AI ASSISTED CODING
ASSIGNMENT - 5.1
ROLL NO. 2303A510J0

Task Description #1 (Privacy in API Usage):

Prompt:
Generate Python code to fetch weather data from a weather API.

Code:

```
import requests

API_KEY = "a3f9c2e4b7d8410f9c6e2a1b5d7f8c90"
city = "Delhi"

url =
f"https://api.openweathermap.org/data/2.5/weather?q={ci
ty}&appid={API_KEY}"
response = requests.get(url)

print(response.json())
```

Output:

```
{
  "weather": [
    {
      "description": "clear sky"
    }
  ],
```

```
  "main": {
    "temp": 303.15
  },
  "name": "Delhi"
}
```

Explanation:

The API key is written directly inside the program. This approach is insecure because anyone who has access to the source code can see and misuse the API key. If the code is shared publicly or uploaded to a repository, the key can be exposed, leading to unauthorized usage and possible service abuse.

Secure version:

```
import os
os.environ["WEATHER_API_KEY"] =
"a3f9c2e4b7d8410f9c6e2a1b5d7f8c90"
```

```
import os
import requests

API_KEY = os.getenv("WEATHER_API_KEY")
city = "Delhi"

url =
f"https://api.openweathermap.org/data/2.5/weather?q={ci
ty}&appid={API_KEY}"
response = requests.get(url)
```

```
print(response.json())
```

Output:

```
{
  "weather": [
    {
      "description": "clear sky"
    }
  ],
  "main": {
    "temp": 301.15
  },
  "name": "Delhi"
}
```

Explanation:
The API key is stored in an environment variable and accessed at runtime. This ensures that the sensitive API key is not visible in the source code. Even if the code is shared, the API key remains protected, improving privacy and security. This method follows best practices for handling sensitive credentials in applications.

Conclusion:
Using environment variables to store API keys is secure and best practice.

Task Description #2 (Privacy & Security in File Handling)

Prompt:

Generate a Python script that asks the user for their name, email, and password, and stores this information in a text file.

Code:

```
# Get user input
name = input("Enter your name: ")
email = input("Enter your email: ")
password = input("Enter your password: ")

# Define the filename
filename = "user_info.txt"

# Open the file in write mode and store the information
with open(filename, "w") as file:
    file.write(f"Name: {name}\n")
    file.write(f"Email: {email}\n")
    file.write(f"Password: {password}\n")

print(f"User information saved to {filename}")
```

Output:

```
Enter your name: rohit
Enter your email: this@gmail.com
Enter your password: something@123#gp
User information saved to user_info.txt
```

Explanation:

The code stores the data without encrypting as a plain text.

Encrypted secure version:

Prompt:

Improve the Python script to store user data securely. Encrypt both the email and password using a secure method. Store the name as plain text. Provide full working code including encryption and decryption for later verification.

Code:

Install cryptography library

```
get_ipython().system('pip install cryptography')
```

Generate and store encryption key

```
from cryptography.fernet import Fernet

# Generate a key
key = Fernet.generate_key()

# Define the filename for the key
key_filename = "secret.key"

# Save the key to a file
with open(key_filename, "wb") as key_file:
    key_file.write(key)

print(f"Encryption key saved to {key_filename}")
```

Encrypt and save user data

```
from cryptography.fernet import Fernet

# 1. Load the encryption key
with open(key_filename, "rb") as key_file:
    key = key_file.read()
fernet = Fernet(key)

# Get user input (these variables are already present from previous
execution)
```

```
name = input("Enter your name: ")
email = input("Enter your email: ")
password = input("Enter your password: ")

# 2. Encode email and password into bytes
encoded_email = email.encode()
encoded_password = password.encode()

# 3. Encrypt the encoded email and password
encrypted_email = fernet.encrypt(encoded_email)
encrypted_password = fernet.encrypt(encoded_password)

# Define the filename
# filename = "user_info.txt" # Already defined

# 4. & 5. Open the file in write mode and store the information
with open(filename, "w") as file:
    file.write(f"Name: {name}\n")
    file.write(f"Encrypted Email: {encrypted_email.decode()}\n") # Decode
bytes to string for writing
    file.write(f"Encrypted Password: {encrypted_password.decode()}\n") #
Decode bytes to string for writing

# 6. Print confirmation message
print(f"User information (with encrypted email and password) saved to
{filename}")
```

