Python Control Structures

1. **If-Else Statement**: Write a program to check if a number is positive, negative, or zero.

Syntax:

num = float(input("Enter a number: "))  
if num > 0:  
 print("The number is positive.")  
elif num < 0:  
 print("The number is negative.")  
else:  
 print("The number is zero.")

output: Enter a number: 10

The number is positive.

Enter a number: -5

The number is negative.

Enter a number: 0

The number is zero

1. **For Loop**: Write a program to print the first 10 natural numbers using a for loop.

print("the first natural numbers=")  
for i in range(1,11):  
 print(i)

output: the first natural numbers=

1

2

3

4

5

6

7

8

9

10

1. **While Loop**: Write a program to print the Fibonacci sequence up to the nth term using a while loop.

**Syntax:**

n=6  
num1 = 0  
num2 = 1  
next\_number = num2  
count = 1  
print("Fibonacci sequence:", end=" ")  
while count <= n:  
 print(next\_number, end=" ")  
 count += 1  
 num1, num2 = num2, next\_number  
 next\_number = num1 + num2  
print()

output: Fibonacci sequence: 1 2 3 5 8 13

1. **Nested Loop**: Write a program to print a multiplication table from 1 to 10.

**Syntax:**

print("Tables:")  
for i in range(1, 11):  
 for j in range(1, 11):  
 print(i, '\*', j, '=', i \* j)

output: Tables:

1 \* 1 = 1

1 \* 2 = 2

1 \* 3 = 3

1 \* 4 = 4

1 \* 5 = 5

1 \* 6 = 6

1 \* 7 = 7

1 \* 8 = 8

1 \* 9 = 9

1 \* 10 = 10 etc up to 10 I got like this 10 \* 1 = 10

10 \* 2 = 20

10 \* 3 = 30

10 \* 4 = 40

10 \* 5 = 50

10 \* 6 = 60

10 \* 7 = 70

10 \* 8 = 80

10 \* 9 = 90

10 \* 10 = 100

1. **List Comprehension**: Write a program to generate a list of squares of numbers from 1 to 10 using list comprehension.

squares = [x \*\* 2 for x in range(1, 11)]  
print("List of squares of numbers from 1 to 10:")  
print(squares)

output:

List of squares of numbers from 1 to 10:

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

1. **Dictionary Comprehension**: Write a program to create a dictionary from two lists: one of keys and one of values.

**Syntax**:

keys = ["manu", "ram", "mouni"]  
values = [2, 6, 7]  
res = {}  
for key in keys:  
 for value in values:  
 res[key] = value  
 values.remove(value)  
 break  
print("Resultant dictionary is : ",res)

output: Resultant dictionary is : {'manu': 2, 'ram': 6, 'mouni': 7}

1. **Break and Continue**: Write a program to iterate over a list of numbers and print each number. If you encounter the number 5, break the loop. If you encounter the number 3, skip to the next iteration.

**Syntax**:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
print("current number:" )  
for num in numbers:  
 if num == 5:  
 print("Encountered 5. Breaking the loop.")  
 break  
 if num == 3:  
 print("Encountered 3. Skipping to the next iteration.")  
 continue  
 print(num)

output:

current number:

1

2

Encountered 3. Skipping to the next iteration.

4

Encountered 5. Breaking the loop.

1. **Functions**: Write a function to calculate the factorial of a number using recursion.

**Syntax:**

def factorial(n):  
 if n == 0:  
 return 1  
 else:  
 return n \* factorial(n - 1)  
num = 5  
print("number : ",num)  
print("Factorial : ",factorial(num))

output:

number : 5

Factorial : 120

1. **Lambda Function**: Write a program to sort a list of tuples based on the second element using a lambda function.

