list3 + test_list4

        # Printing concatenated list
        print ("Concatenated list using + : " str(test_list3))

```

Output:

Concatenated list using + : [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]

Using list comprehension

List comprehension can also accomplish this task of list concatenation. In this case, a new list is created, but this method is a one liner alternative to the loop method discussed above.

```

# Python3 code to demonstrate list
# concatenation using list comprehension
# Initializing lists
test_list1 = [1, 4, 5, 6, 5]
test_list2 = [3, 5, 7, 2, 5]
# using list comprehension to concat
res_list = [y for x in [test_list1, test_list2] for y in x]
# Printing concatenated list
print ("Concatenated list using list comprehension: "+ str(res_list))

```

Output:

Concatenated list using list comprehension: [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]

Method #4 : Using extend()

extend() is the function extended by lists in Python and hence can be used to perform this task. This function performs the inplace extension of first list.

Python3 code to demonstrate list

concatenation using list.extend()

Initializing lists

```
test_list3 = [1, 4, 5, 6, 5]
```

```
test_list4 = [3, 5, 7, 2, 5]
```

using list.extend() to concat

```
test_list3.extend(test_list4)
```

Printing concatenated list

```
print ("Concatenated list using list.extend() : "+ str(test_list3))
```

Output:

Concatenated list using list.extend() : [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]

6. i. What is a Python Tuple? What are the advantages of Tuple over list?

Tuples in Python

A Tuple is a collection of Python objects separated by commas. In some ways a tuple is similar to a list in terms of indexing, nested objects and repetition but a tuple is immutable unlike lists which are mutable.

Creating Tuples

An empty tuple

```
empty_tuple = ()
```

```
print (empty_tuple)
```

OUTPUT

```
()
```

Creating non-empty tuples

One way of creation

```
tup = 'python', 'programming'
```

```
print(tup)
```

Another for doing the same

```
tup = ('python', 'programming')
```

```
print(tup)
```

Output

```
('python', 'programming')
```

```
('python', 'programming')
```

Code for concatenating 2 tuples

```
tuple1 = (0, 1, 2, 3)
tuple2 = ('python', 'tuple')
```

```
# Concatenating above two
print(tuple1 + tuple2)
```

Output:

```
(0, 1, 2, 3, 'python', 'tuple')
```

ii. “Tuples are immutable”. Explain with example.

In Python, tuples are immutable, and "immutable" means the value cannot change. These are well-known, basic facts of Python.

According to the [Python data model](#), "objects are Python's abstraction for data, and all data in a Python program is represented by objects or by relations between objects". Every value in Python is an object, including integers, floats, and Booleans. In Java, these are "primitive data types" and considered separate from "objects". Not so, in Python. So not only is the `datetime.datetime(2018, 2, 4, 19, 38, 54, 798338)` datetime object an object, but the integer `42` is an object and the Boolean `True` is an object.

Every value in Python is an object.

All Python objects have three things: a value, a type, and an identity.

```
>>> spam = 42
>>> spam
42
>>> type(spam)
<class 'int'>
>>> id(spam)
1594282736
```

The variable `spam` refers to an object that has a value of `42`, a type of `int`, and an identity of `1594282736`. An identity is a unique integer, created when the object is created, and never changes for the lifetime of the object. An object's type also cannot change. Only the value of an object may change.

Let's try changing an object's value by entering the following into the interactive shell:

```
>>> spam = 42
>>> spam = 99
```

You may think you've changed the object's value from `42` to `99`, but you haven't. All you've done is made `spam` refer to a new object. You can confirm this by calling the `id()` function and noticing `spam` refers to a completely new object:

```
>>> spam = 42
>>> id(spam)
1594282736
>>> spam = 99
>>> id(spam)
1594284560
```

7. Illustrate the ways of creating the Tuple and the Tuple assignment with suitable programs.

A tuple¹ is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are **immutable**. Tuples are also **comparable** and **hashable** so we can sort lists of them and use tuples as key values in Python dictionaries.

Creation of Tuples

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

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

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

Without the comma Python treats ('a') as an expression with a string in parentheses that evaluates to a string:

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

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

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

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

Most list operators also work on tuples. The bracket operator indexes an element:

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

And the slice operator selects a range of elements.

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

But if you try to modify one of the elements of the tuple, you get an error:

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

TypeError: object doesn't support item assignment
 You can't modify the elements of a tuple.

Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left hand side of an assignment statement. This allows you to assign more than one variable at a time when the left hand side is a sequence.

In this example we have a two element list (which is a sequence) and assign the first and second elements of the sequence to the variables `x` and `y` in a single statement.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

It is not magic, Python *roughly* translates the tuple assignment syntax to be the following:

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

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

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

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

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

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

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

8. What are the accessing elements in a Tuple? Explain with suitable programs.

Example: 1

```
1 Tuple = (3, 5, 6.7, "Python")
2 print("Third element of the Tuple is:", Tuple[2])
```

Output:

Third element of the Tuple is: 6.7

Example: 2

```
1 Tuple = (3, 5, 6.7, "Python")
2 print("First element of the Tuple is:", Tuple[0])
3 print("Last element of the Tuple is:", Tuple[3])
```

Output:

First element of the Tuple is: 3

Last element of the Tuple is: 'Python'

We can also access the items present in the nested tuple with the help of nested indexing.

Example: 3

```
1 Tuple = ("Python", [2, 4, 6], (4, 5.6, "Hi"))
2 print("First element of the tuple is:", Tuple[0][1])
3 print("Items present inside another list or tuple is:", Tuple[2][1])
```

Output:

First element of the tuple is: 'y'

Items present inside another list or tuple is: 5.6

Example: 4

```
1 Tuple = (3, 5, 7.8)
2 print("Last element of the tuple is:", Tuple[-1])
```

Output:

Last element of the tuple is: 7.8

Packing and Unpacking the Tuple

Python provides an important feature called packing and unpacking. In packing, we put the value into a tuple, but in unpacking, we extract all those values stored in the tuples into variables.

Example: 5

```
1 Tuple = ("John", 23567, "Software Engineer")
2 (eName, eID, eTitle) = Tuple

3 print("Packed tuples is:", Tuple)
4 print("Employee name is:", eName)

