Reading and Processing NOAA Climate Records in R

Marc Los Huertos and Isaac Medina

February 10, 2021

1 Introduction

Raw data sets often come with untidy/non-useful formats or information that must first be cleaned or processed before an accurate and useful analysis of the contents can be done. After obtaining a data set there are some preliminary steps you must follow in order to get your data file into working order for your analysis.

1.1 Purpose

This document is intended as a resource and guide to help you:

- upload your data file into the R environment using the Rstudio Server; and
- clean, organize and reformat the data to prepare it for analysis.

2 Preparing CSV file(s)

2.1 Upload CSV Files into Appropriate Rstudio Directory

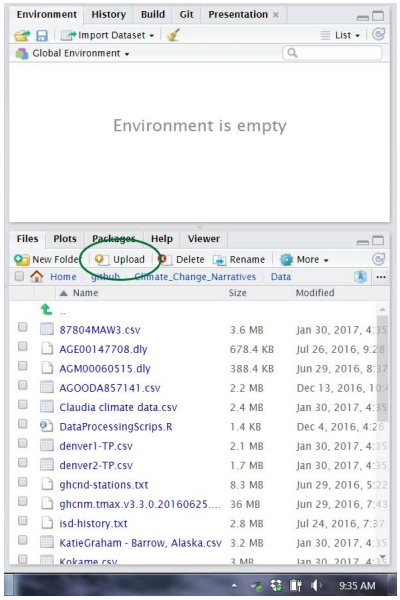
The first step to getting your data into R is to upload it from your computer into the the Rstudio server online but to simplify our work in the future first **locate the file** in your computer and rename it using the following convention:

yourname_region_data

where you fill in your firstname and a short descriptor for the region to which your data pertain.

Now follow these steps to upload the data into the Rstudio server:

1. Use the upload button to select a file from your computer to upload into the rstudio server



2. In the popup window select **Browse** and navigate to the climate data file you downloaded from the CDO website (Step 1—Obtaining Climate Records). It should have the new name you gave it above (yourname_region_data)

3. Click **Open** and then click **OK**.

Your data file is now uploaded to the Rstudio server! All this means is that the actual file is in your workspace on the server.

3 Reading CSV Files into R

3.1 Creating a Record of your R-code

Now that we got the file saved onto the Rstudio Server it's time to tell R to read the file so we can look at the data! This will require telling R to read the file. In order to do this we will use R commands. (remember these commands are to be typed in the R "console")

To begin, create a 'R Markdown' file by navigating the menu in the following order 'File/New File/R Markdown'. Enter a title (can be changed later), enter your name as the author, and make sure you have selected html as an output option, which can also be changed later!

I suggest you save the file and 'knit' it to see what happens, i.e. it works! I will 'knit', which also saves the files every few minutes to make sure the code is doing what I figured it should do, i.e. not give me errors!

Except for the setup options chuck, we can begin altering the R chunks with code that will document your work AND if you have trouble, you can show the

problem to the instructor for trouble shooting!

3.2 Finding the data path to the csv file

In order to tell R to read the file you also have to let it know where the file is (R is funny that way). We use an R command to help us find the specific data path to the file.
In the console type in the following and then follow these steps:

```
file.choose()
```

- press enter on your keyboard
- in the resulting popup window navigate to the your saved data file and **double click** on it
- Look at the console window.. under the command that you just typed you should now see a data path. The data path should look something like this: "/home/CAMPUS/im022012/Singapore_ClimateData.csv"
- **Copy** the data path (including the quotation marks)

I suggest we create an object with the path, for example, often write out something like this:

```
climate_data.csv = ``your file path''
```

where I paste in the results of the file choose into the quoted section.

3.3 Telling R to Read the Data File

Now that we know where the data file is we will tell R to read it! In the console type in the following command and follow the next steps:

```
read.csv()
```

- This time before you press enter on the R command **Paste** in the data path you copied inbetween the parenthesis of the R command (be sure to include the parenthesis). As described above, I recommend using the path object we created, i.e.

  ```
  read.csv(climate_data.csv)
  ```

- press enter
- If your command worked you should see a bunch of text in your console window now. This means R read your file!

3.4 Creating a Data Object

So we made R read our file, however cool this might be, it is still not useful to us. In order to manipulate and do things with the data we need to tell R not only to read the file but also to store it in an object (such as a data frame) that we can manipulate. Follow these steps to tell R to read the file and store it in an object:

This time we will use the following R command

```
climate_data <- read.csv(``the/file/path'')
```

But be sure to use your actual data path!

