Lecture 10: File I/O

CS 50

February 24, 2026

Review: Sequences

- sequences are ordered sets of values
  - ranges are sequences of integers
  - strings are sequences of characters
  - tuples are sequences of arbitrary (possibly mixed) types
  - lists are mutable sequences of arbitrary (possibly mixed) types

- we can perform operations on sequences
  - indexing (e.g., my_list0])
  - slicing (e.g., my_string 1:3])
  - looping (with for loop) (e.g., for char in “hello”:)
  - check membership (e.g., if element in [1, -2, 50, “abc”]:)

Review: Sequences

String Examples

```
string = "Hello world!"
```

- can:
  - loop over contents

```
for char in string:
    print(char)
```

  - check membership

```
check = "!" in string
```

  - get the length

```
length = len(string)
```

  - index into them

```
char = string[3]
```

List Examples

```
my_list = [1,-2,50,"abc"]
```

- can:
  - loop over contents

```
for element in my_list:
    print(element)
```

  - check membership

```
check = 50 in my_list
```

  - get the length

```
length = len(my_list)
```

  - index into them

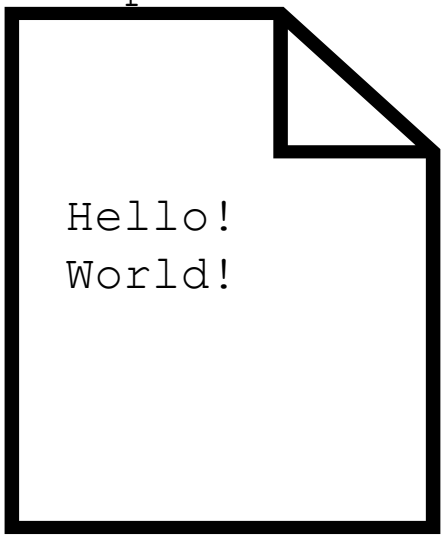
```
x = my_list[3]
```

files are also sequences

- a file is a sequence of strings
- … so we can use the keyword **in** to loop through the lines of a file!

```
for line in file:
    print(line)
```

example1.txt



each line contains \n at end

```
Hello!

World!
```

Reading and Writing Files

- reading from a file:

```
file_in = open("filename.txt", "r")
for line in file_in:
        # do something with each line
file_in.close()
```

  - or can use file_in.readline() to get one line at a time
  - reading from a file keeps going from where it was

Remember to close your files!

Example 1: Reading Files

- Define a function `num_chars` that takes one parameter `filename` and returns the number of characters in that file.

Reading and Writing Files

- reading from a file:

```
file_in = open("filename.txt", "r")
for line in file_in:
      # do something with each line
file_in.close()
```

  - or can use file_in.readline() to get one line at a time
  - reading from a file keeps going from where it was
- writing to a file:

```
file_out = open("filename.txt", "w")
file_out.write("Line 1 in the file\n")
file_out.write("Next line in the file\n")
file_out.close()
```

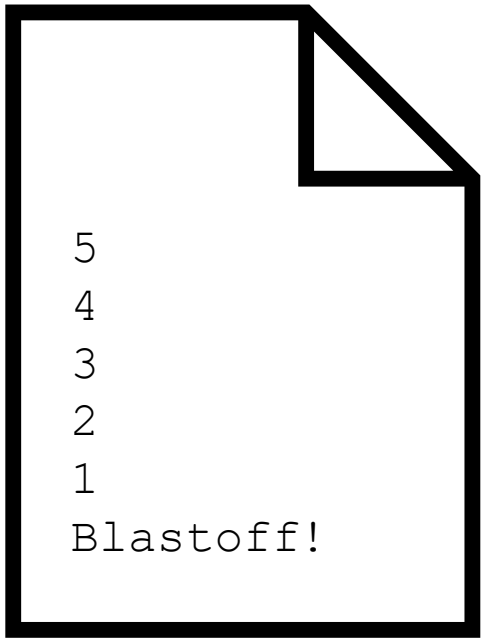
Remember to close your files!

Example 2: Writing Files

- Define a function `write_countdown` that takes one parameter (an integer `num`) and creates a file called `countdown.txt` that contains the numbers num… 1 (each on separate lines) followed by the line "Blastoff!"

```
write_countdown(5)
```

