Exercises

Level 2: Object Oriented Programming

Classes should be in well-structured code files. Every class in its own .py file, and use Python package structure as covered in level 1. The expectation is something like this for this level:

- Base directory containing main() file.
    - **utils** subdirectory containing timer.py with your Timer class
    - **loan** subdirectory containing loan_base.py, loan_pool.py, mortgage.py, auto_loan.py, etc.
    - **asset** subdirectory
    - etc.

Going forward, you should make the determination on how to best structure your code (and code structure can affect your grade up to 10 points per exercise).

2.1: Classes

Timer

1) We've been using **time.time** before and after code blocks to report the difference as the 'time taken'. This exercise is to generalize and encapsulate this into a class, to make things cleaner and re-usable. The steps are as follows:

a. Create a class called **Timer**.

b. Add a **start** method and **end** method. They should work as follows:

```
t = Timer()
t.start()
# Lots of code here
t.end() # This should stop the timer and print the time taken.
```

c. Note that **start** should give an error if the Timer is already started and **end** should give an error if the Timer is not currently running.

d. Add a method to retrieve the last timer result.

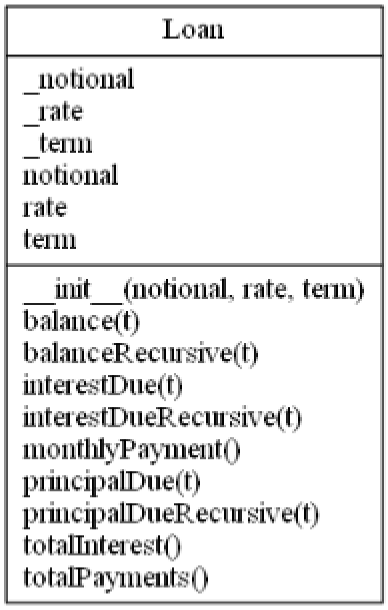
e. Add the ability to configure the Timer to display either seconds, minutes, or hours. The timer result (i.e. from **end** and **retrieveLastResult**) should use whatever the current configuration is.

f. Test your class thoroughly.

This is obviously cleaner than the previous approach of subtracting times everywhere. The remaining downside to this class is that one must still explicitly invoke t.start() and t.end() around the code one wishes to time. We will remedy this when we extend this class using context managers and decorators (see Levels 3 and 5) to make things even cleaner syntactically.

Loan Modeling 101

The following is the class diagram, which is built up over the exercises in this section:



Each exercise should be a separate PyCharm project, with a main() that tests the new/affected functionality for that exercise; each one incrementally builds up the suite of classes.

2) Create a basic Loan class exactly as demonstrated in the lecture (including the setter/getter property methods). Then, extend it with methods that return the following (refer to the slides for any necessary formulas):

- **a.** The monthly payment amount of the Loan (**monthlyPayment**). Even though **monthlyPayment** is likely to be equal for all months, you should still define this with a dummy 'period' parameter, since it's possible some loan types will have a monthly payment dependent on the period.
- **b.** The total payments over the entire Loan (**totalPayments**). This is principal plus interest.
- **c.** The total interest over the entire Loan (**totalInterest)**.

3) Interest due at time T on a loan depends on the outstanding balance. Principal due is the monthly payment less the interest due. Conceptually, these are recursive calculations as one can determine the interest/principal due at time T if one knows the balance at time T-1 (which, in turn, can be determined if one knows the balance at time T-2).

For each of the below functions, implement two versions: A recursive version (per the above statement) and a version that *uses the formulas provided in the slides*:

- **a.** The interest amount due at a given period (**interestDue**).
- **b.** The principal amount due at a given period (**principalDue**).
- **c.** The balance of the loan at a given period (**balance**).

Use your Timer class to time each version of each function; what do you observe? What happens as the time period increases?

4) To demonstrate understanding of class-level methods, do the following:

- **a.** Implement a class-level method called **calcMonthlyPmt**, in the Loan base class. This should calculate a monthly payment based on three parameters: *face*, *rate*, and *term*.
- **b.** Create a class-level function, in the Loan base class, which calculates the balance (**calcBalance**). Input parameters should be *face*, *rate*, *term*, *period*.
- **c.** Test the class-level methods in **main**.
- **d.** Modify the object-level methods for **monthlyPayment** and **balance** to delegate to the class-level methods.
- **e.** Test the object-level methods to ensure they still work correctly.
- **f.** What are the benefits of class-level methods? When are they useful?

