System Requirements Specification Index

For

Financial Analysis System

Version 1.0

IIHT Pvt. Ltd.
fullstack@iiht.com

TABLE OF CONTENTS

Financial Analysis System
System Requirements Specification

1 PROJECT ABSTRACT

Prestige Financial Advisors needs a specialized system for analyzing investment portfolios and financial data. This assignment focuses on implementing functions with appropriate scope management, various argument types, and effective return value handling. Each function will handle specific financial analysis tasks, demonstrating proper variable scope, parameter passing, and structured data returns.

2 BUSINESS REQUIREMENTS:

Screen Name	Console input screen
Problem Statement	1. System must utilize proper function scope management for financial calculations 2. Code must demonstrate various argument passing methods and parameter types 3. System must implement appropriate return value techniques and formats 4. System must perform operations on Investment portfolio data and stock analysis, Risk assessment and return projections, Budget tracking and expense categorization, Financial report generation and formatting

3 CONSTRAINTS

3.1 INPUT REQUIREMENTS

1. Portfolio Data:
    - Input list must contain at least 5 stock dictionaries
    - Each stock must have: ticker, shares, purchase_price, current_price, sector

- Example: `{"ticker": "AAPL", "shares": 10, "purchase_price": 150.0, "current_price": 175.0, "sector": "Technology"}`

2. Transaction Data:
   - Input list must contain at least 10 transaction dictionaries
   - Each transaction must have: date, type, amount, category
   - Example: `{"date": "2023-06-15", "type": "expense", "amount": 125.50, "category": "Utilities"}`
3. Financial Goals:
   - Input list must contain at least 3 financial goal dictionaries
   - Each goal must have: name, target_amount, deadline, priority, current_amount
   - Example: `{"name": "Emergency Fund", "target_amount": 10000, "deadline": "2023-12-31", "priority": "high", "current_amount": 6500}`
4. Market Data:
   - Input data must include historical price information
   - Must include risk-free rate and market benchmark returns
   - Example: `{"risk_free_rate": 0.03, "market_return": 0.08, "volatility": 0.15}`

3.2 FUNCTION SCOPE REQUIREMENTS

1. Variable Scope Management:
   - Must use local variables within functions for calculation steps
   - Must demonstrate proper handling of function scope vs. global scope
   - Must use nonlocal variables in at least one nested function
   - Example: `def analyze_portfolio(portfolio): local_total = sum(stock["value"] for stock in portfolio)`
2. Function Nesting:
   - Must implement at least one function that contains nested functions
   - Must demonstrate closure technique for financial calculation
   - Must maintain proper scope hierarchy
   - Example: `def create_projection_calculator(rate): def calculate(principal): return principal * (1 + rate) ** years`
3. Scope Best Practices:
   - Function parameters must be used instead of relying on global variables

- Results must be returned rather than modifying global state
- Must demonstrate proper namespace management
- Must avoid unnecessary global variables

3.3 Argument Constraints

1. Positional Arguments:
    - Must implement functions with required positional arguments
    - Must demonstrate correct positional argument order
    - Example: `def calculate_roi(initial_investment, final_value, years)`
2. Keyword Arguments:
    - Must implement functions with keyword arguments and defaults
    - Must demonstrate optional parameters with sensible defaults
    - Example: `def analyze_stock(ticker, period="1y", interval="1mo", risk_tolerance=0.05)`
3. Variable Arguments:
    - Must use *args to handle variable number of financial data points
    - Must use **kwargs to handle variable configuration options
    - Example: `def calculate_statistics(*data_points, **options)`
4. Advanced Argument Handling:
    - Must implement at least one function using keyword-only arguments
    - Must implement at least one function using position-only arguments
    - Must demonstrate unpacking lists and dictionaries into function arguments
    - Example: `def generate_report(data, *, format="detailed", output="screen")`

3.4 Return Value Constraints

1. Basic Return Values:
    - Functions must return appropriate data types (numbers, strings, booleans)
    - Must demonstrate early returns for special cases
    - Example: `return total_value if total_value > 0 else 0`
2. Compound Return Values:
    - Must return tuples for related multiple values

- Must return dictionaries for named result sets
- Must return lists for collections of similar results
- Example: `return (expected_return, volatility, sharpe_ratio)`

3. Generator Functions:

- Must implement at least one generator function using yield
- Must demonstrate lazy evaluation for financial data processing
- Example: `def analyze_monthly_returns(data): for month in data: yield calculate_return(month)`

4. Return Value Handling:

- Must demonstrate unpacking returned tuples
- Must demonstrate accessing returned dictionary values
- Must implement proper error handling for return values
- Example: `return_value, risk, ratio = analyze_investment(stock_data)`

3.5 Output Constraints

1. Display Format:

- Each analysis result must have descriptive labels
- Financial metrics must be formatted with appropriate precision
- Monetary values must include currency symbols
- Percentages must include % symbol
- Example: `"Investment Return: $1,250.00 (12.5%)"`

3. Output Format:

- Portfolio performance summary
- Risk analysis breakdown
- Budget category analysis
- Financial goal progress
- Investment recommendations

4. Template code Structure:

1. Portfolio Analysis Functions:

- `calculate_portfolio_value(stocks)` - calculates current portfolio value
- `analyze_portfolio_performance(stocks, *, period="1y")` - analyzes gains/losses
- `calculate_sector_allocation(stocks)` - calculates percentage by sector

- `create_diversification_calculator(risk_profile)` - returns a diversification function

2. Risk Assessment Functions:

- `calculate_volatility(historical_prices)` - calculates price volatility
- `calculate_risk_metrics(returns, risk_free_rate=0.03)` - calculates Sharpe ratio
- `analyze_risk_return(stocks, market_data)` - analyzes risk vs. return
- `generate_risk_report(**options)` - generates risk report with options

3. Budget Analysis Functions:

- `categorize_transactions(*transactions)` - sorts transactions by category
- `analyze_spending_trends(transactions, start_date, end_date)` - finds spending trends
- `calculate_budget_variance(actual, budget)` - calculates budget differences
- `generate_savings_projection(income, expenses, years, savings_rate=0.2)` - projects savings

4. Report Generation Functions:

- `format_currency(amount)` - formats numbers as currency
- `format_percentage(value)` - formats numbers as percentages
- `generate_financial_summary(portfolio, transactions, goals)` - creates overall summary
- `monthly_performance_generator(data)` - yields monthly performance metrics

5. Main Program Function:

- `main()` - main program function executing all demonstrations
- Must execute each function with appropriate arguments
- Must handle return values appropriately
- Must present results in structured format

5. EXECUTION STEPS TO FOLLOW:

1. Run the program
2. Observe portfolio analysis with different argument passing techniques
3. See risk assessment calculations with various return value formats
4. View budget analysis using nested functions and proper scoping
5. Examine the generator function for monthly performance data
6. Observe function closures in action with the diversification calculator
7. See comprehensive financial report generation combining all calculations