Lesson 1, Week 4: text file I/O

AIM

— To learn methods for opening, reading from and writing to text files

After this lesson, you will be able to

* Discuss why file I/O is fundamentally risky for a computer
* Describe the different kinds of access to files: read-only, write-only or read/write access, discarding or retaining file data
* Use `open` to create a file handle, specifying what access is required
* To use use `read` and `readlines` to read from a file with an open file handle
* To use `write` to write to a file with an open file handle
* To use `close` to close a file handle

File I/O in general

So far, you've only dealt with programs that generate their own data. But of course most of the time, working with data means importing data from files and exporting data to files—that is, file I/O.

You should think of a file arriving in your Julia program's workspace as a stranger arriving at your door: it could the package that you've been waiting for, it could be trouble. In the case of the stranger, you may spend a while talking to them before you open the door. For better or for worse, computers tend not to do that sort of thing[1]—instead, the designers of most programming languages, Julia included, put the responsibility for avoiding trouble on you, the programmer.

In practice, this means having a pretty good idea that the files you are about to use are not too large, that the files have no malicious content, and that you know the format of the file. On this course, we

[1] Except when establishing internet connections and the like.

only one format only: flat text files, which consist solely of characters[2], but many other formats are used—in particular, specialised formats limited to specific purposes and domains.

Files to be used for I/O arrive in your computer in many ways: Internet downloads, memory devices such as hard disks and USB sticks, measurement devices, cameras and more. However, once they're there the same thing applies to all of them: you open a connection to the file, your read and or write to the file, and you close the connection.

- The important point in opening the connection to the file is that you use the correct path from your working directory to the directory where the file is. The best is simply to put in the same working directory as your code.
- In Julia, there are many functions that read from and read to a file, once a connection to the file exists. We discuss a few of them below.
- Many of the problems with file I/O come from file connections that remain open unduly long. None of the ways to make sure this never happens are both easy and foolproof. On this course, we recommend that you open a file, read or write to it, and close it. This can be cumbersome and for some purposes completely unsuitable, but is close to foolproof.

Opening and closing files

Long experience in computing have led to the following situation: you as the programmer decide, at the time of opening the connection to a file, whether it is for reading only, or writing only, or both. Additionally, if it is for writing, you decide whether to overwrite the file or to add to it (inserting data in the middle of a file is not possible in Julia).

You open a connection as follows: `<fh> = open(<fname>, <read/write mode>)`.

Here `<fh>` is the name of the connection you make, `<fname>` is the name of the file in question, and `<read/write mode>` is a string that sets the type of access to the file.

The file handle `<fh>` is the name of the connection, and it has to follow the usual rules for naming a variable, and the read/write mode must be one of several options—let us use `?` to see them. Note that the connection is a variable, with a name and a value[3]

The `read` and `readlines` functions

We make available to you a text file containing the first two chapters of Pride and Prejudice by Jane Austen. The file is called `prideandprejudiceextract.txt`. It was created by simply cutting and pasting from the text file of the book at Project Gutenberg: `https://www.gutenberg.org/ebooks/1342`. There are many books available there, completely free—of course you should acknowledge both the website and the author if you use their material, as we have done here.

[2] It is true that some of them are escape sequences.

[3] Of type `IOStream`, if you must know.

We can read the whole text as a single string:

```
fh = open("prideandprejudiceextract.txt")
prideandprejudiceraw = read(fh, String)
close(fh)
```

The file handle `fh` is a name that refers to the *connection* to the file, not the file itself. Recall that a connection is a variable with a name and a value. It exists quite independently of the file.

Note also that there is no read/write code string in the call to `open` — this means that Julia uses the default code string, which is `"r"` and stands for "read-only". One cannot write to a file opened with a read-only file handle.

We can use the `split` function (more or less the opposite of `join`, see the `?` summary) to extract all the lines. This is such a common action that `readlines` is provided to get the same result with far less code. Lots of lines are empty, we use `filter!` to get rid of them. [DEMO]

Writing to a file

First, you need to open it with the correct read/write code string.

Julia uses the read/write code `"w"` to specify that you writing to a blank file. This is fine if you create a file with a new name, but if you happen to use the name of a file already there, the contents of that file will simply be ignored[4] and the file will contain only the data written after you connected with it.