5 print("Employee ID is:", eID)
```

	<pre>6 print("Employee Title is:", eTitle)</pre> <p>Output: Packed tuples is: ("John", 23567, "Software Engineer") Employee name is: John Employee ID is: 23567 Employee Title is: Software Engineer</p>																														
9.	<p>i. Explain the basic Tuple operations with examples.</p> <p>Basic Tuples Operations Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string. In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter –</p> <table><tr><th>Tuple Operation</th><th>Python Expression</th><th>Results</th></tr><tr><td>len() – get length of tuple</td><td>len((1, 2, 3))</td><td>3</td></tr><tr><td>concatenation</td><td>(1, 2, 3) + (4, 5, 6)</td><td>(1, 2, 3, 4, 5, 6)</td></tr><tr><td>Replication</td><td>('Hi!') * 4</td><td>('Hi!', 'Hi!', 'Hi!', 'Hi!')</td></tr><tr><td>Membership</td><td>3 in (1, 2, 3)</td><td>True</td></tr><tr><td>For Loop</td><td>for x in (1, 2, 3): print x,</td><td>1 2 3</td></tr></table> <p>Indexing, Slicing, and Matrixes Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input – L = ('spam', 'Spam', 'SPAM!')</p> <table><tr><th>Python Expression</th><th>Results</th><th>Description</th></tr><tr><td>L[2]</td><td>'SPAM!'</td><td>Offsets start at zero</td></tr><tr><td>L[-2]</td><td>'Spam'</td><td>Negative: count from the right</td></tr><tr><td>L[1:]</td><td>['Spam', 'SPAM!']</td><td>Slicing fetches sections</td></tr></table> <p>ii. Write a program to check whether an element ‘y’ and ‘a’ belongs to the tuple My_tuple = ('p','y','t','h','o','n') and after printing the result, delete the Tuple.</p> <p>iii.</p> <pre>my_tuple = ('p','y','t','h','o','n',)</pre> <pre># In operation # Output: True print('y' in my_tuple)</pre> <pre># Output: False</pre>	Tuple Operation	Python Expression	Results	len() – get length of tuple	len((1, 2, 3))	3	concatenation	(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Replication	('Hi!') * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Membership	3 in (1, 2, 3)	True	For Loop	for x in (1, 2, 3): print x,	1 2 3	Python Expression	Results	Description	L[2]	'SPAM!'	Offsets start at zero	L[-2]	'Spam'	Negative: count from the right	L[1:]	['Spam', 'SPAM!']	Slicing fetches sections
Tuple Operation	Python Expression	Results																													
len() – get length of tuple	len((1, 2, 3))	3																													
concatenation	(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)																													
Replication	('Hi!') * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')																													
Membership	3 in (1, 2, 3)	True																													
For Loop	for x in (1, 2, 3): print x,	1 2 3																													
Python Expression	Results	Description																													
L[2]	'SPAM!'	Offsets start at zero																													
L[-2]	'Spam'	Negative: count from the right																													
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections																													

	<pre>print('a' in my_tuple)</pre> <p># Can delete an entire tuple</p> <pre>del my_tuple</pre>																				
10.	<p>Describe the built in functions with Tuples.</p> <p style="text-align: center;"><i>Built-In Methods</i></p> <table border="1"> <thead> <tr> <th>BUILT-IN FUNCTION</th><th>DESCRIPTION</th></tr> </thead> <tbody> <tr> <td>all()</td><td>Returns true if all element are true or if tuple is empty</td></tr> <tr> <td>any()</td><td>return true if any element of the tuple is true. if tuple is empty, return false</td></tr> <tr> <td>len()</td><td>Returns length of the tuple or size of the tuple</td></tr> <tr> <td>enumerate()</td><td>Returns enumerate object of tuple</td></tr> <tr> <td>max()</td><td>return maximum element of given tuple</td></tr> <tr> <td>min()</td><td>return minimum element of given tuple</td></tr> <tr> <td><u>sum()</u></td><td>Sums up the numbers in the tuple</td></tr> <tr> <td><u>sorted()</u></td><td>input elements in the tuple and return a new sorted list</td></tr> <tr> <td><u>tuple()</u></td><td>Convert an iterable to a tuple.</td></tr> </tbody> </table> <p>Write a program to use Max(), Min() and sorted() methods in Tuple.</p>	BUILT-IN FUNCTION	DESCRIPTION	all()	Returns true if all element are true or if tuple is empty	any()	return true if any element of the tuple is true. if tuple is empty, return false	len()	Returns length of the tuple or size of the tuple	enumerate()	Returns enumerate object of tuple	max()	return maximum element of given tuple	min()	return minimum element of given tuple	<u>sum()</u>	Sums up the numbers in the tuple	<u>sorted()</u>	input elements in the tuple and return a new sorted list	<u>tuple()</u>	Convert an iterable to a tuple.
BUILT-IN FUNCTION	DESCRIPTION																				
all()	Returns true if all element are true or if tuple is empty																				
any()	return true if any element of the tuple is true. if tuple is empty, return false																				
len()	Returns length of the tuple or size of the tuple																				
enumerate()	Returns enumerate object of tuple																				
max()	return maximum element of given tuple																				
min()	return minimum element of given tuple																				
<u>sum()</u>	Sums up the numbers in the tuple																				
<u>sorted()</u>	input elements in the tuple and return a new sorted list																				
<u>tuple()</u>	Convert an iterable to a tuple.																				
11.	<p>Discuss a)Tuples as return values b) Variable Length Argument Tuples</p> <p>Working With Functions: Return Values</p>																				

- Most functions take in arguments, perform some processing and then return a value to the caller. In Python this is achieved with the `return` statement.

```
def square(n):
    return n*n

two_squared = square(2)

# or print it as before

print(square(2))
```

- Python also has the ability to return multiple values from a function call, something missing from many other languages. In this case the return values should be a comma-separated list of values and Python then constructs a *tuple* and returns this to the caller, e.g.

```
def square(x,y):
    return x*x, y*y

t = square(2,3)

print(t) # Produces (4,9)

# Now access the tuple with usual operations
```

is possible to return multiple values from a function in the form of tuple, list, dictionary or an object of a user defined class

Return as tuple

```
>>> def function():
    a=10; b=10
    return a,b
```

```
>>> x=function()
>>> type(x)
<class 'tuple'>
>>> x
(10, 10)
>>> x,y=function()
>>> x,y
(10, 10)
```

Return as list

```
>>> def function():
    a=10; b=10
    return [a,b]
```

```
>>> x=function()
>>> x
[10, 10]
>>> type(x)
```

```
<class 'list'>
```

Return as dictionary

```
>>> def function():
    d=dict()
    a=10; b=10
    d['a']=a; d['b']=b
    return d

>>> x=function()
>>> x
{'a': 10, 'b': 10}
>>> type(x)
<class 'dict'>
```

Write a program to illustrate the comparison operators in Tuple.

Comparing tuples

A comparison operator in Python can work with tuples.

The comparison starts with a first element of each tuple. If they do not compare to =, < or > then it proceed to the second element and so on.

It starts with comparing the first element from each of the tuples

Let's study this with an example-

#case 1

```
a=(5,6)
b=(1,4)
if (a>b):print("a is bigger")
else: print("b is bigger")
```

#case 2

```
a=(5,6)
b=(5,4)
if (a>b):print("a is bigger")
else: print ("b is bigger")
```

#case 3

```
a=(5,6)
b=(6,4)
if (a>b):print("a is bigger")
else: print("b is bigger")
```

12. Write a Python program to perform linear search on a list.

Python Program for Linear Search

Problem: Given an array `arr[]` of `n` elements, write a function to search a given element `x` in `arr[]`.

Examples :

Input : `arr[] = { 10, 20, 80, 30, 60, 50, 110, 100, 130, 170 }`
`x = 110;`

Output : 6

Element `x` is present at index 6

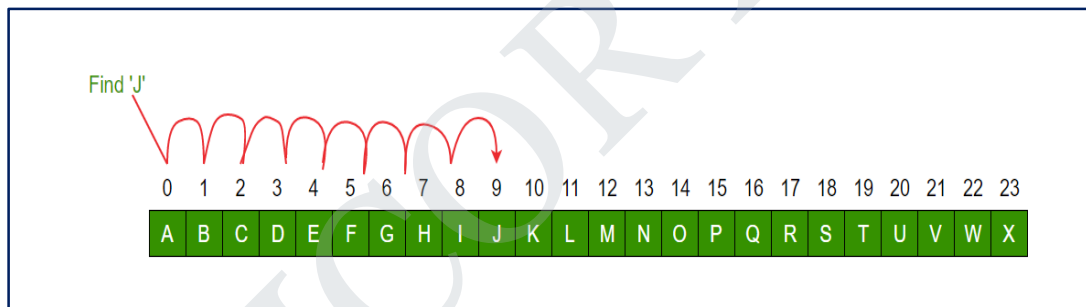
Input : `arr[] = { 10, 20, 80, 30, 60, 50, 110, 100, 130, 170 }`
`x = 175;`

Output : -1

Element `x` is not present in `arr[]`.

A simple approach is to do **linear search**, i.e

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



Example:

Searching an element in a list/array in python

can be simply done using `'in'` operator

if `x` in `arr`: `print arr.index(x)`

Linearly search `x` in `arr[]`

If `x` is present then return its location else return -1

Def `search(arr, x):`

for `i` **in** `range(len(arr)):`

if `arr[i] == x:`
return `i`

return -1

Write a Python program to store 'n' numbers in a list and sort the list using selection sort.

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

