LLM Comparison

Which tool(s) did you use?

I used an LLM, Claude AI.

If you used an LLM, what was your prompt to the LLM?

My prompt:
Build a Python n-branch family tree using three classes: Person, PersonFactory, and FamilyTree.

Person Class

The Person class is responsible for keeping details of each simulated person in the model.

Each member of the family tree is an instance of a Person class.

Create a Person class with the following attributes:

- first_name
- last_name
- Year_born
- year_died
- gender (M or F)
- spouse
- children

Rules and Constraints

- The first two people must be born in 1900, and they will propagate the family tree.
- the tree stops generating when no more children can be born before the year 2200.
- Children are born between ages 20 and 40 of the elder parent, distributed evenly
- FamilyTree generates all children with predetermined birth years
- The number of children is determined by the parent's birth decade, using the expected value from the birth rate data +/- 2 children, but randomly generated
- Single parents have one fewer child than coupled parents
- Spouses who marry into the family keep their own randomly-selected last name
- Spouses are born within +/- 5 years of the person they marry
- First names are selected by frequency-weighted random choice based on the person's birth year and gender
- Last names are selected by frequency-weighted random choice from the last names dataset

PersonFactory Class

Create a PersonFactory class that reads data from CSV files and generates Person instances.

Load data from five CSV files. Assume csv files are in the repo.

- first_names: year, name, gender, frequency
- last_names.csv with columns: last_name, frequency
- Life_expectancy.csv with columns: year, marriage_rate
- marriage_rates.csv with columns: year, marriage_rate
- birth_rates.csv with columns: year, num_children, probability

FamilyTree Class

Create a FamilyTree class using PersonFactory. This will simulate the growing family tree by generating descendants generation by generation.

Once the family tree has been generated, you will allow the user to query or interact with the tree, with the following queries:

- Total number of people in the tree
- Total number of people in the tree by year
- All Duplicate names

Please also list any suggestions or differences in design you would make if you were to add other functionalities.

What differences are there between your implementation and the LLM's?

File reading and data processing: My implementation used pandas and numpy for CSV loading. The LLM used Python's built-in open() and readline() methods to manually parse the CSV files.

Weighted Probability choices: My implementation used mostly numpy's random methods (e.g. np.random.choice with probability weights) for weighted sampling, which integrates naturally with the pandas. The LLM relied primarily on Python's standard random library with manual weighted-choice logic.

Person class methods: The LLM included several additional methods on Person beyond basic attributes: get_age_in_year(year) to calculate a person's age at any point in the simulation, and standard developer utility methods (__repr__, __str__, etc.) for easier debugging and display.

Class initialization and parameters: My FamilyTree class is self-contained with the tree generation logic lives entirely within it, and the constructor only takes max_year. The LLM created the two founding ancestors externally in main() and passed them as constructor arguments to FamilyTree.

Queue: For the tree generation, the LLM used a list to simulate a queue, removing elements from the list head. My implementation used the deque class (queue) for processing the tree.

Tree storage structure: My implementation uses lists to store generated people and their children, and converts the list to a dictionary for the menu lookups. The LLM used a dictionary keyed by generation number, with each Person object maintaining its own children sublist.

Tree termination logic: The LLM maintained a current_year tracker updated by simulating the age of the oldest person currently in the queue. Generation stopped when current_year + minimum_childbearing_age exceeded max_year. My approach is more direct: I compute the birth years of potential children upfront and simply discard any that fall after max_year.

What changes would you make based on the LLM's suggestions?

The most valuable change I may adopt is upgrading the storage from a list to a dictionary keyed by generation number, mapping each generation to its members. This would make generation-level queries direct lookups rather than full list traversals, and it can be segmented further by birth decade, which is relevant to one of the menu items.
Two additional changes worth considering: adding get_age_in_year to Person is useful for any time-based queries for additional features (who was alive in 2XXX, who was of childbearing age in a given decade), and the __repr__/__str__ methods, which may be good for debugging.

What changes would you refuse to make?

I will not use the LLM's file reading approach: using open() and readline() offers no advantage over pandas for this assignment, as it is more code, less readable.

I will also not use the termination logic using a current_year tracker. Directly comparing child birth_years to max_year is simpler.

I will keep my queue instead of using a list that mimics a queue. I think that having a list that mimics a queue is inefficient, as each deletion requires shifting all the other elements left.

Finally, I will not adopt the FamilyTree constructor signature taking two Person objects as parameters would not be adopted. Initializing the founders inside the class keeps the interface cleaner.