What this command is essentially doing is it's telling R to read the file and store it in an object called climate_data ... if we really wanted to we could have actually named the object something else but climate_data will suffice for now.

In R usually "no news is good news" after you hit enter on a command. In other words, if you don't get red colored text describing an error, the program did something – but now we need to figure out if it did something useful! Luckily there is a way to check if your command was successful in creating your object. All you have to do is check the **Environment** tab (in window 2). You should see your data object: climate_data listed there.

4 Confirming the Proper Reading of the CSV file

So we see that the data file is actually an object in R now. But what does it actually look like and what's in it? Since R was made to handle large data sets it tries not to overwhelm you with all the data at once but luckily the developers made a few commands to peak at the data to see if everthing is in order. We will try some of these commands now to look at few observations then we'll evaluate the structure of the dataframe and go on to finally plot the data!

4.0.1 Viewing the 1st 6 Observations

The following command allows you to view the first six observations of data in the data frame you created.

```
head(climate_data)
```

Type it into the console and hit enter! You should get something that looks like the following

```
head(climate_data)
```

```
##             STATION                       STATION_NAME     DATE     PRCP TAVG
## 1 GHCND:SNM00048698 SINGAPORE CHANGI INTERNATIONAL SN 19620512     0.16   81
## 2 GHCND:SNM00048698 SINGAPORE CHANGI INTERNATIONAL SN 19620516     0.08   79
## 3 GHCND:SNM00048698 SINGAPORE CHANGI INTERNATIONAL SN 19620605 -9999.00   84
## 4 GHCND:SNM00048698 SINGAPORE CHANGI INTERNATIONAL SN 19621006     0.39   83
## 5 GHCND:SNM00048698 SINGAPORE CHANGI INTERNATIONAL SN 19621007     0.02   84
## 6 GHCND:SNM00048698 SINGAPORE CHANGI INTERNATIONAL SN 19621012     0.08   78
##    TMAX  TMIN
## 1 -9999    75
## 2    86    75
## 3    93    77
## 4    91 -9999
## 5    91    79
## 6    84 -9999
```

You can see how the data is formatted into columns and rows like in Excel.

4.1 Evaluating the structure of the object

Okay, so you now have created a data frame and taken a peak at it. But what if you want more information about it? Such as how many columns (a.k.a. variables) are in it. Or how many rows (a.k.a. observations) are in it. You can use the following command:

`str(climate_data)`

This function allow you to peer into the structure of the data frame giving output which describes how many variables they're are, how many observations of each as well as other useful information. Below you can see an example of the command. Checking the structure of the data frame is a another way to ensure that the data have been imported in a way that you expect.

```
str(climate_data)

## 'data.frame': 13704 obs. of  7 variables:
##  $ STATION     : Factor w/ 1 level "GHCND:SNM00048698": 1 1 1 1 1 1 1 1 1 1 ...
##  $ STATION_NAME: Factor w/ 1 level "SINGAPORE CHANGI INTERNATIONAL SN": 1 1 1 1 1 1 1 1
##  $ DATE        : int  19620512 19620516 19620605 19621006 19621007 19621012 19621022 1962
##  $ PRCP        : num  0.16 0.08 -9999 0.39 0.02 ...
##  $ TAVG        : int  81 79 84 83 84 78 81 78 79 81 ...
##  $ TMAX        : int  -9999 86 93 91 91 84 91 88 91 -9999 ...
##  $ TMIN        : int  75 75 77 -9999 79 -9999 75 73 73 75 ...
```

4.2 Confirming the Column Names

In R it isn't uncommon to want to manipulate a specific column of data within your data frame. Therefore it is useful to know what the names of the columns in your data frame are. The following command can be used to make R give you the names of the columns in your data frame.

```
names(climate_data)

## [1] "STATION"      "STATION_NAME" "DATE"         "PRCP"         "TAVG"
## [6] "TMAX"         "TMIN"
```

A data frame is essentially a set of "vectors". Which themselves are like lists of numbers or text. You can think of each column as one of the vectors inside your data frame. If you want to access the data in one of the columns specifically, you can use the following command syntax:

`nameofdataframe$columnname`

so for example, you can type in:

`climate_data$TMAX`

And R will spit out the data in just that column.

5 Plotting the Data

Now we will check the data by plotting it (Figure 1).

