[ Index | Exercise 2.4 | Exercise 2.6 ]

# Exercise 2.5

*Objectives:*

- Look at memory allocation behavior of lists and dicts
- Make a custom container

*Files Created:* None

## (a) List growth

Python lists are highly optimized for performing `append()` operations. Each time a list grows, it grabs a larger chunk of memory than it actually needs with the expectation that more data will be added to the list later. If new items are added and space is available, the `append()` operation stores the item without allocating more memory.

Experiment with this feature of lists by using the `sys.getsizeof()` function on a list and appending a few more items.

```
>>> import sys
>>> items = []
>>> sys.getsizeof(items)
64
>>> items.append(1)
>>> sys.getsizeof(items)
96
>>> items.append(2)
>>> sys.getsizeof(items)    # Notice how the size does not increase
96
>>> items.append(3)
>>> sys.getsizeof(items)    # It still doesn't increase here
96
>>> items.append(4)
>>> sys.getsizeof(items)    # Not yet.
96
>>> items.append(5)
>>> sys.getsizeof(items)    # Notice the size has jumped
128
>>>
```

A list stores its items by reference. So, the memory required for each item is a single memory address. On a 64-bit machine, an address is typically 8 bytes. However, if Python has been compiled for 32-bits, it might be 4 bytes and the numbers for the above example will be half of what's shown.

## (b) Dictionary/Class Growth

Python dictionaries (and classes) allow up to 5 values to be stored before their reserved memory doubles. Investigate by making a dictionary and adding a few more values to it:

```
>>> row = { 'route': '22', 'date': '01/01/2001', 'daytype': 'U', 'rides':
7354 }
>>> sys.getsizeof(row)
>>> sys.getsizeof(row)
240
>>> row['a'] = 1
>>> sys.getsizeof(row)
240
>>> row['b'] = 2
>>> sys.getsizeof(row)
368
>>>
```

Does the memory go down if you delete the item you just added?

Food for thought: If you are creating large numbers of records, representing each record as a dictionary might not be the most efficient approach--you could be paying a heavy price for the convenience of having a dictionary. It might be better to consider the use of tuples, named tuples, or classes that define `__slots__`.

## (c) Changing Your Orientation (to Columns)

You can often save a lot of memory if you change your view of data. For example, what happens if you read all of the bus data into a columns using this function?

```
# readrides.py

...

def read_rides_as_columns(filename):
    '''
    Read the bus ride data into 4 lists, representing columns
    '''
    routes = []
    dates = []
    daytypes = []
    numrides = []
    with open(filename) as f:
        rows = csv.reader(f)
        headings = next(rows)     # Skip headers
        for row in rows:
            routes.append(row[0])
            dates.append(row[1])
            daytypes.append(row[2])
            numrides.append(int(row[3]))
    return dict(routes=routes, dates=dates, daytypes=daytypes,
numrides=numrides)
```

In theory, this function should save a lot of memory. Let's analyze it before trying it.

First, the datafile contained 577563 rows of data where each row contained four values. If each row is stored as a dictionary, then those dictionaries are minimally 240 bytes in size.

```
>>> nrows = 577563     # Number of rows in original file
>>> nrows * 240
138615120
>>>
```

So, that's 138MB just for the dictionaries themselves. This does not include any of the values actually stored in the dictionaries.

By switching to columns, the data is stored in 4 separate lists.
Each list requires 8 bytes per item to store a pointer. So, here's a rough estimate of the list requirements:

```
>>> nrows * 4 * 8
18482016
>>>
```

That's about 18MB in list overhead. So, switching to a column orientation should save about 120MB of memory solely from eliminating all of the extra information that needs to be stored in dictionaries.

Try using this function to read the bus data and look at the memory use.

```
>>> import tracemalloc
>>> tracemalloc.start()
>>> columns = read_rides_as_columns('Data/ctabus.csv')
>>> tracemalloc.get_traced_memory()
... look at the result ...
>>>
```

Does the result reflect the expected savings in memory from our rough calculations above?

