ASSIGNMENT-19.4

HT.NO-2203A52100

BATCH-50

QUESTION:

Lab 19: Assignment – Code Translation: Converting Between Programming Languages Week 10

Lab Objectives

- Translate code between programming languages while preserving logic and structure.
- Identify language-specific differences in syntax, typing, and execution models.
- Validate AI-generated translations for correctness and performance.
- Compare procedural, object-oriented, and database query translations across ecosystems.

Learning Outcome

After completing this assignment, students will be able to:

- Integrate external REST APIs using AI-assisted Python scripting. • Implement structured error handling for API failures and invalid inputs.
- Safely parse and validate JSON responses from external services.
- Develop resilient applications capable of handling real-world API limitations.

Task 1 – JavaScript to Python (Array Processing Logic)

Scenario

A frontend application written in JavaScript processes user scores. The backend team wants the same logic implemented in Python for analytics processing.

Task

Translate a JavaScript function that:

• Takes an array of numbers

• Filters values greater than 50

• Returns the average of filtered values

Instructions

• Prompt AI to convert JavaScript array methods (filter, reduce) into equivalent Python logic.

• Ensure list comprehension or Pythonic style is used.

• Compare outputs of both implementations with sample input.

• Document differences in handling arrays between JS and Python.

Expected Output

• Equivalent Python function.

• Same output for identical inputs.

• Clean and Pythonic implementation.

Task 2 – Python OOP to C++ Class Translation

Scenario

A data modeling module is written in Python using OOP. The system is being migrated to a high-performance C++ environment.

Task

Translate a Python class that:

• Contains private attributes

• Uses getter/setter methods

• Includes a method that calculates a derived value

Instructions

• Convert the Python class into a C++ class.

• Ensure:

o Proper access modifiers o Constructor handling

o Correct data types

• Compile and test the C++ version.

• Compare how encapsulation differs between Python and C++.

Expected Output

• Fully working C++ class equivalent to Python version.

• Proper use of access specifiers (private, public).

Task 3 – REST API Call Translation: Python to JavaScript

Scenario

A backend developer wrote an API request using Python requests. The frontend team needs the same functionality in JavaScript (Fetch API).

Task

Translate Python API request code into JavaScript.

Instructions

• Convert:

o GET request logic o JSON parsing o

Error handling (try-except → try-catch)

• Ensure proper async/await usage in JavaScript.

• Compare differences in error handling mechanisms.

Expected Output

• Equivalent JavaScript code using Fetch API.

• Proper async handling.

• Similar output structure.

Task 4 – SQL Aggregation to MongoDB Query Translation

Scenario

A company is migrating from relational databases (SQL) to MongoDB (NoSQL).

Task

Translate an SQL aggregation query into a MongoDB aggregation pipeline.

Example Logic:

• Group records by category

• Calculate average price

• Filter groups where average price > threshold

Instructions

• Provide SQL query to AI.

• Generate equivalent MongoDB query.

• Explain differences between relational and document-based querying.

• Validate results conceptually.

Expected Output

• Working MongoDB aggregation query.

• Clear understanding of grouping and aggregation differences.

Task 5 – Real-Time Application: Algorithm Translation Across Three Languages

Scenario

A tech company maintains codebases in multiple languages (Python, Java, and Go). A common algorithm must be implemented consistently across systems.

Task

Choose an algorithm (e.g., binary search, palindrome check, prime number checker) and translate it across:

• Python

• Java

• One additional language (Go / C# / JavaScript)

Instructions

• Ensure logic remains identical across languages.

• Handle: o Data types o Loop differences o Input/output handling

• Test each version with same input values.

• Briefly document translation challenges.

Expected Output

• Three equivalent working implementations.

• Clear logical consistency.

• Understanding of syntactic vs structural differences.