```
arr[] = 64 25 12 22 11
```

```
// Find the minimum element in arr[0...4] and place it at beginning
```

```
11 25 12 22 64
```

```
// Find the minimum element in arr[1...4]
```

```
// and place it at beginning of arr[1...4]
```

```
11 12 25 22 64
```

```
// Find the minimum element in arr[2...4]
```

```
// and place it at beginning of arr[2...4]
```

```
11 12 22 25 64
```

```
// Find the minimum element in arr[3...4]
```

```
// and place it at beginning of arr[3...4]
```

```
11 12 22 25 64
```

Python program for implementation of Selection # Sort

```
import sys
```

```
A = [64, 25, 12, 22, 11]
```

```
# Traverse through all array elements
```

```
for i in range(len(A)):
```

```
    # Find the minimum element in remaining unsorted array
```

```
    min_idx = i
```

```
    for j in range(i+1, len(A)):
```

```
        if A[min_idx] > A[j]:
```

```
            min_idx = j
```

```
# Swap the found minimum element with the first element
```

```
A[i], A[min_idx] = A[min_idx], A[i]
```

Driver code to test above

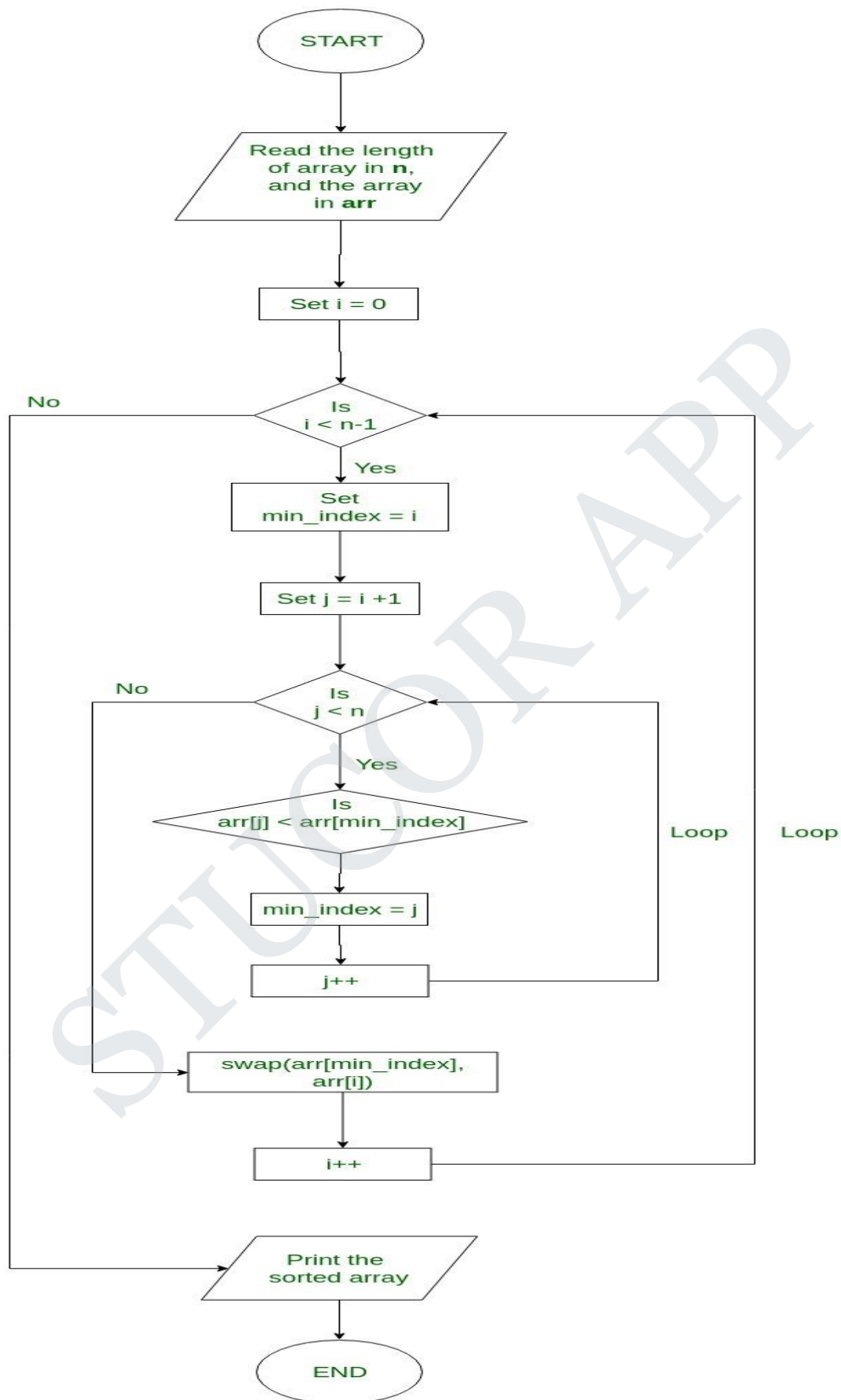
```
print ("Sorted array")
```

```
for i in range(len(A)):
```

```
    print("%d" %A[i]),
```

Output:

