**GE3151-Problem Solving and Python Programming**

**PART-C**

**UNIT I - ALGORITHMIC PROBLEM SOLVING**

**1. Outline the Towers of Hanoi problem. Suggest a solution to the Towers of Hanoi problem with relevant diagrams. (Jan-2018)**

**Towers of Hanoi Problem**

The **Towers of Hanoi** is a classic puzzle that involves three pegs and a set of disks of different sizes. The problem is to move all the disks from one peg to another, following these rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the top disk from one stack and placing it on top of another stack.
3. No disk may be placed on top of a smaller disk.

**Problem Description**

* You are given three pegs: **A**, **B**, and **C**.
* Initially, all disks are stacked on peg **A** in decreasing size (largest at the bottom).
* The goal is to move all disks from **peg A** to **peg C** using **peg B** as an auxiliary peg.

**Solution (Recursive Approach)**

The solution to the problem can be broken down into smaller sub-problems using recursion:

1. Move the top n−1n-1 disks from **peg A** to **peg B**, using **peg C** as auxiliary.
2. Move the nthn^{th} (largest) disk from **peg A** to **peg C**.
3. Move the n−1n-1 disks from **peg B** to **peg C**, using **peg A** as auxiliary.

**Recursive Formula:**

* For n=1n = 1, move the disk directly from **A** to **C**.
* For n>1n > 1, first solve for n−1n-1 disks, then move the largest disk, and then solve for n−1n-1 disks again.

**Algorithm for Towers of Hanoi**

def hanoi(n, source, target, auxiliary):

if n == 1:

print(f"Move disk 1 from {source} to {target}")

return

hanoi(n-1, source, auxiliary, target) # Move n-1 disks to auxiliary

print(f"Move disk {n} from {source} to {target}") # Move nth disk to target

hanoi(n-1, auxiliary, target, source) # Move n-1 disks from auxiliary to target

**🔹 Example (for n=3n = 3):**

* **Disks**: D1,D2,D3D1, D2, D3 (where D3D3 is the largest disk).
* **Pegs**: A (source), B (auxiliary), C (target).

**Steps (for n=3n = 3):**

1. Move D1D1 and D2D2 from A to B (using C as auxiliary).
2. Move D3D3 from A to C.
3. Move D1D1 and D2D2 from B to C (using A as auxiliary).

**Diagram:**

Initial Configuration:

Peg A: [D3, D2, D1]

Peg B: []

Peg C: []

Step 1: Move D1 from A to C.

Peg A: [D3, D2]

Peg B: []

Peg C: [D1]

Step 2: Move D2 from A to B.

Peg A: [D3]

Peg B: [D2]

Peg C: [D1]

Step 3: Move D3 from A to C.

Peg A: []

Peg B: [D2]

Peg C: [D1, D3]

Step 4: Move D2 from B to C.

Peg A: []

Peg B: []

Peg C: [D1, D2, D3]

**🔹 Conclusion:**

The Towers of Hanoi problem is a great example of recursion in algorithm design. The solution involves moving smaller sub-problems (sub-stacks) to auxiliary pegs, and then combining these steps to solve the larger problem.

**2. Identify Simple Strategies for Developing an Algorithm. (Jan-2019)**

**🔹 Simple Strategies for Developing an Algorithm**

Developing an efficient algorithm involves following systematic steps that help to break down the problem and design an effective solution. Some key strategies include:

**1. Understand the Problem**

* **Clarify the problem**: Define what is being asked and what needs to be solved.
* **Input/Output**: Identify the inputs to the algorithm and the expected outputs.

**Example**: In sorting a list of numbers, the input is a list of numbers, and the output is the sorted list.

**2. Break Down the Problem**

* **Divide the problem into smaller sub-problems**: Break complex problems into smaller, manageable parts.
* **Decompose into simpler tasks**: This helps in identifying reusable operations, especially in recursive or iterative solutions.

**Example**: For sorting, a complex solution like Merge Sort divides the list into smaller sub-lists, sorts them individually, and then merges them back together.

**3. Define Data Structures**

* Choose appropriate data structures to represent the input, intermediate steps, and output of the algorithm. Data structures should be chosen based on the nature of the problem.