Syntax:

my\_list = [(1, 3), (5, 2), (7, 8), (6, 4)]  
sorted\_list = sorted(my\_list, key=lambda x: x[1])  
print("Sorted list based on the second element:")  
print(sorted\_list)

output:

Sorted list based on the second element:

[(5, 2), (1, 3), (6, 4), (7, 8)]

1. **Exception Handling**: Write a program to handle a division by zero exception.

**Syntax**:

numerator = 70  
denominator = 0  
if denominator != 0:  
 result = numerator / denominator  
else:  
 print("Error: Cannot divide by zero.")

output: Error: Cannot divide by zero.

### **Python Data Structures**

1. **Lists**: Create a list of the first 10 prime numbers. Write functions to add an element, remove an element, and find an element in the list.

**Syntax**:

def add\_prime(prime\_list, new\_prime):  
 prime\_list.append(new\_prime)  
  
def remove\_prime(prime\_list, prime\_to\_remove):  
 if prime\_to\_remove in prime\_list:  
 prime\_list.remove(prime\_to\_remove)  
  
def find\_prime(prime\_list, prime\_to\_find):  
 if prime\_to\_find in prime\_list:  
 return True  
 else:  
 return False  
  
prime\_numbers = [2, 3, 5, 7, 11, 17, 19, 23, 29, 31]  
  
print("Original list of prime numbers:", prime\_numbers)  
  
new\_prime\_number = 13  
add\_prime(prime\_numbers, new\_prime\_number)  
prime\_to\_remove = 5  
remove\_prime(prime\_numbers, prime\_to\_remove)  
print("Updated list of prime numbers after add and remove operations:", prime\_numbers)  
number\_to\_find = 17  
if find\_prime(prime\_numbers, number\_to\_find):  
 print(number\_to\_find, "is found in the list.")  
else:  
 print(number\_to\_find, "is not found in the list.")

output:

Original list of prime numbers: [2, 3, 5, 7, 11, 17, 19, 23, 29, 31]

Updated list of prime numbers after add and remove operations: [2, 3, 7, 11, 17, 19, 23, 29, 31, 13]

17 is found in the list.

1. **Tuples**: Create a tuple with 5 different elements. Write a program to access and print each element in the tuple.

**Syntax**:

tuple = ("apple", 10,1.00, "manu", [1, 2, 3])  
for i in tuple:  
 print(i)

output:

apple

10

1.0

manu

[1, 2, 3]

1. **Dictionaries**: Create a dictionary with keys as student names and values as their scores. Write a program to find the student with the highest score.

**Syntax**:

student\_scores = {  
 "reddy": 86,  
 "manu": 77,  
 "ram": 68,  
 "manohar": 91,  
 "mouni": 81  
}  
highest\_score\_student = max(student\_scores, key=student\_scores.get)  
highest\_score = student\_scores[highest\_score\_student]  
print("Student with the highest score name :", highest\_score\_student)  
print("Score:", highest\_score)

output:

Student with the highest score name : manohar

Score: 91

1. **Sets**: Create two sets of integers. Write a program to find their union, intersection, and difference.

**Syntax**:

set1 = {10, 20, 30, 40, 50}  
set2 = {40, 50, 60, 70, 80}  
union\_result = set1.union(set2)  
intersection\_result = set1.intersection(set2)  
difference\_set1\_set2 = set1.difference(set2)  
difference\_set2\_set1 = set2.difference(set1)  
print("Union of the sets:", union\_result)  
print("Intersection of the sets:", intersection\_result)  
print("Difference of set1 - set2:", difference\_set1\_set2)  
print("Difference of set2 - set1:", difference\_set2\_set1)

output:

Union of the sets: {70, 40, 10, 80, 50, 20, 60, 30}

Intersection of the sets: {40, 50}

Difference of set1 - set2: {10, 20, 30}

Difference of set2 - set1: {80, 60, 70}

1. **Stacks**: Implement a stack using a list. Write functions for push, pop, and peek operations.

syntax:

stack = []  
stack.append(10)  
stack.append(15)  
stack.append(20)  
print('Initial stack')  
print(stack)  
print('\nElements popped from stack:')  
print(stack.pop())  
print(stack.pop())  
print(stack.pop())  
print('\nStack after elements are popped:')  
print(stack)

output:

Initial stack

[10, 15, 20]

Elements popped from stack:

20

15

10

Stack after elements are popped:

[]

1. **Queues**: Implement a queue using a list. Write functions for enqueue and dequeue operations.

