Lab 2: Julia Quickstart

Functions, Logic, and Packages

CEVE 421/521

Fri., Jan. 19

1 Overview

In this lab we will learn how to work with tabular data in Julia. Specifically, you will get some experience using:

1. `DataFrames.jl` to store tabular data as a DataFrame
2. `CSV.jl` to read CSV files and convert them to DataFrames
3. `DataFramesMeta.jl` to manipulate DataFrames
4. `Plots.jl` and `StatsPlots.jl` to create visualizations

For those of you who took CEVE 543, you'll find most of this familiar! If you find this challenging (e.g., if you're new to programming) please look at the Resources page which has links to tutorials that you can use to get up to speed. Some of you will work extra hard this week, and some of you will have an easier week.

1.1 This repository

In this repository you will find two files.

1. `index.qmd` is the source code for the rendered document on the course website that you may be looking at now. That's this file!
2. `template.qmd` is the template for your lab submission. You should edit it directly.

1.2 Submission

As with Lab 01, you should:

1. push your final code to GitHub (I'll be able to see it via GitHub classroom)
2. submit your rendered PDF or DOCX file to Canvas

2 Setup

Here are some instructions for getting this lab working.

2.1 Clone the repository

First, you'll need to clone this repository to your computer. As with Lab 01, I recommend to use GitHub Desktop or the built-in Git support in VS Code. Remember to use the link from Canvas

(`classroom.github.com/...`).

Next, open the repository in VS Code (you can do this directly from GitHub desktop if you'd like). All instructions from here assume you're in the *root directory* of the repository.

2.2 Install required packages

As we saw in Lab 01, Julia is a modular language with code in packages. Compared to a language like Python, the packages in Julia typically have a narrower scope (for example, instead of a single Pandas package that does everything, there are separate packages for reading CSV files, defining dataframes, using clear syntax for data manipulation, etc.). When we're working with a new lab, we'll need to first install the packages we need.

1. Open the command palette and select `Julia: Start REPL`
2. In the Julia REPL, type `]` to enter package manager mode
3. Type `activate .` to activate the project environment
4. Type `instantiate` to install the packages listed in the `Project.toml` file. This may take a few minutes.[1]

2.3 Rendering and previewing

As we saw in Lab 01, Quarto lets you convert a text file into an output. We've seen PDF, DOCX, and HTML (website) outputs, but there are other options as well.

There are many valid workflows, but my favorite is often to `preview` the document in VS Code while I'm working on it, and then `render` it when I'm done. To preview the document, run the following in your terminal (use the command palette in VS and then select "Terminal: Open a new terminal" or learn the shortcut keys):

```
quarto preview template.qmd
```

You should see that a web browser opens up with the rendered document. As you make changes, they should appear in the browser automatically. You'll see `localhost:XXXX` in your URL bar.

To render this document to PDF or DOCX, you have a few options. My favorite is to use the terminal again. For example, to convert to PDF:

```
quarto render template.qmd --to pdf
```

If you run into issues, try the following two tips

1. Make sure you're typing these commands into the *terminal* and not into the Julia REPL
2. In the Julia REPL, type `]` to enter package manager mode and then type `build IJulia` to rebuild your IJulia kernel (we won't go into details)

2.4 Getting help

If you're getting stuck, please:

[1] Julia precompiles packages when they are installed, and (to a lesser extent) when they are first used. The first time you use a package it may take a moment to load. This is normal, nothing to worry about, and rapidly improving.

1. Come up and ask me questions if we're in lab
2. Post on Canvas discussions
3. If I can't resolve your comment on Canvas, please email me to schedule a 1:1

2.5 Looking ahead

In the future, you'll repeat these steps *for every lab*:

1. `clone` the repository to your computer
2. `activate` the project environment
3. `instantiate` the packages
4. make your changes, saving and `commit`ing regularly as you go
5. `push` your changes to GitHub (you don't have to wait until the end for this – you can `push` multiple times)

3 Refresher: Quarto basics

Before diving in, let's quickly review some Quarto basics. As we saw in the last lab, Quarto is a program that lets you combine text, code, and output in a single document. Quarto files are just text files, typically with the file extension `.qmd`.

By default, all the text in a Quarto file is interpreted as Markdown, a simple markup language for formatting text. You've probably seen Markdown before. You can create headers with `##` (for a section), `###` (for a subsection), and so on. You can make text *italic* with `*italic*` and **bold** with `**bold**`. For more, you can learn more about it here.

When you're authoring your labs, you should take advantage of Markdown features!

3.1 Document metadata

If you open a Quarto file in your text editor (e.g., VS Code) or look at it on GitHub, you'll see that the file starts with some *metadata*. The metadata is a set of key-value pairs that tell Quarto how to render the document. In Lab 01, you edited the `author` field to include your name.