5) To demonstrate understanding of static-level methods, do the following:

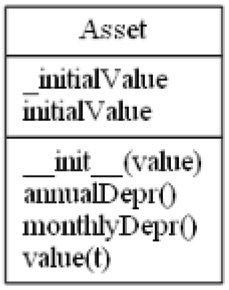
- **a.** Create a static-level method in Loan called **monthlyRate**. This should return the monthly interest rate for a passed-in annual rate.
- **b.** Create another static-level method that does the opposite (returns an annual rate for a passed-in monthly rate).
- **c.** Test the static-level method in **main**.
- **d.** Modify all the Loan methods that rely on the rate to utilize the static-level rate functions.
- **e.** What are the benefits of static-level methods? When are they useful?

6) Create a class called **Asset**. This class will represent the Asset covered by the loan. The class should do the following:

- **a.** Save an initial asset value upon object initialization (i.e. the initial value of a car).
- **b.** Create a method that returns a yearly depreciation rate (i.e., 10%).
- **c.** Create a method that calculates the monthly depreciation rate, from the yearly depreciation rate.
- **d.** Create a method that returns the current value of the asset, for a given period. This is the initial value times total depreciation. Total depreciation at time **t** can be calculated as:

$$(\mathbf{1} - \boldsymbol{monthlyDeprRate})^{\boldsymbol{t}}$$

We will build upon this class in the next level, to make it more useful to our ABS model.



2.2: Intermediate Classes

In addition to watching the video lectures, please be sure to read the provided Tips, Tricks, and Pitfalls document for this level before attempting the below exercises. In specific, the section called Inheritance & Mixin Classes, for more info on Mixin classes. Please also be sure to examine the provided sample code for this level in detail.

Each exercise should be a separate PyCharm project, with a main() that tests the new/affected functionality for that exercise; each one incrementally builds up the suite of classes.

The last page contains the class diagram, which is built up over the exercises in this section.

1) As shown in the lecture, create derived classes as follows:

- **a.** A **FixedRateLoan** class which derives from Loan.
- **b.** A **VariableRateLoan** class which derives from Loan. This will differ from a FixedRateLoan in that it has a rate **dict** instead of a single rate value. This dict will contain *startPeriod* as key and *rate* as value. This should have its own __init__ function that does the following:
  - **i.** Validates that the rate parameter is indeed a dict (if not, print an error).
  - **ii.** Invokes the super-class' __init__ function with all the parameters.

  The result of the above is that a VariableRateLoan's _rate attribute, as well as its rate property getters/setters will be a dict instead of a value. However, the functions that use the rate (i.e. balance) does not yet know how to handle a dict of rates. To handle this, do the following:

  - **i.** Create a **getRate** function in the base Loan class. This should take a period parameter. and return the result of the rate property.
  - **ii.** Override the **getRate** function in VariableRateLoan. This version will calculate the rate from the dict based on the period argument. Tip: Keep in mind that the dict only contains startPeriod for each rate -- the code will need to infer the rate for any period in between.

  Then, modify the Loan class functions (i.e. balance) to use the getRate function to get the rate for the current period.

  Note that the monthly payment and balance formulas are technically different in this Variable case, but we will avoid changing it for simplicity (the focus of the remaining exercises and case study are on fixed rate loans only).

2) Create a **MortgageMixin** class which will contain mortgage-specific methods. In particular, we'd like to implement the concept of Private Mortgage Insurance (PMI). For those unaware, PMI is an extra monthly payment that one must make to compensate for the added risk of having a Loan-to-Value (LTV) ratio of less than 80% (in other words, the loan covers >= 80% of the value of the asset).

