Lab 2: Julia Quickstart

Functions, Logic, and Packages

CEVE 421/521

Fri., Jan. 19

Table of contents

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

1. [DataFrames.jl](https://dataframes.juliadata.org/stable/) to store tabular data as a DataFrame
2. [CSV.jl](https://csv.juliadata.org/stable/) to read CSV files and convert them to DataFrames
3. [DataFramesMeta.jl](https://juliadata.github.io/DataFramesMeta.jl/stable/) to manipulate DataFrames
4. [Plots.jl](https://docs.juliaplots.org/stable/tutorial/) and [StatsPlots.jl](https://github.com/JuliaPlots/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](../../resources.qmd) for some tutorials.

|  |
| --- |
| Instructions |
| Much of this lab is example code with some narration to give you a sense of what’s going on. However, when you see a box like this it means you need to do something! |

As with Lab 01, you should push your final code to GitHub and submit your rendered PDF or DOCX file to Canvas.

## 1 Setup

Here are some instructions for getting this lab working.

### 1.1 Clone the repository

First, you’ll need to clone this repository to your computer. As with [Lab 01](../lab-01/index.qmd), 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.

### 1.2 Install required packages

As we saw in [Lab 01](../lab-01/index.qmd), 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](https://code.visualstudio.com/docs/getstarted/userinterface#_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]](#footnote-32)

### 1.3 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 commiting 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)

## 2 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](https://quarto.org/docs/authoring/markdown-basics.html).

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

### 2.1 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.

### 2.2 LaTeX Math

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

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

renders as

For more, see the “Typesetting Math” section of the [resources page](../../resources.qmd).

### 2.3 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(2π \* x / 1.5 + 0.5) + 0.25  
f(2.1)  
```

yields

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

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

### 2.4 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(2π \* x / 1.5 + 0.5) + 0.25  
f(2.1)  
```

which yields

f(x) = 1.25 \* sin(2π \* 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](https://quarto.org/docs/computations/julia.html).

### 2.5 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](https://retorque.re/zotero-better-bibtex/) plugin to export a .bib file.

See [here](https://quarto.org/docs/authoring/footnotes-and-citations.html) for instructions on using references with Quarto or see the website code for an example. I’ll provide a template for your final project.

## 3 Julia Quickstart

### 3.1 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

### 3.2 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"  
df = CSV.read(fname, DataFrame)  
first(df, 5)

Line 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.

Line 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.

Line 3

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

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"  
df = CSV.read(fname, DataFrame; dateformat=date\_format)  
first(df, 3)

Line 1

This is a string that tells Julia how the dates are formatted. For example, 1928-01-01 00:00. See the [documentation](https://docs.julialang.org/en/v1/stdlib/Dates/#Base.Dates.DateFormat) for more information.

Line 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".

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"  
first(df, 3)

Line 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.

### 3.3 Subsetting and renaming

We want to only keep the first two (for more on the other three, see [here](https://api.tidesandcurrents.noaa.gov/api/prod/responseHelp.html)). 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");

Line 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)

For more on what DataFramesMeta can do, see [this Tweet](https://twitter.com/evalparse/status/1413142675688067089).

### 3.4 Writing a function

We have just done a lot of work to read in our data. However, this just gives us data for the year 1928. In fact, we have a CSV file for each year 1928-2021. To make sure we can read them each in excatly the same way, we want to write a function. This function will take in the year as an argument, and return a DataFrame with the data for that year.

Writing functions is an important part of programming effectively. Let’s write a function that takes in a year and returns a DataFrame with the data for that year, following the steps we’ve explored above.

Before we do that, let’s define a function that will return the filename for a given year. It’s often valuable to stack several functions together.

get\_fname(year::Int) = "data/tidesandcurrents-8638610-$(year)-NAVD-GMT-metric.csv"

function read\_tides(year::Int)  
   
 # define the CSV file corresponding to our year of choice  
 fname = get\_fname(year)  
  
 # a constant, don't change this  
 date\_format = "yyyy-mm-dd HH:MM"  
   
 # <YOUR CODE GOES HERE>  
 # 1. read in the CSV file and save as a dataframe  
 # 2. convert the "Date Time" column to a DateTime object  
 # 3. convert the " Water Level" column to meters  
 # 4. rename the columns to "datetime" and "lsl"  
 # 5. select the "datetime" and "lsl" columns  
 # 6. return the dataframe  
end  
  
# print out the first 10 rows of the 1928 data  
first(read\_tides(1928), 10)

|  |
| --- |
| Instructions |
| Fill out this function. Your function should implement the six steps indicated in the instructions. When it’s done, convert it to a live code block by replacing ```julia``` with ```{julia}```. When you run this code, it should print out the first 10 rows of the 1928 data. Make sure they look right! |