We find some rather odd low temperature values in the plot. We can find some of these with the `min()` function. This function works when there are no missing data, otherwise it will just report NA.

```
min(climate_data$TMAX)

## [1] -9999
```

To get the function to cooperate if there are missing values, add 'na.rm=T' to the function:

```
min(climate_data$TMAX, na.rm=T)

## [1] -9999
```

```
plot(TMAX~DATE, climate_data)
```

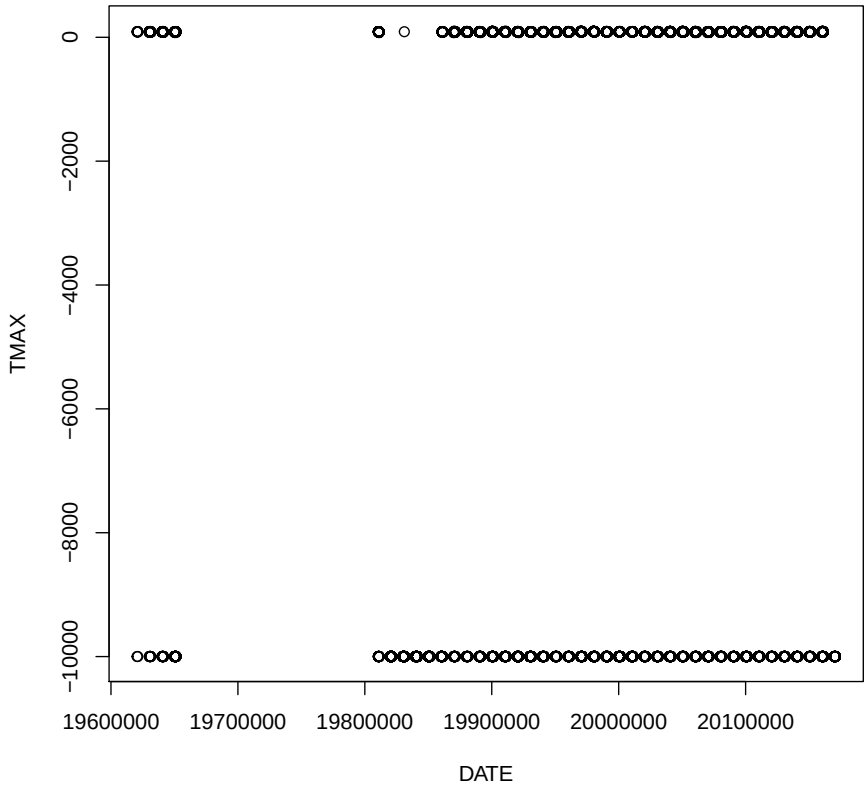


Figure 1: These data are plotted with DATE on the x-asis and TMAX on the y-asis. These data have a fair amount of missing data (-9999). Not all the datasets will report missing data in this way. In some cases, they are just missing.

5.1 Re-assigning Missing Values to NAs

In some of the stations, missing values were coded as -9999? These are used for missing data. Historically, computers didn't have a lot of options for mixing numbers and letters in a variable type while R has some built in flexibility for this. So, to avoid leaving values blank (with all the ambiguous interpretations), the value -9999 is used to sybmolize missing values, since the number is unrealistic in the real world!

If your data do not have missing data, skip these steps!

If your station used -9999 for missing values, we will replace these with NA, which R uses specifically to avoid accidently averaging arbitrary values that are representing missing values.

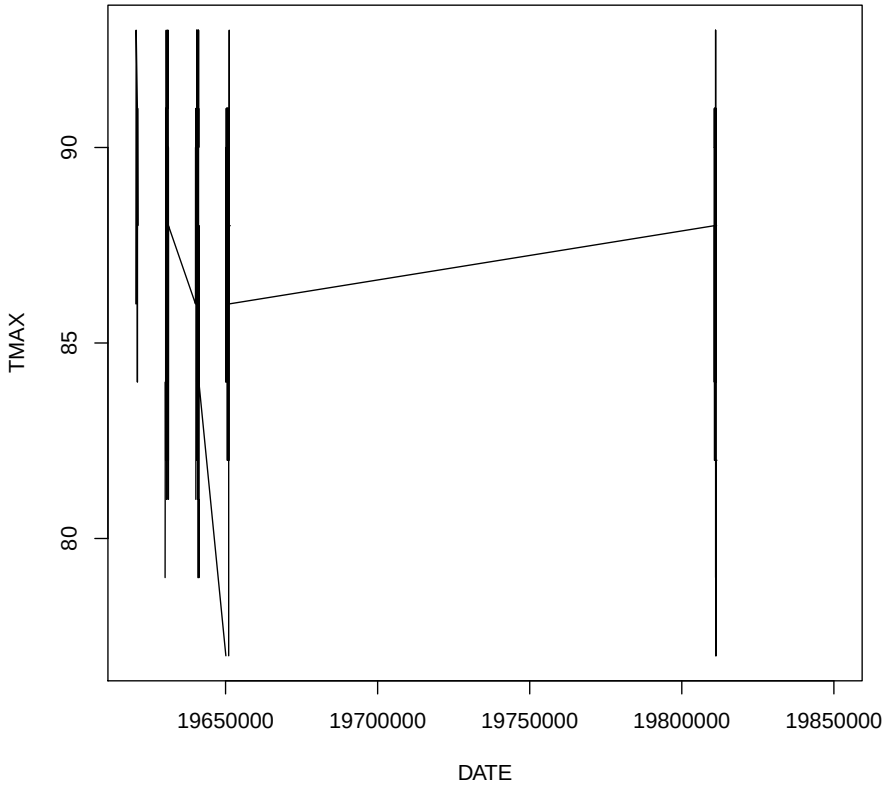
How do we do this?

```
climate_data$TMAX[climate_data$TMAX==-9999] = NA
climate_data$TMIN[climate_data$TMIN==-9999] = NA
```