3.2 LaTeX Math

As in standard Pandoc markdown, you can use LaTeX math in Quarto. For example, `$\alpha$` yields α. You can also use `$$` to create a block equation:

```
$$
P(E) = \{ n \choose k \} p ^k (2-p) ^ {n-k}
$$
```

renders as

$$P(E) = \binom{n}{k} p^k (2-p)^{n-k}$$

For more, see the "Typesetting Math" section of the resources page.

3.3 Source code

Sometimes we want to provide example code in our documents. This is code that is not meant to be run, but is just there to illustrate a point. We do that by wrapping the code in ```. For example:

````
```
f(x) = 1.25 * sin(2  * x / 1.5 + 0.5) + 0.25
f(2.1)
```
````

yields

```
f(x) = 1.25 * sin(2  * x / 1.5 + 0.5) + 0.25
f(2.1)
```

You will typically want to specify the language of the code block, which will tell Quarto how to syntax highlight it. For example, see how the highlighting changes when we specify `julia`:

````
```julia
f(x) = 1.25 * sin(2  * x / 1.5 + 0.5) + 0.25
f(2.1)
```
````

```julia
f(x) = 1.25 * sin(2  * x / 1.5 + 0.5) + 0.25
f(2.1)
```

3.4 Code blocks

Often, we don't just want to show code, but we want to run it and show the output.

````
```{julia}
f(x) = 1.25 * sin(2  * x / 1.5 + 0.5) + 0.25
f(2.1)
```
````

which yields

```julia
f(x) = 1.25 * sin(2  * x / 1.5 + 0.5) + 0.25
f(2.1)
```

```
0.4099583491000567
```

You can run these blocks in Julia by clicking the "Run Cell" button, or by pressing the keyboard shortcut (to see it, open the command palette and search for "Run Cell"). For more on Julia, see here.

3.5 Citations

You can add citations in Quarto. The easiest way is to export a bibliography from Zotero, and then add it to your Quarto document. You can use the Zotero Better BibTeX plugin to export a `.bib` file.

See here for instructions on using references with Quarto or see the website code for an example. I'll provide a template for your final project.

4 Julia Quickstart

4.1 Loading packages

In Julia we say `using` to import a package. By convention we'll put these at the top of our script or notebook in alphabetical order. When you run this cell, you'll see a bunch of activity in your REPL as Julia goes through the following steps:

1. Download a file from the internet that specifies which packages depend on which other packages
2. Solve an optimization problem to identify which versions of which packages (including dependencies, and their dependencies, and so on) are compatible with each other
3. Download the packages and compile them (this may take a few minutes)

```
using CSV
using DataFrames
using DataFramesMeta
using Dates
using Plots
using StatsBase: mean
using StatsPlots
using Unitful
```

We'll also make sure that we have the correct backend set for plotting (don't need to worry about this)

```
Plots.gr()
```

```
Plots.GRBackend()
```

4.2 Read in data

We will use the `CSV.jl` package to read in our data.

> **Tip**
>
> Hover over the numbers on the right of this code for explanations.

```
fname = "data/tidesandcurrents-8638610-1928-NAVD-GMT-metric.csv"    ①
df = CSV.read(fname, DataFrame)                                     ②
first(df, 5)                                                        ③
```

① We define a variable called `fname` that stores the path to our data file. The `data` folder is in the same directory as this notebook.

② We use the `CSV.read` function to read in the data. The first argument is the filename, and the second argument tells Julia to convert the data to a `DataFrame`. We store it as a variable called `df`.

③ We use the `first` function to show the first 5 rows of the DataFrame.

	Date Time	Water Level	Sigma	I	L
	String31	Float64	Float64	Int64	Int64
1	1928-01-01 00:00	-0.547	0.0	0	0
2	1928-01-01 01:00	-0.699	0.0	0	0
3	1928-01-01 02:00	-0.73	0.0	0	0
4	1928-01-01 03:00	-0.669	0.0	0	0
5	1928-01-01 04:00	-0.516	0.0	0	0

This data comes from the NOAA Tides and Currents website, specifically for a station at Sewells Point, VA for the year 1928. NAVD refers to the North American Vertical Datum, which is a reference point for measuring sea level, and GMT refers to Greenwich Mean Time, which is the time zone used in the data (rather than local time).

We can see that our DataFrame has five columns, the first of which is "Date Time". However, the "Date Time" column is being parsed as a `string`. We want it to be a `DateTime` object from the `Dates` package. To do that, we need to tell Julia how the dates are formatted. We could then manually convert, but `CSV.read` has a kewyord argument that we can use

```
date_format = "yyyy-mm-dd HH:MM"                                    ①
df = CSV.read(fname, DataFrame; dateformat=date_format)             ②
first(df, 3)
```

① This is a string that tells Julia how the dates are formatted. For example, `1928-01-01 00:00`. See the documentation for more information.