**Syntax**:

class Queue:  
 def \_\_init\_\_(self):  
 self.queue = []  
 def enqueue(self, item):  
 self.queue.append(item)  
 def dequeue(self):  
 if self.is\_empty():  
 print("Error: Queue is empty. Cannot dequeue.")  
 return None  
 return self.queue.pop(0)  
 def is\_empty(self):  
 return len(self.queue) == 0  
queue = Queue()  
queue.enqueue(1)  
queue.enqueue(2)  
queue.enqueue(3)  
print("Current queue:", queue.queue)  
dequeued\_item = queue.dequeue()  
print("Dequeued item:", dequeued\_item)  
print("Current queue after dequeue:", queue.queue)

output:

Current queue: [1, 2, 3]

Dequeued item: 1

Current queue after dequeue: [2, 3]

1. **Linked List**: Implement a singly linked list with methods to insert, delete, and display elements.

**Syntax**:

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
class LinkedList:  
 def \_\_init\_\_(self):  
 self.head = None  
 def insert(self, data):  
 new\_node = Node(data)  
 if self.head is None:  
 self.head = new\_node  
 return  
 last\_node = self.head  
 while last\_node.next:  
 last\_node = last\_node.next  
 last\_node.next = new\_node  
 def delete(self, data):  
 if self.head is None:  
 print("Error: Linked list is empty. Cannot delete.")  
 return  
 if self.head.data == data:  
 self.head = self.head.next  
 return  
 prev\_node = self.head  
 current\_node = self.head.next  
 while current\_node:  
 if current\_node.data == data:  
 prev\_node.next = current\_node.next  
 return  
 prev\_node = current\_node  
 current\_node = current\_node.next  
 print("Error: Node with data", data, "not found.")  
 def display(self):  
 *"""Display the elements of the linked list."""* if self.head is None:  
 print("Linked list is empty.")  
 return  
 current\_node = self.head  
 while current\_node:  
 print(current\_node.data, end=" ")  
 current\_node = current\_node.next  
 print()  
linked\_list = LinkedList()  
linked\_list.insert(10)  
linked\_list.insert(20)  
linked\_list.insert(30)  
linked\_list.insert(40)  
print("Linked list after insertions:")  
linked\_list.display()  
linked\_list.delete(30)  
print("Linked list after deletion of 3:")  
linked\_list.display()  
linked\_list.delete(2)  
linked\_list.delete(10)  
print("Linked list after deletion of 1:")  
linked\_list.display()

output: Linked list after insertions:

10 20 30 40

Linked list after deletion of 3:

10 20 40

Error: Node with data 2 not found.

Linked list after deletion of 1:

20 40

1. **Binary Tree**: Implement a binary tree with methods for in-order, pre-order, and post-order traversal.
2. class Node:  
    def \_\_init\_\_(self, key):  
    self.left = None  
    self.right = None  
    self.val = key  
   class BinaryTree:  
    def \_\_init\_\_(self):  
    self.root = None  
    def in\_order\_traversal(self, node):  
    result = []  
    if node:  
    result = self.in\_order\_traversal(node.left)  
    result.append(node.val)  
    result = result + self.in\_order\_traversal(node.right)  
    return result  
    def pre\_order\_traversal(self, node):  
    result = []  
    if node:  
    result.append(node.val)  
    result = result + self.pre\_order\_traversal(node.left)  
    result = result + self.pre\_order\_traversal(node.right)  
    return result  
    def post\_order\_traversal(self, node):  
    result = []  
    if node:  
    result = self.post\_order\_traversal(node.left)  
    result = result + self.post\_order\_traversal(node.right)  
    result.append(node.val)  
    return result  
   if \_\_name\_\_ == "\_\_main\_\_":  
    tree = BinaryTree()  
    tree.root = Node(1)  
    tree.root.left = Node(2)  
    tree.root.right = Node(3)  
    tree.root.left.left = Node(4)  
    tree.root.left.right = Node(5)  
    tree.root.right.left = Node(6)  
    tree.root.right.right = Node(7)  
    print("In-order traversal:", tree.in\_order\_traversal(tree.root))  
    print("Pre-order traversal:", tree.pre\_order\_traversal(tree.root))  
    print("Post-order traversal:", tree.post\_order\_traversal(tree.root))