Okay, now we'll check again, but let's plot a just a few years, let's say five years (356 days * 5 years = 1825) or 1825 observations. I am using this to check my dataframe – based on the number of observations that I have, but your data set will vary and you may find this selection of rows to be unhelpful.[1]

```
plot(TMAX~DATE, climate_data[1:1835,], ty='l')
```

[1] IMPORTANT: when we plot with a type (ty) as a line, we define that as the letter 'l' not the number '1'!



In the case of the Singapore data we have a new problem. What's wrong? It appears we have gaps in the data – but we already removed our missing data, why are these big jumps in the data.

As it turns out stations might have one of three types of date formats – and each of them are problematic. The first thing to note is that they are factors instead of begin formatted as a Date. You can check this with the following function:

```
str(climate_data)

## 'data.frame': 13704 obs. of  7 variables:
##  $ STATION     : Factor w/ 1 level "GHCND:SNM00048698": 1 1 1 1 1 1 1 1 1 1 ...
##  $ STATION_NAME: Factor w/ 1 level "SINGAPORE CHANGI INTERNATIONAL SN": 1 1 1 1 1 1 1 1
##  $ DATE        : int  19620512 19620516 19620605 19621006 19621007 19621012 19621022 1962
##  $ PRCP        : num  0.16 0.08 -9999 0.39 0.02 ...
##  $ TAVG        : int  81 79 84 83 84 78 81 78 79 81 ...
##  $ TMAX        : int  NA 86 93 91 91 84 91 88 91 NA ...
```

```
##  $ TMIN        : int  75 75 77 NA 79 NA 75 73 73 75 ...
```

In this case, we find that Date is a factor, which means that we will not be able to determine a trend because R has not recognized the data as a continuous variable. Look carefully at the structure, i.e. str() of the data frame – note the format that the date is in – usually a factor, which is not helpful!

Factors in R are thought of as unique observations of character or numeric strings. But R does not understand the numeric and ordered nature of dates, e.g. 12 months, some months with 28, 29 (leap year), 30, and 31 days. When you think about it, dates are very strange.

So, we need to convert these to a date format that R understands. This is the hardest part of this process. Pay attention to the date formats! If you assume a particular format that is wrong, it can lead to a reasonbly frustrating process. Note: I have found myself missing the details of the stage below leading to the loss of valuable time.

5.2 Determine and Convert Date Format

To create a new format, we have to complete a few steps. Unfortunately, date formats are one of the more obtuse aspects of R, but if you follow along, you should have success, even if you have no clue what you did.

First, we convert the date to a string of character values. Next, we'll convert the strings to a data format. But we need to determine which type of data format we have from the following three choices then we can make the conversion:

MM/DD/YYYY This is a standard data for many US stations. To convert the dates, we first create a vector of just the dates. Then, we can reformat the dates using the syntax below:

```
strDates <- as.character(climate_data$DATE)

climate_data$NewDate <- as.Date(strDates, "%m/%d/%Y")
```

Caution: Be sure the dates were actually converted correctly, using the `head()` or `str()` functions.