**Example**: A queue might be used for breadth-first search, while a stack is used for depth-first search.

**4. Identify the Core Operations**

* Identify and define the basic operations or steps required to solve the problem. This will help in formulating a sequence of steps for the algorithm.

**Example**: In a search algorithm, the basic operation might be “compare the target value with the current element.”

**5. Design the Algorithm Step-by-Step**

* **Sequential steps**: Break down the solution into a series of logical steps, ensuring each step leads to solving the problem.
* **Pseudocode**: Write the algorithm in a language-agnostic pseudocode format to outline the logic before writing it in a specific programming language.

**Example**: For binary search:

1. Compare the middle element with the target value.
2. If equal, return the position.
3. If smaller, search in the right half; if larger, search in the left half.

**6. Optimize the Algorithm**

* **Efficiency**: Ensure that the algorithm performs well in terms of time and space complexity. Avoid unnecessary computations and redundant steps.
* **Minimize steps**: If possible, find ways to optimize the algorithm by reducing unnecessary loops, recursive calls, or data structure manipulations.

**7. Validate and Test**

* **Testing**: Test the algorithm with sample inputs, including edge cases, to ensure that it works correctly.
* **Debugging**: Check for logical errors, performance issues, and correctness.

**3. Mention the Different Types of Iterative Structures Allowed in Python. Explain the Use of Continue and Break Statements with an Example. (May 2019)**

**Types of Iterative Structures in Python**

Python provides several types of loops for iteration:

**1. For Loop**

The **for** loop is used to iterate over a sequence (like a list, tuple, string, or range).

**Syntax:**

for variable in sequence:

# Block of code

**Example:**

for i in range(5):

print(i)

**2. While Loop**

The **while** loop repeats a block of code as long as a condition is true.

**Syntax:**

while condition:

# Block of code

**Example:**

count = 0

while count < 5:

print(count)

count += 1

**The Use of continue and break Statements**

The **continue** and **break** statements are used inside loops to control the flow of iteration.

**1. continue Statement**

* The continue statement is used to skip the current iteration of a loop and continue with the next iteration.

**Example:**

for i in range(1, 6):

if i == 3:

continue # Skip the number 3

print(i)

**Output**:

1

2

4

5

**2. break Statement**

* The break statement is used to terminate the loop prematurely when a condition is met.

**Example:**

for i in range(1, 6):

if i == 4:

break # Exit the loop when i equals 4

print(i)

**Output**:

1

2

3

**Conclusion**

* **For Loop** is ideal for iterating over sequences, and **while loop** is best when the number of iterations is not fixed and depends on a condition.
* **continue** is used to skip the current iteration and move to the next, while **break** stops the loop completely. Both statements enhance the flexibility of loops in Python.

**UNIT II – DATA TYPES, EXPRESSIONS, STATEMENTS**

**1. Write a Python program to swap two variables.**

**Question:** Write a Python program that takes two variables and swaps their values without using a temporary variable.

python

# Program to swap two variables

# Input two variables

a = int(input("Enter the first number (a): "))

b = int(input("Enter the second number (b): "))

# Swapping values using tuple unpacking

a, b = b, a

# Display the swapped values

print("After swapping: a =", a)

print("After swapping: b =", b)

**Explanation:**

* The program prompts the user to input two numbers.
* It swaps the values of a and b using Python’s tuple unpacking feature, which allows swapping without a temporary variable.
* The swapped values are then printed.

**2. Write a Python program to check whether a given year is a leap year or not.**

**Question:** Write a Python program that checks if a given year is a leap year.

python

CopyEdit

# Program to check if a year is a leap year

# Input year

year = int(input("Enter a year: "))

# Check if the year is a leap year

if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):

print(f"{year} is a Leap Year.")

else:

print(f"{year} is not a Leap Year.")

**Explanation:**

* A year is a leap year if it is divisible by 4, but not divisible by 100, unless it is also divisible by 400.
* The program checks the conditions for leap year and prints whether the year is a leap year or not.

