6.2. Reading and saving data

6.2.1. The role of files

In statistical software like SPSS or Stata, or in all typical office applications, you *open* a file, do some work on it, and then *save* the changes to the same file once you are done. You basically “work on that file”.

That’s not how your typical workflow in R or Python looks like. Here, you work on one or multiple data frames (or some other data structures). That means that you might start by *reading* the contents of some file into a data frame, but once that is done, there is no link between the dataframe and that file any more. Once your work is done, you can save your dataframe to a file, of course, but it is a good practice not to overwrite your input file, so that you can always go back to where you started. A typical workflow would rather look like this:

1. Read raw data from file "myrawdata.csv" into data frame "df"
2. Do some operations and analyses on df
3. Save df to file "myfinaldata.csv"

Note that the last step is not even necessary, but may be handy if running the script takes very long, or if you want to re-distribute the resulting file.

The format in which we read files into a data frame and the format to which we save our final data frame also by no means needs to be identical. We can, for example, read data created by someone else in Stata’s proprietary .dta format into a dataframe and later save it to a .csv table.

While we sometimes do not have the choice in which format we get our input data, we have a range of options regarding our output data. We usually prefer formats that are *open* and *interoperable* for this, which ensures that they can be used by as many people as possible, also in the future, and that they are not tied to any specific (proprietary) product.

The most common file formats that are relevant to us are listed in Table 6.1. txt

Table 6.1: Basics of data frame handling

	Used for?	open	interoperable?
txt	plain text	yes	yes
csv	tabular data	yes	yes
json	nested data, key-value pairs	yes	yes
pickle	Python objects	yes	no
RDS/RDA	R objects	yes	no

files are particularly useful for long texts (think of one file containing one newspaper article or even a whole book), but they are bad for storing associated meta data. csv files are the default choice for tabular data, and json files allow us to store nested data in a dictionary-like format (see above).

For the sake of completeness, we also listed the native Python and R formats pickle, RDS, and RDA. Because their lack of interoperability, they are not very suitable for long-term storage or for sharing data, but they can have a place in a workflow as an intermediate step to solve the issue that none of the other formats are able of storing all properties of a dataframe (e.g., the csv file cannot store whether a given column in an R dataframe is to be understood as containing strings such as 'man', 'woman', 'non-binary' or a factor with the three levels man, woman, non-binary). If it is of importance to store an object (such as a dataframe) exactly as-it-is, we can use these formats.

6.2.2. Encodings and dialects

txt files, csv files, and json files are all files that are based on text. Unlike binary file formats, you can read them in any text editor. Try it yourself to understand what is going on under the hood.

Download the example files from XXXX and open them in a text editor of your choice (for example, Notepad++, Atom, emacs, ... [offer some choice here]). As you will see (Figure 6.1), a csv file internally just looks like a bunch of text in which each line represents a row and in which the columns are separated by a comma (hence the name comma seperated values (csv)).

Looking at the data in a text editor is a very good way to find out what happens if reading your files into a data frame does not work as expected - which can happen more frequently than you would expect.

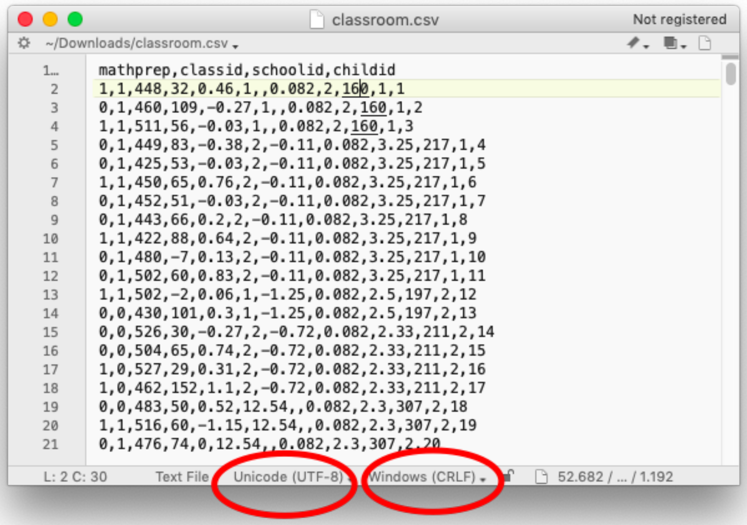
Mostly due to historical reasons, not every text based file (which, as we have seen, includes csv files) is internally stored in the same way.

