[Index | Exercise 2.5 | Exercise 3.1]

Exercise 2.6

Objectives:

- Explore the power of having first-class objects.
- Better understand Python's memory model

Files Created: `reader.py`

In previous exercises, you wrote various functions for reading CSV data in different files. Surely this is a problem that could be generalized in some way. In this exercise, we explore that idea.

(a) First-class Data

In the file `Data/portfolio.csv`, you read data organized as columns that look like this:

```
"AA",100,32.20
"IBM",50,91.10
...
```

In previous code, this data was processed by hard-coding all of the type conversions. For example:

```
rows = csv.reader(f)
for row in rows:
    name   = row[0]
    shares = int(row[1])
    price  = float(row[2])
```

This kind of conversion can also be performed in a more clever manner using some list operations. Make a Python list that contains the conversions you want to carry out on each column:

```
>>> coltypes = [str, int, float]
>>>
```

The reason you can even create this list is that everything in Python is "first-class." So, if you want to have a list of functions, that's fine.

Now, read a row of data from the above file:

```
>>> import csv
>>> f = open('Data/portfolio.csv')
>>> rows = csv.reader(f)
```

```
>>> headers = next(rows)
>>> row = next(rows)
>>> row
['AA', '100', '32.20']
>>>
```

Zip the column types with the row and look at the result:

```
>>> r = list(zip(coltypes, row))
>>> r
[(<class 'str'>, 'AA'), (<class 'int'>, '100'), (<class 'float'>,'32.20')]
>>>
```

You will notice that this has paired a type conversion with a value. For example, `int` is paired with the value `'100'`. Now, try this:

```
>>> record = [func(val) for func, val in zip(coltypes, row)]
>>> record
['AA', 100, 32.2]
>>>
```

Make sure you understand what's happening in the above code. In the loop, the `func` variable is one of the type conversion functions (e.g., `str`, `int`, etc.) and the `val` variable is one of the values like `'AA'`, `'100'`. The expression `func(val)` is converting a value (kind of like a type cast).

You can take it a step further and make dictionaries by using the column headers. For example:

```
>>> dict(zip(headers, record))
{'name': 'AA', 'shares': 100, 'price': 32.2}
>>>
```

If you prefer, you can perform all of these steps at once using a dictionary comprehension:

```
>>> { name:func(val) for name, func, val in zip(headers, coltypes, row) }
{'name': 'AA', 'shares': 100, 'price': 32.2}
>>>
```

(b) Making a Parsing Utility Function

Create a new file `reader.py`. In that file, define a utility function `read_csv_as_dicts()` that reads a file of CSV data into a list of dictionaries where the user specifies the type conversions for each column.

Here is how it should work:

```
>>> import reader
>>> portfolio = reader.read_csv_as_dicts('Data/portfolio.csv',
[str,int,float])
>>> for s in portfolio:
        print(s)

{'name': 'AA', 'shares': 100, 'price': 32.2}
{'name': 'IBM', 'shares': 50, 'price': 91.1}
{'name': 'CAT', 'shares': 150, 'price': 83.44}
{'name': 'MSFT', 'shares': 200, 'price': 51.23}
{'name': 'GE', 'shares': 95, 'price': 40.37}
{'name': 'MSFT', 'shares': 50, 'price': 65.1}
{'name': 'IBM', 'shares': 100, 'price': 70.44}
>>>
```

Or, if you wanted to read the CTA data:

```
>>> rows = reader.read_csv_as_dicts('Data/ctabus.csv', [str,str,str,int])
>>> len(rows)
577563
>>> rows[0]
{'daytype': 'U', 'route': '3', 'rides': 7354, 'date': '01/01/2001'}
>>>
```

(c) Memory Revisited

In the CTA bus data, we determined that there were 181 unique bus routes.

```
>>> routes = { row['route'] for row in rows }
>>> len(routes)
181
>>>
```

Question: How many unique route string objects are contained in the ride data? Instead of building a set of routes, build a set of object ids instead:

```
>>> routeids = { id(row['route']) for row in rows }
>>> len(routeids)
542305
>>>
```