CODE:

```
import json from typing import Dict,
Any, Optional


# ==================================================================
==========
# TASK 1: JavaScript to Python (Array Processing Logic)
# ==================================================================
==========


def filter_and_average(scores):
    """
    Translates JavaScript array processing logic to Python.

    JavaScript equivalent:     function
filterAndAverage(scores) {
return scores
            .filter(val => val > 50)
            .reduce((sum, val) => sum + val, 0) / scores.filter(val => val > 50).length;
    }

    Args:
        scores: List of numbers

    Returns:
        Average of values greater than 50, or 0 if none exist
    """     filtered = [score for score in scores if
score > 50]
```

```
    if len(filtered) == 0:
        return 0

    return sum(filtered) / len(filtered)

# Test Task 1 print("=" * 70) print("TASK 1: JavaScript to
Python (Array Processing)") print("=" * 70) test_scores = [30,
45, 55, 60, 70, 40, 80, 50] result_task1 =
filter_and_average(test_scores) print(f"Input: {test_scores}")
print(f"Filtered values (>50): {[x for x in test_scores if x > 50]}")
print(f"Average of filtered values: {result_task1}") print()

#
==================================================================
==========
# TASK 2: Python OOP to C++ Class Translation
#
==================================================================
==========
class Student:
    """
    Python class demonstrating OOP concepts.
    Will be translated to C++ with equivalent functionality.

    Attributes:
        __name (private): Student name
        __gpa (private): Grade Point Average
        __credits (private): Course credits
    """
```

```
    def __init__(self, name, gpa, credits):
        """Constructor"""
self.__name = name
self.__gpa = gpa        self.__credits
= credits

    def get_name(self):
"""Getter for name"""
return self.__name

    def set_name(self, name):
"""Setter for name"""
self.__name = name

    def get_gpa(self):
"""Getter for GPA"""
return self.__gpa
    def set_gpa(self, gpa):
    """Setter for GPA"""
    if 0 <= gpa <= 4.0:
            self.__gpa = gpa
        else:
            raise ValueError("GPA must be between 0 and 4.0")

    def get_credits(self):
"""Getter for credits"""
return self.__credits
```

```
    def set_credits(self, credits):
"""Setter for credits"""
self.__credits = credits

    def calculate_academic_standing(self):
        """
        Calculates derived value: academic standing.
        Returns: "Excellent" if GPA >= 3.5, "Good" if >= 3.0, "Fair" otherwise
        """         if self.__gpa
>= 3.5:             return
"Excellent"         elif
self.__gpa >= 3.0:
            return "Good"
    else:
            return "Fair"

    def __str__(self):
        return f"Student: {self.__name}, GPA: {self.__gpa}, Credits: {self.__credits}"

# Test Task 2 print("=" * 70) print("TASK 2: Python OOP Class") print("=" *
70) student1 = Student("Alice Johnson", 3.8, 120) print(f"Created:
{student1}") print(f"Academic Standing:
{student1.calculate_academic_standing()}") student1.set_gpa(3.2)
print(f"Updated GPA: {student1.get_gpa()}") print(f"New Academic
Standing: {student1.calculate_academic_standing()}")
print()

#
==================================================================
==========
```

```
# TASK 3: REST API Call Translation (Python to JavaScript concept)
# ==================================================================
==========


def fetch_user_data_python(user_id: int) -> Optional[Dict[str, Any]]:
    """
    Python implementation using requests library for API calls.


    JavaScript Equivalent (Fetch API):
    async function fetchUserData(userId) {
        try {
            const response = await fetch(`https://api.example.com/users/${userId}`);
            if (!response.ok) {             throw new Error(`HTTP error!
    status: ${response.status}`);
            }
            const data = await response.json();
return data;         } catch (error) {
console.error('Error:', error.message);
return null;
        }
    }
"""
try:
        # Note: Using mock data instead of actual API call for testing
if user_id <= 0:
            raise ValueError("User ID must be positive")


        # Simulated API response
mock_response = {
```

```
            "id": user_id,
            "name": "John Doe",
            "email": "john@example.com",
            "status": "active"
        }

        return mock_response

    except ValueError as e:
    print(f"Error: {str(e)}")
    return None

    except Exception as e:
        print(f"Unexpected error: {str(e)}")
    return None

# Test Task 3 print("=" * 70) print("TASK 3: REST API
Call - Python Implementation") print("=" * 70)
result_task3 = fetch_user_data_python(1) if result_task3:
    print(f"API Response: {json.dumps(result_task3, indent=2)}") print()

#
==================================================================
==========
# TASK 4: SQL to MongoDB Query Translation
#
==================================================================
==========

def sql_to_mongodb_explanation():
    """
```