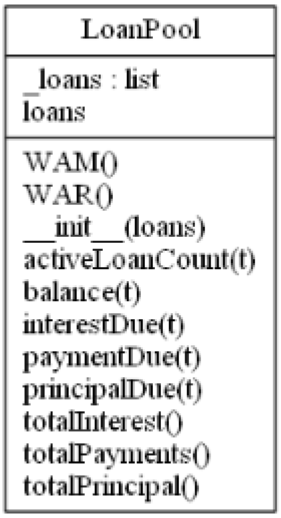
To this end, implement a function called **PMI** that returns 0.0075% of the loan face value, but only if the LTV is < 80%. For now, assume that the initial loan amount is for 100% of the asset value. Additionally, override the base class **monthlyPayment** and **principalDue** functions to account for PMI (Hint: use **super** to avoid duplicating the formulas, and note that the other methods (interestDue, balance, etc.) should not require any changes).

3) Create a **VariableMortgage** and **FixedMortgage** class. These should each derive-from the appropriate base class(es) (TBD by student).

4) Create a fixed **AutoLoan** class. This should derive-from the appropriate base class(es).

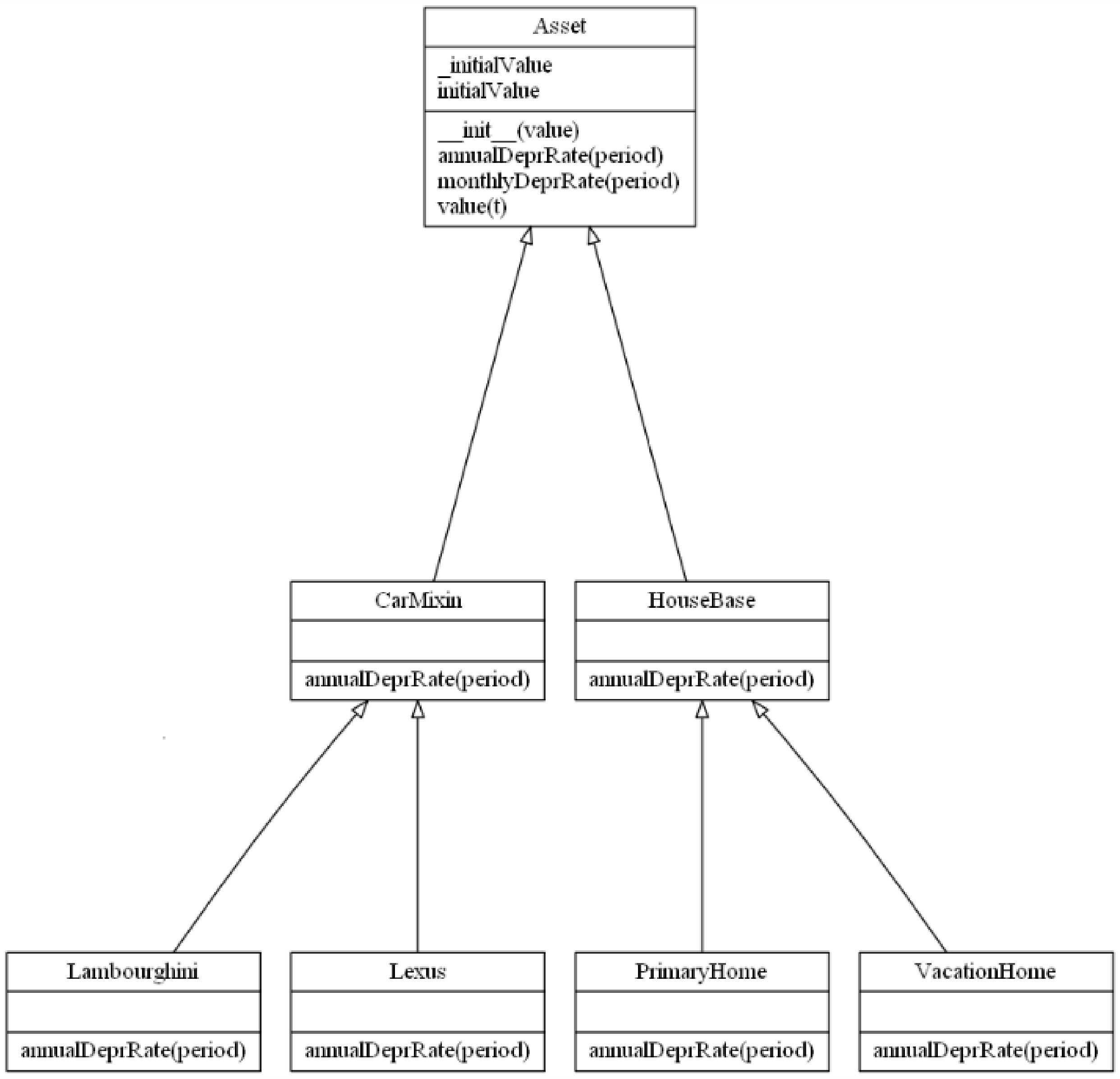
5) Create a **LoanPool** class that can contain and operate on a pool of loans (*composition*). Provide the following functionality:

