[Index | Exercise 3.4 | Exercise 3.6]

Exercise 3.5

Objectives:

- Learn how to use inheritance to write extensible code.
- See a practical use of inheritance by writing a program that must output data in a variety of user-selectable formats such as plain-text, CSV, and HTML.

Files Modified: `tableformat.py`

One major use of classes in Python is in writing code that be extended/adapted in various ways. To illustrate, in Exercise 3.2 you created a function `print_table()` that made tables. You used this to make output from the `portfolio` list. For example:

```
>>> import stock
>>> import reader
>>> import tableformat
>>> portfolio = reader.read_csv_as_instances('Data/portfolio.csv',
stock.Stock)
>>> tableformat.print_table(portfolio, ['name','shares','price'])
      name     shares      price
---------- ---------- ----------
        AA        100       32.2
       IBM         50       91.1
       CAT        150      83.44
      MSFT        200      51.23
        GE         95      40.37
      MSFT         50       65.1
       IBM        100      70.44
>>>
```

Suppose you wanted the `print_table()` function to be able to make tables in any number of output formats such as CSV, XML, HTML, Excel, etc. Trying to modify the function to support all of those output formats at once would be painful. A better way to do this involves moving the output-related formatting code to a class and using inheritance to implement different output formats.

(a) Defining a generic formatter class

Add the following class definition to the `tableformat.py` file:

```
class TableFormatter:
    def headings(self, headers):
        raise NotImplementedError()
```

```
    def row(self, rowdata):
        raise NotImplementedError()
```

Now, modify the `print_table()` function so that it accepts a `TableFormatter` instance and invokes methods on it to produce output:

```
def print_table(records, fields, formatter):
    formatter.headings(fields)
    for r in records:
        rowdata = [getattr(r, fieldname) for fieldname in fields]
        formatter.row(rowdata)
```

These two classes are meant to be used together. For example:

```
>>> import stock, reader, tableformat
>>> portfolio = reader.read_csv_as_instances('Data/portfolio.csv',
stock.Stock)
>>> formatter = tableformat.TableFormatter()
>>> tableformat.print_table(portfolio, ['name', 'shares', 'price'],
formatter)
Traceback (most recent call last):
...
NotImplementedError
>>>
```

For now, it doesn't do much of anything interesting. You'll fix this in the next section.

(b) Implementing a concrete formatter

The `TableFormatter` isn't meant to be used by itself. Instead, it is merely a base for other classes that will implement the formatting. Add the following class to `tableformat.py`:

```
class TextTableFormatter(TableFormatter):
    def headings(self, headers):
        print(' '.join('%10s' % h for h in headers))
        print(('-'*10 + ' ')*len(headers))

    def row(self, rowdata):
        print(' '.join('%10s' % d for d in rowdata))
```

Now, use your new class as follows:

```
>>> import stock, reader, tableformat
>>> portfolio = reader.read_csv_as_instances('Data/portfolio.csv',
stock.Stock)
```

```
>>> formatter = tableformat.TextTableFormatter()
>>> tableformat.print_table(portfolio, ['name','shares','price'],
formatter)
     name     shares      price
---------- ---------- ----------
        AA        100       32.2
       IBM         50       91.1
       CAT        150      83.44
      MSFT        200      51.23
        GE         95      40.37
      MSFT         50       65.1
       IBM        100      70.44
>>>
```

(c) Adding More Implementations

Create a class `CSVTableFormatter` that allows output to be generated in CSV format:

```
>>> import stock, reader, tableformat
>>> portfolio = reader.read_csv_as_instances('Data/portfolio.csv',
stock.Stock)
>>> formatter = tableformat.CSVTableFormatter()
>>> tableformat.print_table(portfolio, ['name','shares','price'],
formatter)
name,shares,price
AA,100,32.2
IBM,50,91.1
CAT,150,83.44
MSFT,200,51.23
GE,95,40.37
MSFT,50,65.1
IBM,100,70.44
>>>
```

Create a class `HTMLTableFormatter` that generates output in HTML format:

```
>>> import stock, reader, tableformat
>>> portfolio = reader.read_csv_as_instances('Data/portfolio.csv',
stock.Stock)
>>> formatter = tableformat.HTMLTableFormatter()
>>> tableformat.print_table(portfolio, ['name','shares','price'],
formatter)
<tr> <th>name</th> <th>shares</th> <th>price</th> </tr>
<tr> <td>AA</td> <td>100</td> <td>32.2</td> </tr>
<tr> <td>IBM</td> <td>50</td> <td>91.1</td> </tr>
<tr> <td>CAT</td> <td>150</td> <td>83.44</td> </tr>
<tr> <td>MSFT</td> <td>200</td> <td>51.23</td> </tr>
<tr> <td>GE</td> <td>95</td> <td>40.37</td> </tr>
<tr> <td>MSFT</td> <td>50</td> <td>65.1</td> </tr>
```

```
<tr> <td>IBM</td> <td>100</td> <td>70.44</td> </tr>
>>>
```

(d) Making it Easier To Choose

One problem with using inheritance is the added complexity of picking different classes to use (e.g., remembering the names, using the right `import` statements, etc.). A factory function can simplify this. Add a function `create_formatter()` to your `tableformat.py` file that allows a user to more easily make a formatter by specifying a format such as `'text'`, `'csv'`, or `'html'`. For example:

```
>>> from tableformat import create_formatter, print_table
>>> formatter = create_formatter('html')
>>> print_table(portfolio, ['name','shares','price'], formatter)
<tr> <th>name</th> <th>shares</th> <th>price</th> </tr>
<tr> <td>AA</td> <td>100</td> <td>32.2</td> </tr>
<tr> <td>IBM</td> <td>50</td> <td>91.1</td> </tr>
<tr> <td>CAT</td> <td>150</td> <td>83.44</td> </tr>
<tr> <td>MSFT</td> <td>200</td> <td>51.23</td> </tr>
<tr> <td>GE</td> <td>95</td> <td>40.37</td> </tr>
<tr> <td>MSFT</td> <td>50</td> <td>65.1</td> </tr>
<tr> <td>IBM</td> <td>100</td> <td>70.44</td> </tr>
>>>
```

Discussion

The `TableFormatter` class in this exercise is an example of something known as an "Abstract Base Class." It's not something that's meant to be used directly. Instead, it's serving as a kind of interface specification for a program component--in this case the various output formats. Essentially, the code that produces the table will be programmed against the abstract base class with the expectation that a user will provide a suitable implementation. As long as all of the required methods have been implemented, it should all just "work" (fingers crossed).

[Solution | Index | Exercise 3.4 | Exercise 3.6]
