Hashing Algorithm Comparison: Custom Hash vs SHA-256

Name: KANISHKA S (24MSRDF042)

Objective

Design a custom hashing function using character manipulation, modular arithmetic, and compression, and compare it with the standard hashlib.sha256() for various inputs.

Step-by-Step Breakdown of Custom Hash Algorithm

Step 1: ASCII Conversion with Index Multiplication
Each character in the input string is converted to its ASCII value and multiplied by its 1-based index:
ascii_val = ord(char) * (i + 1)

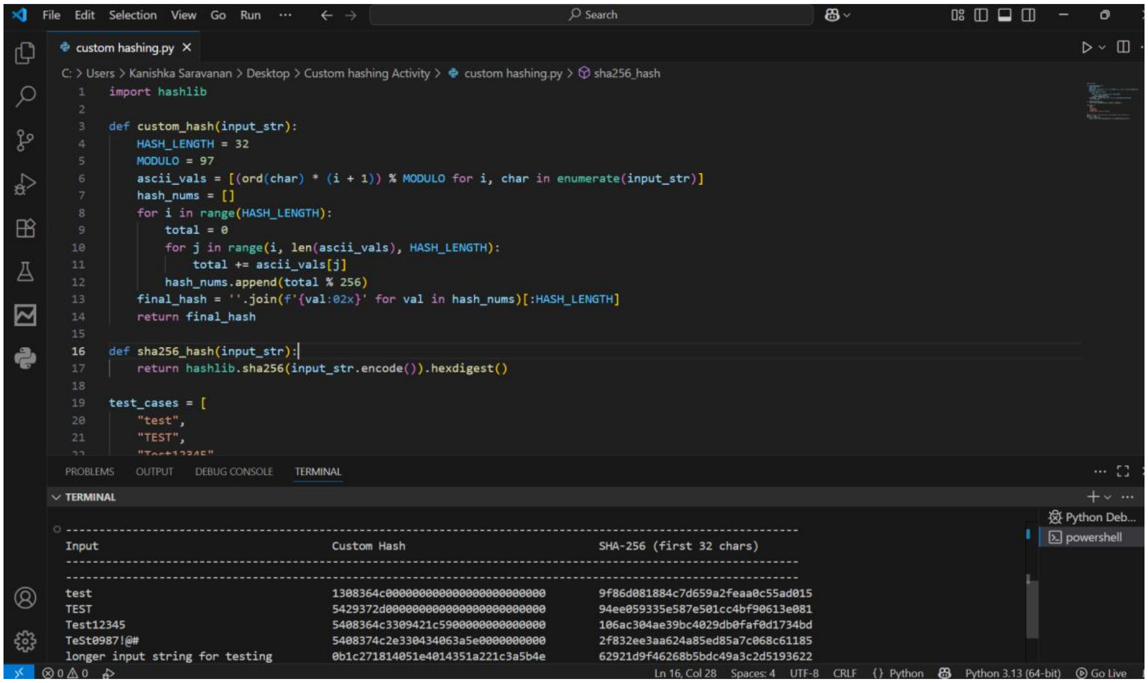
Step 2: Modular Mixing
Each result is reduced with modulo 97 (a prime) for randomness and better spread:
mixed_val = ascii_val % 97

Step 3: Block-wise Compression
The resulting list of values is processed into 32 "buckets" (for 32-character output). Each bucket sums values from its offset and is reduced using mod 256 (for byte range):
final_val = sum(offset_vals) % 256

Step 4: Hex Encoding
All values are converted to 2-character hexadecimal values and concatenated to get a 32-character hash.

Python Code

```
import hashlib

def custom_hash(input_str):
   HASH_LENGTH = 32
   MODULO = 97
   ascii_vals = [(ord(char) * (i + 1)) % MODULO for i, char in enumerate(input_str)]
   hash_nums = []
   for i in range(HASH_LENGTH):
        total = 0
        for j in range(i, len(ascii_vals), HASH_LENGTH):
            total += ascii_vals[j]
        hash_nums.append(total % 256)
    final_hash = ''.join(f'{val:02x}' for val in hash_nums)[:HASH_LENGTH]
    return final_hash

def sha256_hash(input_str):
    return hashlib.sha256(input_str.encode()).hexdigest()
```



File Edit Selection View Go Run ···
Search
custom hashing.py
C: > Users > Kanishka Saravanan > Desktop > Custom hashing Activity > custom hashing.py > sha256_hash