SQL Query:

```
SELECT category, AVG(price) as average_price
FROM products
GROUP BY category
HAVING AVG(price) > 50
ORDER BY average_price DESC;
```

MongoDB Equivalent (Aggregation Pipeline):

```
db.products.aggregate([
    {
        $group: {
            _id: "$category",
average_price: { $avg: "$price" }
        }
    },
    {
        $match: { average_price: { $gt: 50 } }
    },
    {
        $sort: { average_price: -1 }
    }
])    """
return {
    "SQL_Query": """
        SELECT category, AVG(price) as average_price
        FROM products
        GROUP BY category
        HAVING AVG(price) > 50
        ORDER BY average_price DESC;
```

```
        """,
        "MongoDB_Pipeline": [
            {"$group": {"_id": "$category", "average_price": {"$avg": "$price"}}},
            {"$match": {"average_price": {"$gt": 50}}},
            {"$sort": {"average_price": -1}}
        ],
        "Differences": {
            "Grouping": "SQL uses GROUP BY; MongoDB uses $group stage",
            "Filtering": "SQL uses HAVING clause; MongoDB uses $match stage",
           "Sorting": "Both use ORDER BY / $sort, but MongoDB uses -1 for DESC",
            "Document_Structure": "MongoDB returns documents; SQL returns tuples",
"Flexibility": "MongoDB is more flexible with nested documents"
        }
    }

# Test Task 4 print("=" * 70) print("TASK 4: SQL to MongoDB
Query Translation") print("=" * 70) task4_result =
sql_to_mongodb_explanation() print("SQL Query:")
print(task4_result["SQL_Query"].strip()) print("\nMongoDB
Aggregation Pipeline:")
print(json.dumps(task4_result["MongoDB_Pipeline"], indent=2))
print("\nKey Differences:") for key, value in
task4_result["Differences"].items():
    print(f"  • {key}: {value}") print()

#
==========================================================================
==========
# TASK 5: Algorithm Translation Across Three Languages
#
```

==========
==========

def binary_search_python(arr: list, target: int) -> int:

"""

Python implementation of binary search algorithm.

Args:

arr: Sorted list of integers

target: Value to search for

Returns:

Index of target if found, -1 otherwise

""" left, right = 0,

len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1 else:

right = mid - 1

return -1

def binary_search_java_code():

"""Returns Java equivalent of binary search"""

return """

```
    public class BinarySearch {         public static int
binarySearch(int[] arr, int target) {             int left = 0,
right = arr.length - 1;

            while (left <= right) {
int mid = (left + right) / 2;

                if (arr[mid] == target) {
return mid;
                } else if (arr[mid] < target) {
left = mid + 1;                 } else {
right = mid - 1;
                }
            }
return -1;
        }
    }
    """

def binary_search_go_code():
    """Returns Go equivalent of binary search"""
return """     package main

    func BinarySearch(arr []int, target int) int {
left, right := 0, len(arr)-1

        for left <= right {
mid := (left + right) / 2
```

```
            if arr[mid] == target {
return mid
            } else if arr[mid] < target {
left = mid + 1            } else {
right = mid - 1
            }
        }

        return -1
    }
    """

# Test Task 5 print("=" * 70) print("TASK 5: Binary Search Algorithm - Three
Language Implementation") print("=" * 70)

test_array = [2, 5, 8, 12, 16, 23, 38, 45, 56, 67] test_target
= 23

# Python Version result_python =
binary_search_python(test_array, test_target) print(f"Python
Implementation:") print(f"  Array: {test_array}") print(f"
Target: {test_target}")
print(f"  Result: Index {result_python}") print()

# Java Version Code print("Java
Implementation:")
print(binary_search_java_code())
print()
```

