**AI LAB 3:**

**HOMETASK:**

**PACKAGE1---MODULE1**

def functionAdd(x, y):  
 return x + y  
def subtract(x, y):  
 return x - y

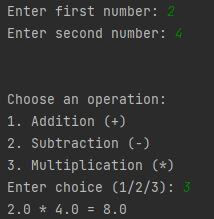
**PACKAGE2---MODULE2**

def multiply(x, y):  
 return x \* y

**MAIN.PY:**

from Package1 import Module1;  
from Package1.Module1 import functionAdd, subtract  
from Package2 import Module2;  
from Package2.Module2 import multiply  
  
num1 = float(input("Enter first number: "))  
num2 = float(input("Enter second number: "))  
print("\n")  
print("Choose an operation:")  
print("1. Addition (+)")  
print("2. Subtraction (-)")  
print("3. Multiplication (\*)")  
choice = input("Enter choice (1/2/3): ")  
  
if choice == '1':  
 print(num1, "+", num2, "=", functionAdd(num1, num2))  
elif choice == '2':  
 print(num1, "-", num2, "=", subtract(num1, num2))  
elif choice == '3':  
 print(num1, "\*", num2, "=", multiply(num1, num2))  
else:  
 print("Invalid input")  
  
Module1.functionAdd()  
Module1.subtract()  
Module2.multiply()

**OUTPUT:**

****

**LAB TASK:**

class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def display(self):  
 print(f"Name: {self.name}, Age: {self.age}")  
  
  
class Student(Person):  
 def \_\_init\_\_(self, name, age, student\_id):  
 super().\_\_init\_\_(name, age)  
 self.student\_id = student\_id  
  
 def display(self):  
 super().display()  
 print(f"Student ID: {self.student\_id}")  
  
person = Person("Alice", 25)  
student = Student("Bob", 20, 12345)  
person.display()  
student.display()

**OUTPUT:**

**C:\Users\hp\AppData\Local\Programs\Python\Python39\python.exe C:\Users\hp\PycharmProjects\pythonProject1\inheritance\file2.py**

**Name: Alice, Age: 25**

**Name: Bob, Age: 20**

**Student ID: 12345**

**LAB TASK 2:**

**SIMPLE GRAPHS:**

class graph:  
 def \_\_init\_\_(self,gdict=None):  
 if gdict is None:  
 gdict = []  
 self.gdict = gdict  
  
 def getVertices(self):  
 return list(self.gdict.keys())  
  
 def edges(self):  
 return self.findedges()  
  
 def AddEdge(self, edge):  
 edge = set(edge)  
 (vrtx1, vrtx2) = tuple(edge)  
 if vrtx1 in self.gdict:  
 self.gdict[vrtx1].append(vrtx2)  
 else:  
 self.gdict[vrtx1] = [vrtx2]  
 def findedges(self):  
 edgename = []  
 for vrtx in self.gdict:  
 for nxtvrtx in self.gdict[vrtx]:  
 if {nxtvrtx, vrtx} not in edgename:  
 edgename.append({vrtx, nxtvrtx})  
 return edgename  
  
graph\_elements = {  
 "a" : ["b","c"],  
 "b" : ["a", "d"],  
 "c" : ["a", "d"],  
 "d" : ["e"],  
 "e" : ["d"]  
}  
g = graph(graph\_elements)  
print(g.getVertices())  
g.AddEdge({'a','e'})  
g.AddEdge({'a','c'})  
print(g.edges())

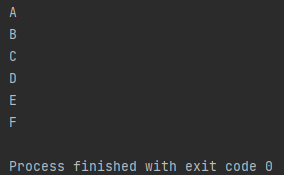
**OUTPUT:**

****

**SIMPLE BFS:**

from collections import deque  
  
  
def bfs(graph, start):  
 visited = set() # keep track of visited nodes  
 queue = deque([start]) # initialize the queue with the start node  
 visited.add(start) # mark start node as visited  
  
 while queue: # while there are nodes in the queue  
 node = queue.popleft() # remove the first node from the queue  
 print(node) # print the node (or process it in some other way)  
 for neighbor in graph[node]: # iterate over the neighbors of the node  
 if neighbor not in visited: # if neighbor has not been visited  
 visited.add(neighbor) # mark it as visited  
 queue.append(neighbor) # add it to the queue  
  
  
# Example usage  
graph = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []  
}  
  
bfs(graph, 'A')

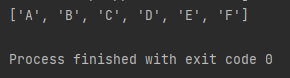
**OUTPUT:**

****

**TRAVERSAL BFS:**

from collections import deque  
  
# define graph as a dictionary  
graph = {'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []}  
  
def bfs(graph, start):  
 visited = [] # list to keep track of visited nodes  
 queue = deque([start]) # queue for BFS  
  
 while queue:  
 node = queue.popleft()  
 if node not in visited:  
 visited.append(node)  
 neighbors = graph[node]  
 for neighbor in neighbors:  
 queue.append(neighbor)  
 return visited  
  
# example usage  
print(bfs(graph, 'A'))

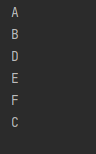
**OUTPUT:**

****

**DFS:**

def dfs(graph, start, visited=None):  
 if visited is None:  
 visited = set() # keep track of visited nodes  
 visited.add(start) # mark the start node as visited  
 print(start) # print the node (or process it in some other way)  
 for neighbor in graph[start]: # iterate over the neighbors of the node  
 if neighbor not in visited: # if neighbor has not been visited  
 dfs(graph, neighbor, visited) # recursively call dfs on the neighbor  
  
  
# Example usage  
graph = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []  
}  
  
dfs(graph, 'A')

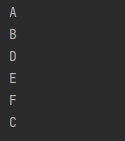
**OUTPUT:**

****

**DFS TRAVERSAL**

def dfs\_traversal(graph, start):  
 visited = set() # keep track of visited nodes  
 stack = [start] # initialize the stack with the start node  
  
 while stack: # while there are nodes in the stack  
 node = stack.pop() # remove the last node from the stack  
 if node not in visited:  
 visited.add(node) # mark the node as visited  
 print(node) # print the node (or process it in some other way)  
 neighbors = graph[node] # get the neighbors of the node  
 stack.extend(reversed(neighbors)) # add the neighbors to the stack in reverse order  
  
  
# Example usage  
graph = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []  
}  
  
dfs\_traversal(graph, 'A')

**OUTPUT:**

****