In other words, when you specify `"w"`, you specify that existing data should be destroyed.

To add to an existing file, use the `"a"` read/write code string. [DEMO: the `?` guidance on]

Once you have an appropriate file handle, use the `write` function:

```
f = open("newfile.txt", "w")
write(f, "what a boring test string")
close(f)
```

DEMO: try several writes, try opening and closing several times, try with `"a"` read-/write code string; use `\n` to get new lines.

Review and summary

* File I/O has three stages: opening a connection to a file, reading and/or writing to the file, closing the connection
* Files that stay open when they should be closed lead to computer problems
* The programmer decides, at the time of opening the connection, whether the access is for reading, for writing, or for both
* If the access is for writing, the programmer must also decide whether to keep existing data in the file or not

[4] The file content doesn't immediately disappear, however. This is why recovery of accidentally erased files is sometimes possible.

* The syntax is `<file handle> open(<file name>, <read/write code>)`
* If you omit the read/write code string, the default `"r"` is used, which means "read-only"
* Read the file as a single string[5] with `read(<file handle>, String)`
* Read the file as an array of strings with `readlines(<file handle>, String)`
* To write to file from scratch (destroying any data it may have contained), open it with `"w"`
* To append to a file (retaining all the data it contains), open it with `"a"`
* Use the syntax `write(<file handle>, <string value>)` to write a plain text file
* Appending happens on the same line as the last line in a file, unless that line ends with a newline character

[5] This assumes it is a text file, of course.

Lesson 2, Week 4:
Obtaining and manipulating the words in a text file

AIM

— To read a text file, extract the words only, and manipulate them

After this lesson, you will be able to

* Depunctuate a text file line by line
* Turn a file of depunctuated text strings into a file of words
* Count various words and phrases in a file of words

Words only: removing the punctuation

Reading in the data

Reminder: the file `prideandprejudiceextract.txt` is used to provide data for this week's work. Let's read in the data with `open` and `readlines` :

```
fh = open("prideandprejudiceextract.txt")
pridelines = readlines(fh)
```

`close(fh)` and then use `filter!` to get rid of the empty lines:

```
filter!(x -> !isempty(x), pridelines)
```

Take small steps

A good habit: work on small pieces of data when you develop code[1]. So before depunctuating the whole file, let's pick a line with several marks, for instance line 10:

[1] If possible!

`sampleline = pridelins[10]`
and let's see it character by character: `samplechars = [x for x in sampleline]`.

Removing the full stop

So an easy character to work on would be the full stop. We can again use `filter!`. It has to operate on an array, so we use `samplechars`. The test `x == '.'` is true when the (local) name `x` is bound to the value `'.'` (programmers usually use the shorthand "when x is a full stop"), but we want `filter!` to retain all the other values, so the anonymous function we need is `x -> !(x == '.')`. Putting it all together, we reach `filter!(x -> !(x == '.'), samplechars)`. [DEMO]

But that the result is an array, so `join` is needed to get the string back:
`join(filter!(x -> !(x == '.'), samplechars))`.

Scaling up: removing the other marks

To filter out the commas as well, we note that the logical expression we need is `!(x == '.'  || x == ',')` [DEMO]

That leaves the quote characters. Note that they are not double quote characters—but then how to find out what they are? You can simply index for it: `sampleline[1]`. [DEMO]

Unfortunately, unless we use `'\u201c'`, that doesn't help us to type it in. One thing to try is whether it might come up as a "\ + letters + Tab" character, and it turns out that simply starting with \quo does the trick. [DEMO]

For the full text, we also need to remove the question marks, colons, semi-colons, exclamation marks—but there's more, see below

Scaling up to the full text

We now face two problems: filtering the other marks, and iterating over all the lines. Let's start with iterating over all the lines. For this, we can use a loop starting with `for line in pridelines` but then we have a problem: `line` is a local variable inside the loop. But we want the results of the depunctuation available globally!