```
Sorted array:  11 12 22 25 64
```



Flowchart for Selection Sort

13. Explain the properties of Dictionary keys with examples.

Properties of Dictionary Keys

More than one entry per key is not allowed (no duplicate key is allowed)

The values in the **dictionary** can be of any type while the keys must be immutable like numbers, tuples or strings.

Differentiate a List and Dictionary. Explain the Dictionary Methods with examples.

Python: Dictionary and its properties

Dictionary is a generalization of a List. Dictionary stores (key, value) pair for easier value access in future using the key. We see how it is different from a list and its properties.

What is a Dictionary?

In Python, Dictionary is a generalization of a List. Values in a list are accessed using index that start from 0, whereas values in a Dictionary are accessed using a **key**. A key can be any immutable python object. Dictionaries are surrounded by curly braces.

Dictionaries are yet another widely used data-structures in Python, just like Lists. It would be interesting to know how it is possible to have various types of keys in dictionary and yet have a fast lookup while accessing the value for that key. Python creates a hash for the key and created internal hash: value map.

In dictionary, we can define our own keys.

In `dict2 = {'i': 'one', 'ii': 'two', 'iii': 'three', 'iv': 'four'}` , keys are ['i', 'ii', 'iii', 'iv'].

An example comparing a **list** and **dict** would be helpful:

Example of Python List	Example of Python Dictionary
<pre>>>> list1 = ['zero', 'one', 'two']</pre> <p>The first element of list1 can be accessed using index list1[0]</p> <pre>>>> list1[0] 'zero'</pre>	<pre>>>> dict1 = {0: 'zero', 1: 'one', 2: 'two'} >>> dict1[0] 'zero' >>> dict1[1] 'one'</pre>
<p>Accessing an element in a list requires us to use specific index, which is an integer.</p>	<p>However, we can have any other immutable Python object for accessing an element in case of dictionary! For example:</p> <pre>>>> dict2 = {'i': 'one', 'ii': 'two', 'iii': 'three', 'iv': 'four'} >>> dict2['ii'] 'two'</pre>

Declaring a Dictionary	Accessing elements of a Dictionary	Updating a value against a key in Python dict
Empty dictionary is declared by	We use square brackets along with	Dictionaries are mutable, just like

<p>empty curly braces.</p> <pre>>>> dict1 = {} >>> dict1 {} >>> type(dict1) <type 'dict'></pre>	<p>the key whose value we want to access.</p> <p>Examples:</p> <pre>>>> month = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'June': 6} >>> month['Apr'] 4</pre>	<p>lists. We can update a value at using a specific key.</p> <pre>>>> month = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'June': 6} >>> month['Jan'] = 'One' >>> month.get('Jan') 'One'</pre>
<p>Unlike a List or Tuple, Dictionary does not maintain the order in which you added the keys and values. For instance, if you try to print the dict in the above example, you'll see a random order.</p>		
<p>Dictionary methods</p> <p>Generalized form of declaring a dictionary is d = {key1: value1, key2: value2 }:</p> <pre>>>> month = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'June': 6}</pre>		
<p>dict.keys()</p>	<p>dict1.values()</p>	<p>dict1.items()</p>
<p>dict.keys() returns a list of all the keys of the dictionary.</p> <pre>>>> month.keys() ['Mar', 'Feb', 'Apr', 'June', 'Jan', 'May']</pre> <p>Similarly, we can get a list of the all the values of the dictionary.</p>	<p>dict.values() returns a list of all the values of the dictionary.</p> <pre>>>> month.values() [3, 2, 4, 6, 1, 5]</pre>	<p>dict.items() returns a list of (key, value) tuples of the dictionary.</p> <pre>>>> month.items() [('Mar', 3), ('Feb', 2), ('Apr', 4), ('June', 6), ('Jan', 1), ('May', 5)]</pre>
<p>Looping over a Dictionary in Python</p> <p>dict.items() returns a list of (key, value) tuples of the dictionary. We can use it iterate/loop over the dictionary.</p> <pre>>>> month.items() [('Mar', 3), ('Feb', 2), ('Apr', 4), ('June', 6), ('Jan', 1), ('May', 5)]</pre> <p>for key, value in month.items(): print key, value</p> <pre>Mar 3 Feb 2 Apr 4 June 6 Jan One May 5 July 7</pre>		<p>Deleting elements of a Dictionary</p> <p>We can delete a value in a dict by using a key. For example:</p> <pre>>>> month = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'June': 6} >>> del month['Mar'] >>> month {'Feb': 2, 'Apr': 4, 'June': 6, 'Jan': 1, 'May': 5}</pre> <p>We can also delete an entire dict using del</p> <pre>>>> del month</pre>

14.	Write a Python program named weather that is passed a dictionary of daily temperatures and returns the average temperature over the weekend for the weekly temperatures given.
-----	--

STUCOR APP

GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING																	
UNIT 5 - FILES, MODULES, PACKAGES																	
SYLLABUS																	
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.																	
Part-A																	
Q. No.	Q&A																
1.	<p>Point out different modes of file opening. Python File I/O: Read and Write Files in Python</p> <table border="1"> <thead> <tr> <th>Mode</th><th>Description</th></tr> </thead> <tbody> <tr> <td>'r'</td><td>Open a file for reading. (default)</td></tr> <tr> <td>'w'</td><td>Open file for writing. Creates a new file if it does not exist or truncates the file if it exists.</td></tr> <tr> <td>'x'</td><td>Open a file for exclusive creation. If the file already exists, the operation fails.</td></tr> <tr> <td>'a'</td><td>Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.</td></tr> <tr> <td>'t'</td><td>Open in text mode. (default)</td></tr> <tr> <td>'b'</td><td>Open in binary mode.</td></tr> <tr> <td>'+'</td><td>Open a file for updating (reading and writing)</td></tr> </tbody> </table> <p> <code>f = open("test.txt")</code> # equivalent to 'r' or 'rt' <code>f = open("test.txt", 'w')</code> # write in text mode <code>f = open("img.bmp", 'r+b')</code> # read and write in binary mod </p>	Mode	Description	'r'	Open a file for reading. (default)	'w'	Open file for writing. Creates a new file if it does not exist or truncates the file if it exists.	'x'	Open a file for exclusive creation. If the file already exists, the operation fails.	'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.	't'	Open in text mode. (default)	'b'	Open in binary mode.	'+'	Open a file for updating (reading and writing)
Mode	Description																
'r'	Open a file for reading. (default)																
'w'	Open file for writing. Creates a new file if it does not exist or truncates the file if it exists.																
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.																
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.																
't'	Open in text mode. (default)																
'b'	Open in binary mode.																
'+'	Open a file for updating (reading and writing)																
2.	<p>Define the access modes access_mode - The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).</p>																
3.	<p>Distinguish between files and modules. Any Python file is a module, its name being the file's base name without the .py extension. A package is a collection of Python modules: while a module is a single Python file, a package is a directory of Python modules containing an additional <code>__init__.py</code> file, to distinguish a package from a directory that just happens to contain a bunch of Python scripts.</p>																
4.	<p>Define read and write file Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python. Read and Write ('r+') : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.</p>																
5.	Describe renaming and delete.																

	<pre>#Syntax for renaming a file import os # Rename a file from test1.txt to test2.txt os.rename("test1.txt", "test2.txt")</pre> <p>Syntax</p> <pre>os.remove(file_name)</pre> <p>Example</p> <pre># Following is the example to delete an existing file test2.txt: #!/usr/bin/python import os # Delete file test2.txt os.remove("text2.txt")</pre>
6.	<p>Discover the format operator available in files.</p> <p>Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".</p>
7.	<p>Explain with example the need for exceptions.</p> <p>What is an Exception?</p> <p>An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash.</p> <p>Why use Exceptions?</p> <p>Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.</p>
8.	<p>Explain Built-in exceptions.</p> <p>Python Built-in Exceptions</p>

Exception	Cause of Error
AssertionError	Raised when <code>assert</code> statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the <code>input()</code> functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's <code>close()</code> method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by <code>next()</code> function to indicate that there is no further item to be returned by iterator.

	<p>SyntaxError Raised by parser when syntax error is encountered.</p> <p>IndentationError Raised when there is incorrect indentation.</p> <p>TabError Raised when indentation consists of inconsistent tabs and spaces.</p> <p>SystemError Raised when interpreter detects internal error.</p> <p>SystemExit Raised by <code>sys.exit()</code> function.</p> <p>TypeError Raised when a function or operation is applied to an object of incorrect type.</p> <p>UnboundLocalError Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.</p> <p>UnicodeError Raised when a Unicode-related encoding or decoding error occurs.</p> <p>UnicodeEncodeError Raised when a Unicode-related error occurs during encoding.</p> <p>UnicodeDecodeError Raised when a Unicode-related error occurs during decoding.</p> <p>UnicodeTranslateError Raised when a Unicode-related error occurs during translating.</p> <p>ValueError Raised when a function gets argument of correct type but improper value.</p> <p>ZeroDivisionError Raised when second operand of division or modulo operation is zero.</p> <p>Python Built-in Exceptions</p>
9.	<p>Difference between built-in exceptions and handling exception</p> <p>Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.</p> <p>However, sometimes you may need to create custom exceptions that serves your purpose.</p>

User-defined exceptions

Though Python has many built-in exception covering a lot of error scenarios but sometimes you as a user would want to create your own exception for a specific scenario in order to make error messages more relevant to the context. Such exceptions are called user-defined exceptions or custom exceptions.

User-defined exception Python example

Suppose you have a Python function that take age as a parameter and tells whether a person is eligible to vote or not. Voting age is 18 or more.

If person is not eligible to vote you want to raise an exception using [raise statement](#), for this scenario you want to write a custom exception named “InvalidAgeError”.

```
# Custom exception
class InvalidAgeError(Exception):
    def __init__(self, arg):
        self.msg = arg

def vote_eligibility(age):
    if age < 18:
        raise InvalidAgeError("Person not eligible to vote, age is " + str(age))
    else:
        print('Person can vote, age is', age)

try:
    vote_eligibility(22)
    vote_eligibility(14)
except InvalidAgeError as error:
    print(error)
```

Output

```
Person can vote, age is 22
Person not eligible to vote, age is 14
```

10. Write a program to write a data in a file for both write and append modes.

```
f = open("demofile1.txt", "w")
f.write("I am writing to a new file")
f.close()
```

```
#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

OUTPUT

```
"I am writing to a file"
```

	<pre>f = open("demofile2.txt", "a") f.write("Now the file has more content!") f.close() #open and read the file after the appending: f = open("demofile2.txt", "r") print(f.read())</pre> <p>OUTPUT</p> <p>Hello! Welcome to demofile2.txt This file is for testing purposes. Good Luck!Now the file has more content!</p>
11.	<p>How to import statements?</p> <p>Modules are Python .py files that consist of Python code. Any Python file can be referenced as a module. A Python file called hello.py has the module name of hello that can be imported into other Python files or used on the Python command line interpreter.</p> <p>Modules can define functions, classes, and variables that you can reference in other Python .py files or via the Python command line interpreter.</p> <p>In Python, modules are accessed by using the import statement. When you do this, you execute the code of the module, keeping the scopes of the definitions so that your current file(s) can make use of these.</p> <p>When Python imports a module called hello for example, the interpreter will first search for a built-in module called hello. If a built-in module is not found, the Python interpreter will then search for a file named hello.py in a list of directories that it receives from the sys.path variable.</p> <p>Importing Modules</p> <p>To make use of the functions in a module, you'll need to import the module with an import statement.</p> <p>An import statement is made up of the import keyword along with the name of the module.</p> <p>In a Python file, this will be declared at the top of the code, under any shebang lines or general comments.</p> <p>So, in the Python program file my_rand_int.py we would import the random module to generate random numbers in this manner:</p> <pre>my_rand_int.py import random</pre>
12.	Express about namespace and scoping

What is a Namespace in Python?