Decrypt and verify user data

```
from cryptography.fernet import Fernet

# 1. & 2. Load the encryption key
with open(key_filename, "rb") as key_file:
    key = key_file.read()

# 3. Initialize a Fernet object
fernet = Fernet(key)

# 4. Open user_info.txt in read mode and read its contents
with open(filename, "r") as file:
```

```
    lines = file.readlines()

# 5. Extract the name, encrypted email, and encrypted password
retrieved_name = lines[0].split(": ")[1].strip()
encrypted_email_str = lines[1].split(": ")[1].strip()
encrypted_password_str = lines[2].split(": ")[1].strip()

# Encode the encrypted strings back to bytes for decryption
encrypted_email_bytes = encrypted_email_str.encode()
encrypted_password_bytes = encrypted_password_str.encode()

# 6. Decrypt the email and password
decrypted_email_bytes = fernet.decrypt(encrypted_email_bytes)
decrypted_password_bytes = fernet.decrypt(encrypted_password_bytes)

# 7. Decode the decrypted email and password to UTF-8 strings
decrypted_email = decrypted_email_bytes.decode()
decrypted_password = decrypted_password_bytes.decode()

# 8. Print the retrieved information
print("\n--- Decrypted User Information ---")
print(f"Name: {retrieved_name}")
print(f"Decrypted Email: {decrypted_email}")
print(f"Decrypted Password: {decrypted_password}")
print("-----------------------------------")
```

Output:

```
Requirement already satisfied: cryptography in
/usr/local/lib/python3.12/dist-packages (43.0.3)
Requirement already satisfied: cffi>=1.12 in
/usr/local/lib/python3.12/dist-packages (from cryptography) (2.0.0)
Requirement already satisfied: pycparser in
/usr/local/lib/python3.12/dist-packages (from cffi>=1.12->cryptography) (2.23)
```

```
Encryption key saved to secret.key
```

```
Enter your name: rohit
Enter your email: this@gmail.com
Enter your password: something@123#gp
User information (with encrypted email and password) saved to user_info.txt
```

```
--- Decrypted User Information ---
Name: rohit
Decrypted Email: this@gmail.com
Decrypted Password: something@123#gp
---------------------------------
```

Explanation:

The code first sets up the cryptography library for secure data handling. It then generates and stores a unique encryption key, which is crucial for both encrypting and decrypting sensitive user information. Next, it takes user inputs, encrypts the email and password, and saves them along with the plaintext name into a file. Finally, it demonstrates how to load the key and decrypt the stored information, verifying that the original data can be securely retrieved.

Conclusion:

The initial AI-generated script stored passwords in plain text, creating serious privacy risks. By using hashing Fernet, passwords are now stored securely, preventing unauthorized access and protecting user data.

Task Description #3 (Transparency in Algorithm Design) Prompt:

Generate a Python function to check if a number is an Armstrong number.

Provide clear comments for each line explaining what it does. Also, give a step-by-step explanation of how the function works.

Code:

```
def is_armstrong(number):
    """
    Check if the given number is an Armstrong number.
    An Armstrong number is a number that is equal to the sum
    of its own digits raised to the power of the number of digits.
    Example: 153 -> 1^3 + 5^3 + 3^3 = 153
    """
    # Convert the number to string to easily iterate over each digit
    digits = str(number)

    # Count the number of digits
    num_digits = len(digits)

    # Initialize sum of powers
    total = 0

    # Loop over each digit
    for digit in digits:
```

```
        # Raise digit to the power of num_digits
and add to total
        total += int(digit) ** num_digits

    # Compare the total with original number
    return total == number


# Example usage
num = 153
if is_armstrong(num):
    print(f"{num} is an Armstrong number")
else:
    print(f"{num} is not an Armstrong number")
```

Output:

```
153 is an Armstrong number
```

Explanation:

Line	Explanation
`def is_armstrong(number):`	Defines a function that checks if a number is an Armstrong number.

`digits = str(number)`	Converts the number to a string so each digit can be accessed individually.
`num_digits = len(digits)`	Counts how many digits are in the number.
`total = 0`	Initializes a variable to store the sum of powers of digits.
`for digit in digits:`	Loops through each digit in the number.
`total += int(digit) ** num_digits`	Raises the digit to the power of the number of digits and adds it to the total.
`return total == number`	Returns `True` if the sum equals the original number; otherwise `False`.
`num = 153 ...`	Example code to test the function and print whether the number is Armstrong.

Conclusion:

Explanation matches code functionality:

- Each line in the explanation correctly describes what the code does.

- Conversion to string, power calculation, and total comparison are all covered

---

Task Description #4 (Transparency in Algorithm Comparison)

Prompt:

Generate Python code for QuickSort and BubbleSort.

Include detailed comments explaining how each step works.

Also, explain clearly the differences between the two algorithms in terms of logic and efficiency.

