**UNIVERSITY OF CALCUTTA**

B.Sc Semester 5 Honours Examination-2024

Under CBCS System

University Registration Number: 012-1114-1749-21

University Roll Number: 213012-21-0396

Subject: Computer Science

Subject Code: CMSA

Paper: Python Lab

Paper Code:CC-12

Semester: 5

Year: 2024

Index

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sl No. | Date | Assignment | Page No. | Teacher’s Signature |
| 01 | 20.09.23 | Assignment 1 | 01 |  |
| 02 | 22.09.23 | Assignment 2 | 02 |  |
| 03 | 29.09.23 | Assignment 3 | 03 |  |
| 04 | 04.10.23 | Assignment 4 | 04 |  |
| 05 | 11.10.23 | Assignment 5 | 05 |  |
| 06 | 13.10.23 | Assignment 6 | 10 |  |
| 07 | 22.11.23 | Assignment 7 | 12 |  |
| 08 | 01.12.23 | Assignment 8 | 13 |  |
| 09 | 06.12.23 | Assignment 9 | 15 |  |
| 10 | 13.12.23 | Assignment 10 | 17 |  |
| 11 | 20.12.23 | Assignment 11 | 18 |  |
| 12 | 20.12.23 | Assignment 12 | 23 |  |

Assignment 1

Exploring the complex data type and their operation, eg: finding the modulus and phase angle of a complex number and print values.

Code:

import cmath

def main():

print("Enter the real and imaginary parts of a complex number: ")

real = float(input("Real part: "))

imag = float(input("Imaginary part: "))

complex\_num = complex(real, imag)

print(f"The modulus of the complex number is: {abs(complex\_num):.4f} units")

print(f"The phase angle of the complex number is: {cmath.phase(complex\_num):.4f} rad")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

Enter the real and imaginary parts of a complex number:

Real part: 3

Imaginary part: 4

The modulus of the complex number is: 5.0000 units

The phase angle of the complex number is: 0.9273 rad

Assignment 2

Ask the user to enter two numbers, and output the sum, product, difference, and the GCD.

Code:

def main():

num1 = int(input("Enter the first number: "))

num2 = int(input("Enter the second number: "))

print(f"The sum of the numbers is: {num1+num2}")

print(f"The product of the numbers is: {num1\*num2}")

print(f"The difference of the numbers is: {num1-num2}")

print(f"The GCD of the numbers is: {gcd(num1,num2)}")

def gcd(num1,num2):

#non-recursive

while num1 != 0:

num1,num2 = num2%num1,num1

return num2

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

Enter the first number: 25

Enter the second number: 65

The sum of the numbers is: 90

The product of the numbers is: 1625

The difference of the numbers is: -40

The GCD of the numbers is: 5

Assignment 3

Ask the user for two strings, print a new string where the first string is reversed, and the second string is converted to upper case. Sample strings: “Pets“, “party”, output: “steP PARTY”. Only use string slicing and + operators.

Code:

def main():

string1 = input("Enter the first string: ")

string2 = input("Enter the second string: ")

result\_string = string1[::-1] + " " + string2.upper()

print(result\_string)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

Output-1

Enter the first string: Pets

Enter the second string: party

steP PARTY

Output-2

Enter the first string: Hello

Enter the second string: World

olleH WORLD

Assignment 4

From a list of words, join all the words in the odd and even indices to form two strings. Use list slicing and join methods. .

Code:

def main():

words = input("Enter the words: ").split()

even\_words = words[::2]

odd\_words = words[1::2]

even\_string = " ".join(even\_words)

odd\_string = " ".join(odd\_words)

print("Even string:", even\_string)

print("Odd string:", odd\_string)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

Output-1

Enter the words: Hello World

Even string: Hello

Odd string: World

Output-2

Enter the words: Python is a programming language

Even string: Python a language

Odd string: is programming

Assignment 5

Simulate a stack and a queue using lists. Note that the queue deletion operation won`t run in O(1) time.

Code:

def main():

stack = []

queue = []

while True:

print("--------------------------")

print("1. Push to stack")

print("2. Pop from stack")

print("3. Push to queue")

print("4. Pop from queue")

print("5. Exit")

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

if choice == 1:

element = input("Enter the element: ")

stack.append(element)

elif choice == 2:

if len(stack) == 0:

print("Stack is empty!")

else:

print("Popped element:", stack.pop())

elif choice == 3:

element = input("Enter the element: ")

queue.append(element)

elif choice == 4:

if len(queue) == 0:

print("Queue is empty!")

else:

print("Popped element:", queue.pop(0))

elif choice == 5:

break

else:

print("Invalid choice!")

print("Current stack:", stack)

print("Current queue:", queue)

print("--------------------------")

print("Final stack:", stack)

print("Final queue:", queue)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 1

