Week 1 Deliverable: AI-Assisted Financial Calculator Module in Python

Overview

Students will develop a comprehensive financial calculator module in Python using AI coding assistants (Cline or any other IDE extension). This exercise introduces collaborative AI-driven development while building practical financial computation skills using Python's numerical libraries.

SECTION 1: Project Setup & Structure

Implementation Order: Start Here

1.1 Project Directory Structure

Create the following directory structure:

```
Unset
financial_calculator/
├── __init__.py
├── compound_interest.py
├── loan_calculator.py
├── investment_projections.py
├── utils/
│   ├── __init__.py
│   ├── validators.py
│   └── formatters.py
├── tests/
│   ├── test_compound_interest.py
│   ├── test_loan_calculator.py
│   └── test_investment_projections.py
└── examples/
```

```
    └── usage_examples.py
```

1.2 Environment Setup

- Create `requirements.txt` with dependencies:
  - NumPy for calculations
  - Pandas for data structures
  - matplotlib/seaborn for optional visualizations
  - pytest and pytest-cov for testing
  - pydantic for input validation

1.3 Initial AI Collaboration Setup

- Start your `development_log.md` file
- Document your first AI interaction session

SECTION 2: Core Utility Functions

Implementation Order: Build Foundation First

2.1 Input Validation (`utils/validators.py`)

- Create custom validators or use pydantic
- Implement validation for financial inputs (positive amounts, valid rates, etc.)
- Create custom exception classes for financial calculation errors

2.2 Output Formatting (`utils/formatters.py`)

- Currency formatting functions
- Percentage formatting
- Data structure formatting for results

SECTION 3: Financial Functions Implementation

Implementation Order: Core Business Logic

3.1 Compound Interest Calculator (`compound_interest.py`)

Required Functions:

```python
def calculate_compound_interest(principal: float, rate: float, time: float, n: int = 12) -> dict[str, float]:
    """Calculate compound interest with various compounding frequencies"""
```

Implementation Requirements:

- Support for annual (n=1), semi-annual (n=2), quarterly (n=4), monthly (n=12), and daily (n=365) compounding
- Handle both future value and present value calculations
- Include continuous compounding option using `math.e`

Minimum Required Functions:

- `calculate_future_value()`
- `calculate_present_value()`
- `calculate_continuous_compound()`
- `calculate_effective_annual_rate()`

3.2 Loan Payment Calculator (`loan_calculator.py`)

```python
def calculate_loan_payment(principal: float, annual_rate: float, years: int):
    """Calculate monthly payment for fixed-rate loans"""
```

Implementation Requirements:

- Calculate monthly payments using the amortization formula
- Generate amortization schedules as pandas DataFrames
- Support extra payment scenarios and early payoff calculations
- Return payment breakdowns (principal vs. interest over time)

Minimum Required Functions:

- `calculate_monthly_payment()`
- `generate_amortization_schedule()`
- `calculate_total_interest()`
- `calculate_payoff_time_with_extra_payments()`

3.3 Investment Projection Tools (`investment_projections.py`)

```python
def project_investment_growth(initial: float, monthly_contrib: float, annual_return: float, years: int):
    """Project investment growth with regular contributions"""
```

Implementation Requirements:

- Dollar-cost averaging calculator with variable contribution schedules
- Portfolio growth projections with adjustable return rates
- Retirement savings calculator with inflation adjustments
- Support for varying market conditions (bear/bull market scenarios)

Minimum Required Functions:

- `project_investment_growth()`
- `calculate_dollar_cost_average_returns()`
- `project_retirement_savings()`
- `simulate_portfolio_scenarios()`

SECTION 4: Documentation Standards

Implementation Order: Document As You Code

4.1 Code Documentation Requirements

- **Type hints** for all functions using Python 3.9+ syntax
- **Docstrings** following Google or NumPy style

Example Documentation Format:

Python

```
def calculate_compound_interest(
    principal: float,
    rate: float,
    time: float,
    n: int = 12
) -> dict[str, float]:
    """
    Calculate compound interest with various compounding
frequencies.

    Args:
        principal: Initial investment amount
        rate: Annual interest rate as decimal (e.g., 0.05 for 5%)
        time: Investment period in years
        n: Compounding frequency per year (default: 12 for
monthly)

    Returns:
        Dictionary containing:
        - 'future_value': Final amount after interest
        - 'interest_earned': Total interest earned
        - 'effective_rate': Effective annual rate

    Raises:
        ValueError: If principal, rate, or time are negative

    Example:
        >>> calculate_compound_interest(1000, 0.05, 10, 12)
        {'future_value': 1647.01, 'interest_earned': 647.01, ...}
    """
```

4.2 Project Documentation

- **README.md** including:
  - Installation instructions with `requirements.txt`
  - Quick start guide with code snippets
  - API reference linking to detailed documentation

SECTION 5: Testing Implementation

Implementation Order: Test After Each Module

5.1 Testing Framework Setup

- Use pytest framework with fixtures for common test data
- Target test coverage of at least 80% using `pytest-cov`

5.2 Test Categories

- **Unit tests** for individual functions
- **Property-based tests** using `hypothesis` for mathematical properties
- **Integration tests** for combined calculations
- **Performance tests** for large-scale calculations

5.3 Example Test Structure

Python

```
def test_compound_interest_positive_values():
    """Test compound interest with typical positive values"""

def test_compound_interest_edge_cases():
    """Test with zero rates, negative values, extreme inputs"""

@pytest.mark.parametrize("principal,rate,time,expected", [...])
def test_compound_interest_known_values(principal, rate, time,
expected):
    """Test against known financial calculations"""
```

SECTION 6: AI Collaboration Documentation

Implementation Order: Continuous Throughout Development

6.1 Development Log Requirements (`development_log.md`)

Document the following throughout your development process:

- Session-by-session AI tool usage
- Effective prompts for generating financial formulas
- Examples of AI-suggested optimizations
- Debugging assistance from AI tools

6.2 AI-Generated Code Markers

Mark AI-assisted code sections:

```python
# AI-assisted: Claude suggested this optimization
# Original approach used nested loops, AI suggested vectorized NumPy operation
future_values = principal * (1 + rate/n) ** (n * time_periods)
```

SECTION 7: Quality Assurance

Implementation Order: Before Final Submission

7.1 Code Quality Standards

- **Error handling** with custom exception classes
- Effective use of Python idioms and features
- Modular, reusable design

7.2 Performance Considerations

- Handle large datasets efficiently
- Use vectorized NumPy operations where possible
- Optimize for common use cases

SECTION 8: Bonus Challenges

Implementation Order: Optional Extensions

8.1 Command Line Interface

- CLI Interface using `argparse` or `click`

```shell
python -m financial_calculator compound --principal 1000 --rate 0.05 --time 10
```

8.2 Data Visualization

- **matplotlib visualizations:**
  - Loan amortization charts
  - Investment growth curves
  - Comparison plots for different scenarios

8.3 Advanced Features

- **Web API** using FastAPI for calculator endpoints
- **Excel Export** functionality using `openpyxl`
- **Monte Carlo Simulation** for investment risk analysis

Evaluation Criteria

Category	Weight	Focus Areas
Functionality & Accuracy	30%	Correct financial formulas, edge case handling, performance
Code Quality	20%	PEP 8 compliance, Python idioms, modular design
Documentation	20%	Clear docstrings, comprehensive README, inline comments
Testing	20%	Coverage metrics, test quality, testing strategy
AI Collaboration	10%	Thoughtful AI usage, iterative improvement, critical evaluation

---

Submission Requirements

Final Deliverables:

1. **GitHub repository** with clean commit history showing AI collaboration

2. **requirements.txt** with all dependencies
3. **Running instructions** in README
4. **Development log** as markdown file
5. **Optional:** Jupyter notebook demonstrating usage