### 3.5 Combining files

Now that we have the ability to read in the data corresponding to any year, we can read them all in and combine into a single DataFrame. First, let’s read in all the data.

|  |
| --- |
| Instructions |
| Update the code blocks below, then replace ```julia``` with ```{julia}```. |

years = 1928:2021 # all the years of data  
annual\_data = # 1. call the read\_tides function on each year  
typeof(annual\_data) # should be a vector of DataFrames

Next, we’ll use the vcat function to combine all the data into a single DataFrame.

df = vcat(annual\_data...) # don't change this  
first(df, 5)

last(df, 5) # check the last 5 years and ensure they are the last 3 hours of 2021-12-31

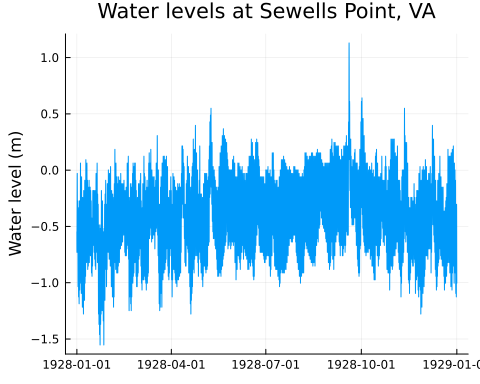
Finally, we’ll make sure we drop any missing data.

dropmissing!(df) # drop any missing data

### 3.6 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="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)  
t\_end = Dates.DateTime(1928, 10, 31, 23)  
df\_month = @subset(df, t\_start .<= :datetime .<= t\_end)  
first(df\_month, 3)

Line 1

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

Line 3

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.

Now we can plot it as above:

plot(  
 df\_month.datetime,  
 df\_month.lsl;  
 title="Water levels at Sewells Point, VA",  
 ylabel="Water level",  
 label=false,  
)



|  |
| --- |
| Instructions |
| Change the t\_start and t\_end dates so that you plot March 2020. What do you notice? |

### 3.7 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]](#footnote-78)

df[!, :month] = Dates.month.(df.datetime)  
dropmissing!(df, :lsl)  
df\_bymonth = groupby(df, :month)  
df\_climatology = combine(df\_bymonth, :lsl => mean => :lsl\_avg);

Line 1

This creates a new column called :month that is the month of each observation.

Line 2

This will discard any rows in df that have a missing value of :lsl. This is necessary because the mean function will return missing if any of the values are missing.

Line 3

This creates a GroupedDataFrame object that contains all the data grouped by month.

Line 4

This takes the grouped data and calculates the mean of the :lsl column for each month. The general syntax is combine(grouped\_df, :column => function).

We can now plot the climatology.

plot(  
 df\_climatology.month,  
 df\_climatology.lsl\_avg;  
 xticks=1:12,  
 xlabel="Month",  
 ylabel="Average Water level",  
 linewidth=3,  
 label=false,  
)

Line 2

We can use df.colname instead of df[!, :colname]. The latter is more robust but the former is easier to type.

Line 4

Setting xticks will set the x-axis ticks to the values in the vector. We can use this to make sure the x-axis ticks are labeled with the months.

Line 7

We can set the line width to make the plot easier to read.



|  |
| --- |
| Instructions |
| 1. Use the full dataset to plot the climatology using data from all years 2. Next, create a new Markdown header (## Groupby Day of Year) plot the average water level for each day of the year (Dates.dayofyear from 1 to 366). 3. What do you notice? |

Boxplots are ways to visualize the distribution of data. They show the median (the line in the middle of the box), the interquartile range (the box), and the range of the data (the whiskers). Outliers are shown as dots. We can use the boxplot function from the StatsPlots.jl package:

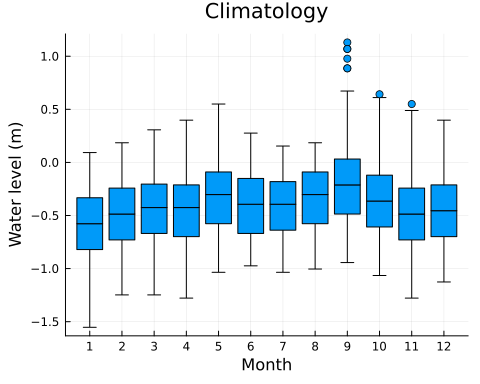
boxplot(  
 df[!, :month],  
 df[!, :lsl];  
 xticks=1:12,  
 xlabel="Month",  
 ylabel="Water level",  
 label=false,  
 title="Climatology",  
)

Line 2

We are back to df[!, :colname] syntax. Both work!

Line 8

We can set the title using the title keyword argument.



|  |
| --- |
| Instructions |
| Repeat this analysis grouping by year rather than month. What do you notice from the boxplots? |

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. [↑](#footnote-ref-32)
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. [↑](#footnote-ref-78)