Q.1 Demonstrate the working of Hash function for the following data: Integer, float, character, string, tuple, list

```
# Integer
print("Integer:", hash(42))

# Float
print("Float:", hash(3.14))

# Character
print("Character:", hash('a'))

# String
print("String:", hash("OpenAI"))

# Tuple
print("Tuple:", hash((1, 2, 3)))

# List (Lists are mutable and not hashable)
try:
    print("List:", hash([1, 2, 3]))
except TypeError as e:
    print("List: Error -", e)
```

```
Integer: 42
Float: 322818021289917443
Character: -8898068660708084394
String: 8343873355699456581
Tuple: 529344067295497451
List: Error - unhashable type: 'list'
```

Q.2 Take a number list and check the hash index assigned to the set of values of the list.

```
num_list = [10, 20, 30]

# Convert to tuple
tuple_form = tuple(num_list)
print("Hash index (tuple):", hash(tuple_form))
```

```
Hash index (tuple): 3952409569436607343
```

Q.3 Take any expression of your choice (e.g. hello world), and find the hash value assigned to the expression using ord()function used to get the ordinal value of any character.

```
expression = "hello world"
hash_value = sum(ord(char) for char in expression)
print(hash_value)
```

```
1116
```

Q.4 Demonstrate that Mutable objects like lists, dictionaries, and sets cannot be hashed with the hash() function.

```
val = {
    "list": [1, 2, 3],
    "dict": {"name": "vishal", "age": 22},
    "sets": {"vishal", "pal"}
}

for i in val:
    try:
        print("Hash value:", hash(val[i]))  # <-- This was missing a closing parenthesis
```

```
    except TypeError as e:
        print("Error:", e)
```

```
Error: unhashable type: 'list'
Error: unhashable type: 'dict'
Error: unhashable type: 'set'
```

Start coding or generate with AI.

Q.1 Write and execute the Python code to create a (i) Simple Hash Table (ii) Hash Table with Collision.

```
# Simple Hash Table Implementation

class SimpleHashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def insert(self, key, value):
        index = key % self.size
        self.table[index] = value

    def display(self):
        for i, val in enumerate(self.table):
            print(f"Index {i}: {val}")

# Create a hash table with no collisions
hash_table = SimpleHashTable(10)
hash_table.insert(1, "Apple")
hash_table.insert(2, "Banana")
hash_table.insert(3, "Cherry")

print("Simple Hash Table (No Collisions):")
hash_table.display()
```

```
Simple Hash Table (No Collisions):
Index 0: None
Index 1: Apple
Index 2: Banana
Index 3: Cherry
Index 4: None
Index 5: None
Index 6: None
Index 7: None
Index 8: None
Index 9: None
```

```
# Hash Table with Collision using Chaining

class HashTableWithCollision:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]

    def insert(self, key, value):
        index = key % self.size
        self.table[index].append((key, value))  # Append to chain

    def display(self):
        for i, items in enumerate(self.table):
            print(f"Index {i}: {items}")

# Create a hash table where collisions will occur
hash_table_c = HashTableWithCollision(5)
hash_table_c.insert(1, "Apple")
hash_table_c.insert(6, "Banana")  # 1 % 5 == 6 % 5 == 1
hash_table_c.insert(11, "Cherry")  # 11 % 5 == 1

print("\nHash Table with Collisions (Handled via Chaining):")
hash_table_c.display()
```

```

Hash Table with Collisions (Handled via Chaining):
Index 0: []
```

```
Index 1: [(1, 'Apple'), (6, 'Banana'), (11, 'Cherry')]
Index 2: []
Index 3: []
Index 4: []
```

Q. 2 Create a Hash Table with Collision and Demonstrate following different Open Addressing types Collision Handling Technique :