**3. Write a Python program to convert Celsius to Fahrenheit.**

**Question:** Write a Python program that converts a given temperature in Celsius to Fahrenheit using the formula:

Fahrenheit=(Celsius×1.8)+32\text{Fahrenheit} = (\text{Celsius} \times 1.8) + 32Fahrenheit=(Celsius×1.8)+32

python

CopyEdit

# Program to convert Celsius to Fahrenheit

# Input temperature in Celsius

celsius = float(input("Enter temperature in Celsius: "))

# Convert to Fahrenheit

fahrenheit = (celsius \* 1.8) + 32

# Display the result

print(f"{celsius}°C is equal to {fahrenheit}°F.")

**Explanation:**

* The program prompts the user to input a temperature in Celsius.
* It then applies the conversion formula to calculate the equivalent temperature in Fahrenheit.
* The result is displayed to the user.

**UNIT III - CONTROL FLOW, FUNCTIONS, STRINGS**

**1. Explain in detail about list as arrays.**

**Explanation:**

In Python, lists can function similarly to arrays in other programming languages. A list is a collection of elements that can be of different data types, whereas arrays in some languages are typically used to store elements of the same data type. However, in Python, a list can hold elements of multiple data types, which makes it more flexible than a traditional array.

**Key Points:**

1. **Homogeneous vs. Heterogeneous**: Unlike arrays in languages like C or Java, Python lists can store heterogeneous data types (different types in the same list). In contrast, arrays in languages like C or Java require all elements to be of the same data type.
2. **Accessing elements**: Similar to arrays, elements in a Python list are accessed via their index positions. Python uses zero-based indexing, which means the first element is at index 0, the second element at index 1, and so on.
3. **Advantages**:
   * Python lists can be resized dynamically, unlike static arrays in languages like C.
   * They support various methods such as append(), remove(), insert(), and others to manipulate the elements.
4. **Limitations**: While lists are flexible, they are less efficient in memory and speed compared to arrays in languages like C when handling large datasets. For numerical computations, Python provides the array module and the numpy library, which are more optimized for performance.

**Example of a list functioning as an array:**

python

CopyEdit

# List used as an array

arr = [1, 2, 3, 4, 5]

# Accessing elements in the list

print(arr[0]) # Output: 1

print(arr[4]) # Output: 5

# Modifying elements

arr[2] = 10

print(arr) # Output: [1, 2, 10, 4, 5]

**2. a) Write a Python program to perform linear search.**

**Question:** Write a Python program to perform linear search on a list.

python

CopyEdit

# Program to perform Linear Search

def linear\_search(arr, target):

for i in range(len(arr)):

if arr[i] == target:

return i # Return index if found

return -1 # Return -1 if target is not found

# Sample list and target

arr = [10, 20, 30, 40, 50]target = 30

# Perform linear search

result = linear\_search(arr, target)

if result != -1:

print(f"Element {target} found at index {result}")

else:

print(f"Element {target} not found.")

**Explanation:**

* The program defines a function linear\_search that iterates through the list arr. If the target element is found, it returns the index of the element; otherwise, it returns -1.
* The program demonstrates how to use the linear\_search function on a sample list.

**2. b) Write a Python program to perform binary search.**

**Question:** Write a Python program to perform binary search on a sorted list.

python

CopyEdit

# Program to perform Binary Search

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

while low <= high:

mid = (low + high) // 2

if arr[mid] == target:

return mid # Return index if found

elif arr[mid] < target:

low = mid + 1 # Search in the right half

else:

high = mid - 1 # Search in the left half

return -1 # Return -1 if target is not found

# Sample sorted list and target

arr = [10, 20, 30, 40, 50]

target = 40

# Perform binary search

result = binary\_search(arr, target)

if result != -1:

print(f"Element {target} found at index {result}")

else:

print(f"Element {target} not found.")

**Explanation:**

* Binary search works only on sorted lists. It repeatedly divides the search interval in half, reducing the number of elements to search through.
* The program defines a function binary\_search that performs this iterative approach to search for the target in the list arr.

**3. a) Appraise with an example nested if and elif header in Python (6).**

**Question:** Explain with an example the use of nested if and elif statements in Python.

python

CopyEdit

# Program to demonstrate nested if and elif statements

age = int(input("Enter your age: "))

if age >= 18:

print("You are eligible to vote.")

if age >= 60:

print("You are eligible for senior citizen benefits.")

else:

print("You are not eligible for senior citizen benefits.")

else:

print("You are not eligible to vote.")

if age >= 16:

print("You can get a driving license.")

else:

print("You cannot get a driving license.")