No problem: we create a global array, start it empty, and use `push!` inside the loop to extend it:

```
depunclines = []
for line in pridelines
 newline = join(filter!(x -> !(x == '.'  || x == ',' || x == '\' || x == '"'), [x for x in line]))
push!(depunclines, newline)
```

```
end
```

The logical test in `filter!` is getting longer and longer. One could repeat it several time, for different punctuation marks each time. But an alternative is to just to break it up over several lines:

```
depunclines = [];
for line in pridelines
  newline = join(filter!(x -> !(x == '.'  || x == ','
        || x == '\' || x == '"' || x == '?'  || x == '!'
        || x == '_' || x == ':'  || x == ';' || x = '\'' ),
        [x for x in line]));
push!(depunclines, newline)
end
```

Escaping the single quote character this way is not the best thing to do. Another option, only slightly better, is to replace it with a space, which we leave to the exercises.

Turning the array of strings into an array of words

So now we have the array of `depunclines`. Each line is a string, eg. `depunclines[10]`, which we can split into words with `split(depunclines[10], ' ')`.

Actually, we can turn use `join` to join up all the strings in `depunclines` into one string, and then split it in the same way.

Naively, we could try `split(join(depunclines), ' ')`. However, although in the main it looks good it seems to join some words that should stay separate. Debugging time! The reason is that there is no space between the end of one line and the start of the next. We can put space in with the `join` function, leading to

```
pridewords = split(join(depunclines, ' '), ' ')
```

Counting some of the interesting words and phrases

A simple way to get all occurrences of Mr Bingley's name is by using the `filter` function: `filter(x -> x == "and", pridewords)`.

[DEMO: do this for other name, common words like "I", "the", "and", etc.]

Even more interesting is to search for phrases. One might for instance ask for the phrases "Mr Bennet" and "Mrs Bennet". The problem now is that a `pridewords` does not contain phrases, but only words. So we need a different approach.

One way is to check every pair of words. We loop over the file, from the first to the second-last word, and compare the current pair to the target. However, we cannot simply use `for word in pridewords`

because that only gives us one word at a time. The solution is to use a range, and to index the pair we need as a sub-array. The code below looks for the phrase “Mrs Bennet”

```
count = 0; numwords = length(pridewords)
for wordnumber in 1:numwords-1
  if join(pridewords[wordnumber:wordnumber+1], ' ') == "Mrs Bennet"
    global count = count + 1
  end
end
count
```

Review and summary

* Use `filter!` with test functions similar to `x -> !(x == '.')` to remove punctuation marks
* Filter out the punctuation line by line
* Form an array of strings containing words only into an array of words
* Count given words and given phrases

Lesson 3, Week 4: More playing with text

AIM

— Playful creativity, using what we've learnt so far

After this lesson, you will be able to

* Play by replacing specific words
* Play by replacing specific letters
* Use the function `rand` and the array `[true, false]` to simulate the toss of a fair coin
* Play by replacing randomly selected words

Reminder: we use the array `pridewords`, which contains the words but none of the punctuation from Pride and Prejudice extract in the file `prideandprejudiceextract.txt` which we provide along with the other course material. Please use the code from the previous lesson to make the array available.

Deliberate word replacements

A pleasant game is to replace words with other strings. There are actually many ways to do so, but here we will use the same method as for counting phrases: a `for` loop over all the words, but not directly as in `for word in pridewords`. Rather, we will loop over a range and use indexing.

```
for i in 1:length(pridewords)
    if pridewords[i] == "Bennet"
  newwords1[i] = "Klunderclap"
  end
end
println(join(newwords1, ' '))
```

[DEMO: illustrate this with full file, modify to do only the first 200 words, replace also "and", "is" and "the"]

[DEMO: instead of replacing word matches, put in regular replacements by using a range like `1:5:length(pridwords)`]

Deleting some letters only

For this one has to go back to the original string of the whole file, or else concatenate all of words into one string file (retaining the spaces, otherwise it is an unfunny mess). One can play in similar ways to the word replacement, it is particularly interesting to filter out all the occurences of the letter "e". This is left as an exercise.

Using random numbers to guide swops and/or deletions and/or replacements of words

Monte Carlo simulation is a general term in science for creating patterns that are the result of random processes. It is a vast topic, but we can play a little with using it here.

