[Index | Exercise 1.6 | Exercise 2.2]

Exercise 2.1

Objectives:

- Figure out the most memory-efficient way to store a lot of data.
- Learn about different ways of representing records including tuples, dictionaries, classes, and named tuples.

In this exercise, we look at different choices for representing data structures with an eye towards memory use and efficiency. A lot of people use Python to perform various kinds of data analysis so knowing about different options and their tradeoffs is useful information.

(a) Stuck on the bus

The file `Data/ctabus.csv` is a CSV file containing daily ridership data for the Chicago Transit Authority (CTA) bus system from January 1, 2001 to August 31, 2013. It contains approximately 577000 rows of data. Use Python to view a few lines of data to see what it looks like:

```
>>> f = open('Data/ctabus.csv')
>>> next(f)
'route,date,daytype,rides\n'
>>> next(f)
'3,01/01/2001,U,7354\n'
>>> next(f)
'4,01/01/2001,U,9288\n'
>>>
```

There are 4 columns of data.

- route: Column 0. The bus route name.
- date: Column 1. A date string of the form MM/DD/YYYY.
- daytype: Column 2. A day type code (U=Sunday/Holiday, A=Saturday, W=Weekday)
- rides: Column 3. Total number of riders (integer)

The `rides` column records the total number of people who boarded a bus on that route on a given day. Thus, from the example, 7354 people rode the number 3 bus on January 1, 2001.

(b) Basic memory use of text

Let's get a baseline of the memory required to work with this datafile. First, restart Python and try a very simple experiment of simply grabbing the file and storing its data in a single string:

```
>>> # --- RESTART
>>> import tracemalloc
>>> f = open('Data/ctabus.csv')
```

```
>>> tracemalloc.start()
>>> data = f.read()
>>> len(data)
12361039
>>> current, peak = tracemalloc.get_traced_memory()
>>> current
12369664
>>> peak
24730766
>>>
```

Your results might vary somewhat, but you should see current memory use in the range of 12MB with a peak of 24MB.

What happens if you read the entire file into a list of strings instead? Restart Python and try this:

```
>>> # --- RESTART
>>> import tracemalloc
>>> f = open('Data/ctabus.csv')
>>> tracemalloc.start()
>>> lines = f.readlines()
>>> len(lines)
577564
>>> current, peak = tracemalloc.get_traced_memory()
>>> current
45828030
>>> peak
45867371
>>>
```

You should see the memory use go up significantly into the range of 40-50MB. Point to ponder: what might be the source of that extra overhead?

(c) A List of Tuples

In practice, you might read the data into a list and convert each line into some other data structure. Here is a program `readrides.py` that reads the entire file into a list of tuples using the `csv` module:

```
# readrides.py

import csv

def read_rides_as_tuples(filename):
    '''
    Read the bus ride data as a list of tuples
    '''
    records = []
    with open(filename) as f:
        rows = csv.reader(f)
```

```
        headings = next(rows)     # Skip headers
        for row in rows:
            route = row[0]
            date = row[1]
            daytype = row[2]
            rides = int(row[3])
            record = (route, date, daytype, rides)
            records.append(record)
    return records

if __name__ == '__main__':
    import tracemalloc
    tracemalloc.start()
    rows = read_rides_as_tuples('Data/ctabus.csv')
    print('Memory Use: Current %d, Peak %d' %
tracemalloc.get_traced_memory())
```

Run this program using `python3 -i readrides.py` and look at the resulting contents of `rows`. You should get a list of tuples like this:

```
>>> len(rows)
577563
>>> rows[0]
('3', '01/01/2001', 'U', 7354)
>>> rows[1]
('4', '01/01/2001', 'U', 9288)
```

Look at the resulting memory use. It should be substantially higher than in part (b).

(d) Memory Use of Other Data Structures

Python has many different choices for representing data structures. For example:

```
# A tuple
row = (route, date, daytype, rides)

# A dictionary
row = {
    'route': route,
    'date': date,
    'daytype': daytype,
    'rides': rides,
}

# A class
class Row:
    def __init__(self, route, date, daytype, rides):
        self.route = route
        self.date = date
```

```
        self.daytype = daytype
        self.rides = rides

# A named tuple
from collections import namedtuple
Row = namedtuple('Row', ['route', 'date', 'daytype', 'rides'])

# A class with __slots__
class Row:
    __slots__ = ['route', 'date', 'daytype', 'rides']
    def __init__(self, route, date, daytype, rides):
        self.route = route
        self.date = date
        self.daytype = daytype
        self.rides = rides
```

Your task is as follows: Create different versions of the `read_rides()` function that use each of these data structures to represent a single row of data. Then, find out the resulting memory use of each option. Find out which approach offers the most efficient storage if you were working with a lot of data all at once.

[Solution | Index | Exercise 1.6 | Exercise 2.2]
