420-LCU-05 Programming in Python - Lab Exercise 8

November 13, 2017

Goals for this lab:

- Examine and work with inheritance.
- Overload a few new operators.

Your submission for this exercise should be either a text or Word file. You do NOT need to copy the questions into the answer file, just write answers and/or copy and paste results from your IDLE session. Any code you include can also be pasted into your submission - there is no need for a separate `.py` file.

You should also show your work by copying and pasting the entire IDLE session at the end of your submission.

1 Mortgages

As you probably know, most people get a mortgage loan in order to purchase a house. As with any loan, you pay interest on the balance of the loan, and most mortgages are designed to be paid over a long period, usually 25 or 30 years. Therefore, the total amount you pay to the bank is substantially more than the original loan amount.

In the USA, especially in the period leading up to the financial crisis in 2008, there were many new, complicated mortgage types introduced. It could be quite difficult for a prospective home purchaser to evaluate the total cost of each type of mortgage. Our goal is to write a program that allows someone to evaluate the relative costs of different types of mortgages.

To make this easier, we use a simple class hierarchy to represent three different mortgage types.

The point of this exercise is to provide some experience in the development of a set of related classes, not to make you an expert on mortgages

1. Examine the included file `mortgage.py`. Take note of the following details:

   - The function `findPayment`. This is a "helper" function which is used to compute the monthly payment for a given mortgage amount, interest rate, and load period.

     It is implemented as a global function rather than a method, since it is somewhat independent of the specific attributes of the classes.

     You do not have to understand the math of this function! Consider it a black box.

   - The class `Mortgage`. This forms the base class of each of our specific mortgage types. This is an example of what is sometimes called an "abstract class." An abstract class is a class which is not intended to be used directly to create objects, but which exists only to act as a base class for some set of derived classes. The class is *abstract* because it is missing some crucial details, such as a working implementation of a method or some crucial data value(s).

     The `Mortgage` class has several methods:

     - `__init__` is the constructor. It creates several variable attributes corresponding to the initial loan amount, the monthly interest rate, the duration of the loan in months, a list of payments that have been made, a list of the outstanding balance at each month, the minimum payment due each month, and a description of the mortgage.
     - `makePayment` records a mortgage payment. Part of each payment covers the amount of interest due on the loan balance, and the remainder is used to reduce the loan balance.
     - `getTotalPaid` returns the total amount paid by the mortgage holder up to this point.
     - `__str__` returns the 'legend' or description of the mortgage.

   - The derived class `FixedRate`. This implements the "standard" mortgage, which used a fixed interest rate over the entire lifetime of the mortage.

     Since this is essentially the same as the `Mortgage` class, it only overrides the `__init__` method. It calls the method of the base class to make sure the attributes are properly initialized.

     Otherwise, the new `__init__` method only replaces the empty 'legend' value from the base class with a specific description.

- The derived class `FixedRateWithPoints`, which implements a mortgage in which the mortgage borrower must pay in a certain percentage when they arrange the mortgage, in return for a lower overall borrowing rate.

  As with the `FixedRate` mortgage, this mortgage overrides only the `__init__` method, where it sets up both the 'legend' attribute and changes the initial value of the 'paid' list to reflect the points paid by the borrower.

- The derived class `TeaserRate`, which implements a more complicated mortgage product. In a teaser or two-rate mortgage, there are two rates. The first rate is lower but only applies for the first few years of the mortgage. The second, higher rate increases the payment required for the rest of the duration of the mortgage.

  This class has to override the `makePayment` method in order to correctly implement the logic of the mortgage. Like the other derived classes, it also overrides `__init__`.

2. Run the program and take note of the results. Include these in the document you submit (just cut and paste).

3. Look at the `__init__` methods of each class. Explain why is the call to the base class method is done before the specific additional work done in the constructor.

4. In contrast with the constructor, the `makePayment` method in the `TeaserRate` class calls the base class implementation only after performing its extra tasks. Explain why this is done this way.