output: In-order traversal: [4, 2, 5, 1, 6, 3, 7]

Pre-order traversal: [1, 2, 4, 5, 3, 6, 7]

Post-order traversal: [4, 5, 2, 6, 7, 3, 1]

**19.Graphs**: Represent a graph using an adjacency list and write a program to perform a breadth-first search (BFS).

Syntax:

from collections import defaultdict, deque  
class Graph:  
 def \_\_init\_\_(self):  
 self.adjacency\_list = defaultdict(list)  
 def add\_edge(self, u, v):  
 self.adjacency\_list[u].append(v)  
 self.adjacency\_list[v].append(u)  
 def bfs(self, start\_node):  
 visited = set()  
 queue = deque([start\_node])  
 while queue:  
 node = queue.popleft()  
 if node not in visited:  
 print(node, end=" ")  
 visited.add(node)  
 for neighbor in self.adjacency\_list[node]:  
 if neighbor not in visited:  
 queue.append(neighbor)  
graph = Graph()  
graph.add\_edge(0, 1)  
graph.add\_edge(0, 2)  
graph.add\_edge(1, 2)  
graph.add\_edge(2, 3)  
graph.add\_edge(3, 3)  
print("BFS starting from node 2:")  
graph.bfs(2)

output:

BFS starting from node 2:

2 0 1 3

**20.Hash Table**: Implement a basic hash table with functions for inserting, deleting, and searching elements.

class HashTable:  
 def \_\_init\_\_(self, size):  
 self.size = size  
 self.table = [None] \* size  
 def \_hash(self, key):  
 return hash(key) % self.size  
 def insert(self, key, value):  
 index = self.\_hash(key)  
 if self.table[index] is None:  
 self.table[index] = [(key, value)]  
 else:  
 self.table[index].append((key, value))  
 def delete(self, key):  
 index = self.\_hash(key)  
 if self.table[index] is not None:  
 for i, (existing\_key, existing\_value) in enumerate(self.table[index]):  
 if existing\_key == key:  
 del self.table[index][i]  
 break  
 def search(self, key):  
 index = self.\_hash(key)  
 if self.table[index] is not None:  
 for existing\_key, existing\_value in self.table[index]:  
 if existing\_key == key:  
 return existing\_value  
 return None  
hash\_table = HashTable(10)  
hash\_table.insert("apple", 20)  
hash\_table.insert("banana", 30)  
hash\_table.insert("cherry", 40)  
print("Search 'banana':", hash\_table.search("banana"))  
print("Search 'grape':", hash\_table.search("grape"))  
hash\_table.delete("banana")  
print("Search 'banana' after deletion:", hash\_table.search("banana"))

output:

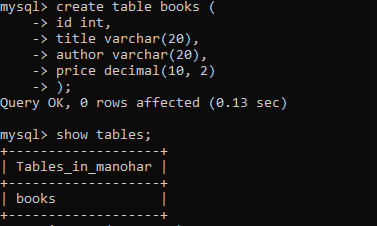
Search 'banana': 30

Search 'grape': None

Search 'banana' after deletion: None

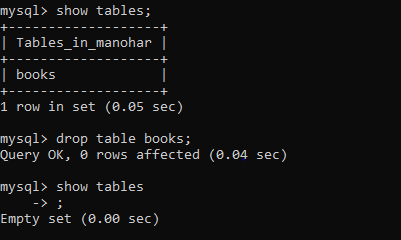
### **SQL DDL (Data Definition Language)**

1. **Create Table**: Write a SQL statement to create a table named Books with columns id, title, author, and price.