For a long time, it was common to *encode* in such a way that one character mapped to one byte. That was easy from a programming perspective (after all, the n-th character of a text can directly be read from and written to the n-th byte of a file) and also storage-efficient. But given that a byte consists of 8 bits, that means that there are only 256 possible characters. All letters in the alphabet in uppercase, again in lower case, numbers, punctuation, some control characters - and you are out of characters. Due to this limitation, there were different encodings or codepages for different languages that told a program which value should be interpreted as which character.

We all know the phenomenon of garbled special characters, like German umlauts or Scandinavian characters like ø, å, or œ being displayed as something completely different. In these cases, a different encoding was used for saving them than for reading them.

In principle, this issue has been solved due to the advent of Unicode. Unicode allows to handle all characters from all scripts, including emoticons, Korean and Chinese characters, and so on. The most popular encoding for Unicode characters is called UTF-8, and it has been around for decades.

To avoid any data loss, it is advisable to make sure that your whole workflow uses UTF-8 files. By far most modern applications support UTF-8, even though some still by default use a different encoding (e.g., 'Windows-1252') to store data. As Figure 6.1 illustrates, you can use a text editor to find out what encoding your data has, and many editors also offer an option to change the encoding. However, you cannot recover what has been lost (e.g., if at one point you saved your data with an encoding that



```
mathprep,classid,schoolid,childid
1,1,448,32,0.46,1,,0.082,2,160,1,1
0,1,460,109,-0.27,1,,0.082,2,160,1,2
1,1,511,56,-0.03,1,,0.082,2,160,1,3
0,1,449,83,-0.38,2,-0.11,0.082,3.25,217,1,4
0,1,425,53,-0.03,2,-0.11,0.082,3.25,217,1,5
1,1,450,65,0.76,2,-0.11,0.082,3.25,217,1,6
0,1,452,51,-0.03,2,-0.11,0.082,3.25,217,1,7
0,1,443,66,0.2,2,-0.11,0.082,3.25,217,1,8
1,1,422,88,0.64,2,-0.11,0.082,3.25,217,1,9
0,1,480,-7,0.13,2,-0.11,0.082,3.25,217,1,10
0,1,502,60,0.83,2,-0.11,0.082,3.25,217,1,11
1,1,502,-2,0.06,1,-1.25,0.082,2.5,197,2,12
0,0,430,101,0.3,1,-1.25,0.082,2.5,197,2,13
0,0,526,30,-0.27,2,-0.72,0.082,2.33,211,2,14
0,0,504,65,0.74,2,-0.72,0.082,2.33,211,2,15
1,0,527,29,0.31,2,-0.72,0.082,2.33,211,2,16
1,0,462,152,1.1,2,-0.72,0.082,2.33,211,2,17
0,0,483,50,0.52,12.54,,0.082,2.3,307,2,18
1,1,516,60,-1.15,12.54,,0.082,2.3,307,2,19
0,1,476,74,0,12.54,,0.082,2.3,307,2,20
```

Figure 6.1: A csv file opened in a text editor, illustrating that the columns are separated by commas, and showing the encoding and the line endings.

only allows 256 different characters, it follows logically that you cannot recover that information).

As we will show in the practical code examples below, you can also force Python and R to use a specific encoding, which can come in handy if your data arrives in a legacy encoding.

Related to the different encodings a file can have, but less problematical, are different conventions of how a *line ending* is denoted. Windows-based programs have been using a Carriage Return followed by a Line Feed (denoted as `\r\n`), very old versions of MacOS used a Carriage Return only (`\r`), and newer versions of MacOS as well as Linux use a Line Feed only (`\n`). In our field, the Linux (or Unix) style line endings have become most dominant, and Python 3 even automatically converts Windows style line endings to Unix style line endings when reading a file - even on Windows itself.

