CSV

Comma Separated Values (CSV) is a common data file format that represents data as a row of values, separated by a delimiter, which is typically a comma. Data in spreadsheets and databases matches this format nicely, so CSV is often used as an export format. If you work in data science, CSV is ubiquitous, so it makes sense to spend some time learning more about the format and developing skills to manipulate data once its read into memory.

Here is some example CSV data representing financial information from Apple Computer.

```
Date,Open,High,Low,Close,Volume,Adj Close
2009-12-31,213.13,213.35,210.56,210.73,88102700,28.40
2009-12-30,208.83,212.00,208.31,211.64,103021100,28.52
2009-12-29,212.63,212.72,208.73,209.10,111301400,28.18
2009-12-28,211.72,213.95,209.61,211.61,161141400,28.51
```

header many CSV files start with an initial header row, which gives column names for the data

data data in CSVs is separated by commas, but any delimiter can be used.

Python and CSV: Readers

Suppose the the contents of the above CSV were in a file called `aapl.csv`. One could open that CSV and stream through the data using the `csv` module and the following syntax.

```
import csv

with open('aapl.csv', 'r') as fin:
    print(list(csv.reader(fin)))
```

The `reader` object is iterable. Each row is a list of strings that were split by the delimiter, which by default is the comma. This yields the following output.

```
[['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close'],
 ['2009-12-31', '213.13', '213.35', '210.56', '210.73', '88102700', '28.40'],
 ['2009-12-30', '208.83', '212.00', '208.31', '211.64', '103021100', '28.52'],
 ['2009-12-29', '212.63', '212.72', '208.73', '209.10', '111301400', '28.18'],
 ['2009-12-28', '211.72', '213.95', '209.61', '211.61', '161141400', '28.51']]
```

If I wanted to find the highest stock price over the time period given by this data, I could write:

```
import csv
HIGHPRICECOL = 2

with open('aapl.csv', 'r') as fin:
    data = list(csv.reader(fin))
    prices = [row[HIGHPRICECOL] for row in data[1:]]
    print(max(prices))
```

This code makes several assumptions and uses some new python constructs, all of which are worth mentioning:

- all the data is held in memory at once so this is not a *streaming* algorithm;
- we use a list comprehension when assigning the prices variable; and
- we assume that we know the column index for price, which is 2.

Let's write this without loading all the data into memory and without knowing which column the 'High' price occupies:

```
import csv
COLNAME = "High"

with open('aapl.csv', 'r') as fin:
  maxprice = float('-inf')
  maxpricecol = None

  for rownum, row in enumerate(csv.reader(fin)):
    if rownum == 0:
      maxpricecol = row.index(COLNAME)
    else:
      maxprice = max(maxprice, float(row[maxpricecol]))

  print(maxprice)
```

This code also uses some Python constructs that are new to us. The first is `float('-inf')`, which is a Python way of specifying a value that is *always* smaller than any other value. The `maxpricecol` variable we declare and initialize to `None`. The `index` method returns the index or position of the value ``` ``High'' ``` in the row. Again, note that this algorithm is streaming, so we only consider one line at a time.

Practice

Imagine I have a file called `realestate.csv` that contains real estate transactions in Sacramento over 5 days. The format of the data is

`street,city,zip,state,beds,baths,sqft,type,sale_date,price,lat,long`.

Write a short script that finds that average sale price for every transaction in the file. You know that price occurs at index 9, that the `statistics.mean` function is available, and that the data can easily fit into memory.

```
import sys
import csv
import statistics
PRICECOL = 9

def mean_sale(filename):
  with open(filename, 'r') as fin:
    rows = list(csv.reader(fin))[1:]
    prices = [float(row[PRICECOL]) for row in rows]
    return(statistics.mean(prices))

if __name__ == '__main__':
  print(mean_sale(sys.argv[1]))
```

Now imagine writing a function that returned the mean sale price of houses over 2000 square feet. Square footage is given by the column at index 6.

```
import csv
import sys
import statistics
```

```

PRICECOL = 9
SQFTCOL = 6
SQFTMIN = 2000

def mean_sale_high(filename):
    with open(filename, 'r') as fin:
        rows = list(csv.reader(fin))[1:]
        prices = []
        for row in rows:
            if int(row[SQFTCOL]) > SQFTMIN:
                prices.append(float(row[PRICECOL]))
        return(statistics.mean(prices))

if __name__ == '__main__':
    print(mean_sale(sys.argv[1]))
```

You can also use an `if` statement in the list comprehension for some truly beautiful code.

```
def mean_sale_high2(filename):
    with open(filename, 'r') as fin:
        rows = list(csv.reader(fin))[1:]
        prices = [float(row[9]) for row in rows if int(row[6]) > 2000]
        print(statistics.mean(prices))
```

CSV Data in Strings

Suppose that the CSV data, however, is in a string `data`, instead of a file. In this case, one would use the `io.StringIO` type to wrap the string inside something that behaves like a *file object*. You can think of this as *buffering* the string.

```
import csv
import io

data = 'purple,cow,moo\nhappy,moose,grunt'
reader = csv.reader(io.StringIO(data))
for row in reader:
    print("*".join(row))
```

Reader Options

There are many options when creating a CSV reader. Here are some, with definitions coming directly from the API[1]:

`delimiter` A one-character string used to separate fields. It defaults to ','.

`escapechar` On reading, the `escapechar` removes any special meaning from the following character. It defaults to `None`, which disables escaping.

`lineterminator` The string used to terminate lines produced by the writer. It defaults to `'\r\n'`. Note The reader is hard-coded to recognize either `'\r'` or `'\n'` as end-of-line, and ignores line terminator. This behavior may change in the future.

[1]`https://docs.python.org/3.4/library/csv.html#csv-fmt-params`

As an extreme example, suppose we wanted to represent a bunch of data that was just commas. One could use a different delimiter

```
,|,,|,
,,|,|,,
```

and use `csv.reader(filename.csv, delimiter="|")` to create the correct reader. We could also escape the commas

```
\,,\,\,,\,
\,\,,\,,\,\,
```

and use `csv.reader(filename.csv, escapechar="\\")` to create the correct reader. Notice that we need to escape the backslash inside the character string.

Writers

CSV Writer objects accept any object that has a `write` method (file objects, `StringIO` objects, etc.) and formats CSV data using the `writerow` or `writerows` method. Here's an example. Suppose that data is a list of NESCAC school information.

```
data = [['Williams', 'Ephs', 'Purple Cows'],
              ['Middlebury', 'Panthers', 'Panther']]
```

To write this to the file called `nescac.csv` we would use the following code

```
import csv
with open('nescac.csv', 'w', newline='') as csvfile:
   writer = csv.writer(csvfile, delimiter=',')
   writer.writerow(['School', 'Nickname', 'Mascot'])
   writer.writerows(data)
```

Practice

Suppose you had a list of constellations and their galactic coordinates (right ascension and declination) in CSV format.

```
constellation, right ascension, declination
Sagittarius,19,-25
Taurus, 4.9, 19
Perseus, 3, 45
```

Write a function that takes a filename `file` in CSV format and returns a list of constellations. Suppose that you know one of the headers is labelled `constellation`, but not which one. Suppose further that you can easily fit all the data in memory.

```
with open(file, newline='') as fp:
  data = [row for row in csv.reader(file)]
  col = data[0].index('constellation')
  return [row[col] for row in data[1:]]
```