Think about this for a moment--there are only 181 distinct route names, but the resulting list of dictionaries contains 542305 different route strings. Maybe this is something that could be fixed with a bit of caching or

object reuse. As it turns out, Python has a function that can be used to cache strings, `sys.intern()`. For example:

```
>>> a = 'hello world'
>>> b = 'hello world'
>>> a is b
False
>>> import sys
>>> a = sys.intern(a)
>>> b = sys.intern(b)
>>> a is b
True
>>>
```

Restart Python and try this:

```
>>> # ------------------ RESTART ---------```
>>> import tracemalloc
>>> tracemalloc.start()
>>> from sys import intern
>>> import reader
>>> rows = reader.read_csv_as_dicts('Data/ctabus.csv', [intern, str, str,
int])
>>> routeids = { id(row['route']) for row in rows }
>>> len(routeids)
181
>>>
```

Take a look at the memory use.

```
>>> tracemalloc.get_traced_memory()
... look at result ...
>>>
```

The memory should drop quite a bit. Observation: There is a lot of repetition involving dates as well. What happens if you also cache the date strings?

```
>>> # ------------------ RESTART ---------```
>>> import tracemalloc
>>> tracemalloc.start()
>>> from sys import intern
>>> import reader
>>> rows = reader.read_csv_as_dicts('Data/ctabus.csv', [intern, intern,
str, int])
>>> tracemalloc.get_traced_memory()
```

```
... look at result ...
>>>
```

(d) Special Challenge Project

In Exercise 2.5, we created a class `RideData` that stored all of the bus data in columns, but actually presented the data to a user as a sequence of dictionaries. It saved a lot of memory through various forms of magic.

Can you generalize that idea? Specifically, can you make a general purpose function `read_csv_as_columns()` that works like this:

```
>>> data = read_csv_as_columns('Data/ctabus.csv', types=[str, str, str,
int])
>>> data
<__main__.DataCollection object at 0x102b45048>
>>> len(data)
577563
>>> data[0]
{'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': 7354}
>>> data[1]
{'route': '4', 'date': '01/01/2001', 'daytype': 'U', 'rides': 9288}
>>> data[2]
{'route': '6', 'date': '01/01/2001', 'daytype': 'U', 'rides': 6048}
>>>
```

This function is supposed to be general purpose--you can give it any file and a list of the column types and it will read the data. The data is read into a class `DataCollection` that stores the data as columns internally. The data presents itself as a sequence of dictionaries when accessed however.

Try using this function with the string interning trick in the last part. How much memory does it take to store all of the ride data now? Can you still use this function with your earlier CTA analysis code?

(e) Deep Thought

In this exercise, you have written two functions, `read_csv_as_dicts()` and `read_csv_as_columns()`. These functions present data to the user in the same way. For example:

```
>>> data1 = read_csv_as_dicts('Data/ctabus.csv', [str, str, str, int])
>>> len(data1)
577563
>>> data1[0]
{'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': 7354}
>>> data1[1]
{'route': '4', 'date': '01/01/2001', 'daytype': 'U', 'rides': 9288}
>>>

>>> data2 = read_csv_as_columns('Data/ctabus.csv', [str, str, str, int])
```

```
>>> len(data2)
577563
>>> data2[0]
{'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': 7354}
>>> data2[1]
{'route': '4', 'date': '01/01/2001', 'daytype': 'U', 'rides': 9288}
>>>
```

In fact, you can use either function in the CTA data analysis code that you wrote. Yet, under the covers completely different things are happening. The `read_csv_as_columns()` function is storing the data in a different representation. It's relying on Python sequence protocols (magic methods) to present information to you in a more useful way.

This is really part of a much bigger programming concept of "Data Abstraction". When writing programs, the way in which data is presented is often more important than how the data is actually put together under the hood. Although we're presenting the data as a sequence of dictionaries, there's a great deal of flexibility in how that actually happens behind the scenes. That's a powerful idea and something to think about when writing your own programs.

[Solution | Index | Exercise 2.5 | Exercise 3.1]