- **a.** A method to get the total loan principal.
- **b.** A method to get the total loan balance for a given period.
- **c.** Methods to get the aggregate principal, interest, and total payment due in a given period.
- **d.** A method that returns the number of 'active' loans. Active loans are loans that have a balance greater than zero.
- **e.** Methods to calculate the Weighted Average Maturity (WAM) and Weighted Average Rate (WAR) of the loans. You may port over the previously implemented global functions.



6) This exercise focuses on creating the individual **Asset** derived classes. Do the following:

- **a.** Modify the **annualDeprRate** method of the **Asset** class to trigger a not-implemented error. This ensures that no one can directly instantiate an Asset object (makes it *abstract*).
- **b.** Create a **Car** class, derived from **Asset**. Derive multiple car types from **Car** (i.e. Civic, Lexus, Lamborghini, etc.). Give each one a different depreciation rate.
- **c.** Create a **HouseBase** class, derived from **Asset**. Derive **PrimaryHome** and **VacationHome** from **House**. Give each one a different depreciation rate (note, a vacation home will depreciate slower than a primary home since it is often vacant).



7) Now that we have our **Loan** and **Asset** classes, let's incorporate the asset into the loan. As a loan is 'on an' asset, which is similar to 'has a', we use composition instead of derivation. To this end:

   a. Add an *asset* parameter to the base loan **__init__** function, which saves it down into an object-level attribute. The one caveat here is that we must to ensure that the *asset* parameter indeed contains an **Asset** object (or any of its derived classes). If it's not an **Asset** type, you should print an error message to the user, and leave the function.

   b. Modify **MortgageMixin** to initialize with a *home* parameter. In this case, we need to ensure that the value of *home* is indeed a primary home, vacation home, or any other derived **HouseBase** type. Modify the **PMI** function to calculate LTV based on the asset initial value (instead of the loan's face value).

   c. Do the same for the **AutoLoan** class.

   d. Create a method called **recoveryValue** in the **Loan** base class. This method should return the current asset value for the given period, times a *recovery multiplier* of 0.6.

