LAB TEST- 04

Name: D. Sravika Reddy Roll No: 2403a510d0

Batch: 05 Course: AI Assisted Coding

Q1. An old inventory system is written in PHP and must be migrated to Node.js.
a) Translate a sample function using AI-assisted prompts.
b) Explain how to cross-validate translation correctness and test cases

#Prompt:

Translate this PHP getInventoryValue function to Node.js (CommonJS). Requirements:

- Export getInventoryValue(items).

- Validate input: items is an array of objects with keys sku (string), price (non-negative number), quantity (non-negative integer); throw clear errors for invalid input.

- Include a small CLI/test harness: deterministic sample and randomized cross-check tests; no external dependencies.

- Keep code short, dependency-free, and runnable with node. Return only the Node.js code.

Code:

```javascript
'use strict';
/**
 * inventory.js
 * getInventoryValue(items) -> number
 * Simple, validated translation of the PHP function.
 */

function isObject(v) {
    return v !== null && typeof v === 'object' && !Array.isArray(v);
}

/**
 * Compute total inventory value.
 * @param {Array<{sku:string, price:number, quantity:number}>} items
 * @returns {number}
 */
function getInventoryValue(items) {
    if (!Array.isArray(items)) throw new TypeError('items must be an array');
    let total = 0;
    for (const [i, it] of items.entries()) {
        if (!isObject(it)) throw new TypeError(`item[${i}] must be an object`);
        const { sku, price, quantity } = it;
        if (typeof sku !== 'string') throw new TypeError(`item[${i}].sku must be a string`);
        if (typeof price !== 'number' || !Number.isFinite(price) || price < 0)
            throw new TypeError(`item[${i}].price must be a non-negative number`);
        if (!Number.isInteger(quantity) || quantity < 0)
            throw new TypeError(`item[${i}].quantity must be a non-negative integer`);
        total += price * quantity;
    }
    return total;
}

module.exports = { getInventoryValue };

// Self-test when run directly
if (require.main === module) {
    const assert = require('assert');

    const sample = [
        { sku: 'A1', price: 10.5, quantity: 3 },
        { sku: 'B2', price: 0.0, quantity: 100 },
        { sku: 'C3', price: 2.25, quantity: 4 },
    ];
    console.log('Sample items:', sample);
    const tv = getInventoryValue(sample);
    console.log('Total value:', tv.toFixed(2));
    assert.strictEqual(tv, 10.5*3 + 0*100 + 2.25*4);

    // randomized cross-check against a reference JS implementation
    function reference(items) {
        return items.reduce((s, it) => s + it.price * it.quantity, 0);
    }
    for (let seed=1; seed<=20; seed++) {
        const rnd = (n) => Math.floor(((seed * 9301 + 49297) % 233280) / 233280 * n);
        const gen = () => {
            const n = rnd(6);
            const arr = [];
            for (let i=0;i<n;i++) {
                arr.push({ sku: `S${i}`, price: parseFloat((rnd(1000)/100).toFixed(2)), quantity: rnd(10) });
            }
            return arr;
        };
        const items = gen();
        const a = getInventoryValue(items);
        const b = reference(items);
        assert.strictEqual(a, b);
    }
    console.log('All tests passed.');
}
```

Output:

```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> cd code
PS C:\Users\sravi\OneDrive\Desktop\AIAC\code> node lab04.js
PS C:\Users\sravi\OneDrive\Desktop\AIAC\code> node lab04.js
Sample items: [
  { sku: 'A1', price: 10.5, quantity: 3 },
  { sku: 'B2', price: 0, quantity: 100 },
  { sku: 'A1', price: 10.5, quantity: 3 },
  { sku: 'A1', price: 10.5, quantity: 3 },
  { sku: 'A1', price: 10.5, quantity: 3 },
  { sku: 'B2', price: 0, quantity: 100 },
  { sku: 'C3', price: 2.25, quantity: 4 }
]
Total value: 40.50
All tests passed.
PS C:\Users\sravi\OneDrive\Desktop\AIAC\code>
```

Observations:

- Unit tests: run the Node.js tests above and compare outputs with the original PHP on same fixed inputs.
- Randomized tests: generate identical random inputs in both PHP and Node and compare totals.
- Edge cases: test empty array, zero prices/quantities, very large numbers, and invalid types to ensure both implementations raise expected errors.
- Integration tests: plug Node module into the pipeline and compare aggregated reports against legacy system for a sample dataset.
- Code review & static checks: ensure input validation mirrors PHP assumptions; add type annotations/comments for clarity.