**Explanation:**

* The program uses a nested if statement where inside the first if block, there is another if that checks whether the user is eligible for senior citizen benefits based on their age.
* The use of elif is not shown directly here, but could be added if there are more conditions to check.

**3. b) Explain with an example while loop, break statement and continue statement in Python.**

**Question:** Explain the use of while loop, break statement, and continue statement in Python.

python

CopyEdit

# Program demonstrating while loop, break and continue statements

count = 0

# While loop example

while count < 10:

count += 1

# Skip even numbers using continue

if count % 2 == 0:

continue

# Print odd numbers

print(count)

# Exit loop after printing 7 using break

if count == 7:

break

**Explanation:**

* The while loop runs as long as the condition count < 10 is true.
* Inside the loop, if count is even, the continue statement is used to skip that iteration, i.e., it does not print even numbers.
* If the value of count reaches 7, the break statement is used to exit the loop prematurely.

**UNIT IV LISTS, TUPLES, DICTIONARIES**

**1. Appraise the operations for dynamically manipulating dictionaries (12)**

**Question:** Explain how to dynamically manipulate dictionaries in Python, including adding, updating, and removing elements.

**Explanation:**

A **dictionary** in Python is an unordered collection of key-value pairs. It is mutable, meaning that we can dynamically modify its contents during runtime. The operations that can be performed on dictionaries include adding, updating, and deleting elements.

Here are some key operations:

1. **Adding elements**:
   * We can add a new key-value pair to the dictionary using the assignment operator (=).
2. **Updating elements**:
   * To update an existing element, we simply assign a new value to an existing key.
3. **Removing elements**:
   * We can remove a key-value pair using the del statement or the pop() method.
4. **Using dictionary methods**:
   * get(key): Retrieves the value for the given key (returns None if the key doesn’t exist).
   * pop(key): Removes and returns the value of the specified key.
   * clear(): Removes all elements from the dictionary.
   * keys(): Returns a list of all keys in the dictionary.
   * values(): Returns a list of all values in the dictionary.
   * items(): Returns a list of all key-value pairs in the dictionary.

**Example:**

python

CopyEdit

# Initialize an empty dictionary

student\_info = {}

# Adding elements to the dictionary

student\_info["name"] = "John"

student\_info["age"] = 21

student\_info["department"] = "Computer Science"

# Updating elements

student\_info["age"] = 22 # Update age

# Removing elements

del student\_info["department"] # Remove key-value pair by key

# Using dictionary methods

print("Keys:", student\_info.keys()) # Output: dict\_keys(['name', 'age'])

print("Values:", student\_info.values()) # Output: dict\_values(['John', 22])

print("Items:", student\_info.items()) # Output: dict\_items([('name', 'John'), ('age', 22)])

# Using pop() method

removed\_value = student\_info.pop("name") # Remove and return value of the "name" key

print("Removed value:", removed\_value) # Output: Removed value: John

# Clear all elements

student\_info.clear() # Clears the dictionary

print("After clearing:", student\_info) # Output: {}

**Explanation:**

* In the above code, we demonstrate how to add, update, and delete items in a dictionary.
* We also showcase some common methods used to retrieve keys, values, and items from the dictionary, and how to clear the entire dictionary.

**2. Write a Python program to perform linear search on a list (8)**

**Question:** Write a Python program that performs a linear search on a list and returns the index of the target element if found.

python

CopyEdit

# Program to perform Linear Search

def linear\_search(arr, target):

for i in range(len(arr)):

if arr[i] == target:

return i # Return index if found

return -1 # Return -1 if target is not found

# Sample list and target

arr = [10, 20, 30, 40, 50]

target = 30

# Perform linear search

result = linear\_search(arr, target)

if result != -1:

print(f"Element {target} found at index {result}")

else:

print(f"Element {target} not found.")

**Explanation:**

* The function linear\_search iterates through each element in the list and checks if the current element matches the target.
* If the target is found, it returns the index of the element; otherwise, it returns -1.