Namespace is a collection of names.

In Python, you can imagine a namespace as a mapping of every name, you have defined, to corresponding objects.

Different namespaces can co-exist at a given time but are completely isolated.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long we don't exit.

This is the reason that built-in functions like `id()`, `print()` etc. are always available to us from any part of the program. Each [module](#) creates its own global namespace.

These different namespaces are isolated. Hence, the same name that may exist in different modules do not collide.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar, is the case with class. Following diagram may help to clarify this concept.



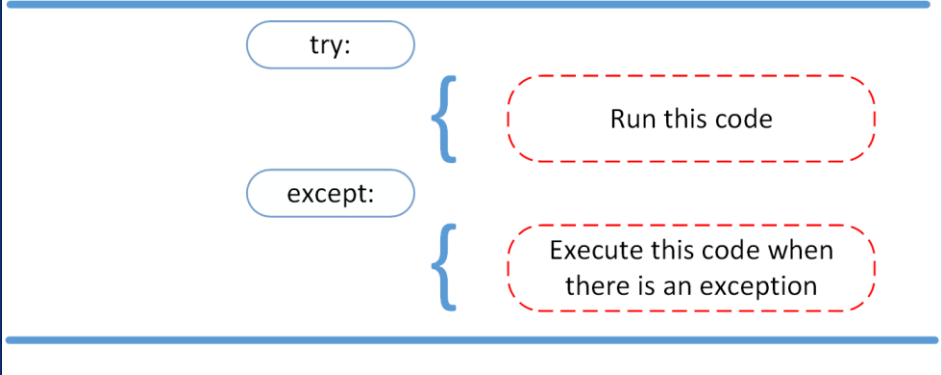
Python Variable Scope

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

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

	<p>At any given moment, there are at least three nested scopes.</p> <p>Scope of the current function which has local names Scope of the module which has global names Outermost scope which has built-in names</p> <p>When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.</p> <p>Example of scope of variables</p> <pre>def outer_function(): a = 20 def inner_function(): a = 30 print('a =',a) inner_function() print('a =',a) a = 10 outer_function() print('a =',a)</pre> <p>OUTPUT</p> <pre>a = 30 a = 20 a = 10</pre>
13.	<p>Differentiate global and local</p> <pre># This function has a variable with # name same as s. def f(): s = "I am Local" print s # Global scope s = "I am Global" f() print s</pre> <p>OUTPUT</p> <pre>I am Local I am Global</pre>
14.	<p>Identify what are packages in Python.</p> <p>Any Python file is a module, its name being the file's base name/module's name property without the .py extension. A package is a collection of Python modules, i.e., a package is a directory of Python modules containing an additional <code>__init__.py</code> file. The <code>__init__.py</code> distinguishes a package from a</p>

	<p>directory that just happens to contain a bunch of Python scripts.</p> <pre>>>> import datetime >>> from datetime import date >>> date.today() datetime.date(2017, 9, 1)</pre>
15.	<p>Examine buffering</p> <p>What is the use of buffering in python's built-in open() function? The optional buffering argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size (in bytes). A negative buffering means to use the system default. If omitted, the system default is used.</p> <p>Example</p> <pre>filedata = open(file.txt,"r",0) or filedata = open(file.txt,"r",1) or filedata = open(file.txt,"r",2)</pre>
16.	<p>Discuss file.isatty[]</p> <p>Python File isatty() Method</p> <p>Example</p> <p>Check if the file is connected to a terminal device:</p> <pre>f = open("demofile.txt", "r") print(f.isatty()) OUTPUT False</pre>
17.	<p>Discover except Clause with Multiple exception</p> <p>The try and except Block: Handling Exceptions</p> <p>The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a “normal” part of the program. The code that follows the except statement is the program's response to any exceptions in the preceding try clause.</p>

	 <p>The diagram illustrates the structure of a try-except block. It shows a 'try:' label followed by a blue curly brace, and an 'except:' label followed by another blue curly brace. To the right of the first brace is a red dashed box containing the text 'Run this code'. To the right of the second brace is a red dashed box containing the text 'Execute this code when there is an exception'.</p>
18.	<p>Create a Python script to display the current date and time.</p> <p>Example : Python get today's date</p> <pre>1. from datetime import date 2. import time.time 3. 4. today = date.today() 5. print("Today's date:", today)</pre> <p>OUTPUT Today's date: 2019-12-10 >>></p> <pre>from datetime import datetime import pytz print(datetime.now(pytz.timezone('Asia/Kolkata')))</pre> <p>OUTPUT 2019-12-10 21:02:30 + 5 : 30</p>
19.	<p>Analyze the object as return values.</p> <p>The return statement makes a python function to exit and hand back a value to its caller. The objective of functions in general is to take in inputs and return something. A return statement, once executed, immediately halts execution of a function, even if it is not the last statement in the function.</p> <p>Functions that return values are sometimes called fruitful functions.</p> <pre>def sum(a,b): return a+b sum(5,16)</pre> <p>Output 21</p> <p>Everything in python, almost everything is an object. Lists, dictionaries, tuples are also python objects. The code below shows a python function that returns a python object; a dictionary</p> <pre># This function returns a dictionary def foo(): d = dict(); d['str'] = "Tutorialspoint"</pre>

	<pre>d['x'] = 50 return d print foo()</pre> <p>Output</p> <pre>{'x': 50, 'str': 'Tutorialspoint'}</pre>
20.	<p>Discuss a modular design</p> <p>Modular programming is a software design technique, which is based on the general principal of modular design. Modular design is an approach which has been proven as indispensable in engineering even long before the first computers. Modular design means that a complex system is broken down into smaller parts or components, i.e. modules. These components can be independently created and tested. In many cases, they can be even used in other systems as well.</p> <p>Importing Modules</p> <p>Python module : every file, which has the file extension .py and consists of proper Python code, can be seen or is a module. There is no special syntax required to make such a file a module. A module can contain arbitrary objects, for example files, classes or attributes. All those objects can be accessed after an import. There are different ways to import a modules. We demonstrate this with the math module:</p> <pre>import math</pre> <p>The module math provides mathematical constants and functions, e.g. π (math.pi), the sine function (math.sin()) and the cosine function (math.cos()). Every attribute or function can only be accessed by putting "math." in front of the name:</p> <pre>>>> math.pi 3.141592653589793 >>> math.sin(math.pi/2) 1.0 >>> math.cos(math.pi/2) 6.123031769111886e-17 >>> math.cos(math.pi) -1.0</pre>
PART-B	
1.	<p>Write a Python program to demonstrate the file I/O operations. The most basic tasks involved in file manipulation are reading data from files and writing or appending data to files.</p> <p>Reading and Writing Files in Python</p> <p>In our first example we want to show how to read data from a file. The way of telling Python that we want to read from a file is to use the open function. The first parameter is the name of the file we want to read and with the second parameter, assigned to the value "r", we state that we want to read from the file:</p> <pre>fobj = open("wordsworth.txt", "r")</pre>

The "r" is optional. An open() command with just a file name is opened for reading per default. The open() function returns a file object, which offers attributes and methods.

```
fobj = open("wordsworth.txt.txt")
```

After we have finished working with a file, we have to close it again by using the file object method close():

```
fobj.close()
```

Now we want to finally open and read a file. The method rstrip() in the following example is used to strip off whitespaces (newlines included) from the right side of the string "line":

```
fobj = open("ad_lesbiam.txt")
for line in fobj:
    print(line.rstrip())
fobj.close()
```

OUTPUT

```
I wandered lonely as a cloud
That floats on high o'er vales and hills,
When all at once I saw a crowd,
A host, of golden daffodils;
Beside the lake, beneath the trees,
Fluttering and dancing in the breeze.
```

2. Discuss the different modes for opening a file and closing a file.
Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
1. >>> f = open("test.txt") # open file in current directory
2. >>> f = open("C:/Python33/README.txt") # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

Mode	Description
------	-------------

'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

Python File Modes

```

1. f = open("test.txt") # equivalent to 'r' or 'rt'
2. f = open("test.txt",'w') # write in text mode
3. f = open("img.bmp",'r+b') # read and write in binary mode

1. f = open("test.txt",mode = 'r',encoding = 'utf-8')
```

Closing a file in Python

When we are done with operations to the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file and is done using Python close() method.

Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

```

1. f = open("test.txt",encoding = 'utf-8')
2. # perform file operations
3. f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a `try...finally` block.

```

1. try:
2.     f = open("test.txt",encoding = 'utf-8')
```

	<pre> 3. # perform file operations 4. finally: 5. f.close() </pre> <p>This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.</p>
3.	<p>i) Write a program to catch a divide by zero exception. Add a finally block too.</p> <pre> > def divide(x, y): ... try: ... result = x / y ... except ZeroDivisionError: ... print("division by zero!") ... else: ... print("result is", result) ... finally: ... print("executing finally clause") ... >>> divide(2, 1) result is 2.0 executing finally clause >>> divide(2, 0) division by zero! executing finally clause >>> divide("2", "1") executing finally clause Traceback (most recent call last): File "<stdin>", line 1, in <module> File "<stdin>", line 3, in divide TypeError: unsupported operand type(s) for /: 'str' and 'str' </pre> <p>ii) Write a function to print the hash of any given file in Python.</p> <pre> # Python 3 code to demonstrate # SHA hash algorithms. import hashlib # initializing string str = "GreekandLatin" # encoding GreekandLatin using encode() # then sending to SHA256() result = hashlib.sha256(str.encode()) # printing the equivalent hexadecimal value. print("The hexadecimal equivalent of SHA256 is : ") print(result.hexdigest()) </pre>

	<pre> print ("\r") # initializing string str = "GreekandLatin" # encoding GreekandLatin using encode() # then sending to SHA384() result = hashlib.sha384(str.encode()) # printing the equivalent hexadecimal value. print("The hexadecimal equivalent of SHA384 is : ") print(result.hexdigest()) print ("\r") </pre> <p>OUTPUT</p> <p>The hexadecimal equivalent of SHA256 is : bed3b89c643693e40b1bb6f8ae65cb75eb5925e03918179e801f79e399980efc</p> <p>The hexadecimal equivalent of SHA384 is : 29d4ccd433fbbba7a40e73fdd89b55cd1cbb8cc0707f6b0e565c62809a680956a2a799f6ff9a47ad0ae36107e4cf9116b</p>
4.	<p>i) Describe the use of try block and except block in python with syntax.</p> <p>What is Exception?</p> <p>An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.</p> <p>When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.</p> <p>Handling an exception</p> <p>If you have some <i>suspicious</i> code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.</p> <p>Syntax</p> <p>Here is simple syntax of <i>try....except...else</i> blocks –</p> <pre> try: You do your operations here except ExceptionI: If there is ExceptionI, then execute this block. except ExceptionII: If there is ExceptionII, then execute this block. else: If there is no exception then execute this block. </pre> <p>EXAMPLE</p>

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all –

```
#!/usr/bin/python3

try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print ("Error: can't find file or read data")
else:
    print ("Written content in the file successfully")
    fh.close()
```

This produces the following result –

Written content in the file successfully

- ii) Describe with an example exceptions with arguments in python.

Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```
try:
    You do your operations here;
    .....
except ExceptionType, Argument:
    You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception –

```
#!/usr/bin/python

# Define a function here.
def temp_convert(var):
    try:
        return int(var)
    except ValueError, Argument:
        print "The argument does not contain numbers\n", Argument
```


	<p># Call above function here. temp_convert("xyz");</p> <p>This produces the following result –</p> <p>The argument does not contain numbers invalid literal for int() with base 10: 'xyz'</p>																		
5.	<p>Explain about the files related methods</p> <p>A file object is created using <i>open</i> function and here is a list of functions which can be called on this object –</p> <table border="1" data-bbox="256 674 1555 1917"> <thead> <tr> <th data-bbox="256 674 370 779">Sr.No.</th><th data-bbox="370 674 1555 779">Methods with Description</th></tr> </thead> <tbody> <tr> <td data-bbox="256 779 370 936">1</td><td data-bbox="370 779 1555 936"> <u>file.close()</u> Close the file. A closed file cannot be read or written any more. </td></tr> <tr> <td data-bbox="256 936 370 1066">2</td><td data-bbox="370 936 1555 1066"> <u>file.flush()</u> Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects. </td></tr> <tr> <td data-bbox="256 1066 370 1255">3</td><td data-bbox="370 1066 1555 1255"> <u>file.fileno()</u> Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system. </td></tr> <tr> <td data-bbox="256 1255 370 1386">4</td><td data-bbox="370 1255 1555 1386"> <u>file.isatty()</u> Returns True if the file is connected to a tty(-like) device, else False. </td></tr> <tr> <td data-bbox="256 1386 370 1516">5</td><td data-bbox="370 1386 1555 1516"> <u>file.next()</u> Returns the next line from the file each time it is being called. </td></tr> <tr> <td data-bbox="256 1516 370 1646">6</td><td data-bbox="370 1516 1555 1646"> <u>file.read([size])</u> Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes). </td></tr> <tr> <td data-bbox="256 1646 370 1776">7</td><td data-bbox="370 1646 1555 1776"> <u>file.readline([size])</u> Reads one entire line from the file. A trailing newline character is kept in the string. </td></tr> <tr> <td data-bbox="256 1776 370 1917">8</td><td data-bbox="370 1776 1555 1917"> <u>file.readlines([sizehint])</u> Reads until EOF using readline() and return a list containing the lines. If the optional sizehint </td></tr> </tbody> </table>	Sr.No.	Methods with Description	1	<u>file.close()</u> Close the file. A closed file cannot be read or written any more.	2	<u>file.flush()</u> Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.	3	<u>file.fileno()</u> Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.	4	<u>file.isatty()</u> Returns True if the file is connected to a tty(-like) device, else False.	5	<u>file.next()</u> Returns the next line from the file each time it is being called.	6	<u>file.read([size])</u> Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).	7	<u>file.readline([size])</u> Reads one entire line from the file. A trailing newline character is kept in the string.	8	<u>file.readlines([sizehint])</u> Reads until EOF using readline() and return a list containing the lines. If the optional sizehint
Sr.No.	Methods with Description																		
1	<u>file.close()</u> Close the file. A closed file cannot be read or written any more.																		
2	<u>file.flush()</u> Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.																		
3	<u>file.fileno()</u> Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.																		
4	<u>file.isatty()</u> Returns True if the file is connected to a tty(-like) device, else False.																		
5	<u>file.next()</u> Returns the next line from the file each time it is being called.																		
6	<u>file.read([size])</u> Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).																		
7	<u>file.readline([size])</u> Reads one entire line from the file. A trailing newline character is kept in the string.																		
8	<u>file.readlines([sizehint])</u> Reads until EOF using readline() and return a list containing the lines. If the optional sizehint																		

		argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.
9	<u>file.seek(offset[, whence])</u> Sets the file's current position	
10	<u>file.tell()</u> Returns the file's current position	
11	<u>file.truncate([size])</u> Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.	
12	<u>file.write(str)</u> Writes a string to the file. There is no return value.	
13	<u>file.writelines(sequence)</u> Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings.	
6.	<p>i) Structure Renaming a file</p> <p>Python os.rename() method</p> <p>OS module in Python provides functions for interacting with the operating system. OS comes under Python's standard utility modules. This module provides a portable way of using operating system dependent functionality.</p> <p><i>os.rename()</i> method in Python is used to rename a file or directory.</p> <p>This method renames a source file/ directory to specified destination file/directory.</p> <p>Syntax: <i>os.rename(source, destination, *, src_dir_fd = None, dst_dir_fd = None)</i></p> <p>Parameters:</p> <p>source: A path-like object representing the file system path. This is the source file path which is to be renamed.</p> <p>destination: A path-like object representing the file system path.</p> <p>src_dir_fd (optional): A file descriptor referring to a directory.</p> <p>dst_dir_fd (optional): A file descriptor referring to a directory.</p> <p>Return Type: This method does not return any value.</p> <p>Example: Use of <i>os.rename()</i> method</p>	