5. Create a new mortgage, a fixed rate with an interest rate of 7.5%. Add this to the overall comparison. Run the modified program and include the results in your report.

6. One way to save money on a mortgage is to make a *prepayment*, which means you pay a bit more than the standard payment each month. A small increase in the amount paid, $50 for example, will cause you to pay considerably less over the life of the mortgage. However, not all mortgages allow this.

   Add functionality to the classes to permit prepayments, and check the results. The technique you should use is as follows:

   (a) Change the definition of the `makePayment` method to look like `makePayment(self, extra = 0)`.

   (b) Update the code in the base class implementation of `makePayment` to take off the extra amount in addition to the regular payment.

   (c) Modify the `compareMortgages` function to take a parameter specifying the additional payment, and pass this to the `makePayment` method.

   Run the modified program with a prepayment equal to the last two digits of your ID number (use 100 if the last two digits of your ID happen to be zero). What are the new total payments of the four mortgages?

2 Overriding operators

As we've mentioned, the `float` type in Python has a limit to both the size and precision of the numbers it can represent. One way to represent fractional values is to explicitly store both an integer numerator and denominator. Because Python allows integers to have any number of digits, we can theoretically represent any rational number this way.

I've provided a basic class that implements this idea in `fraction.py`. In this part, you will use this class and add some functionality to integrate this nicely with Python's math operators.

1. First take a look at the class. The most complicated part is the `__init__` method. The constructor performs several functions - it checks that the numerator and denominator are both integers, and raises an error if the denominator is zero (don't worry about the details of this error business yet). Finally it uses a local `gcd` function to simplify the numerator and denominator.

   Notice that `__init__` does not need or use a `return` statement. Why do you suppose this is?

2. In the Python shell, try importing the fraction class and create two fraction objects. Try adding them together and printing the results. Copy and paste what you see into your lab report.

3. I have provided some test code in the `fraction.py` class. If you run the module from IDLE you should get an error, because the class does not yet implement everything it should.

4. We'll now make the class complete. First, we'll add a `__str__` method. It should do the following: If the denominator of the fraction is one, the fraction is just an integer. In this case the method should return a string representation of the numerator. For any other denominator value, return a string of the form "num/den". You can use `format` or any other method to implement this.

5. Check your method. One way to experiment with this is to go back to the shell, restart, and re-import your updated fraction class. Now try executing the following code in the shell:

```
>>> a = fraction(1, 2)
>>> b = fraction
>>> a
>>> b
>>> print(a)
>>> print(b)
```

   Try this and show what gets printed.

6. Re-run the test code in the module. If your method is correct, the main program should now print `__str__ OK`

7. Now add an implementation of the `__eq__` method. This should be simple - it will look very much like the implementation in the `point` class. Copy and paste your code into the report.

8. Finally, add an implementation of the `__mul__` method, which is similar to the `__add__` method already provided. Copy and paste your code into the report.

9. Now run the entire module and test program again. It should now complete without error. Copy and paste the output into your lab report.

3 Optional

If you have extra time, try the following.

1. Obviously it might be nice if we could multiply or add an integer and a fraction directly, such as the following:

```
>>> a = fraction(1, 2)
>>> c = a + 2 # add 1/2 and 2
>>> print(c) # should give 5/2
```

   Extend the `__add__` and `__mul__` methods to allow this. There are hints about this in the slides for lecture 14 - look for examples using the `isinstance` builtin function. Add some test cases for these features.

2. We haven't handled subtraction or division. For this you need to implement `__sub__` and `__truediv__`. Add some test cases for these features.

3. OK, you still need more? Implement the `__bool__` and `__float__` methods to handle conversion of fractions to Boolean or floating-point. For boolean, define any fraction with a zero numerator to be `False`. For float, just divide the numerator by the denominator and return the result.

4. Still here? OK, go try to implement the comparison operators (e.g. `>=`, <, etc.). For this you need to implement four different functions, `__le__`, `__lt__`, `__ge__`, and `__gt__`. Add some test code to verify that these work correctly.