A third difference is the use of so-called *byte-order markers* (BOM). In essence, a BOM is an additional byte added to the beginning of a text file to indicate that it is a utf-encoded file and to indicate in which order the bytes are to be read (the so-called endianness). While informative, this can cause trouble if your program does not expect that byte to be there. In that case, you might either want to remove it or explicitly specify the encoding as such (e.g., 'utf-8-bom' instead of 'utf-8' in the examples below).

In short, the most standard form in which you probably want to encode your data is in UTF-8 with Linux-style line endings without the use of a byte-order marker.

In the case of reading and writing csv files, we thus need to know the encoding, and potentially also the line ending conventions and the presence of a byte-order marker. However, there are also some additional variations that we need to consider. There is no definite definition of how a csv file needs to look like, but there are multiple dialects that are widely used. They mainly differ in to aspects: the delimiter that is chosen, and the quoting an/or escaping of values.

First, even though csv stands for comma separated values, one could use other characters instead of a comma to separate the columns. In fact, because many countries use a comma instead of a dot to as a decimal separator ($10.30 vs 10,30€), in many countries a semicolon (';')is used instead of a comma as column delimiter. To avoid the possible confusion, others use a tab character (`\t`) to seperate columns. Sometimes, these files are then called a tab-seperated file, and instead of .csv, they may have an ending such as .tsv, .tab, or even .txt. However, this does not change the way how you can read them - but what you need to know is whether your columns are seperated by `,`, `;`, or `\t`.

Second, there may be different ways of how to deal with strings as values in a csv file. For instance, it may be that a specific value contains the same character that is also used as a delimiter. These cases are usually resolved by either putting all strings into quotes, putting only strings that contain such ambiguities in quotes, or by prepending the ambigous character with a specific escape character. Most likely, all of this is just handled automatically under the hood, but in case of problems, you might want to look into this and check out the documentation of the packages you are using on how to specify which strategy is to be used.

Let's get practical and try out reading and writing files into a data frame (Example 6.2).

Python Code

```python
# If the csv file looks like pandas expects it, this simple
        command works:
df = pd.read_csv('http://cssbook.net/d/mediause.csv')

# But we can also explicitly specify the encoding, delimiters
        , and so on:
df = pd.read_csv('http://cssbook.net/d/mediause.csv',
        encoding = 'utf-8', delimiter = ',')
# There are many other options you could specifiy (for
        instance, whether your data have a header row. Just
        enter pd.read_csv? to get an overview.

# Save dataframe to a csv:
df.to_csv('mynewcsvfile.csv')
```

R Code

```r
library(tidyverse) #for csv files
library(haven) # for other file types

# If the csv file looks like pandas expects it, this simple
        command works:
df = read_csv('http://cssbook.net/d/mediause.csv')

# But we can also explicitly specify delimiters, etc.
df = read_delim('http://cssbook.net/d/mediause.csv', delim =
        ',')
# There are many other options you could specifiy. Just enter
         ?read_csv to get an overview.

# Save dataframe to a csv:
write_csv(df,'mynewcsvfile.csv')
```

Example 6.2: Reading files into a dataframe

Of course, we can read more than just csv files. In the Python example, you can use tabcompletion to get an overview of all file formats Python supports: Type `pd.read` and then press the TAB key to get a list of all supported files. For instance, you could to `pd.read_excel('test.xlsx')`, `df3 = pd.read_stata('test.dta')`, or `df4 = pd.read_json('test.json')` Similarily, for R, you can hit TAB after typing `haven::` to get an overview over functions such as `read_spss`.

