EML4930/EML6934: Lecture 09

Reading, writing, and modifying text and CSV files in Python

Charles Jekel

October 26, 2017

Reminder: Quiz at the end of next class (11/02/2017)

- Quiz will cover anything from lecture 00 - lecture 05
- You should be able to write the code to plot a curve in Python from memory!
- Sample quiz questions will be posted
- I view lectures 00 - 05 as the basics of Python (these are the lectures the final exam will be about! + this lecture for reading and writing files) Lectures 06 - 13 build upon these basics for advanced cases.

Let's take a look at HW 09

- You are presented with a practical structural optimization problem
- You must create a couple functions to aid in the optimization
- One function will take read an input template and write a modified version of the file
- The other function reads specific values from a large output file
- This HW is to prepare you for *black-box* optimization (think of the input template and output file your results from some expensive simulation that you'll be running!)

Topics for today's lecture

`https://docs.python.org/3/tutorial/inputoutput.html`

- with statement
- open function
- methods for file objects
- read file line by line
- string manipulation
- itertools islice efficient loops
- read a csv file
- write a csv file

I've created a demo.txt file which contains the following

This is line 1 of a demo txt file.

This is line 2 of a demo txt file.

This is line 3 of a demo txt file.

This is line 4 of a demo txt file.

This is line 5 of a demo txt file.

This is line 6 of a demo txt file.

This is line 7 of a demo txt file.

This is line 8 of a demo txt file.

This is line 9 of a demo txt file.

This is line 10 of a demo txt file.

I replace_me_1 Python.

I replace_me_2 MATLAB.

The open() function in Python to load files.

open() returns a file object as is most commonly used as

```
open(filename, mode)
```

The following code will create a file object f by reading demo.txt.

```
f = open('demo.txt', 'r')
```

Typically the file object are opened in text mode, which means that you work with Python strings.

modes for open() text files

Mode	Description
'r'	open file read only
	open file read only (assumed if left blank)
'w'	open file for writing only (existing files will be erased)
'a'	open file for appending (data is written at the end of file)
'r+'	open file for both reading and writing

Unix and Windows end statements - text mode

- Each line in the file object will have some kind of end line statement
- a Windows file with have an end of string statement that looks like '\r \n'
- Unix file will have an end of string statement that looks like '\n'
- these statements indicate where on the text file does a new line begin
- When writing files Python automatically specifies platform dependent line endings.

modes for open() binary files

Mode	Description
'rb'	open file read only
'b'	open file read only (assumed if left blank)
'wb'	open file for writing only (existing files will be erased)
'ab'	open file for appending (data is written at the end of file)
'r+b'	open file for both reading and writing

use .close() to close a file

Let's say you opened demo.txt as

```
f = open('demo.txt', 'r')
```

The file will stay open (eating memory) until you explicitly close it.

If you don't explicitly close a file, Python's garbage collection will eventually destroy the object and close the file for you. This is dangerous because it won't be consistent!

To close the text file we run

```
f.close()
```

open() and .close() in summary

- use open(filename, mode)
- by default files are opened in text mode
- you can also open binary files
- use .close() to close a file

Now that I’ve shown you open() and .close()

Never ever use open() and .close().

It’s bad to forget to close a file. So bad that you should never use open(). Rather we will be using the with statement which closes files automatically!

Let’s take a look at the with statement

What is a with statement?

`https://docs.python.org/3/reference/compound_stmts.html#with`

The with statement is used to wrap the execution of a block with methods defined by a context manager (see section With Statement Context Managers). This allows common try-except-finally usage patterns to be encapsulated for convenient reuse.

How to open a file: use a with statement

```
# start a with statement, this opens the
with open('demo.txt', 'r') as f:
    # let's read the text data
    read_data = f.read()
    # let's print the read data
    print(read_data)
# once the indent is removed, the file is closed
# f is no longer in memory!, though read_data is
```

methods of file objects

Method	Description
.read()	return the entire contexts of a file as a single string
.read(size)	returns the contexts of your file for specified size
.readline()	reads a single line from the file (iterator)
.readlines()	read the entire file as a list, where each line is list item
.write(string)	write the string to the file (requires write mode)
.tell()	returns an integer indicating the current position in f
.seek(off,from)	go to some integer offset in file , from is either 1,2,or 3

Note: The end of a file will be indicated by an empty string ”, however a blank line will contain '\n'.

Reading a file line by line is easy in Python

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
with open('demo.txt', 'r') as f:
    for line in f:
        print(line)
```

Methods for string manipulation - join

Remember how I said strings were really objects long ago?

Let's pretend we have some list of strings

```
data = ['a', 'b', 'c', 'd']
```

we can use .join() to make a single string containing all of the strings

```
data_flat_string = ''.join(data)
```

```
print(data_flat_string)
```

will return a single string of 'abcd'

Methods for string manipulation - split

We can use the .split(string_separator) to break a string into a list of strings, separated by string_separator

Example: break up the string wherever there is a space

```
a = 'This is line 4 of a demo txt file.'
b = a.split(' ')
print(b)
```

b would be the following list

```
['This', 'is', 'line', '4', 'of', 'a', 'demo', 'txt', 'file.']
```

Methods for string manipulation - split

Example: break up the string wherever there is the lowercase letter i

```
c = 'This is line 4 of a demo txt file.'
d = c.split('i')
print(d)
```

d would be the following list

```
['Th', 's ', 's l', 'ne 4 of a demo txt f', 'le.']
```

Methods for string manipulation - replace

We can use .replace(my_old_string, my_new_string) to replace all occurrences of my_old_string with my_new_string.

Let's do a more complicated example, where we replace the instances of replace_me_1 and replace_me_2 of demo.txt with my own custom strings. Then we'll save the modified file as final.txt

Example: replacing strings

