CS 2316 Data Manipulation for Engineers

Comma-Seaprated Values Files

Christopher Simpkins
chris.simpkins@gatech.edu

Text File IO

- File IO is done in Python with the built-in `File` object which is returned by the built-in `open` function
- Use the `'w'` open mode for writing

```
$ python
>>> f = open("hello.txt",'w') # open for writing, create if necessary
>>> f.write("Hello, file!\n") # write string to file; notice \n ending
>>> f.close()                 # close file, causing it to write to disk
>>> exit()
$ cat hello.txt
Hello, file!
```

- Use the `'r'` open mode for reading

```
$ python
>>> f = open("hello.txt", "r") # open for reading in text mode
>>> contents = f.read()        # slurp the whole file into memory
>>> contents
'Hello, file!\n'
>>> exit()
```

Reading Lines from Text Files

- Text files often have data split into lines
- the `readlines()` function reads all lines into memory as a list

```
>>> f = open('lines.txt', 'r')
>>> f.readlines()
['line 1\n', 'line 2\n', 'line 3\n']
```

- `readline()` reads one line at a time, returns empty string when fully read
- re-open file or use `seek()` to go back to beginning of file

```
>>> f = open('lines.txt', 'r')
>>> f.readline()
'line 1\n'
>>> f.readline()
'line 2\n'
>>> f.readline()
'line 3\n'
>>> f.readline()
''
>>> f.seek(0)
>>> f.readline()
'line 1\n'
```

Processing Lines in a Text File

Could use `readlines()` and iterate through list it returns

```
>>> f = open('lines.txt', 'r')
>>> for line in f.readlines():
...     print line
...
line 1
line 2
line 3
```

Better to use the built-in file iterator

```
>>> for line in open('lines.txt', 'r'):
...     print line
...
line 1
line 2
line 3
```

Files are Buffered

Try a little experiment. create a subdirectory named `foo`, cd to your new empty `foo` directory, lauch a Python shell, create open a new file named `bar`, and write something to it:

```
$ mkdir foo
$ cd foo
$ python3
Python 3.4.0 (v3.4.0:04f714765c13, Mar 15 2014, 23:02:41) ...
>>> bar = open("bar", 'w')
>>> bar.write("last call!")
10
>>>
```

At this point, open another command shell or use your graphical file explorer to view the contents of the `bar` file. It's empty. Now go back to your Python shell and do:

```
>>> bar.close()
```

Now view the contents of the `bar` file again. It has the text from the previous `write()` call. Files are buffered, and the buffer isn't (guaranteed to be) flushed to disk until the file object is closed or the

Context Management with `with`

Python has *context managers* to close resources automatically. A contaxt manager has the form

> `with` *expression* `as` *variable:*
>
> *block*

which is equivalent to

> *variable = expression*
>
> *block*
>
> *variable*`.close()`

For example, the previous bar example is:

```
>>> with open('bar', 'w') as bar:
...     bar.write('last call!')
...
```

And the file is closed and flushed to disk automatically after the block under the `with` statement finishes.

Comma-Separated Value Files

Say we have data in a comma-separated value file

```
$ cat capitals.dat # could be .dat, .csv, or anything
Japan,Tokyo
France,Paris
Germany,Berlin
U.S.A.,Washington, D.C
```

Can use line-by-line file reading with the split() function we saw earlier to process comma-separated value files

```
$ python
>>> capitals = {} # initialize a dictionary to hold our capitals data
>>> for line in open('capitals.dat', 'r'): # for each line in file
...     k, v = line.split(',')             # split into key and value
...     capitals[k] = v                    # add key:value to dict
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: too many values to unpack
```

Why didn't it work?

CSV Separator Characters

We can troubleshoot in the Python interpreter

```
>>> for line in open('capitals.dat', 'r'):
...     print line.split(',')
...
['Japan', 'Tokyo\n']
['France', 'Paris\n']
['Germany', 'Berlin\n']
['U.S.A.', 'Washington', ' D.C\n']
```

- There's a comma in Washington, D.C. that was taken as a separator
- So let's change the capitals.dat file to use semicolons as the separators

```
$ cat capitals.dat
Japan;Tokyo
France;Paris
Germany;Berlin
U.S.A.;Washington, D.C
```

CSV Files in Practice

Now our capitals.dat file is readable as a "comma"-separated value file

```
>>> capitals = {}
>>> for line in open('capitals.dat', 'r'):
...     k, v = line.split(';')
...     capitals[k] = v
...
>>> capitals
{'Japan': ' Tokyo\n', 'U.S.A.': ' Washington, D.C\n', 'Germany': '
    Berlin\n', 'France': ' Paris\n'}
```

- But the values have leading whitespace and trailing `'\n'` characters from the data file
- We can make our code more robust with `strip()`, which removes leading and trailing whitespace and non-printing chars

```
>>> for line in open('capitals.dat', 'r'):
...     k, v = line.split(';')
...     capitals[k.strip()] = v.strip()
...
>>> capitals
{'Japan': 'Tokyo', 'U.S.A.': 'Washington, D.C', 'Germany': 'Berlin',
    'France': 'Paris'}
```

The csv Module

The best way to process CSV files is with the csv module.

```
>>> import csv
>>> scripters = [
...     ['Perl', 'Larry Wall'],
...     ['Python', 'Guido Van Rossum'],
...     ['Ruby', 'Yukihiro Matsumoto']
... ]
>>> with open('scripters', 'wt') as fout:
...     csvout = csv.writer(fout)
...     csvout.writerows(scripters)
...
>>> ^D
$ cat scripters
Perl,Larry Wall
Python,Guido Van Rossum
Ruby,Yukihiro Matsumoto
```

- The with statement is a context manager
- After the with block ends, the file is automatically closed

Reading CSV Files

We can read our scripters file with

```
>>> import csv
>>> with open('scripters', 'r') as fin:
...     csvin = csv.reader(fin)
...     scripters = [line for line in csvin]
...
>>> scripters
[['Perl', 'Larry Wall'], ['Python', 'Guido Van Rossum'], ['Ruby',
    'Yukihiro Matsumoto']]
>>>
```

Column Headers in CSV Files

Use a `DictReader` to store the records from the CSV file in a `dict`.

```
>>> import csv
>>> with open('scripters', 'r') as fin:
...     csvin = csv.DictReader(fin, fieldnames=['langauge', 'creator'])
...     scripters = [line for line in csvin]
...
>>> scripters
[{'creator': 'Larry Wall', 'langauge': 'Perl'}, {'creator': 'Guido Van
    Rossum', 'langauge': 'Python'}, {'creator': 'Yukihiro Matsumoto',
    'langauge': 'Ruby'}]
```

And we can use a `DictWriter` to write a CSV file with a header line.

```
>>> with open('scripters', 'w') as fout:
...     csvout = csv.DictWriter(fout, fieldnames=['langauge',
    'creator'])
...     csvout.writeheader()
...     csvout.writerows(scripters)
...
>>> ^D
$ cat scripters
langauge,creator
Perl,Larry Wall
Python,Guido Van Rossum
```

CSV Details

CSV files can be complex.

- Different delimiters can be used.
- Delimiter characters can appear in fields.
- Fields can be surrounded with "quotes".
- Different operating systems may use different line endings.

The CSV module handles all of these issues for you. Read the CSV module documentation to become familiar with its capabilities.