6.2.3. File handling beyond data frames

Dataframes are a very useful data structure for organizing and analyzing data, and will come back in many examples in this book. However, not all things that we might want to read from a file needs to go into a dataframe. Imagine if we have a list of words that we later want to remove from some texts (so-called stopwords, see CHAPTERXXXXXXXXXXX). We could make a list (or vector) of such words directly in our code. But if we have more than a couple of such words, it is easier and more readable to keep them in an external file. We could create a file `stopwords.txt` in a text editor with one of such words per line:

```
and
or
a
an
```

If you are too lazy to create this list yourself, you could also download one from http://cssbook.net/d/stopwords.txt and save it in the same directory as your Python or R script.

Then, you can read this file into a vector or list (see Example 6.3).

Python Code

```python
# bad: define stopword list in the code itself
stopwords_in_code = ['and', 'or', 'a', 'an', 'the']

# good: read stopword list from external text file
stopwords_from_file = [f.strip() for f in open('../datasets/
        stopwords.txt').readlines()]
```

R Code

```r
# bad: define stopword list in the code itself
stopwords_in_code = c('and', 'or', 'a', 'an', 'the')

# good: read stopword list from external text file
stopwords_from_file = scan('stopwords.txt', what='string')
```

Example 6.3: Reading files without dataframes

Example 6.4 provides you with some more elaborate code examples that allows us to dig a bit deeper into the general way of handling files.

In the Python example, we can open a file and assign a handle to it that allows us to refer to it (the name of the handle is arbitrary, let's just call it `f` here. Then, we can use a for loop to iterate over all lines in the file and add it to a list

The `mode = 'r'` specifies that we want to read from the file. `mode = 'w'` would open the file for writing, create it if necessary, and immediately deletes all content that may have been in there if the file already existed (!). Note that the `.strip( )` is necessary to remove the line ending itself, and also possible whitespace at the beginning or end of a line. If we wanted to save our stopwords, we could do this in a similar way: We first open the file (this time, for writing), and then use the file handle's methods to write to it. We are not limited to plain text files, here. For instance, we can use the same approach to read json files into a python dict or to store a python

Python Code

```python
stopwords = []
with open('stopwords.txt', mode = 'r', encoding = 'utf-8') as f:
    for line in f:
        stopwords.append(line.strip())

# Let's modify the stopword list and save it to a new file:
stopwords.extend(["somenewstopword", "andanotherone"])
with open('newstopwords.txt', mode = 'w', encoding = 'utf-8') as f:
    f.writelines(stopwords)

# Instead of .writelines(which writes multiple lines at once), we can also .write() a single string to a file.

# We can use the same to read json files into a python dict or to store a python dict into a json file:

import json
somedict = {'a':'somevalue',
            'b': {'level2a':'anothervalue', 'level2b':{1:'x', 2: 'y'}},
            'c':[1,2,3,4]}

with open('test.json', mode = 'w', encoding = 'utf-8') as f:
    json.dump(somedict, f)

with open('test.json', mode = 'r', encoding = 'utf-8') as f:
    mydict = json.load(f)
```

R Code

```r
stopwords = scan('stopwords.txt', what='string')

# Let's modify the stopword list and save it to a new file:
stopwords = c(stopwords_from_file, "somenewstopword", "andanotherone")
fileConn<-file("newstopwords.txt")
writeLines(stopwords, fileConn)
close(fileConn)
```

Example 6.4: More examples for reading from and writing to files. Because dictionaries do not exist in R and handling non-rectangular data is more uncommon in R, we do not show an example for direct interaction with JSON files without involvement of a dataframe in the R example.

dict into a json file.

We could also combine this with a for loop that goes over all files in a dictionary. Imagine we have a folder full of positive movie reviews, and another one full of negative movie reviews that we want to use to train a machine learning classifier (see Section 12.2). Let's further assume that all these reviews are saved as `.txt` files. We can iterate over all of them, as shown in Example 12.2. If you want to read text files into a dataframe in R, the *readtext* package may be interesting for you.

6.3. Gathering data from online sources

Many data that are interesting to those analyzing communication are nowadays gathered online. In Chapter ADDREFERENCELATER, you will learn how to use APIs to retrieve data from web services, and how to write your own web scraper to auto-