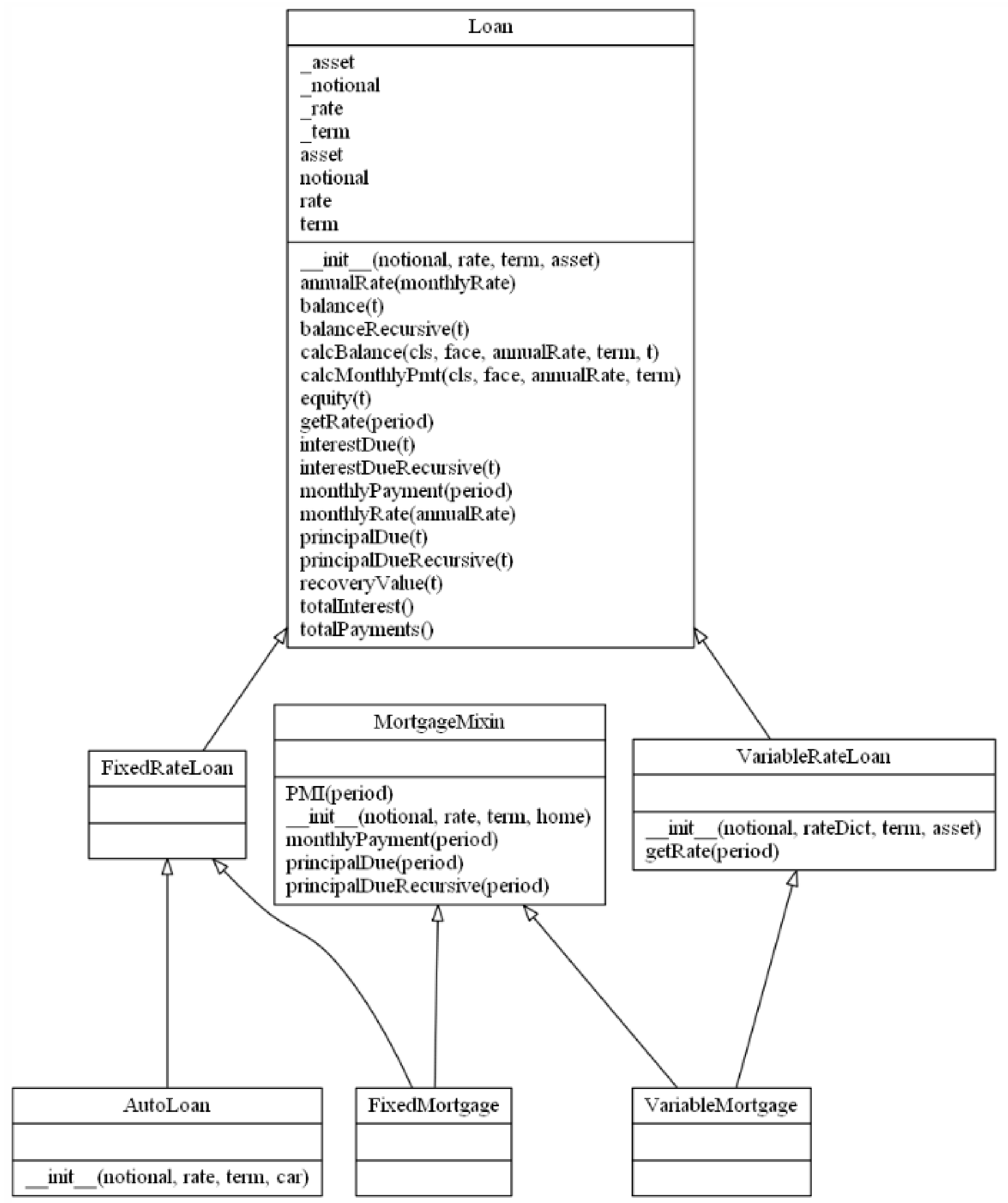
   e. Create a method called **equity** in the **Loan** base class. This should return the available equity (the asset value less the loan balance).

   f. In **main**, instantiate different **Loan** types with different assets and test the new functionality.

Note that the 'recovery value' of an asset, in terms of a loan, is the amount of money the lender can recover if the borrower defaults (forecloses). The lender will usually auction off the property. The 'multiplier' is necessary, as the lender is not likely to receive full market value of the property in an auction. The above is an overly simplistic model, as the recovery rates vary across asset classes and markets (the subject of a different course).

For now, this implementation is for completeness of the Loan classes (they're not very useful yet). We will start to understand the benefit of these **Asset** classes later, when we implement our full ABS model for the final project.

The following is the class diagram, built up from all the above exercises:


Loan
_asset
_notional
_rate
_term
asset
notional
rate
term
__init__(notional, rate, term, asset)
annualRate(monthlyRate)
balance(t)
balanceRecursive(t)
calcBalance(cls, face, annualRate, term, t)
calcMonthlyPmt(cls, face, annualRate, term)
equity(t)
getRate(period)
interestDue(t)
interestDueRecursive(t)
monthlyPayment(period)
monthlyRate(annualRate)
principalDue(t)
principalDueRecursive(t)
recoveryValue(t)
totalInterest()
totalPayments()
FixedRateLoan
MortgageMixin
PMI(period)
__init__(notional, rate, term, home)
monthlyPayment(period)
principalDue(period)
principalDueRecursive(period)
VariableRateLoan
__init__(notional, rateDict, term, asset)
getRate(period)
AutoLoan
__init__(notional, rate, term, car)
FixedMortgage
VariableMortgage