```
with open('demo.txt', 'r') as d, open('final.txt', 'w') as f:
    # read all of demo.txt into memory
    my_txt = d.read()
    # replace 'replace_me_1'
    my_txt = my_txt.replace('replace_me_1', 'love')
    # replace 'replace_me_2'
    my_txt = my_txt.replace('replace_me_2', 'blah')
    # write my file to final.txt
    f.write(my_txt)
```

Example: replacing strings results of final.txt

```
This is line 1 of a demo txt file.
This is line 2 of a demo txt file.
This is line 3 of a demo txt file.
This is line 4 of a demo txt file.
This is line 5 of a demo txt file.
This is line 6 of a demo txt file.
This is line 7 of a demo txt file.
This is line 8 of a demo txt file.
This is line 9 of a demo txt file.
This is line 10 of a demo txt file.
I love Python.
I blah MATLAB.
```

Summary of string manipulation and read write so far

- reading and writing text files in Python is easy
- use with statements! Don't use open() .close()!
- strings can be easily modified using the the string methods within string objects
- I just showed you how to open a file, perform automatic find and replace, and then save as a new file in like five lines of code
- Python Python Python!

Efficient loops using itertools.islice

itertools is a built-in library for efficient iterators

`https://docs.python.org/3/library/itertools.html#module-itertools`

islice let's us iterate through some large data, while only loading certain sections into memory!

methods of use:

```python
from itertools import islice
islice(iterable, stop) # starts from the beginning until stop
islice(iterable, start, stop) # iterate from start to stop
islice(iterable, start, stop, step)
```

Like with other Python libraries, start is inclusive, while stop is exclusive.

Example: islice demo on demo.txt

Let’s only read lines 5 - 10, line by line. The trick here is Python line numbering starts at zero!.

```
from itertools import islice
with open('demo.txt', 'r') as d:
    for line in islice(d, 4, 10):
        print(line)
```

Why is this an efficient loop? because I’ve only loaded the particular lines I want to read into memory, and skipped all other lines!

Example: a list of numbers as floats from demo.txt

In this examples I read the first 10 lines of demo.txt. My intention is to store numbers from each of the 10 lines as floats. I take advantage that the number is always the fourth item when I split the line by a space.

```python
with open('demo.txt', 'r') as d:
    my_numbers = []
    # open the first 10 lines
    for line in islice(d,0,10):
        # split the line by spaces
        temp = line.split(' ')
        # the nubmer is always the fourth item in the temp list
        my_numbers.append(float(temp[3]))
        # float(temp[3]) coverts temp[3] to a float
```

This creates a list of floats. my_numbers is [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0] as read from demo.txt (you can do something similar on the HW!)

Summary of efficient loops

- Use itertools.islice if you need to loop through a few specific lines in a text file efficiently
- islice is efficient because it won't load the entire text file into memory, just the parts you need
- you can take advantage of splitting a line by spaces to often read numbers from formatted text files!
- you have enough information to now complete the HW

csv - the built in library for working with CSV files

`https://docs.python.org/3/library/csv.html`

The csv library is built on the principles established thus far in this lecture.

I've create a demo.csv to work through on this problem. This .csv file was created using Microsoft Excel.

I'm going to demonstrate simple reading of a csv file into a Python list, and simple writing a list as a csv file.

Example: reading demo.csv into a list

```
with open('demo.csv', 'r') as my_csv:
    my_data = [] # my blank list
    # you should specify the delimiter
    my_csv_data = csv.reader(my_csv, delimiter=',')
    # you need to iterate through the csv one row at a time
    for row in my_csv_data:
        my_data.append(row)
```

In this example, the list my_data will contain the demo.csv as a Python list.

Example: writing a Python list to csv file

In this example I write a random x y numbers to a xy.csv file.

```
import numpy as np
x = np.random.random(10); y = np.random.random(10)
# convert to strings
x=x.astype('string'); y=y.astype('string')
with open('xy.csv', 'w') as my_csv:
    my_csv_write = csv.writer(my_csv, delimiter=',')
    # write the header
    my_csv_write.writerow(['x','y'])
    # write the csv row by row
    for row in zip(x,y):
        my_csv_write.writerow(row)
```

Example: dictionary reading from header

In this example I print just the 'radius(mm)' values

```
with open('demo.csv', 'r') as my_csv:
    my_data = []
    # this assumes the top row is keywords of a dictionary
    my_csv_data = csv.DictReader(my_csv, delimiter=',')
    for row in my_csv_data:
        # I can specify the specific keyword
        print(row['radius(mm)'])
```

Example: dictionary writing

In this example I write a random w z numbers to a wz_dict.csv file using the dictionary format.

```
w = np.random.random(10); z = np.random.random(10)
# convert to strings
w=w.astype('string'); z=z.astype('string')
with open('wz_dict.csv', 'w') as my_csv:
    # specify the header
    fieldnames = ['w', 'z']
    my_csv_write = csv.DictWriter(my_csv, delimiter=',',
        fieldnames=fieldnames)
    # write the header
    my_csv_write.writeheader()
    # write the csv row by row as dictionary
    for row in zip(x,y):
        my_csv_write.writerow({'w': row[0], 'z': row[1]})
```

Summary of CSV

- there is a built in csv library specific for csv files
- it's a bit clunky, so clunky that most people prefer to import csv files with pandas (library for data frames)
- you load files as if they are plain text files, then use the csv.reader to read the files row by row
- you generally write csv files row by row
- the dictionary functionality is meant to simplify writing and reading csv files to dictionaries