The function `rand` can be used to pick an element randomly from an array. Hence `rand([true, false])` is a simulation of a fair coin. Let's toss a coin for every word in the first few hundred:

```
numwords = 200; newwords = pridewords[1:numwords]
for i in 1:numwords
  if rand([true, false])
    newwords1[i] = "Klunderclap"
  end
end
```

[DEMO: using `rand([true, false, false, false, false])` one can replace every fifth word; replacing words with blank space or ... might also be suggestive]

One can really play wonderfully with randomness! This is just a start.

Review and summary

The only new thing we learnt in this lesson was the functon `rand` and that `rand([true, false])` simulates a toss of a fair coin.

Otherwise it was all play!

Lesson 4, Week 4: Looking back, looking forward

First of all, congratulations!

You've made it this far, well done! Programming is hard work[1] and if you've got here and done most of the exercises and assignments, you deserve plenty of plaudits!

The looking back: what we covered, how and why. And what we didn't

We covered quite a lot of the basics of Julia: values[2], types[3], variables, functions[4], local and global scope, logical tests, structures of branching code[5].

We applied it all to text we created ourselves and also to one sample text file[6], even learning a bit of Monte Carlo simulation doing so. It's been quite a journey and one could do very substantial projects without using anything beyond the matter of this course.

But in truth, there is much, much more to Julia than that. It is, after all, a very recently created language, building on all that has gone before and aiming providing a very wide audience with ease of writing code, efficiency n extending code, and the possibility of blindingly fast code.

Here's an indication, roughly in order of how accessible the topics will be to you right after finishing this course[7]: improving your coding style; accessing and using Julia packages; making plots and other graphics with Julia; learning how to collaborate with others[8]; user-defined types; Julia's system of modules; high-speed performance; meta-programming . . . there is still much more than that.

But really, don't be intimidated, you need not take all of that or indeed any of that on. You've already come a significant way, and maybe you have gained all you ever want from learning to program. It's fine to pat yourself on the back and do something else[9].

[1] Hard for almost everyone—that's why successful programmers earn good money.
[2] Characters, strings, numbers, arrays, and `true` and `false`
[3] `String, Char, Int64, Float64, Bool` and a few `Array` types also
[4] Including operators. User-defined in three ways: the `function` keyword, inline functions, and anonymous functions.
[5] with `if`, `while` and `for`
[6] Go on, you can do similar on a text file you find or make yourself!
[7] Assuming that this is your only experience of programming so far
[8] The Julia community is very pleasant, polite and helpful; GitHub is a particularly good platform for collaboration
[9] The branching of paths is not limited to code!

If you want to continue with Julia ...

There are many online tutorials and courses. You will gain a lot from read the Julia documentation at `https://docs.julialang.org/`. A very good way to learn more and enjoy yourself at the same time is to do a project that interests you. Finally, you can easily get help online via `http://discourse.julialang.org/` and other resources, for example Stackoverflow at `http://stackoverflow.com/`.

Of course, do search through the packages available for Julia at `https://pkg.julialang.org/`—you'll find plenty to explore and enjoy, and many have tutorials that you will find accessible. You might even find one you like so much that you want to help develop it[10].

I would strongly recommend that you learn to use at least one of IJulia and Juno. They are alternatives to the way we did things on this course: the REPL + `.jl` files.

IJulia is a very clean interface, a so-called notebook, and it allows you to combine detailed discussions with elegant bits of code. Juan Klopper and I use it for the course "Scientific Programming in Julia" which you will find on Coursera, reach them via `http://coursera.org/`.

Juno is a so-called IDE, an integrated development interface, all on one screen you get a REPL, an editor, graphics if you need it, your file system and even other filesystems (particularly on `github.com`) and more. It takes a while to learn how to use it well, though when you do it can really push up your productivity.

What about options other than Julia?

Much as we love Julia, we recognise there many factors that influence choice of language for continuing your adventure with programming. Your workplace may demand a different language, you may have a specialised purpose that is better served by another language, you may feel that one of the better-known languages would be better for your job prospects. And of course, you may simply be curious to see what other languages are like and see whether you like any of them better.

Goodbye and good luck!

[10] It's not that hard, start the online docs. Package developers appreciate help with documentation especially, and as we read we all spot ways that the documentation could improve. It isn't too hard to learn how to make pull requests that eventually are incorporated in the online material.