**1. Finance: Option Pricing (Black-Scholes Model)**

This snippet calculates the price of a European call option using the Black-Scholes formula. It involves mathematical functions and statistical concepts.

**Business Relevance:** Core concept in financial derivatives pricing, risk management, and investment strategies.

**Code Snippet:**

Python

import numpy as np

from scipy.stats import norm

def black\_scholes\_call(S, K, T, r, sigma):

"""

Calculates the Black-Scholes price for a European call option.

S: Current stock price

K: Strike price

T: Time to maturity (in years)

r: Risk-free interest rate (annual)

sigma: Volatility of the stock (annual)

"""

if S <= 0 or K <= 0 or T <= 0 or sigma <= 0:

return 0.0 # Basic handling for invalid inputs

# Calculate d1 and d2

d1 = (np.log(S / K) + (r + 0.5 \* sigma \*\* 2) \* T) / (sigma \* np.sqrt(T))

d2 = d1 - sigma \* np.sqrt(T)

# Calculate N(d1) and N(d2) - cumulative standard normal distribution

N\_d1 = norm.cdf(d1)

N\_d2 = norm.cdf(d2)

# Calculate call option price

call\_price = (S \* N\_d1 - K \* np.exp(-r \* T) \* N\_d2)

return call\_price

# --- Example Usage ---

stock\_price = 100 # Current price

strike\_price = 105 # Price at which we can buy

time\_to\_expiry = 1 # 1 year

risk\_free\_rate = 0.05 # 5%

volatility = 0.2 # 20%

price = black\_scholes\_call(stock\_price, strike\_price, time\_to\_expiry, risk\_free\_rate, volatility)

print(f"The estimated call option price is: ${price:.2f}")

# Example with a potential issue (T=0)

price\_at\_expiry = black\_scholes\_call(stock\_price, strike\_price, 0, risk\_free\_rate, volatility)

# print(f"Price at expiry (T=0) returns: ${price\_at\_expiry:.2f}") # This will cause a division by zero without the initial check!

**Why it's good for demo:**

* **Explain:** The model can explain the Black-Scholes formula, its components (d1​,d2​), the use of scipy.stats.norm, and the overall financial concept.
* **Refactor:** Variable names could be more descriptive (S -> current\_asset\_price), error handling could be more robust (raising ValueError instead of returning 0), and the d1​/d2​ calculation could be broken down.
* **Debug:** If you remove the initial if check, passing T=0 causes a ZeroDivisionError. The model should identify this and suggest a fix. You could also introduce a typo in the formula.

**2. Logistics: Simple Greedy TSP Route Finder**

This snippet attempts to find an approximate solution to the Traveling Salesperson Problem (TSP) using a greedy "nearest neighbor" approach.

**Business Relevance:** Route optimization, delivery planning, supply chain management, cost reduction.

**Code Snippet:**

Python

import math

def calculate\_distance(point1, point2):

"""Calculates Euclidean distance between two points (x, y)."""

return math.sqrt((point1[0] - point2[0])\*\*2 + (point1[1] - point2[1])\*\*2)

def find\_shortest\_route\_greedy(locations):

"""

Finds an approximate shortest route using the nearest neighbor heuristic.

Starts at the first location and always goes to the nearest unvisited one.

"""

if not locations:

return []

num\_locations = len(locations)

unvisited = list(locations.keys())

route = []

current\_location\_name = unvisited[0] # Always start at the first one

route.append(current\_location\_name)

unvisited.remove(current\_location\_name)

while unvisited:

current\_coords = locations[current\_location\_name]

nearest\_dist = float('inf')

nearest\_location = None

for next\_location in unvisited:

dist = calculate\_distance(current\_coords, locations[next\_location])

if dist < nearest\_dist:

nearest\_dist = dist

nearest\_location = next\_location

if nearest\_location:

current\_location\_name = nearest\_location

route.append(current\_location\_name)

unvisited.remove(current\_location\_name)

else:

break # Should not happen if all are reachable

# It doesn't return to the start - a potential 'bug' or 'feature' to discuss

return route

# --- Example Usage ---

delivery\_points = {

"Warehouse A": (0, 0),

"Customer 1": (1, 5),

"Customer 2": (8, 2),

"Pickup X": (3, 9),

"Customer 3": (10, 1),

"Warehouse B": (5, 5) # Maybe the end point?

}

path = find\_shortest\_route\_greedy(delivery\_points)

print(f"Greedy Route Found: {' -> '.join(path)}")

**Why it's good for demo:**

* **Explain:** The model can explain the TSP concept, the greedy algorithm approach, its limitations (not guaranteed optimal), and the distance calculation.
* **Refactor:** Could use a more efficient data structure, add a check for returning to the start, use Haversine for real-world distances (if coords were lat/lon), improve variable names.
* **Debug:** What if locations is empty? What if a location isn't removed correctly (infinite loop)? What if the distance calculation is wrong? Does it handle a single location?

