Lab 02

Working with tabular data in Julia

2023-09-01

Table of contents

# 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](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

## 1.1 Exploratory modeling

Technically, what we are doing today is called *exploratory modeling* or *exploratory data analysis.* The latter is more common, but the former makes clear that all visualizations include some (usually implicit) conceptual model of the data. You will find more resources if you search for “exploratory data analysis”, however.

## 1.2 How to use this template

I have provided you with a lot of template code.

Any code block that starts with

```{julia}

is a “live” [code block](https://quarto.org/docs/computations/julia.html#code-blocks) and *will execute (run)*.

* These blocks will automatically run when you render the document
* You can click “Run Cell” to run the cell
* Make sure you understand what the code does and ask questions (in-person or on Canvas) if you don’t
* Do not change these blocks unless instructed

Any code block that starts with

```julia

is [source code](https://quarto.org/docs/authoring/markdown-basics.html#source-code) and *will not run*.

* You should change these blocks to complete the lab, filling in blanks and adding code as needed.
* To make it a cell that you can run, add the brackets

Changes you are expected to make are marked as follows:

|  |
| --- |
| Important |
| This is an instruction. |

## 1.3 Getting help

You may find the following Quarto documentation pages helpful:

* [Using Julia](https://quarto.org/docs/authoring/markdown-basics.html#source-code)
* [Markdown Basic](https://quarto.org/docs/authoring/markdown-basics.html)

Ask questions in class or post them on the Lab02 discussion on Canvas. The sections of this lab are numbered, so refer to the number when asking questions on Canvas to make it easier for us to help you.

# 2. Setup

## 2.1 Clone the respository

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/...).

## 2.2 Install required packages

1. In VS Code, open the command palette (Windows: Ctrl+Shift+P, Mac: Cmd+Shift+P) and select Julia: Start REPL.
2. In the Julia REPL, type ] to enter the package manager.
3. Type activate . to activate the project environment.
4. Type instantiate to install the required packages. This may take a moment.[[1]](#footnote-36)

## 2.3 Using statement

In Julia we say using to import a package. Typically we want to do this as early as possible in a script or notebook.

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

That this is a code block code and will run. If you have not yet installed the packages, you will see an error message. Don’t change this block – see instructions to install and instantiate.

## 2.4 Check

To make sure everything is working, you should open the command palette and select Quarto: Render HTML. This will generate a HTML file from this notebook. This is a good way to check that everything is working before you start editing code.

# 3. Reading data

We will use the CSV.jl package to read in our data. First, let’s define the file name

fname = "data/tidesandcurrents-8638610-1928-NAVD-GMT-metric.csv";

Line 1

Recall the semi-colon at the end of the line suppresses output. This is a good habit to get into when defining variables.

Next, we will use the CSV.jl package to read in the data.

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

Line 1

CSV.read is a function that takes two arguments: the file name and the type of object to read the data into. In this case, we want a DataFrame.

Line 2

first is a function that takes two arguments: the object to look at and the number of items to display. In this case, we want to look at 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).

## 3.1 Dates

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".

## 3.2 Water levels

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)

|  |
| --- |
| Code explanation |
| 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. 2. \*= 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. 3. 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.

function read\_tides(year::Int)  
 fname = "data/tidesandcurrents-8638610-$(year)-NAVD-GMT-metric.csv" # don't change this  
 date\_format = "yyyy-mm-dd HH:MM" # don't change this  
 # your code 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 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.

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

|  |
| --- |
| Instructions |
| 1. Call the read\_tides function on each year 2. Turn the two code blocks in this section into live code blocks 3. Run the code and make sure the first 5 rows and last five rows look right |

# 4. Plots

Now we’ll create some plots of our data.

|  |
| --- |
| Note |
| We have followed a bad naming convention for a good reason. When we read in the 1928 data we called it df. When we read in the data from all years, we also called it df (which will overwrite the old definition of the variable). This is normally bad practice! **We want use clear, descriptive, and specific variable names!** However, it means that we can make plots here using the 1928 data. Once you are able to read in the data for all years, these plots will automatically update to show data for all years. If you want to rename these variables, please go for it! |

## 4.1 Time series plot

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.