Code:

```
# -------------------- BubbleSort
---------------------
def bubble_sort(arr):
    """
    BubbleSort repeatedly swaps adjacent
elements if they are in the wrong order.
    This continues until the list is sorted.
```

```
    """
    n = len(arr)
    for i in range(n):
        # Track if any swap happened in this
pass
        swapped = False
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if arr[j] > arr[j + 1]:
                # Swap if they are in wrong
order
                arr[j], arr[j + 1] = arr[j + 1],
arr[j]
                swapped = True
        # If no swaps happened, the array is
already sorted
        if not swapped:
            break
    return arr


# -------------------- QuickSort
--------------------
def quick_sort(arr):
    """
    QuickSort works by selecting a 'pivot'
element and partitioning
```

```
    the array into elements less than pivot and
elements greater than pivot.
    It then recursively sorts the sub-arrays.
    """
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]  # Choose the first
element as pivot
        # Partition into two sublists
        less = [x for x in arr[1:] if x <=
pivot]
        greater = [x for x in arr[1:] if x >
pivot]
        # Recursively sort sublists and combine
        return quick_sort(less) + [pivot] +
quick_sort(greater)


# -------------------- Example Usage
--------------------
arr1 = [64, 34, 25, 12, 22, 11, 90]
arr2 = arr1.copy()

print("BubbleSort Result:", bubble_sort(arr1))
print("QuickSort Result:", quick_sort(arr2))
```

Output:

```
BubbleSort Result: [11, 12, 22, 25, 34, 64, 90]
QuickSort Result: [11, 12, 22, 25, 34, 64, 90]
```

Comparison:

Feature	BubbleSort	QuickSort
Logic	Repeatedly compares adjacent elements and swaps them if they are in wrong order until the list is sorted.	Selects a pivot, partitions the array into elements less than and greater than pivot, and recursively sorts the partitions.
Implementation	Iterative, simple to implement.	Recursive, divide-and-conquer approach.

Best Case Time Complexity	**O(n) (if already sorted, with optimized swap check)**	**O(n log n)**
Average Case Time Complexity	**$O(n^2)$**	**O(n log n)**
Worst Case Time Complexity	**$O(n^2)$**	**$O(n^2)$ (rare, e.g., pivot is smallest/largest element)**
Space Complexity	**O(1) (in-place)**	**O(log n) (recursive call stack)**
Stability	**Stable (does not change relative order of equal elements)**	**Not stable by default**
Key Difference	**Simple but inefficient for large arrays.**	**Efficient for large arrays, faster due to divide-and-conquer strategy.**

Task Description #5 (Transparency in AI Recommendations)

Prompt:

Generate a Python product recommendation system.

The system should suggest products to a user based on their past purchases or preferences.

For each recommendation, provide a clear explanation of why the product is suggested.

Code:

```
# Sample product recommendation system
# Data: Users' past purchases
user_purchases = {
    "Alice": ["Laptop", "Mouse", "Keyboard"],
    "Bob": ["Shampoo", "Conditioner", "Soap"],
    "Charlie": ["Laptop", "Headphones"]
}

# Product categories for recommendation logic
product_categories = {
    "Electronics": ["Laptop", "Mouse",
"Keyboard", "Headphones", "Monitor"],
    "Personal Care": ["Shampoo", "Conditioner",
"Soap", "Lotion"],
```

```
    "Books": ["Novel", "Comics", "Biography"]
}

def recommend_products(user):
    """
    Recommend products to the user based on
their purchase history.
    Also provide reasons for each
recommendation.
    """
    purchased = user_purchases.get(user, [])
    recommendations = []

    # Recommend products from the same category
    for category, products in
product_categories.items():
        # Check if the user bought something in
this category
        if any(item in products for item in
purchased):
            for product in products:
                if product not in purchased:
                    reason = f"Because you
bought similar items in {category}"

recommendations.append((product, reason))
```

```
    return recommendations

# Example usage
user = "Alice"
results = recommend_products(user)

print(f"Recommendations for {user}:")
for product, reason in results:
    print(f"- {product}: {reason}")
```

Output:

```
Recommendations for Alice:
- Headphones: Because you bought similar items
in Electronics
- Monitor: Because you bought similar items in
Electronics
```

Explanation:

The system checks a user's past purchases and identifies the categories they belong to. It then recommends products from the same categories that the user hasn't bought yet. Each recommendation includes a reason, explaining that it was suggested because the user bought similar items in that category. This makes the system's logic transparent and easy to understand.

Explanation Quality:

- **Each recommendation is accompanied by a reason that is easy to understand.**

- **The user can clearly see the logic behind suggestions (“Because you bought similar items in [category]”).**

- **This makes the system transparent and interpretable, unlike black-box recommendation systems.**

Transparency Score: High — the reasoning is explicit and readable.