countdown.txt

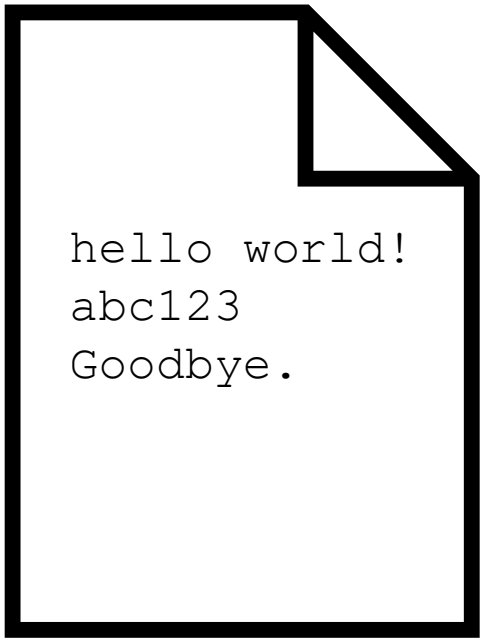


Exercise 1

- Define a function `combine_files` that takes three parameters (`infile1`, `infile2`, and `outfile`), all of which are strings, and creates a new file named `outfile` whose contents are the contents of the file named `infile1` followed by the contents of the file named `infile2`.

Exercise 2

- Define a function `reverse_file` that takes two parameters (`infile` and `outfile`), both of which are strings, and creates a new file named `outfile` whose contents are the contents of the file named `infile` but with the order of the lines reversed.



What about non-text files?

- Other types of files are just files with a particular format
- In some cases, they are formatted sequences of characters
  - Can manipulate these in Python using what we know
- In other cases, they are formatted sequences of non-printable character-like things (called bytes or binary)
  - Can manipulate these in Python too (but beyond the scope of this class)

Example 3: Image Files (.ppm)

- Pixels represent small areas of color
  - Red, Green, and Blue
  - Each value is the brightness for the color
  - Can make ~any color from RGB

- A PPM image consists of
  - Header
  - Body that consists of rows of pixels

header

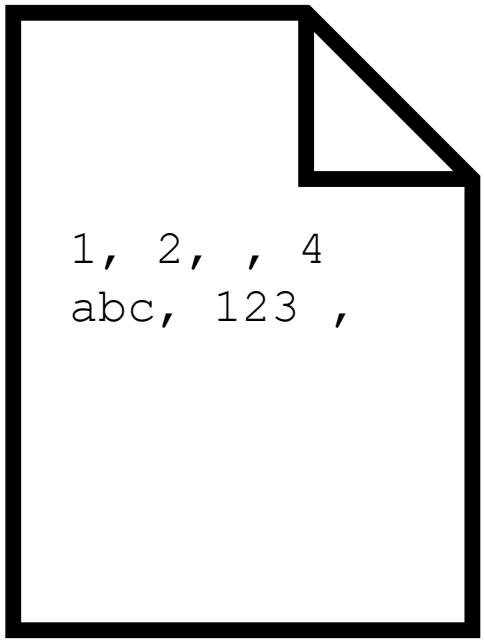
body

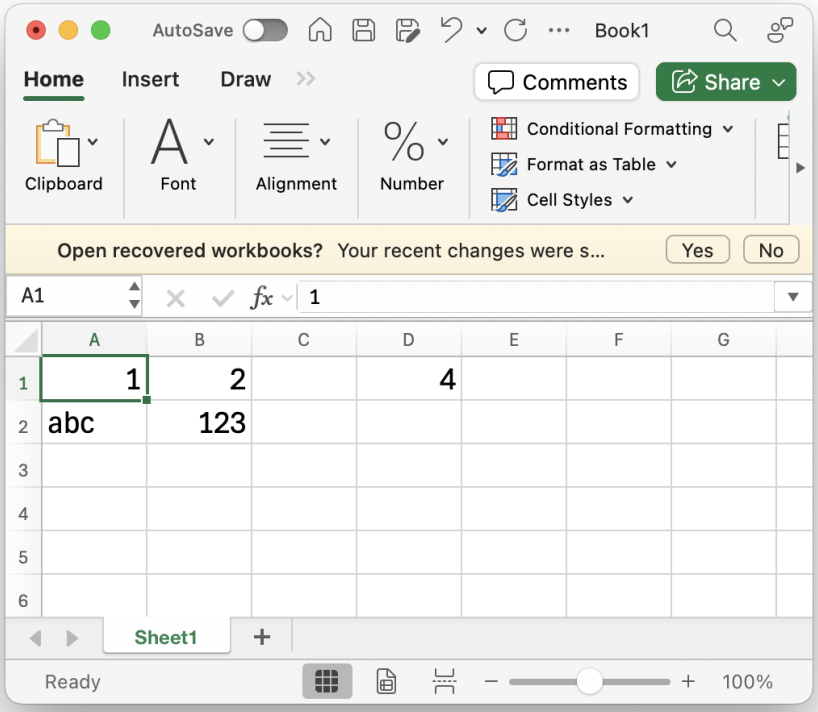
```
P3
3 2
255
255 0 0    0 255 0      0 0 255
122 23 55 128 200 100 0 0 0
```