```
# Python program to explain os.rename() method

# importing os module
import os

# Source file path
source = 'GeeksforGeeks/file.txt'

# destination file path
dest = 'GeekforGeeks/newfile.txt'

# Now rename the source path
# to destination path
# using os.rename() method
os.rename(source, dest)
print("Source path renamed to destination path successfully.")
```

Output:

Source path renamed to destination path successfully.

7. i) Describe the import statements
ii) Describe the from...import statements

Importing Modules

To make use of the functions in a module, you'll need to import the module with an import statement.

An import statement is made up of the import keyword along with the name of the module.

In a Python file, this will be declared at the top of the code, under any shebang lines or general comments.

. This means that we will have to refer to the function in dot notation, as in [module].[function].

In practice, with the example of the random module, this may look like a function such as:

Example

So, in the Python program file my_rand_int.py we would import the random module to generate random numbers in this manner:

```
my_rand_int.py
```

```
import random
```

When we import a module, we are making it available to us in our current program as a separate namespace

Let's create a for loop to show how we will call a function of the random module within our my_rand_int.py program:

```
my_rand_int.py
```

```
import random
```

```
for i in range(10):
```

- `random.randint()` which calls the function to return a random integer, or
- `random.randrange()` which calls the function to return a random element from a specified range.

The **import** statement allows you to import one or more modules into your Python program, letting you make use of the definitions constructed in those modules.

Using **from ... import**

To refer to items from a module within your program's namespace, you can use the **from ... import** statement. When you import modules this way, you can refer to the functions by name rather than through dot notation

In this construction, you can specify which definitions to reference directly.

In other programs, you may see the **import** statement take in references to everything defined within the module by using an asterisk (*) as a wildcard, but this is discouraged by PEP 8.

Let's first look at importing one specific function, `randint()` from the `random` module:

Output

6
9
1
14
3
22
10
1
15
9

my_rand_int.py

from random import randint

Here, we first call the **from** keyword, then `random` for the module. Next, we use the **import** keyword and call the specific function we would like to use.

Now, when we implement this function within our program, we will no longer write the function in dot notation as `random.randint()` but instead will just write `randint()`:

my_rand_int.py

from random import randint

for i in range(10):

`print(randint(1, 25))`

When you run the program, you'll receive output similar to what we received earlier.

Using the **from ... import** construction allows us to reference the defined elements of a module within our program's namespace, letting us avoid dot notation.

	<h2>Aliasing Modules</h2> <p>It is possible to modify the names of modules and their functions within Python by using the <code>as</code> keyword.</p> <p>You may want to change a name because you have already used the same name for something else in your program, another module you have imported also uses that name, or you may want to abbreviate a longer name that you are using a lot.</p> <p>The construction of this statement looks like this:</p> <pre>import [module] as [another_name]</pre> <p>Let's modify the name of the <code>math</code> module in our <code>my_math.py</code> program file. We'll change the module name of <code>math</code> to <code>m</code> in order to abbreviate it. Our modified program will look like this:</p>	<div>my_math.py</div> <pre>import math as m</pre> <pre>print(m.pi) print(m.e)</pre> <p>Within the program, we now refer to the <code>pi</code> constant as <code>m.pi</code> rather than <code>math.pi</code>.</p> <p>For some modules, it is commonplace to use aliases. The <code>matplotlib.pyplot</code> module's official documentation calls for use of <code>plt</code> as an alias:</p> <pre>import matplotlib.pyplot as plt</pre> <p>This allows programmers to append the shorter word <code>plt</code> to any of the functions available within the module, as in <code>plt.show()</code>.</p>	
8.	<p>Describe in detail locating modules.</p> <h3>Locating Python modules</h3> <h3>Finding Modules: The Path</h3> <p>For modules to be available for use, the Python interpreter must be able to locate the module file. Python has a set of directories in which it looks for module files. This set of directories is called the <i>search path</i>, and is analogous to the <code>PATH</code> environment variable used by an operating system to locate an executable file.</p> <p>Python's search path is built from a number of sources:</p> <ul style="list-style-type: none"> • <code>PYTHONHOME</code> is used to define directories that are part of the Python installation. If this environment variable is not defined, then a standard directory structure is used. For Windows, the standard location is based on the directory into which Python is installed. For most Linux environments, Python is installed under <code>/usr/local</code>, and the libraries can be found there. For Mac OS, the home directory is under <code>/Library/Frameworks/Python.framework</code>. • <code>PYTHONPATH</code> is used to add directories to the path. This environment variable is formatted like the OS <code>PATH</code> variable, with a series of filenames separated by <code>:</code>'s (or <code>;</code>'s for Windows). 		

- Script Directory. If you run a Python script, that script's directory is placed first on the search path so that locally-defined modules will be used instead of built-in modules of the same name.
- The site module's locations are also added. (This can be disabled by starting Python with the -S option.) The site module will use the PYTHONHOME location(s) to create up to four additional directories. Generally, the most interesting one is the site-packages directory. This directory is a handy place to put additional modules you've downloaded. Additionally, this directory can contain .PTH files. The site module reads .PTH files and puts the named directories onto the search path.

The search path is defined by the path variable in the sys module. If we import sys, we can display sys.path. This is very handy for debugging. When debugging shell scripts, it can help to run 'python -c 'import sys; print sys.path' just to see parts of the Python environment settings.

Installing a module, then, is a matter of assuring that the module appears on the search path. There are four central methods for doing this.

- Some packages will suggest you create a directory and place the package in that directory. This may be done by downloading and unzipping a file. It may be done by using Subversion and synchronizing your subversion copy with the copy on a server. Either way, you will likely only need to create an operating system link to this directory and place that link in site-packages directory.
- Some packages will suggest you download (or use subversion) to create a temporary copy. They will provide you with a script — typically based on setup.py — which moves files into the correct locations. This is called the distutils distribution. This will generally copy the module files to the site-packages directory.
- Some packages will rely on setuptools. This is a package from the [Python Enterprise Application Kit](#) that extends distutils to further automates download and installation. This tool, also, works by moving the working library modules to the site-packages directory.
- Extending the search path. Either set the PYTHONPATH environment variable, or put .PTH files in the site-packages directory.

Windows Environment

In the Windows environment, the Python_Path symbol in the Windows registry is used to locate modules.

9. Identify the various methods used to delete the elements from the dictionary

Python | Ways to remove a key from dictionary

Dictionary is used in manifold practical applications such as day-day programming, web development and AI/ML programming as well, making it a useful container overall. Hence, knowing shorthands for achieving different tasks related to dictionary usage always is a plus. This article deals with one such task of deleting a dictionary key-value pair from a dictionary.

Method 1 : Using del

del keyword can be used to in-place delete the key that is present in the dictionary. One drawback that can be thought of using this is that it raises an exception if the key is not found and hence non-existence of key has to be handled.

Method 2 : Using pop()

pop() can be used to delete a key and its value in-place. Advantage over using del is that it provides the mechanism to print desired value if tried to remove a non-existing dict. pair. Second, it also returns the value of key that is being removed in addition to performing a simple delete

<p>Code #1 : Demonstrating key-value pair deletion using del</p> <pre># Python code to demonstrate # removal of dict. pair # using del # Initializing dictionary test_dict = {"Arushi" : 22, "Anuradha" : 21, "Mani" : 21} # Printing dictionary before removal print ("The dictionary before performing remove is : " + str(test_dict)) # Using del to remove a dict # removes Mani del test_dict['Mani'] # Printing dictionary after removal print ("The dictionary after remove is : " + str(test_dict)) # Using del to remove a dict # raises exception del test_dict['Manjeet']</pre> <p>Output : The dictionary before performing remove is : {'Anuradha': 21, 'Haritha': 21, 'Arushi': 22, 'Mani': 21} The dictionary after remove is : {'Anuradha': 21, 'Haritha': 21, 'Arushi': 22}</p> <p>Exception : Traceback (most recent call last): File "/home/44db951e7011423359af4861d475458aipy/line 20, in del test_dict['Manjeet'] KeyError: 'Manjeet'</p>	<p>operation.</p> <p>Code #2 : Demonstrating key-value pair deletion using pop()</p> <pre># Python code to demonstrate # removal of dict. pair # using pop() test_dict = {"Arushi" : 22, "Anuradha" : 21, "Mani" : 21, "Haritha" : 21} # Initializing dictionary test_dict = {"Arushi" : 22, "Anuradha" : 21, "Mani" : 21, "Haritha" : 21} # Printing dictionary before removal print ("The dictionary before performing remove is : " + str(test_dict)) # Using pop() to remove a dict. pair # removes Mani removed_value = test_dict.pop('Mani') # Printing dictionary after removal print ("The dictionary after remove is : " + str(test_dict)) print ("The removed key's value is : " + str(removed_value)) print ('\r') # Using pop() to remove a dict. pair # doesn't raise exception # assigns 'No Key found' to removed_value removed_value = test_dict.pop('Manjeet', 'No Key found') # Printing dictionary after removal print ("The dictionary after remove is : " + str(test_dict)) print ("The removed key's value is : " + str(removed_value))</pre> <p>Output : The dictionary before performing remove is : {'Arushi': 22, 'Anuradha': 21, 'Mani': 21, 'Haritha': 21} The dictionary after remove is : {'Arushi': 22, 'Anuradha': 21, 'Haritha': 21} The removed key's value is : 21 The dictionary after remove is : {'Arushi': 22, 'Anuradha': 21, 'Haritha': 21} The removed key's value is : No Key found</p>	
<p align="center">Method 3 : Using items() + dict comprehension</p> <p>items() coupled with dict comprehension can also help us achieve task of key-value pair deletion but, it</p>		

has drawback of not being an inplace dict. technique. Actually a new dict is created except for the key we don't wish to include.

Code #3 : Demonstrating key-value pair deletion using `items()` + dict. comprehension