Enter the element: 2

Current stack: ['2']

Current queue: []

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 1

Enter the element: 4

Current stack: ['2', '4']

Current queue: []

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 1

Enter the element: 8

Current stack: ['2', '4', '8']

Current queue: []

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 3

Enter the element: 1

Current stack: ['2', '4', '8']

Current queue: ['1']

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 3

Enter the element: 3

Current stack: ['2', '4', '8']

Current queue: ['1', '3']

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 3

Enter the element: 5

Current stack: ['2', '4', '8']

Current queue: ['1', '3', '5']

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 2

Popped element: 8

Current stack: ['2', '4']

Current queue: ['1', '3', '5']

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 4

Popped element: 1

Current stack: ['2', '4']

Current queue: ['3', '5']

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 2

Popped element: 4

Current stack: ['2']

Current queue: ['3', '5']

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 4

Popped element: 3

Current stack: ['2']

Current queue: ['5']

--------------------------

1. Push to stack

2. Pop from stack

3. Push to queue

4. Pop from queue

5. Exit

Enter your choice: 5

Final stack: ['2']

Final queue: ['5']

--------------------------

Assignment 6

Explore the ‘re’ module, especially re.split, re.join, re.search and re.match methods. .

Code:

import re

def main():

text = "Hello, World! How are you?"

result = re.split(r'[^\w]+', text) # Split based on non-word characters

print(result)

# Output: ['Hello', 'World', 'How', 'are', 'you']

words = ['Hello', 'World', 'How', 'are', 'you']

result = ' '.join(words)

print(result)

# Output: Hello World How are you

text = "The quick brown fox jumps over the lazy dog"

pattern = r'fox'

searchres = re.search(pattern, text)

if searchres:

print("Pattern found:", searchres.group())

else:

print("Pattern not found")

# Output: Pattern found: fox

text = "The quick brown fox jumps over the lazy dog"

pattern = r'fox'

matchres = re.match(pattern, text)

if matchres:

print("Pattern found:", matchres.group())

else:

print("Pattern not found")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

['Hello', 'World', 'How', 'are', 'you']

Hello World How are you

Pattern found: fox

Pattern not found

['my', 'name', 'is', 'sayantan', 'ghosh']

my name is sayantan ghosh

Pattern found: fox

Pattern not found

Assignment 7

Use list comprehension to find all the odd numbers and numbers divisible by 3 from a list of numbers.

Code:

def main():

numbers = [x for x in range(1, 20)]# List comprehension

print("Original list:",numbers)

odd\_numbers = [x for x in numbers if x % 2 != 0]# List comprehension with condition

print("Odd Numbers:",odd\_numbers)

divisible\_by\_3 = [x for x in numbers if x % 3 == 0]# List comprehension with condition

print("Numbers that are divisible by 3:",divisible\_by\_3)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

Original list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Odd Numbers: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

Numbers that are divisible by 3: [3, 6, 9, 12, 15, 18]

Assignment 8

Implement popular sorting algorithms like quick sort and merge sort to sort lists of numbers. .

Code:

# Quick sort

def quickSort(arr):

if len(arr) <= 1:

return arr

else:

pivot = arr[0]

less\_than\_pivot = [x for x in arr[1:] if x <= pivot]

greater\_than\_pivot = [x for x in arr[1:] if x > pivot]

return quickSort(less\_than\_pivot) + [pivot] + quickSort(greater\_than\_pivot)

# Merge sort

def mergeSort(arr):

if len(arr) <= 1:

return arr

middle = len(arr) // 2

left\_half = arr[:middle]

right\_half = arr[middle:]

left\_half = mergeSort(left\_half)

right\_half = mergeSort(right\_half)

return merge(left\_half, right\_half)

# Merge function for merge sort algorithm

def merge(left, right):

result = []

i = j = 0

while i < len(left) and j < len(right):

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

result.extend(left[i:])

result.extend(right[j:])

return result

def main():

arr1 = [56,98,12,7,9,23,1,25,45,60]

n = len(arr1)

print("Unsorted array :",arr1)

q\_sort=quickSort(arr1)

print("Sorted array using quicksort:",q\_sort)

print("---------------------------------------------------------------")

arr2 = [56,98,12,7,9,23,1,25,45,60]

print("Unsorted array :",arr2)

m\_sort=mergeSort(arr2)

print("Sorted array using mergesort:",m\_sort)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Assignment 9

Write two functions that simulate the toss of a fair coin, and the roll of an unbiased ‘n’ sided die using the random module.