Go Version Code print("Go

Implementation:")

print(binary_search_go_code())

print()

OUTPUT:

```
PS C:\Users\krish\OneDrive\Documents\AI\Py.files> & c:/Users/krish/OneDrive/Documents/AI/Py.files/.venv/Scripts/python.exe "c:/Users/krish/OneDrive/Documents/AI/Py.files/19.4 (1).py"
 ======================================================================
 TASK 1: JavaScript to Python (Array Processing)
 ======================================================================
 Input: [30, 45, 55, 60, 70, 40, 80, 50]
 Filtered values (>50): [55, 60, 70, 80]
 Average of filtered values: 66.25

 ======================================================================
 TASK 2: Python OOP Class
 ======================================================================
 Created: Student: Alice Johnson, GPA: 3.8, Credits: 120
 Academic Standing: Excellent
 Updated GPA: 3.2
 New Academic Standing: Good

 ======================================================================
 TASK 3: REST API Call - Python Implementation
 ======================================================================
 API Response: {
   "id": 1,
   "name": "John Doe",
   "email": "john@example.com",
   "status": "active"
 }
```

```
 ======================================================================
 TASK 4: SQL to MongoDB Query Translation
 ======================================================================
 SQL Query:
 SELECT category, AVG(price) as average_price
            FROM products
            GROUP BY category
            HAVING AVG(price) > 50
            ORDER BY average_price DESC;

 MongoDB Aggregation Pipeline:
 [
   {
     "$group": {
       "_id": "$category",
       "average_price": {
         "$avg": "$price"
       }
     }
   },
   {
     "$match": {
       "average_price": {
         "$gt": 50
       }
     }
   },
   {
     "$sort": {
       "average_price": -1
     }
   }
 ]

 Key Differences:
   • Grouping: SQL uses GROUP BY; MongoDB uses $group stage
   • Filtering: SQL uses HAVING clause; MongoDB uses $match stage
   • Sorting: Both use ORDER BY / $sort, but MongoDB uses -1 for DESC
   • Document_Structure: MongoDB returns documents; SQL returns tuples
   • Flexibility: MongoDB is more flexible with nested documents
```

```
======================================================================
TASK 5: Binary Search Algorithm - Three Language Implementation
======================================================================
Python Implementation:
  Array: [2, 5, 8, 12, 16, 23, 38, 45, 56, 67]
  Target: 23
  Result: Index 5

Java Implementation:

    public class BinarySearch {
        public static int binarySearch(int[] arr, int target) {
            int left = 0, right = arr.length - 1;

            while (left <= right) {
                int mid = (left + right) / 2;

                if (arr[mid] == target) {
                    return mid;
                } else if (arr[mid] < target) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            return -1;
        }
    }


Go Implementation:

    package main

    func BinarySearch(arr []int, target int) int {
        left, right := 0, len(arr)-1

        for left <= right {
            mid := (left + right) / 2

            if arr[mid] == target {
                return mid
            } else if arr[mid] < target {
                left = mid + 1
            } else {
                right = mid - 1
            }
        }

        return -1
    }

PS C:\Users\krish\OneDrive\Documents\AI\Py.files> & c:\Users\krish\OneDrive\Documents\AI\Py.files\.venv\Scripts\Activate.ps1
(.venv) PS C:\Users\krish\OneDrive\Documents\AI\Py.files>
```