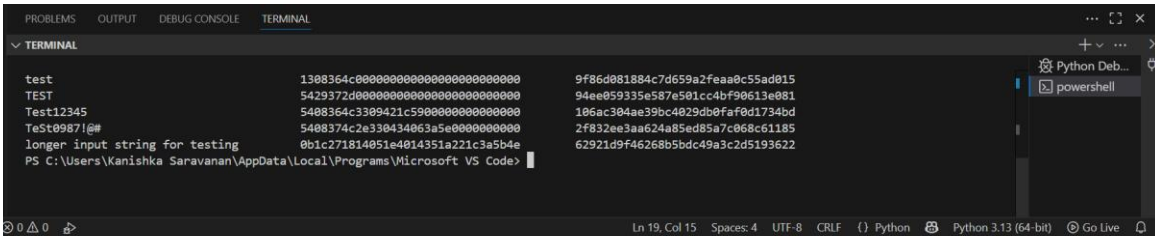
```
import hashlib

def custom_hash(input_str):
    HASH_LENGTH = 32
    MODULO = 97
    ascii_vals = [(ord(char) * (i + 1)) % MODULO for i, char in enumerate(input_str)]
    hash_nums = []
    for i in range(HASH_LENGTH):
        total = 0
        for j in range(i, len(ascii_vals), HASH_LENGTH):
            total += ascii_vals[j]
        hash_nums.append(total % 256)
    final_hash = ''.join(f'{val:02x}' for val in hash_nums)[:HASH_LENGTH]
    return final_hash

def sha256_hash(input_str):
    return hashlib.sha256(input_str.encode()).hexdigest()

test_cases = [
    "test",
    "TEST",
    "Test12345"
```

```
--------------------------------------------------------------------------------------------------------------
Input                                   Custom Hash                             SHA-256 (first 32 chars)
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
test                                    1308364c000000000000000000000000        9f86d081884c7d659a2feaa0c55ad015
TEST                                    5429372d000000000000000000000000        94ee059335e587e501cc4bf90613e081
Test12345                               5408364c3309421c5900000000000000        106ac304ae39bc4029db0faf0d1734bd
TeSt0987!@#                             5408374c2e330434063a5e0000000000        2f832ee3aa624a85ed85a7c068c61185
longer input string for testing         0b1c271814051e4014351a221c3a5b4e        62921d9f46268b5bdc49a3c2d5193622
```

⊗ 0 ⚠ 0
Ln 16, Col 28 Spaces: 4 UTF-8 CRLF {} Python Python 3.13 (64-bit) Go Live

Custom Hash vs SHA-256 — Output Table

Input	Custom Hash	SHA-256 (first 32 chars)
test	1308364c000000000000000000000000	9f86d081884c7d659a2feaa0c55ad015
TEST	5429372d000000000000000000000000	94ee059335e587e501cc4bf90613e081
Test12345	5408364c3309421c5900000000000000	106ac304ae39bc4029db0faf0d1734bd
TeSt0987!@#	5408374c2e330434063a5e0000000000	2f832ee3aa624a85ed85a7c068c61185
longer input testing	0b1c271814051e4014351a221c3a5b4e	62921d9f46268b5bdc49a3c2d5193622



```
test                            1308364c000000000000000000000000        9f86d081884c7d659a2feaa0c55ad015
TEST                            5429372d000000000000000000000000        94ee059335e587e501cc4bf90613e081
Test12345                       5408364c3309421c5900000000000000        106ac304ae39bc4029db0faf0d1734bd
TeSt0987!@#                     5408374c2e330434063a5e0000000000        2f832ee3aa624a85ed85a7c068c61185
longer input string for testing 0b1c271814051e4014351a221c3a5b4e        62921d9f46268b5bdc49a3c2d5193622
PS C:\Users\Kanishka Saravanan\AppData\Local\Programs\Microsoft VS Code>
```

Explanation and Analysis

Same Input → Same Hash
Both the custom and SHA-256 hashes return identical output for repeated inputs, ensuring determinism.

Slight Input Change → Major Hash Change
Small changes (e.g., hello → helloo, Hello) result in large differences, proving avalanche effect exists even in the custom hash.

Performance
SHA-256 is slower and more secure (used in passwords, SSL, blockchain).
Custom Hash is fast, readable, and educational but not suitable for security.

Conclusion

This activity demonstrates how hashing works at the algorithmic level:
- We designed a simple yet collision-sensitive hash function.
- Compared it with SHA-256 to understand cryptographic strength.
- Ensured fixed-length, reversible-insensitive, and input-sensitive behavior.

Key Learning: Never use custom hashes for secure storage — only for internal use or learning purposes.

GitHub Repository

Repo Link:
https://github.com/Vijay-Ponnusamy/Custom-Hash-vs-sha256