② `dateformat` is a *keyword argument* while `date_format` is a variable whose value is `"yyyy-mm-dd HH:MM"`. We could equivalently write `dateformat="yyyy-mm-dd HH:MM"`.

	Date Time	Water Level	Sigma	I	L
	DateTime	Float64	Float64	Int64	Int64
1	1928-01-01T00:00:00	-0.547	0.0	0	0
2	1928-01-01T01:00:00	-0.699	0.0	0	0
3	1928-01-01T02:00:00	-0.73	0.0	0	0

The next column is "Water Level", which is the height of the water above the reference point (NAVD) in meters. We can see that this is being parsed as a float, which is what we want . However, you have to *know* that the data is in meters rather than inches or feet or something else. To explicitly add information about the units, we can use the `Unitful` package.

```
df[!, " Water Level"] .*= 1u"m"                                    ①
first(df, 3)
```

① We select the column with water levels using its name. The `!` means "all rows". Thus, `df[!, " Water Level"]` is a vector of all the water levels stored. `*=` means to multiply in place. For example, if `x=2` then `x *= 2` is equivalent to `x = x * 2`. `.*=` is a vector syntax, meaning do the multiplication to each element of the vector individually. `1u"m"` is a `Unitful` object that represents 1 meter. We multiply the water levels by this to convert them to meters.

	Date Time	Water Level	Sigma	I	L
	DateTime	Quantity…	Float64	Int64	Int64
1	1928-01-01T00:00:00	-0.547 m	0.0	0	0
2	1928-01-01T01:00:00	-0.699 m	0.0	0	0
3	1928-01-01T02:00:00	-0.73 m	0.0	0	0

4.3 Subsetting and renaming

We want to only keep the first two (for more on the other three, see here). We can also rename the columns to make them easier to work with (spaces in variable names are annoying). To do this, we use the `@rename` function:

```
df = @rename(df, :datetime = $"Date Time", :lsl = $" Water Level");    ①
```

① The `$` is needed here because the right hand side is a string, not a `Symbol`.

Then, we can use the `@select` function to do select the columns we want. Notice how the first argument to `select` is the `DataFrame` and the subsequent arguments are column names. Notice also that our column names were strings (`"Date Time"`), but we can also use symbols (`:datetime`).

```
df = @select(df, :datetime, :lsl)
first(df, 3)
```

	datetime	lsl
	DateTime	Quantity…
1	1928-01-01T00:00:00	-0.547 m
2	1928-01-01T01:00:00	-0.699 m
3	1928-01-01T02:00:00	-0.73 m

For more on what `DataFramesMeta` can do, see this Tweet.

4.4 Time series plot

Now we're ready to make some plots of our data. Let's start with a simple time series plot of the water levels. Our data is collected hourly, so we have a lot of data points! Still, we can plot them all.

```
plot(
    df.datetime,
    df.lsl;
```

```
    title="Hourly Water levels at Sewells Point, VA",
    ylabel="Water level",
    label=false,
)
```

Focusing on the entire time series means we can't dig into the details. Let's zoom in on a single month (October 1928) using the `@subset` function.

```
t_start = Dates.DateTime(1928, 10, 1, 0)                                    ①
t_end = Dates.DateTime(1928, 10, 31, 23)
df_month = @subset(df, t_start .<= :datetime .<= t_end)                     ②
first(df_month, 3)
```

① This creates a `DateTime` object for the start of October 1928 at 0 hours. Defining it clearly here aids readability.

② This selects all the rows where the `:datetime` column is between `t_start` and `t_end`. The `.` syntax is called *dot broadcasting* and is a way to apply a function to each element of a vector.

	datetime	lsl
	DateTime	Quantity…
1	1928-10-01T00:00:00	0.215 m
2	1928-10-01T01:00:00	0.429 m
3	1928-10-01T02:00:00	0.581 m

Now we can plot it as above:

```
plot(
    df_month.datetime,
    df_month.lsl;
    title="Water levels at Sewells Point, VA",
    ylabel="Water level",                                                   ①
    label=false,                                                            ②
)
```

4.5 Climatology

An essential idea in working with tabular data (and other data formats) is "split-apply-combine". Essentially: split the data into groups, apply some function to each group, and then combine the results.

We can use this workflow to answer an interesting question: **what is the average water level for each month?**[2] Of course, we're only looking at one year of data here – we should ideally look at a long record!

```
df[!, :month] = Dates.month.(df.datetime)                                   ①
dropmissing!(df, :lsl)                                                      ②
```

[2]To do a better job, we should separate out the long-term trend from the seasonal cycle. This is called de-trending and is a common technique in climate science. We can worry more about this later.