```
# Python code to demonstrate
# removal of dict. pair
# using items() + dict comprehension

# Initializing dictionary
test_dict = {"Arushi" : 22, "Anuradha" : 21, "Mani" : 21, "Haritha" : 21}

# Printing dictionary before removal
print ("The dictionary before performing remove is : " + str(test_dict))

# Using items() + dict comprehension to remove a dict. pair
# removes Mani
new_dict = {key:val for key, val in test_dict.items() if key != 'Mani'}

# Printing dictionary after removal
print ("The dictionary after remove is : " + str(new_dict))
```

Output :

The dictionary before performing remove is : {'Anuradha': 21, 'Haritha': 21, 'Arushi': 22, 'Mani': 21}

The dictionary after remove is : {'Anuradha': 21, 'Haritha': 21, 'Arushi': 22}

10. Describe in detail exception handling with sample program

Python Exceptions

An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.

Whenever an exception occurs, the program halts the execution, and thus the further code is not executed. Therefore, an exception is the error which python script is unable to tackle with.

Python provides us with the way to handle the Exception so that the other part of the code can be executed without any disruption. However, if we do not handle the exception, the interpreter doesn't

Common Exceptions

A list of common exceptions that can be thrown from a normal python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.

	execute all the code that exists after the that.	5. EOFError: It occurs when the end of the file is reached, and yet operations are being performed.	
	<p>Problem without handling exceptions</p> <p>Example</p> <pre>a = int(input("Enter a:")) b = int(input("Enter b:")) c = a/b; print("a/b = %d"%c)</pre> <p>#other code:</p> <pre>print("Hi I am other part of the program")</pre> <p>Output:</p> <pre>Enter a:10 Enter b:0 Traceback (most recent call last): File "exception-test.py", line 3, in <module> c = a/b; ZeroDivisionError: division by zero</pre>	<p>Handling Zero divide Exception</p> <p>Example</p> <pre>try: a = int(input("Enter a:")) b = int(input("Enter b:")) c = a/b; print("a/b = %d"%c) except: print("can't divide by zero") else: print("Hi I am else block")</pre> <p>Output:</p> <pre>Enter a:10 Enter b:0 can't divide by zero</pre>	
	<p>Exception handling in python</p> <p>If the python program contains suspicious code that may throw the exception, we must place that code in the try block. The try block must be followed with the except statement which contains a block of code that will be executed if there is some exception in the try block.</p> <p>Syntax</p> <pre>try: #block of code</pre> <pre>except Exception1: #block of code</pre>	<p>We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.</p> <p>The syntax to use the else statement with the try-except statement is given below.</p> <pre>try: #block of code</pre> <pre>except Exception1: #block of code</pre> <pre>else: #this code executes if no except block is executed</pre>	

	<pre> except Exception2: #block of code #other code </pre>	
11.	<p>Write a program to find the one's complement of binary number using file.</p> <pre> # Python3 program to print 1's and 2's # complement of a binary number # Returns '0' for '1' and '1' for '0' def flip(c): return '1' if (c == '0') else '0' # Print 1's and 2's complement of # binary number represented by "bin" def printOneAndTwosComplement(bin): n = len(bin) ones = "" twos = "" # for ones complement flip every bit for i in range(n): ones += flip(bin[i]) # for two's complement go from right # to left in ones complement and if # we get 1 make, we make them 0 and # keep going left when we get first # 0, make that 1 and go out of loop ones = list(ones.strip("")) twos = list(ones) for i in range(n - 1, -1, -1): if (ones[i] == '1'): twos[i] = '0' else: twos[i] = '1' break # If No break : all are 1 as in 111 or 11111 # in such case, add extra 1 at beginning if (i == -1): </pre>	

	<pre> twos.insert(0, '1') print("1's complement: ", *ones, sep = "") print("2's complement: ", *twos, sep = "") # Driver Code if __name__ == '__main__': bin = "111100" printOneAndTwosComplement(bin.strip("")) OUTPUT 1's complement: 000011 2's complement: 000100 </pre>
12.	<p>Write a program to display a pyramid # Python program to # print Diamond shape</p> <pre> # Function to print # Diamond shape def Pyramid(rows): n = 0 for i in range(1, rows + 1): # loop to print spaces for j in range (1, (rows - i) + 1): print(end = " ") # loop to print star while n != (2 * i - 1): print("*", end = "") n = n + 1 n = 0 # line break print() # Driver Code # number of rows input rows = 10 Pyramid(rows) </pre>
13.	<p>Write a program to find the number of instances of different digits in a given number # Python program to find the frequency of each element of an array</p> <pre> # Input size of array n = int(input('Enter the number of elements : ')) arr = [] freq = [-1] * n # Input elements in array print('\nEnter elements in array: ') for i in range(n): temp = int(input()) # Initially initialize frequencies to -1 arr.append(temp) </pre>

	<pre> for i in range(0, n): count = 1 for j in range(i+1, n): if(arr[i] == arr[j]): count = count + 1 freq[j] = 0 if(freq[i] != 0): freq[i] = count # Print frequency of each element print("\nFrequency of all elements of array : \n"); for i in range(0, n): if(freq[i] != 0): print(arr[i], ' occurs ', freq[i], ' times') OUTPUT Enter a number : 43829 Frequency of all elements of array : 4 occurs -1 times 3 occurs -1 times 8 occurs -1 times 2 occurs -1 times 9 occurs 1 times </pre>
14.	<p>Describe in detail printing to the screen.</p> <p>Screen output</p> <p>Text output is one of the basics in Python programming. Not all Programs have graphical user interfaces, text screens often suffice.</p> <p>You can output to the terminal with print function. This function displays text on your screen, it won't print.</p> <p>The terminal is a very simple interface for Python programs. While not as shiny as a GUI or web app, it's good enough to cover the basics in.</p> <p>Print function</p> <p>Create a new program (text file) in your IDE or code editor. Name the file <i>hello.py</i>. It only needs one line of code.</p> <p>To output text to the screen you will need this line::</p> <pre>print("Hello World")</pre> <p>Run the program (from terminal: python hello.py) If you run the program:</p> <pre>Hello World</pre>

Print newline

The program above prints everything on a single line. At some point you'll want to write multiple lines.

To write multiple lines, add the '\n' character:

```
print("This is First Line" \n This is Second Line ")
```

OUTPUT

This is First Line

This is Second Line

In **Python** strings, the backslash "\" is a special character, also called the "**escape**" character. It is used in representing certain whitespace **characters**: "\t" is a tab, "\n" is a newline, and "\r" is a carriage return.

Print variables

To print variables:

```
x = 3
print(x)
```

OUTPUT

3

To print multiple variables on one line:

```
x = 2
y = 3
print("x = {}, y = {}".format(x,y))
```

OUTPUT:

x = 2, y = 3

STUCOR APP