**3. Web Scraping: Basic Stock Price with Selenium**

This snippet uses Selenium to open a browser, go to Yahoo Finance, and try to extract the current price of a stock. **Note:** This requires selenium and a webdriver (like chromedriver) to be installed and configured. Web scraping is fragile as websites change.

**Business Relevance:** Competitive intelligence, price monitoring, market data collection, sentiment analysis (if scraping news).

**Code Snippet:**

Python

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

import time

def get\_stock\_price(ticker="AAPL"):

"""

Attempts to scrape the current stock price from Yahoo Finance using Selenium.

NOTE: Web selectors can break often! This is for demo purposes.

"""

price = "Not found"

driver = None # Initialize to None

try:

# Setup - requires chromedriver or another webdriver

options = webdriver.ChromeOptions()

options.add\_argument('--headless') # Run without opening a visible browser window

options.add\_argument('--no-sandbox')

options.add\_argument('--disable-dev-shm-usage')

driver = webdriver.Chrome(options=options)

driver.get(f"https://finance.yahoo.com/quote/{ticker}")

# --- This selector is an EXAMPLE and LIKELY TO BREAK ---

# It's better to find a more robust one or handle cookie popups etc.

# This one targets the main price element often found on Yahoo Finance.

price\_element\_selector = f'[data-symbol="{ticker}"][data-field="regularMarketPrice"]'

# Wait for the element to be present (max 10 seconds)

wait = WebDriverWait(driver, 10)

price\_element = wait.until(

EC.presence\_of\_element\_located((By.CSS\_SELECTOR, price\_element\_selector))

)

price = price\_element.text

except Exception as e:

print(f"An error occurred: {e}")

# Poor error handling - just prints, doesn't specify \*what\* failed.

price = "Error"

finally:

if driver:

driver.quit() # Always close the browser

return price

# --- Example Usage ---

apple\_price = get\_stock\_price("AAPL")

print(f"Apple (AAPL) Price: {apple\_price}")

google\_price = get\_stock\_price("GOOGL")

print(f"Google (GOOGL) Price: {google\_price}")

**Why it's good for demo:**

* **Explain:** The model can explain web scraping, Selenium, WebDrivers, CSS selectors, explicit waits, and the try...except...finally block.
* **Refactor:** Improve error handling (more specific exceptions, retries), make selectors more robust (if possible), handle cookie pop-ups/consent banners, abstract the driver setup.
* **Debug:** The *most likely* issue is the CSS selector being outdated. The model could be asked to "debug" this, potentially suggesting ways to find new selectors or alternative approaches. It could also find issues with missing webdriver or selenium installations (though it can't fix *that*).

**4. Data Analysis: Simple Sales Aggregation (Pandas)**

This snippet uses Pandas to perform a basic sales data aggregation.

**Business Relevance:** Sales reporting, business intelligence, performance analysis, data-driven decision-making.

**Code Snippet:**

Python

import pandas as pd

def analyze\_sales(data):

"""

Takes a list of sales data (dictionaries) and calculates total sales per region.

"""

if not data:

return {}

df = pd.DataFrame(data)

# Check for required columns

if 'Region' not in df.columns or 'Sales' not in df.columns:

print("Error: Data must contain 'Region' and 'Sales' columns.")

return {}

# Inefficient way - using a loop (Good for refactoring!)

regional\_sales = {}

for index, row in df.iterrows():

region = row['Region']

sales = row['Sales']

if region in regional\_sales:

regional\_sales[region] += sales

else:

regional\_sales[region] = sales

# A better way (commented out - can be used as a hint or refactor target)

# regional\_sales = df.groupby('Region')['Sales'].sum().to\_dict()

return regional\_sales

# --- Example Usage ---

sales\_records = [

{'OrderID': 101, 'Region': 'North', 'Sales': 1500, 'Product': 'A'},

{'OrderID': 102, 'Region': 'South', 'Sales': 800, 'Product': 'B'},

{'OrderID': 103, 'Region': 'North', 'Sales': 1200, 'Product': 'C'},

{'OrderID': 104, 'Region': 'East', 'Sales': 2200, 'Product': 'A'},

{'OrderID': 105, 'Region': 'South', 'Sales': 950, 'Product': 'C'},

{'OrderID': 106, 'Region': 'North', 'Sales': 300, 'Product': 'B'},

]

analysis = analyze\_sales(sales\_records)

print("Total Sales per Region:")

for region, total in analysis.items():

print(f"- {region}: ${total}")

**Why it's good for demo:**

* **Explain:** The model can explain Pandas DataFrames, the iterrows method (and why it's often discouraged), and the concept of groupby.
* **Refactor:** The most obvious refactoring is to replace the iterrows loop with the much more efficient and "Pythonic" df.groupby('Region')['Sales'].sum(). It can also suggest adding input validation or better error handling.
* **Debug:** What if 'Sales' isn't a number? What if the input data is malformed? What if Region or Sales columns are missing? (The code has a basic check, but it could be improved).