## (d) Making a Custom Container - The Great Fake Out

Storing the data in columns offers a much better memory savings, but the data is now rather weird to work with. In fact, none of our earlier analysis code from Exercise 2.2 can work with columns. The reason everything is broken is that you've broken the data abstraction that was used in earlier exercises--namely the assumption that data is stored as a list of dictionaries.

This can be fixed if you're willing to make a custom container object that "fakes" it. Let's do that.

The earlier analysis code assumes the data is stored in a sequence of records. Each record is represented as a dictionary. Let's start by making a new "Sequence" class. In this class, we store the four columns of data that were being using in the `read_rides_as_columns()` function.

```
# readrides.py

import collections
...
class RideData(collections.abc.Sequence):
    def __init__(self):
        self.routes = []      # Columns
        self.dates = []
        self.daytypes = []
        self.numrides = []
```

Try creating a `RideData` instance. You'll find that it fails with an error message like this:

```
>>> records = RideData()
Traceback (most recent call last):
...
TypeError: Can't instantiate abstract class RideData with abstract methods
__getitem__, __len__
>>>
```

Carefully read the error message. It tells us what we need to implement. Let's add a `__len__()` and `__getitem__()` method. In the `__getitem__()` method, we'll make a dictionary. In addition, we'll create an `append()` method that takes a dictionary and unpacks it into 4 separate `append()` operations.

```
# readrides.py
...

class RideData(collections.abc.Sequence):
    def __init__(self):
        # Each value is a list with all of the values (a column)
        self.routes = []
        self.dates = []
        self.daytypes = []
        self.numrides = []

    def __len__(self):
        # All lists assumed to have the same length
        return len(self.routes)

    def __getitem__(self, index):
        return { 'route': self.routes[index],
                 'date': self.dates[index],
                 'daytype': self.daytypes[index],
                 'rides': self.numrides[index] }
```

```
    def append(self, d):
        self.routes.append(d['route'])
        self.dates.append(d['date'])
        self.daytypes.append(d['daytype'])
        self.numrides.append(d['rides'])
```

If you've done this correctly, you should be able to drop this object into the previously written `read_rides_as_dicts()` function. It involves changing only one line of code:

```
# readrides.py
...

def read_rides_as_dicts(filename):
    '''
    Read the bus ride data as a list of dicts
    '''
    records = RideData()      # <--- CHANGE THIS
    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': route,
                'date': date,
                'daytype': daytype,
                'rides' : rides
                }
            records.append(record)
    return records
```

If you've done this right, old code should work exactly as it did before. For example:

```
>>> rows = readrides.read_rides_as_dicts('Data/ctabus.csv')
>>> rows
<readrides.RideData object at 0x10f5054a8>
>>> len(rows)
577563
>>> rows[0]
{'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': 7354}
>>> rows[1]
{'route': '4', 'date': '01/01/2001', 'daytype': 'U', 'rides': 9288}
>>> rows[2]
{'route': '6', 'date': '01/01/2001', 'daytype': 'U', 'rides': 6048}
>>>
```

Run your earlier CTA code from Exercise 2.2. It should work without modification, but use substantially less memory.

## (e) Challenge

What happens when you take a slice of ride data?

```
>>> r = rows[0:10]
>>> r
... look at result ...
>>>
```

It's probably going to look a little crazy. Can you modify the `RideData` class so that it produces a proper slice that looks like a list of dictionaries? For example, like this:

```
>>> rows = readrides.read_rides_as_columns('Data/ctabus.csv')
>>> rows
<readrides.RideData object at 0x10f5054a8>
>>> len(rows)
577563
>>> r = rows[0:10]
>>> r
<readrides.RideData object at 0x10f5068c8>
>>> len(r)
10
>>> r[0]
{'route': '3', 'date': '01/01/2001', 'daytype': 'U', 'rides': 7354}
>>> r[1]
{'route': '4', 'date': '01/01/2001', 'daytype': 'U', 'rides': 9288}
>>>
```

[ Solution | Index | Exercise 2.4 | Exercise 2.6 ]

---