**3. Demonstrate with code the various operations that can be performed on tuples (May 2019)**

**Question:** Explain and demonstrate various operations that can be performed on tuples.

**Explanation:**

A **tuple** in Python is an immutable sequence of values. Tuples are similar to lists, but they cannot be modified after they are created. Common operations that can be performed on tuples include:

1. **Accessing elements**: Using indices to access individual elements.
2. **Concatenation**: Combining two or more tuples.
3. **Repetition**: Repeating a tuple multiple times.
4. **Slicing**: Extracting a portion of a tuple.
5. **Iteration**: Iterating over the elements of a tuple.
6. **Counting occurrences**: Counting how many times a value appears in the tuple.
7. **Finding index**: Finding the index of a value in the tuple.

**Example:**

python

CopyEdit

# Creating a tuple

fruits = ("apple", "banana", "cherry", "apple", "grapes")

# Accessing elements

print(f"First element: {fruits[0]}") # Output: apple

print(f"Last element: {fruits[-1]}") # Output: grapes

# Concatenating tuples

new\_fruits = fruits + ("orange", "mango")

print("Concatenated tuple:", new\_fruits) # Output: ('apple', 'banana', 'cherry', 'apple', 'grapes', 'orange', 'mango')

# Repeating a tuple

repeated\_fruits = fruits \* 2

print("Repeated tuple:", repeated\_fruits) # Output: ('apple', 'banana', 'cherry', 'apple', 'grapes', 'apple', 'banana', 'cherry', 'apple', 'grapes')

# Slicing a tuple

sliced\_fruits = fruits[1:4]

print("Sliced tuple:", sliced\_fruits) # Output: ('banana', 'cherry', 'apple')

# Iterating over a tuple

print("Iterating over fruits:")

for fruit in fruits:

print(fruit)

# Counting occurrences of an element

apple\_count = fruits.count("apple")

print(f"Number of 'apple' in the tuple: {apple\_count}") # Output: 2

# Finding index of an element

apple\_index = fruits.index("apple")

print(f"Index of 'apple': {apple\_index}") # Output: 0

**Explanation:**

* We demonstrate various operations on tuples, including accessing elements, concatenating tuples, repeating elements, and slicing.
* We also show how to iterate over a tuple, count the occurrences of an element, and find the index of an element.

**UNIT V-FILES, MODULES, PACKAGES**

**1. (a) Describe how exceptions are handled in Python with necessary examples. (8)**

**Exceptions in Python:**

In Python, **exceptions** are errors that occur during the execution of the program. These errors disrupt the normal flow of the program and are caught using **exception handling** techniques. Python provides a mechanism to catch and handle exceptions using the try, except, else, and finally blocks.

**Structure of Exception Handling:**

1. **try block**: Contains the code that might raise an exception.
2. **except block**: Contains the code to handle the exception if one occurs.
3. **else block**: Executes if no exceptions are raised in the try block.
4. **finally block**: Executes no matter what, whether an exception was raised or not.

**Example:**

python

CopyEdit

def divide\_numbers(a, b):

try:

result = a / b

except ZeroDivisionError as e:

print("Error: Division by zero is not allowed.")

return None

except Exception as e:

print(f"An error occurred: {e}")

return None

else:

print(f"Result: {result}")

finally:

print("Execution completed.")

# Test the function with valid inputs

divide\_numbers(10, 2)

# Test the function with a division by zero

divide\_numbers(10, 0)

**Explanation:**

* In this example, the try block attempts to divide two numbers. If a division by zero occurs, it raises a ZeroDivisionError, which is caught by the first except block.
* If any other exception occurs, it is caught by the general except block.
* The else block runs only if no exception occurs.
* The finally block always runs, even if an exception occurs.

**1. (b) Discuss about the use of format operator in file processing. (8)**

**File Processing using Format Operator in Python:**

The **format operator** (%) is used in Python for formatting strings. It allows us to embed variables into strings with specified formatting. When used in file processing, it can help in inserting values into files dynamically, making file contents more readable and formatted.