Code:

import random

def tossCoin():

return random.choice(["Heads","Tails"])

def rollDie(n):

return random.randint(1,n)

def main():

while True:

print("1. Toss a coin")

print("2. Roll a die")

print("3. Exit")

choice = input("Enter your choice: ")

if choice == "1":

print("Tossing a coin...")

print("Result:",tossCoin())

elif choice == "2":

n = int(input("Enter the number of sides of the die: "))

print(f"Rolling a {n} sided dice...")

print("Result:",rollDie(n))

elif choice == "3":

break

else:

print("Invalid choice!")

print("---------------------------------------------------------------")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

1. Toss a coin

2. Roll a die

3. Exit

Enter your choice: 1

Tossing a coin...

Result: Tails

---------------------------------------------------------------

1. Toss a coin

2. Roll a die

3. Exit

Enter your choice: 2

Enter the number of sides of the die: 6

Rolling a 6 sided dice...

Result: 1

---------------------------------------------------------------

1. Toss a coin

2. Roll a die

3. Exit

Enter your choice: 2

Enter the number of sides of the die: 12

Rolling a 12 sided dice...

Result: 2

---------------------------------------------------------------

1. Toss a coin

2. Roll a die

3. Exit

Enter your choice: 3

Assignment 10

Invert a dictionary such the previous keys become values and values keys.

Eg: if the initial and inverted dictionaries are d1 and d2, where d1 = {1: ‘a’, 2: ‘b’, 3: 120}, then d2 = {‘a’: 1, 2: ‘b’, 120: 3}.

Code:

def main():

d1 = {1: 'a', 2: 'b', 3: 120}

print("Original dictionary:",d1)

d2 = {v:k for k,v in d1.items()}# Dictionary comprehension

#not using dict comprehension

d3={}

for k,v in d1.items():

d3[v]=k

print("Inverted dictionary using Dict comprehension:",d2)

print("Inverted dictionary without using Dict comprehension:",d3)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

Original dictionary: {1: 'a', 2: 'b', 3: 120}

Inverted dictionary using Dict comprehension: {'a': 1, 'b': 2, 120: 3}

Inverted dictionary without using Dict comprehension: {'a': 1, 'b': 2, 120: 3}

Assignment 11

Create a ‘Graph’ class to store and manipulate graphs. It should have the following functions:

i. Read an edge list file, where each edge (u, v) appears exactly once in the file as space separated values.

ii. Add and remove nodes and edges

iii. Print nodes, and edges in a user readable format

iv. Finding all the neighbors of a node

v. Finding all the connected components and storing them as individual Graph objects inside the class

vi. Finding single source shortest paths using Breadth First Search

Code:

from collections import defaultdict, deque

class Graph:

def \_\_init\_\_(self):

self.graph = defaultdict(list)

def read\_edge\_list(self, file\_path):

with open(file\_path, 'r') as file:

for line in file:

u, v = map(int,line.split())

self.add\_edge(u, v)

def add\_edge(self, u, v):

self.graph[u].append(v)

self.graph[v].append(u)

def remove\_edge(self, u, v):

self.graph[u].remove(v)

self.graph[v].remove(u)

def add\_node(self, node):

if node not in self.graph:

self.graph[node] = []

def remove\_node(self, node):

del self.graph[node]

for key, value in self.graph.items():

if node in value:

value.remove(node)

def print\_nodes(self):

print("Nodes:", list(self.graph.keys()))

def print\_edges(self):

print("Edges:")

for node, neighbors in self.graph.items():

for neighbor in neighbors:

print(f"({node}, {neighbor})",end=" ")

def find\_neighbors(self, node):

return self.graph[node]

def find\_connected\_components(self):

visited = set()

components = []

for node in self.graph:

if node not in visited:

component = self.bfs(node, visited)

components.append(component)

return components

def bfs(self, start, visited):

component = []

queue = deque([start])

visited.add(start)

while queue:

current\_node = queue.popleft()

component.append(current\_node)

for neighbor in self.graph[current\_node]:

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

return component

def bfs\_shortest\_paths(self, start):

visited = set()

distance = {node: float('inf') for node in self.graph}

distance[start] = 0

queue = deque([start])

while queue:

current\_node = queue.popleft()

for neighbor in self.graph[current\_node]:

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

distance[neighbor] = distance[current\_node] + 1

return distance

def main():

# Create a Graph object

graph = Graph()

# Read edge list from a file

graph.read\_edge\_list("ass11\_edge\_list.txt")

# Print initial nodes and edges

print("Initial Graph:")

graph.print\_nodes()

graph.print\_edges()

# Add a new node and edge

graph.add\_node(6)

graph.add\_edge(6, 5)

graph.add\_edge(6, 2)

# Print nodes and edges after modification

print("\nGraph after Modification:")

graph.print\_nodes()

graph.print\_edges()

