Python File Input/Output Reference

CPSC 231 - FALL 2018

This supplementary handout describes a simple means of reading data from text files and writing data to text files that you might find useful for Assignment #3. Our official course text covers this topic as well, but not until Chapter 3. We thought it best to get a head start.

Reading and writing text files is easy to do with Python using the built-in `open()` function. It returns to you a *file object* that allows you to perform input and output operations on the file. Suppose you have a text file, `locations.txt`, whose contents are shown on the right. It is simple enough to read the contents of the file into an array of `str` objects, with each line of the file as a separate string.

The contents of our text file, `locations.txt`:

```
Calgary,51.0 N,114.1 W
Vancouver,49.3 N,123.1 W
Toronto,43.7 N,79.4 W
```

file-input.py

```
input_file = open('locations.txt')
# read each line in the file and store as an array of str objects
lines = input_file.readlines()
input_file.close()

num_lines = len(lines)
print('Read', num_lines, 'lines of text:')
for i in range(num_lines):
    line_of_text = lines[i]
    print(line_of_text)
```

Output of running `file-input.py`:

```
Read 3 lines of text:
Calgary,51.0 N,114.1 W

Vancouver,49.3 N,123.1 W

Toronto,43.7 N,79.4 W
```

If you run this program, it will echo the contents of the file to your console. But notice how there are extra blank lines in the output. This happens because the `readlines()` function keeps the newline characters from your file at the end of each `str`, which gets printed along with the newline that the `print()` function inserts. To get rid of this extra newline, you can use the `rstrip()` function, which removes whitespace at the end of the string, before you print the line, like this:

```
line_without_newline = line_of_text.rstrip()
```

Now suppose you wanted to work with the data values, which are separated by commas in our input file, rather than with the whole line of text. Python provides an easy way to accomplish this through the string object's `split()` method. It will chop up a string at the delimiter you provide, and give you back an array of the substrings you want. The listing below shows how you might use this function.

file-input-split.py

```
input_file = open('locations.txt')
lines = input_file.readlines()
input_file.close()

# we can also iterate through elements of an array like this
for line_of_text in lines:
    line_without_newline = line_of_text.rstrip()
    elements = line_without_newline.split(',')
    city = elements[0]
    latitude = elements[1]
    longitude = elements[2]
    print(city, 'is at', latitude, 'and', longitude)
```

Output of `file-input-split.py`:

```
Calgary is at 51.0 N and 114.1 W
Vancouver is at 49.3 N and 123.1 W
Toronto is at 43.7 N and 79.4 W
```

Writing text to an output file is nearly as simple through the same method. You can pass a `'w'` to the `open()` function to tell it that you want to write to the file. Then simply call `write()` on the file object to write to your file. Note that the `write()` method only accepts string objects, but you can convert other data types to strings by using the `str()` function. An example use is shown below.

file-output.py

```
# open powers2.txt for writing
output_file = open('powers2.txt', 'w')
output_file.write('The first 10 powers of two are:\n')
output_file.write(str(2**1))
for i in range(2, 11):
    output_file.write(',' + str(2**i))
# write a final newline character for the 2nd line of text
output_file.write('\n')
output_file.close()
```

The contents of the file `powers2.txt` written by our program:

```
The first 10 powers of two are:
2,4,8,16,32,64,128,256,512,1024
```

Note that, unlike the `print()` function, `write()` does not automatically add a newline character to the end of the output. It works more like the `stdio.write()` function from our Booksite Library. If you want a new line in your file output, you must explicitly write a `\n` as shown in the example. Remember that it's always a good idea to close the file by calling `close()` on the file object when you're done writing to it!

The Booksite Library from our course text also provides a mechanism for reading from and writing to text files through their input and output streams. These are presented on pages 380–387 with a good example in Program 3.1.11, which you're free to use if you prefer. It's easier to understand exactly what's happening in these programs after we discuss objects and object-oriented programming, which is why the text defers file input/output streams to Chapter 3. However, it should be simple enough for us to do what we need to do with our files if we just follow the syntax in this handout, or in your text, as carefully as we can for now.