#cols & #rows

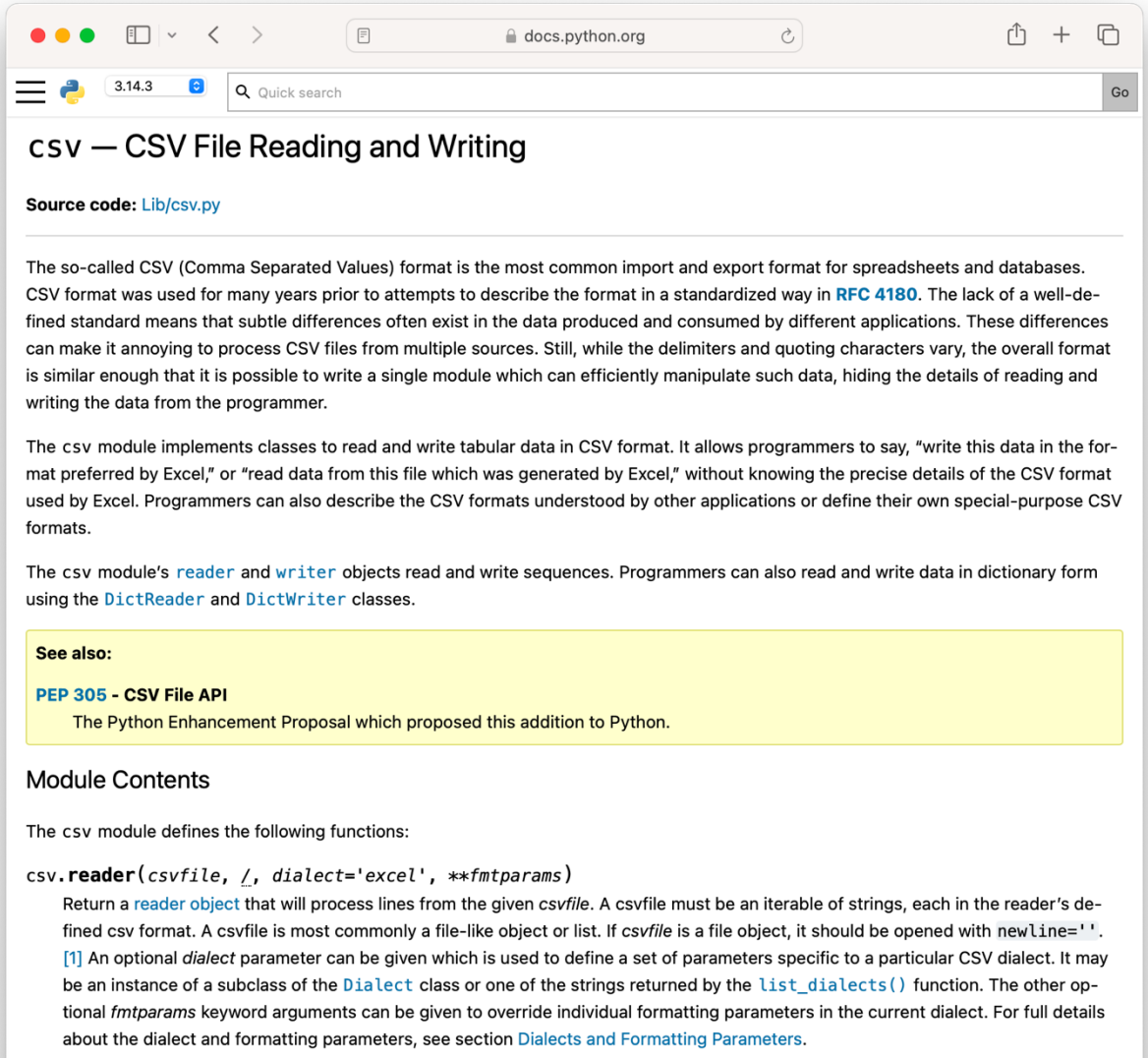
Example 4: Spreadsheets (.csv)

- A CSV file stores each row of the spreadsheet as one line in the file
- values in different columns are separated by commas





CSV Reader and CSV Writer



csv — CSV File Reading and Writing

Source code: Lib/csv.py

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in **RFC 4180**. The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The `csv` module implements classes to read and write tabular data in CSV format. It allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The `csv` module's `reader` and `writer` objects read and write sequences. Programmers can also read and write data in dictionary form using the `DictReader` and `DictWriter` classes.

See also:

PEP 305 - CSV File API
The Python Enhancement Proposal which proposed this addition to Python.

Module Contents

The `csv` module defines the following functions:

`csv.reader(csvfile, /, dialect='excel', **fmtparams)`
Return a reader object that will process lines from the given *csvfile*. A csvfile must be an iterable of strings, each in the reader's defined csv format. A csvfile is most commonly a file-like object or list. If *csvfile* is a file object, it should be opened with `newline=''`. [1] An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.