# Find neighbors of a node

print("\nNeighbors of Node 6:", graph.find\_neighbors(6))

print("Neighbors of Node 3:", graph.find\_neighbors(3))

# Find connected components

components = graph.find\_connected\_components()

print("\nConnected Components:", components)

# Find shortest paths from a source node

shortest\_paths = graph.bfs\_shortest\_paths(1)

print("\nShortest Paths from Node 1:", shortest\_paths)

if \_\_name\_\_ == "\_\_main\_\_":

main()

[Content in ass11\_edge\_list.txt]

1 2

2 4

3 4

1 3

4 5

Output:

Initial Graph:

Nodes: [1, 2, 4, 3, 5]

Edges:

(1, 2) (1, 3) (2, 1) (2, 4) (4, 2) (4, 3) (4, 5) (3, 4) (3, 1) (5, 4)

Graph after Modification:

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

Edges:

(1, 2) (1, 3) (2, 1) (2, 4) (2, 6) (4, 2) (4, 3) (4, 5) (3, 4) (3, 1) (5, 4) (5, 6) (6, 5) (6, 2)

Neighbors of Node 6: [5, 2]

Neighbors of Node 3: [4, 1]

Connected Components: [[1, 2, 3, 4, 6, 5]]

Shortest Paths from Node 1: {1: 2, 2: 1, 4: 2, 3: 1, 5: 3, 6: 2}

Assignment 12

Make a ‘DiGraph’ class to handle directed graphs which inherits from the ‘Graph’ class. In addition to all of the functionalities of (a), it should support

the following operations

i. Finding the predecessors and successors of a node

ii. Creating a new ‘DiGraph’ object where all the edges are reversed

iii. Finding the strongly connected components

Code:

from ass11 import \*;

class DiGraph(Graph):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def predecessors(self, node):

return [key for key, value in self.graph.items() if node in value]

def successors(self, node):

return self.graph[node]

def reverse\_edges(self):

reversed\_graph = DiGraph()

for node, neighbors in self.graph.items():

for neighbor in neighbors:

reversed\_graph.add\_edge(neighbor, node)

return reversed\_graph

def strongly\_connected\_components(self):

components = []

visited = set()

for node in self.graph:

if node not in visited:

component = self.dfs(node, visited)

components.append(component)

return components

def dfs(self, node, visited):

component = []

stack = [node]

while stack:

current\_node = stack.pop()

if current\_node not in visited:

component.append(current\_node)

visited.add(current\_node)

for neighbor in self.graph[current\_node]:

stack.append(neighbor)

return component

def main():

# Create a DiGraph object

digraph = DiGraph()

# Read edge list from a file

digraph.read\_edge\_list("ass11\_edge\_list.txt")

# Print initial nodes and edges

print("Initial DiGraph:")

digraph.print\_nodes()

digraph.print\_edges()

# Add a new node and edge

digraph.add\_node(7)

digraph.add\_edge(6, 7)

# Print nodes and edges after modification

print("\nDiGraph after Modification (adding node 7 and connecting it to 6):")

digraph.print\_nodes()

digraph.print\_edges()

# Find predecessors and successors of a node

node\_to\_check = 6

print(f"\nPredecessors of Node {node\_to\_check}: {digraph.predecessors(node\_to\_check)}")

print(f"Successors of Node {node\_to\_check}: {digraph.successors(node\_to\_check)}")

# Reverse the edges and visualize the reversed graph

reversed\_digraph = digraph.reverse\_edges()

print("\nReversed DiGraph:")

reversed\_digraph.print\_edges()

# Find strongly connected components

components = digraph.strongly\_connected\_components()

print("\nStrongly Connected Components:", components)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:

Initial DiGraph:

Nodes: [1, 2, 4, 3, 5]

Edges:

(1, 2) (1, 3) (2, 1) (2, 4) (4, 2) (4, 3) (4, 5) (3, 4) (3, 1) (5, 4)

DiGraph after Modification (adding node 7 and connecting it to 6):

Nodes: [1, 2, 4, 3, 5, 7, 6]

Edges:

(1, 2) (1, 3) (2, 1) (2, 4) (4, 2) (4, 3) (4, 5) (3, 4) (3, 1) (5, 4) (7, 6) (6, 7)

Predecessors of Node 6: [7]

Successors of Node 6: [7]

Reversed DiGraph:

Edges:

(2, 1) (2, 1) (2, 4) (2, 4) (1, 2) (1, 3) (1, 2) (1, 3) (3, 1) (3, 4) (3, 4) (3, 1) (4, 2) (4, 2) (4, 3) (4, 5) (4, 3) (4, 5) (5, 4) (5, 4) (6, 7) (6, 7) (7, 6) (7, 6)

Strongly Connected Components: [[1, 3, 4, 5, 2], [7, 6]]