**Example of using the format operator in file writing:**

python

CopyEdit

name = "Alice"

age = 25

occupation = "Engineer"

# Open file in write mode

with open("user\_info.txt", "w") as file:

file.write("Name: %s\nAge: %d\nOccupation: %s\n" % (name, age, occupation))

# Open the file and read the content

with open("user\_info.txt", "r") as file:

content = file.read()

print(content)

**Explanation:**

* The %s is used for inserting a string, %d is used for inserting an integer into the file. These placeholders are replaced with the values provided in the tuple (name, age, occupation).
* The file is opened in write mode, and the formatted string is written into the file.
* Then, we read and print the content of the file.

**2. (a) What are exceptions? Explain the method to handle them with an example. (8)**

**Exceptions in Python:**

An **exception** is an event that disrupts the normal flow of a program's execution. Examples of exceptions include division by zero, accessing a non-existent file, or trying to use a variable that has not been initialized.

Python provides a robust mechanism to handle exceptions using try, except, else, and finally blocks.

**Example:**

python

CopyEdit

def safe\_divide(a, b):

try:

result = a / b

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

return None

except TypeError:

print("Error: Both arguments must be numbers.")

return None

else:

print(f"Division result: {result}")

finally:

print("Execution completed.")

# Test the function

safe\_divide(10, 2) # No error

safe\_divide(10, 0) # Division by zero error

safe\_divide(10, "a") # TypeError

**Explanation:**

* The try block contains code that may raise an exception.
* The except block catches the specific exceptions (like ZeroDivisionError or TypeError) and handles them.
* The else block executes if no exception occurs.
* The finally block ensures that some code (like cleanup) runs after the try-except, regardless of success or failure.

**2. (b) Write a Python program to count the number of words in a text file. (8)**

**Python Program to Count Words in a File:**

python

CopyEdit

def count\_words\_in\_file(filename):

try:

with open(filename, "r") as file:

content = file.read()

words = content.split() # Split the content into words

word\_count = len(words) # Count the number of words

print(f"Number of words in the file: {word\_count}")

except FileNotFoundError:

print("The file does not exist.")

except Exception as e:

print(f"An error occurred: {e}")

# Test the function

count\_words\_in\_file("sample.txt")

**Explanation:**

* The count\_words\_in\_file function opens a file in read mode and splits the content into words using the split() method.
* It counts the number of words by determining the length of the list words.
* The try-except block handles potential errors, such as the file not being found.

**3. (a) How to merge multiple files to a new file using Python (6)**

**Merging Multiple Files to a New File in Python:**

python

CopyEdit

def merge\_files(file\_list, output\_filename):

try:

with open(output\_filename, "w") as output\_file:

for filename in file\_list:

with open(filename, "r") as input\_file:

content = input\_file.read()

output\_file.write(content + "\n") # Add newline after each file content

print(f"Files merged into {output\_filename}")

except FileNotFoundError:

print("One or more files not found.")

except Exception as e:

print(f"An error occurred: {e}")

# List of files to merge

file\_list = ["file1.txt", "file2.txt", "file3.txt"]

merge\_files(file\_list, "merged\_output.txt")

**Explanation:**

* The function merge\_files reads the content of each file in the file\_list and writes it into the new output\_filename.
* Each file's content is read using the read() method, and the content is written to the output file.

**3. (b) What are modules in Python? How will you import them? Explain the concept by creating and importing a module (10)**

**Modules in Python:**

A **module** in Python is a file containing Python definitions, functions, variables, and runnable code. Modules allow you to organize your code logically and reuse it in different programs.

**Creating a Module:**

Create a Python file (let's say math\_operations.py) that defines some functions:

python

CopyEdit

# math\_operations.py

def add(a, b):

return a + b

def subtract(a, b):

return a - b

**Importing and Using the Module:**

python

CopyEdit

# main\_program.py

# Import the math\_operations module

import math\_operations

result\_add = math\_operations.add(5, 3)

result\_sub = math\_operations.subtract(10, 4)

print(f"Addition Result: {result\_add}")

print(f"Subtraction Result: {result\_sub}")

**Explanation:**

* In math\_operations.py, we define two functions: add() and subtract().
* In main\_program.py, we import the math\_operations module and use its functions by calling math\_operations.add() and math\_operations.subtract().

**Import Methods:**

* import module\_name: Imports the entire module.
* from module\_name import function\_name: Imports specific functions from the module.
* import module\_name as alias: Imports the module and assigns it an alias.