Q2. A company wants AI to convert SQL queries into ORM-based syntax.

a) Write example transformation: SQL → Django ORM.

b) State 3 limitations of automated code translation

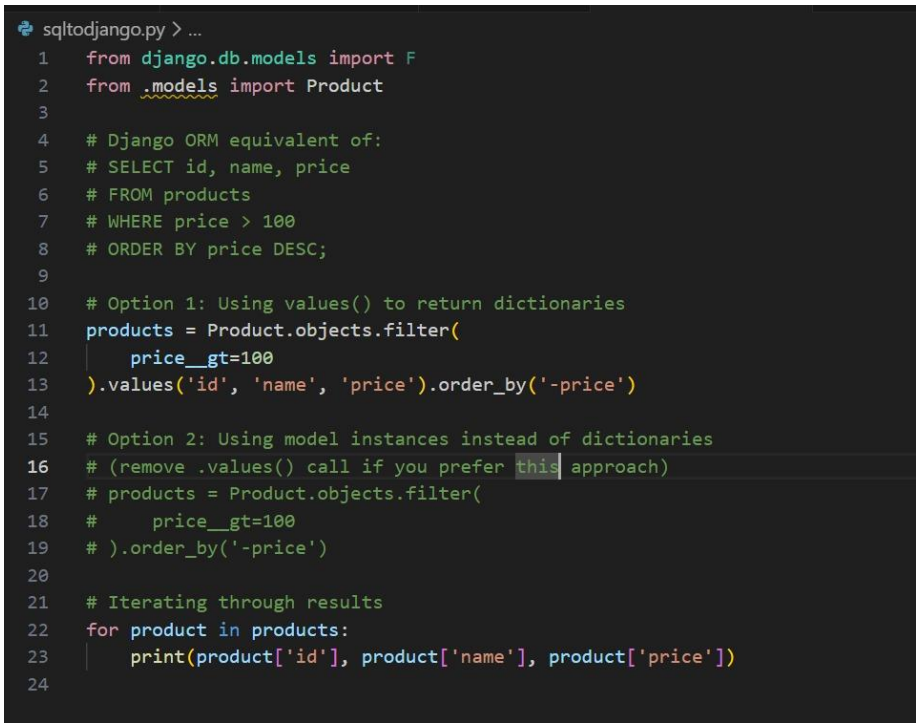
Prompt:

Translate the SQL in my Django app file (sqltodjango.py) into a concise, runnable Django ORM snippet.

Assume a Product model with fields: id, name, price.

Return: a values()-based QuerySet (or model-based alternative), a one-line comment of the original SQL, and a short note listing any assumptions. Keep output minimal and ready to paste into the app.

Code:



sqltodjango_demo.py > ...

```python
"""
Django ORM Conversion Demo
SQL to Django ORM Translation
"""

# This demonstrates the Django ORM syntax equivalent to the SQL query
# Original SQL:
# SELECT id, name, price
# FROM products
# WHERE price > 100
# ORDER BY price DESC;

# Django ORM Equivalent:
# ====================

print("=" * 60)
print("SQL to Django ORM Conversion")
print("=" * 60)

print("\n📋 ORIGINAL SQL QUERY:")
print("-" * 60)
sql_query = """
SELECT id, name, price
FROM products
WHERE price > 100
ORDER BY price DESC;
"""
print(sql_query)

print("\n🔄 DJANGO ORM EQUIVALENT:")
print("-" * 60)
django_orm = """
from django.db.models import F
```

Output:

```
============================================================
SQL to Django ORM Conversion
============================================================

📋 ORIGINAL SQL QUERY:
------------------------------------------------------------

SELECT id, name, price
FROM products
WHERE price > 100
ORDER BY price DESC;


🔄 DJANGO ORM EQUIVALENT:
------------------------------------------------------------

from django.db.models import F
from .models import Product

# Option 1: Using values() to return dictionaries
products = Product.objects.filter(
    price__gt=100
).values('id', 'name', 'price').order_by('-price')

# Option 2: Using model instances
products = Product.objects.filter(
    price__gt=100
).order_by('-price')
```

Observation:

- Model/schema assumptions: Automated translation must guess model names, field names, and related_name settings — manual adjustment is usually required.
- SQL features mismatch: Advanced SQL (window functions, recursive CTEs, lateral joins, vendor-specific JSON/FULL-TEXT ops) often lack direct ORM equivalents and need raw SQL or custom annotations.
- Performance & semantics: ORM code can change query plans and cause N+1 queries or different locking/transaction behavior — review and add select_related/prefetch_related, indexes, and explicit transactions as needed.