1. Probing
    - Linear Probing
    - Quadratic Probing
2. Double Hashing

```
class LinearProbingHashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def insert(self, key, value):
        index = key % self.size
        start_index = index

        while self.table[index] is not None:
            index = (index + 1) % self.size
            if index == start_index:
                print("Table is full!")
                return

        self.table[index] = (key, value)

    def display(self):
        print("Linear Probing:")
        for i, val in enumerate(self.table):
            print(f"Index {i}: {val}")
```

```
class QuadraticProbingHashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def insert(self, key, value):
        index = key % self.size
        i = 1
        start_index = index

        while self.table[index] is not None:
            index = (start_index + i * i) % self.size
            i += 1
            if i == self.size:
                print("Table is full!")
                return

        self.table[index] = (key, value)

    def display(self):
        print("\nQuadratic Probing:")
        for i, val in enumerate(self.table):
            print(f"Index {i}: {val}")
```

```
class DoubleHashingHashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def hash2(self, key):
        return 7 - (key % 7)  # Second hash function (must not be 0)

    def insert(self, key, value):
        index = key % self.size
        step = self.hash2(key)
        i = 0

        while self.table[index] is not None:
            index = (index + step) % self.size
```

```
            i += 1
            if i == self.size:
                print("Table is full!")
                return

        self.table[index] = (key, value)

    def display(self):
        print("\nDouble Hashing:")
        for i, val in enumerate(self.table):
            print(f"Index {i}: {val}")
```

```
keys = [10, 20, 30, 40, 50]  # All will collide at index 0 if table size is 10

# Create tables
linear = LinearProbingHashTable(10)
quadratic = QuadraticProbingHashTable(10)
double_hash = DoubleHashingHashTable(10)

# Insert same keys in all tables
for k in keys:
    linear.insert(k, f"Value{k}")
    quadratic.insert(k, f"Value{k}")
    double_hash.insert(k, f"Value{k}")

# Display results
linear.display()
quadratic.display()
double_hash.display()
```

```
Linear Probing:
Index 0: (10, 'Value10')
Index 1: (20, 'Value20')
Index 2: (30, 'Value30')
Index 3: (40, 'Value40')
Index 4: (50, 'Value50')
Index 5: None
Index 6: None
Index 7: None
Index 8: None
Index 9: None

Quadratic Probing:
Index 0: (10, 'Value10')
Index 1: (20, 'Value20')
Index 2: None
Index 3: None
Index 4: (30, 'Value30')
Index 5: None
Index 6: (50, 'Value50')
Index 7: None
Index 8: None
Index 9: (40, 'Value40')

Double Hashing:
Index 0: (10, 'Value10')
Index 1: (20, 'Value20')
Index 2: (40, 'Value40')
Index 3: None
Index 4: None
Index 5: (30, 'Value30')
Index 6: (50, 'Value50')
Index 7: None
Index 8: None
Index 9: None
```

Q.1 Create a Hash Table with Collision and Demonstrate how Separate Chaining is used for Collision Handling

```
class SeparateChainingHashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]  # Initialize empty chains

    def insert(self, key, value):
        index = key % self.size
        # Append the key-value pair to the list (chain) at that index
        self.table[index].append((key, value))
```

```
    def display(self):
        print("Hash Table with Separate Chaining:")
        for i, chain in enumerate(self.table):
            print(f"Index {i}: {chain}")
```

```
Start coding or generate with AI.
```

Q1: Hash Table – Rehashing

```
class RehashingHashTable:
    def __init__(self, initial_size=5):
        self.size = initial_size
        self.count = 0
        self.table = [None] * self.size

    def _hash(self, key):
        return key % self.size

    def _rehash(self):
        print(f"\nRehashing... Old size: {self.size}")
        old_table = self.table
        self.size *= 2
        self.table = [None] * self.size
        self.count = 0

        for item in old_table:
            if item is not None:
                self.insert(*item)

        print(f"New size: {self.size}")

    def insert(self, key, value):
        if self.count / self.size > 0.7:
            self._rehash()

        index = self._hash(key)
        while self.table[index] is not None:
            index = (index + 1) % self.size

        self.table[index] = (key, value)
        self.count += 1

    def display(self):
        print("\nHash Table with Rehashing:")
        for i, item in enumerate(self.table):
            print(f"Index {i}: {item}")
```

Q1: Hash Table – Universal Hashing

```
import random

class UniversalHashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size
        self.p = 109345121  # large prime
        self.a = random.randint(1, self.p - 1)
        self.b = random.randint(0, self.p - 1)

    def _hash(self, key):
        return ((self.a * key + self.b) % self.p) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        while self.table[index] is not None:
            index = (index + 1) % self.size
        self.table[index] = (key, value)

    def display(self):
        print("\nUniversal Hash Table:")
        for i, val in enumerate(self.table):
            print(f"Index {i}: {val}")
```

```
Start coding or generate with AI.
```