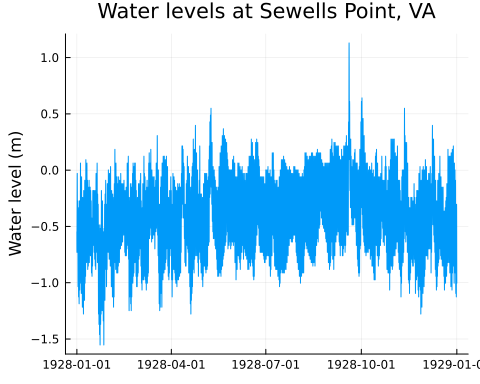
plot(  
 df.datetime,  
 df.lsl;  
 title="Water levels at Sewells Point, VA",  
 ylabel="Water level",  
 label=false,  
)

Line 5

Because we are using the Unitful pacakge, the y-axis label will automatically include the units!!!

Line 6

We are only plotting one “series” (data set), so we don’t need a legend.



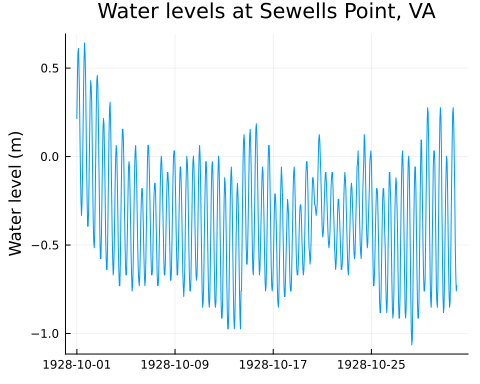
## 4.2 Zooming in

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)

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,  
)



## 4.3 Instructions

Change the start and end dates to plot March 2020. What do you notice? :::

## 4.4 Groupby

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-77)

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

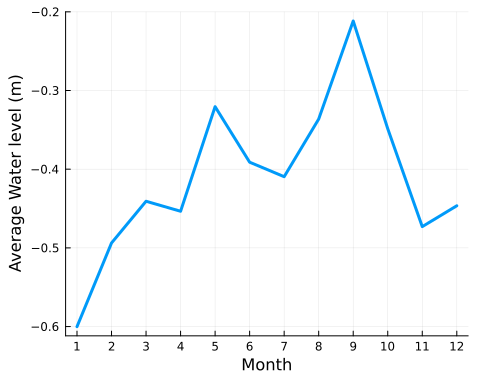
We can use df.colname instead of df[!, :colname]. The latter is more robust but the former is easier to type.

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? |

## 4.5 Boxplot

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:

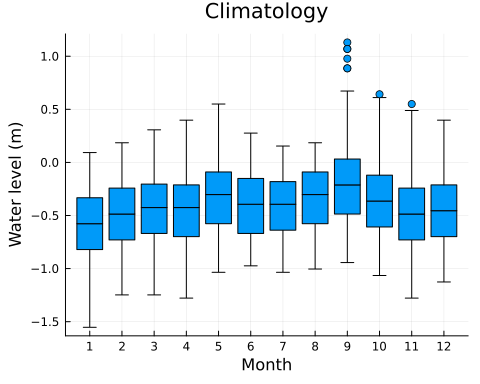
boxplot(  
 df[!, :month],  
 df[!, :lsl];  
 xticks=1:12,  
 xlabel="Month",  
 ylabel="Water level",  
 label=false,  
 title="Climatology",  
)

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? |

## 4.6 Submission

1. Remove all of the instructions blocks (from the first ::: {.callout-important} to the following :::)
2. Remove all of the text I have written (including this block) so that *all the text is your own*. This makes it much easier to grade and to follow what is yours.
   1. Do not delete any of the headers
      1. Thus, the beginning of the document should have the headers for the overview, exploratory modeling, instructions, and getting help, but these should not have any text
      2. This will ensure all documents have the same numbering to make it easier to troubleshoot any issues
   2. Do not delete any of the code
3. Make sure your code runs (click the “Run All” button in the command palette)
4. Render your notebook as HTML (optional)
   1. Open comand palette
   2. Select Quarto: Render HTML
   3. This will create a website that you can view in your browser. The address will be in your terminal like

* Watching files for changes  
  Browse at http://localhost:4200/labs/lab02/instructions.html
  1. Make sure your notebook looks right in the browser.

1. Render your hnotebook as DOCX (required)
   1. Open comand palette
   2. Select Quarto: Render DOCX
   3. It may give you a link to click on in order to download the file. Click it and it will be downloaded to your computer (probably in your Downloads folder)
2. Submit the .docx file to Canvas.
   1. Proofread before you submit!

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-36)
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-77)