*This file contains some notes describing certain aspects of this **Asset Backed Security** model project.*

This project was built in 2 main sections: First the loan folder, then the Tranches folder.

Some testing files are present throughout the Root project file that test certain elements of the code.

This is an ongoing project and I hope to continue make improvements in the future, correcting any errors I may find along the way as well as expanding this project to implement additional functionality.

- Part 1: The loan folder

  The loan folder contains 8 python scripts that build off of each other. The order they were created in is as follows:

  1. loan_base.py
     - This script builds the Loan base class. Many of the classes in this folder are derived from this class. This script contains functionality for all types of loans, which are initialized with asset type (determined from derived classes), face value, rate (fixed or variable rate), and start/end dates of the loan.
     - This class includes getter/setter properties, as well as instance methods for generic payment formulas (including annuity formulas, recursive definitions, etc), class methods that are invoked in some of the calculations, and static methods for some rate conversions.
     - It also contains a memoize decorator I've written at the top of the file. Its primary purpose (so far) is to store results from long computations, namely the recursive definitions.
  2. loan_pool.py
     - This script builds the LoanPool class, which is a composition of Loan objects. It contains methods that apply operations on the pool of loans, including WAR (weighted average return) and WAM (weighted average maturity) methods, as well as some functionality to determine loan defaults based on assigned default probabilities towards the end of the file.
  3. loans.py
     - This script contains derived classes to help define both fixed and variable rate loans. These classes override the rate method in the base class.
  4. mortgage.py
     - This script creates a MortgageMixin class, which contains some mortgage functionality that can be applied on House objects (see next few files). It also defines the derived AutoLoan class, which currently handles fixed rate loans.
  5. asset_base.py
     - This defines the Asset base class, which allows an Asset object to be initialized with an initial value.
  6. car_class.py
     - This script derives the Car class from the Asset base class. It contains a list of car models that can be added to (some dummy interest rates were assigned at random to each model; these can be updated).
  7. house_base.py
     - This script derives the HouseBase class from the Asset class.
  8. house_derived_classes.py
     - This script adds functionality to the HouseBase class by further specifying the asset type.

  A file called new_main.py is also contained in this loan folder, which demonstrates some functionality of the above scripts.

- Part 2: The Tranches folder

  The first section of this folder defines classes to add tranche-related functionality. The second section implements some loan functions as well to implement a payment waterfall throughout periods by tranche.

  This folder also contains a "few tests" folder, to test some of the methods written in this section.

  1. base_tranche_class.py
     - This script defines the base Tranche class. A Tranche object is initialized with a notional, a rate, and a subordination flag. Tranches are sorted lexicographically, and paid out by subordination level. See all scripts for details.
  2. standard_tranche_class.py
     - This script derives the StandardTranche class from the base Tranche class. It also initializes some dictionaries used to store payment information for the upcoming StructuredSecurities class.
  3. structured_securities_class.py
     - Much like the LoanPool class, this StructuredSecurities class is a composition of Tranche objects. It contains methods that apply operations to the pool of StructuredSecurities objects.
  4. waterfall_function.py
     - This script defines the doWaterfall function. It creates a payment waterfall, displaying Interest Due, Interest Paid, Interest Shortfall, Principal Paid, and Balance, for each period (as a row), and for each tranche (side by side).
       After the waterfall, the IRR, AL and DIRR are displayed for each tranche.
       This function also displays the number of defaulted loans, and the total recovery from these loans since not all loans default at the same time, and recovery value is period dependent (see loan_base.py).

- Part 3: runMonte function

  The goal of this function is to "correct" the tranche rates to find what values most accurately reflect the underlying financial metrics (AL, and DIRR specifically in this code) of the given loan pool that gets passed into this function. Details can be found in run_monte.py, in the root folder of this project, but the general process is described below.

  1. The simulateWaterfall function runs a simulation some specified number of times (Monte Carlo simulation) to determine the average AL (or the WAL, weighted average life) and the average DIRR (reduction in yield). This is because this code is set up to randomly default some loans in the pool based on some assigned probability, so we run the Monte Carlo simulation to predict these parameters with greater accuracy.
  2. Next, we keep adjusting the tranche rates to their "correct" value using a yield formula, where coefficients are assigned based on tranche type:

$$\text{yield} = \frac{\frac{7}{1+0.08e^{-0.19(\frac{\text{WAL}}{12})}} + 0.019\sqrt{\left(\frac{\text{WAL}}{12}\right)(\text{DIRR} * 100)}}{100} \tag{1}$$

  3. Next, the new rate can be calculated as follows:

$$\text{newRate} = \text{oldRate} + \text{coeff} * (\text{yield} - \text{oldRate}) \tag{2}$$

  4. The loop breaks, and we consider the adjusted rates to be correct when the following condition is met:

$$\frac{\sum_{\text{tranches}} \text{trancheNotional} * \left|\frac{\text{oldRate}-\text{newRate}}{\text{oldRate}}\right|}{\text{totalNotional}} < \text{tolerance} \tag{3}$$

Future implementations and necessary corrections:

- Currently, there are two fixes that need to be made. There is a bug causing principal payments from the makePayments method in the StructuredSecurities class to not be correctly registered. Also, the timeIncrease method needs to be fixed for the waterfall to display all periods correctly. All subsequent code has been tested separately, and functions as required (i.e. outputting metrics and yield curve).
- Once fixed, I hope to add more payout methods on top of the currently implemented Sequential and Pro Rata modes.
- I want to make the model more comprehensive by including more Asset types, which would involve initializing more models and specific interest rates.
- Any other additions I think of will also be implemented when possible.