[Index | Exercise 3.7 | Exercise 4.1]

Exercise 3.8

Objectives:

- Learn about mixin classes and cooperative inheritance

Files Modified: `tableformat.py`

(a) The Trouble with Column Formatting

If you go all the way back to Exercise 3.1, you wrote a function `print_portfolio()` that produced a table like this:

```
>>> portfolio = read_portfolio('Data/portfolio.csv')
>>> print_portfolio(portfolio)
      name     shares      price
---------- ---------- ----------
        AA        100      32.20
       IBM         50      91.10
       CAT        150      83.44
      MSFT        200      51.23
        GE         95      40.37
      MSFT         50      65.10
       IBM        100      70.44
>>>
```

The `print_table()` function developed in the last several exercises almost replaces this functionality--almost. The one problem that it has is that it can't precisely format the content of each column. For example, notice how the values in the `price` column are precisely formatted with 2 decimal points. The `TableFormatter` class and related subclasses can't do that.

One way to fix it would be to modify the `print_table()` function to accept an additional formats argument. For example, maybe something like this:

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

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

```
>>> print_table(portfolio,
                ['name','shares','price'],
                ['%s','%d','%0.2f'],
                formatter)

      name     shares      price
---------- ---------- ----------
        AA        100      32.20
       IBM         50      91.10
       CAT        150      83.44
      MSFT        200      51.23
        GE         95      40.37
      MSFT         50      65.10
       IBM        100      70.44
>>>
```

Yes, you could modify `print_table()` like this, but is that the right place to do it? The whole idea of all of the `TableFormatter` classes is that they could be used in different kinds of applications. Column formatting is something that could be useful elsewhere, not just in the `print_table()` function.

Another possible approach might be to change the interface to the `TableFormatter` class in some way. For example, maybe adding a third method to apply formatting.

```
class TableFormatter:
    def headings(self, headers):
        ...
    def format(self, rowdata):
        ...
    def row(self, rowdata):
        ...
```

The problem here is that any time you change the interface on a class, you're going to have to refactor all of the existing code to work with it. Specifically, you'd have to modify all of the already written `TableFormatter` subclasses and all of the code written to use them. Let's not do that.

As an alternative, a user could use inheritance to customize a specific formatter in order to inject some formatting into it. For example, try this experiment:

```
>>> from tableformat import TextTableFormatter, print_table
>>> class PortfolioFormatter(TextTableFormatter):
        def row(self, rowdata):
            formats = ['%s','%d','%0.2f']
            rowdata = [(fmt % d) for fmt, d in zip(formats, rowdata)]
            super().row(rowdata)

>>> formatter = PortfolioFormatter()
>>> print_table(portfolio, ['name','shares','price'], formatter)
      name     shares      price
---------- ---------- ----------
```

```
          AA        100      32.20
         IBM         50      91.10
         CAT        150      83.44
        MSFT        200      51.23
          GE         95      40.37
        MSFT         50      65.10
         IBM        100      70.44
>>>
```

Yes, that works, but it's also a bit clumsy and weird. The user has to pick a specific formatter to customize. On top of that, they have to implement the actual column formatting code themselves. Surely there is a different way to do this.

(b) Going Sideways

In the `tableformat.py` file, add the following class definition:

```
class ColumnFormatMixin:
    formats = []
    def row(self, rowdata):
        rowdata = [(fmt % d) for fmt, d in zip(self.formats, rowdata)]
        super().row(rowdata)
```

This class contains a single method `row()` that applies formatting to the row contents. A class variable `formats` is used to hold the format codes. This class is used via multiple inheritance. For example:

```
>>> import stock, reader
>>> portfolio = reader.read_csv_as_instances('Data/portfolio.csv',
stock.Stock)
>>> from tableformat import TextTableFormatter, ColumnFormatMixin,
print_table
>>> class PortfolioFormatter(ColumnFormatMixin, TextTableFormatter):
        formats = ['%s', '%d', '%0.2f']

>>> formatter = PortfolioFormatter()
>>> print_table(portfolio, ['name','shares','price'], formatter)
      name     shares      price
---------- ---------- ----------
        AA        100      32.20
       IBM         50      91.10
       CAT        150      83.44
      MSFT        200      51.23
        GE         95      40.37
      MSFT         50      65.10
       IBM        100      70.44
```

This whole approach works because the `ColumnFormatMixin` class is meant to be mixed together with another class that provides the required `row()` method.

Make another class that makes a formatter print the table headers in all-caps:

```
class UpperHeadersMixin:
    def headings(self, headers):
        super().headings([h.upper() for h in headers])
```

Try it out and notice that the headers are now uppercase:

```
>>> from tableformat import TextTableFormatter, UpperHeadersMixin
>>> class PortfolioFormatter(UpperHeadersMixin, TextTableFormatter):
        pass

>>> formatter = PortfolioFormatter()
>>> 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
>>>
```

This is really the whole idea on "mixins." The creator of a library can provide a basic set of classes such as `TextTableFormatter`, `CSVTableFormatter`, and so forth to start. Then, a collection of add-on classes can be provided to make those classes behave in different ways.

(c) Making it Sane

Using mixins can be a useful tool for framework builders for reducing the amount of code that needs to be written. However, forcing users to remember how to properly compose classes and use multiple inheritance can fry their brains. In Exercise 3.5, you wrote a function `create_formatter()` that made it easier to create a custom formatter. Take that function and extend it to understand a few optional arguments related to the mixin classes. For example:

```
>>> from tableformat import create_formatter
>>> formatter = create_formatter('csv', column_formats=
['"%s"','%d','%0.2f'])
>>> print_table(portfolio, ['name','shares','price'], formatter)
name,shares,price
"AA",100,32.20
"IBM",50,91.10
```

```
"CAT",150,83.44
"MSFT",200,51.23
"GE",95,40.37
"MSFT",50,65.10
"IBM",100,70.44

>>> formatter = create_formatter('text', upper_headers=True)
>>> 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
>>>
```

Under the covers the `create_formatter()` function will properly compose the classes and return a proper `TableFormatter` instance.

[Solution | Index | Exercise 3.7 | Exercise 4.1]