1. **Drop Table**: Write a SQL statement to drop the Books table.

**Ans**:



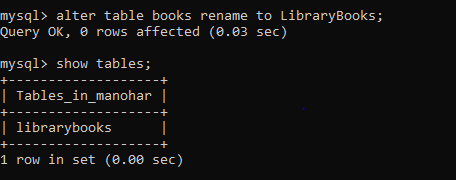
1. **Alter Table**: Write a SQL statement to add a new column published\_date to the Books table.

**Ans:**



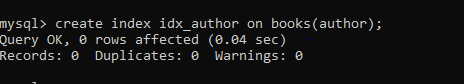
1. **Rename Table**: Write a SQL statement to rename the Books table to LibraryBooks.

**Ans:**



1. **Create Index**: Write a SQL statement to create an index on the author column of the Books table.

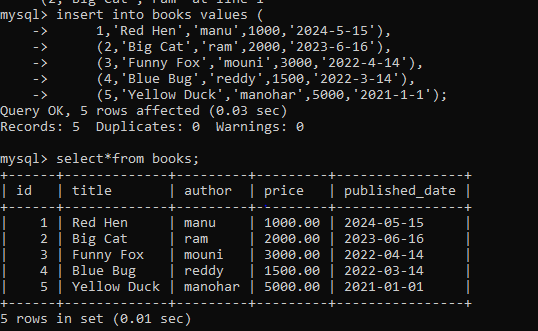
**Ans**:



### **SQL DML (Data Manipulation Language)**

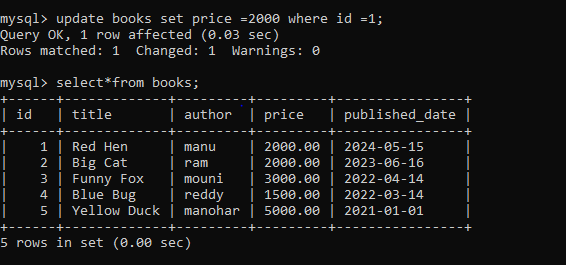
1. **Insert Data**: Write a SQL statement to insert a new record into the Books table.

**Ans**:



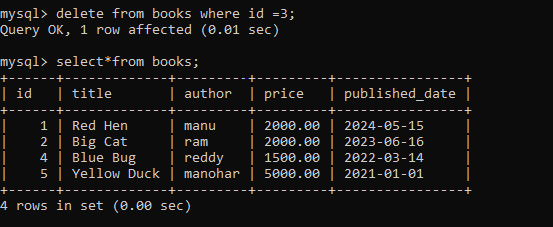
1. **Update Data**: Write a SQL statement to update the price of a book in the Books table based on its id.

**Ans:**



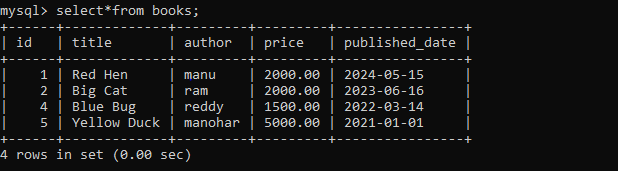
1. **Delete Data**: Write a SQL statement to delete a book from the Books table based on its id.

**Ans**:



1. **Select Data**: Write a SQL statement to select all records from the Books table.

**Ans**:



1. **Join Tables**: Write a SQL statement to join two tables, Books and Authors, on a common column and select relevant data.

**Ans**:

