[Index | Exercise 3.6 | Exercise 3.8]

Exercise 3.7

Objectives:

- Type checking and interfaces
- Abstract base classes

Files Modified: `tableformat.py`

In Exercise 3.5, we modified the `tableformat.py` file to have a `TableFormatter` class and to use various subclasses for different output formats. In this exercise, we extend that code a bit more.

(a) Interfaces and type checking

Modify the `print_table()` function so that it checks if the supplied formatter instance inherits from `TableFormatter`. If not, raise a `TypeError`.

Your new code should catch situations like this:

```
>>> import stock, reader, tableformat
>>> portfolio = reader.read_csv_as_instances('Data/portfolio.csv',
stock.Stock)
>>> class MyFormatter:
        def headings(self,headers): pass
        def row(self,rowdata): pass

>>> tableformat.print_table(portfolio, ['name','shares','price'],
MyFormatter())
Traceback (most recent call last):
...
TypeError: Expected a TableFormatter
>>>
```

Adding a check like this might add some degree of safety to the program. However you should still be aware that type-checking is rather weak in Python. There is no guarantee that the object passed as a formatter will work correctly even if it happens to inherit from the proper base class. This next part addresses that issue.

(b) Abstract Base Classes

Modify the `TableFormatter` base class so that it is defined as a proper abstract base class using the `abc` module. Once you have done that, try this experiment:

```
>>> class NewFormatter(TableFormatter):
        def headers(self, headings):
```

```
            pass
        def row(self, rowdata):
            pass

>>> f = NewFormatter()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class NewFormatter with abstract
methods headings
>>>
```

Here, the abstract base class caught a spelling error in the class--the fact that the `headings()` method was incorrectly given as `headers()`.

(c) Algorithm Template Classes

The file `reader.py` contains two functions, `read_csv_as_dicts()` and `read_csv_as_instances()`. Both of those functions are almost identical--there is just one tiny bit of code that's different. Maybe that code could be consolidated into a class definition of some sort. Add the following class to the `reader.py` file:

```
# reader.py


import csv
from abc import ABC, abstractmethod

class CSVParser(ABC):

    def parse(self, filename):
        records = []
        with open(filename) as f:
            rows = csv.reader(f)
            headers = next(rows)
            for row in rows:
                record = self.make_record(headers, row)
                records.append(record)
        return records

    @abstractmethod
    def make_record(self, headers, row):
        pass
```

This code provides a shell (or template) of the CSV parsing functionality. To use it, you subclass it, add any additional attributes you might need, and redefine the `make_record()` method. For example:

```
class DictCSVParser(CSVParser):
    def __init__(self, types):
```

```
        self.types = types

    def make_record(self, headers, row):
        return { name: func(val) for name, func, val in zip(headers,
self.types, row) }

class InstanceCSVParser(CSVParser):
    def __init__(self, cls):
        self.cls = cls

    def make_record(self, headers, row):
        return self.cls.from_row(row)
```

Add the above classes to the `reader.py` file. Here's how you would use one of them:

```
>>> from reader import DictCSVParser
>>> parser = DictCSVParser([str, int, float])
>>> port = parser.parse('Data/portfolio.csv')
>>>
```

It works, but it's kind of annoying. To fix this, reimplement the `read_csv_as_dicts()` and `read_csv_as_instances()` functions to use these classes. Your refactored code should work exactly the same way that it did before. For example:

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

[Solution | Index | Exercise 3.6 | Exercise 3.8]