MM-DD-YYYY Sometimes, the dates are formatted by month-day-year, where we can use the code below to reformat the dates:

```
strDates <- as.character(climate_data$DATE)

climate_data$NewDate <- as.Date(strDates, "%m-%d-%Y")
```

YYYY/MM/DD Sometimes, the dates are formatted by year/month/day, where we can use the code below to reformat the dates:

```
strDates <- as.character(climate_data$DATE)

climate_data$NewDate <- as.Date(strDates, "%Y/%m/%d")
```

YYYY-MM-DD Sometimes, the dates are formatted by year-month-dayy, where we can use the code below to reformat the dates:

```
strDates <- as.character(climate_data$DATE)

climate_data$NewDate <- as.Date(strDates, "%Y-%m-%d")
```

YYYYMMDD In certain cases, the dates are formatted as a long number starting with year. Although these dates sort correctly, they create big gaps at the new year: Let's say that the data have a year change between 1913 and 1914. The date format in the NOAA data are YYYYMMDD, or year, month, and day with 4, 2, 2 digits, respectively. Thus, the last day of 1913 is 19131231 or Dec, 31, 1913. The next day is January 1st or 19140101. But when you plot these on the x-axis, the order of the values should be 19131231 $\rightarrow$ 19131232 $\rightarrow$ 19131233 $\rightarrow$ 19131234, etc but there is no 32nd, 33rd or 34th of December. Instead the dates go from 19131231 $\rightarrow$ 19140101. We have lots of numbers that are skipped, but no coded as missing, but missing all the same. So, now we need to convert our dates to something more sensible. In R, that means creating a variable with a format that expects dates, thus doesn't plot numbers that are impossible dates!

```
strDates <- as.character(climate_data$DATE)

climate_data$NewDate <- as.Date(strDates, "%Y%m%d")
```

2-digit Years Is a total mess. See Marc for help!

Mixed Formats ... see Valentina's data!

Doesn't seem to work, I had to brut force it... will work on this next summer!

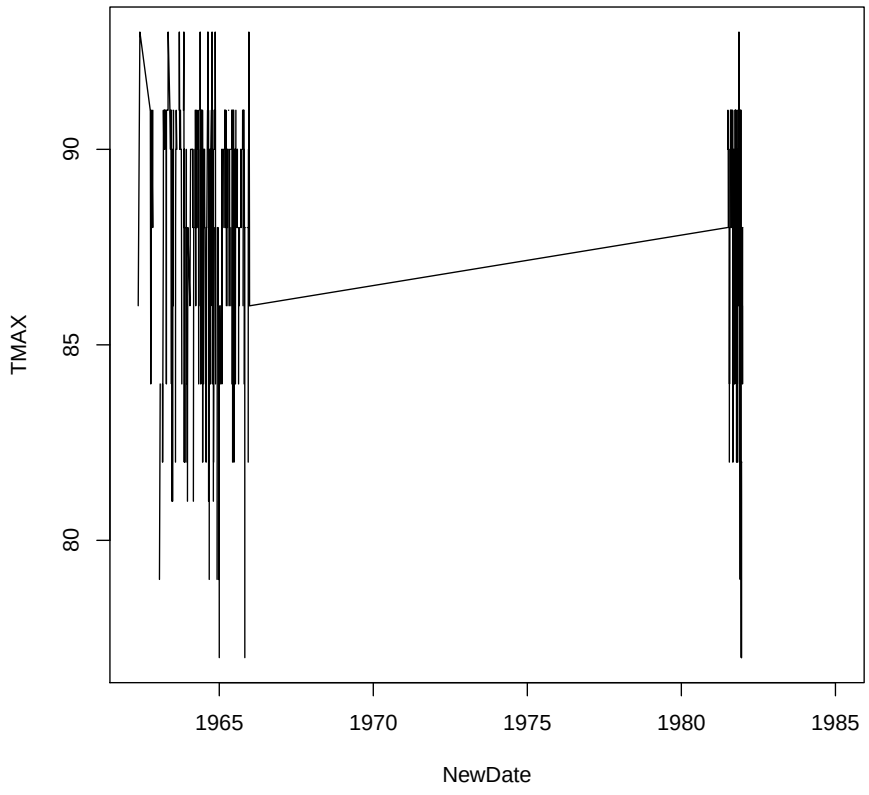
```
strDates <- as.character(climate_data$DATE)
head(strDates)

## [1] "19620512" "19620516" "19620605" "19621006" "19621007" "19621012"

climate_data$NewDate <- as.Date(strDates, "%Y%m%d", tryFormats = c("%Y-%m-%d", "%Y/%m/%
    optional = FALSE)
```

```
climate_data$NewDate <- as.Date(strDates, "%Y%m%d", tryFormats = c("%Y-%m-%d", "%m/%d/%
    optional = FALSE)
```

5.3 Checking the New Dates

To check the NewDate variable, I like to make sure I can get a plot that makes sense. In the code below I have subsetted the data for the first 1835 observations – however, you may not have that many, so you'll get an error. Remove the code with the square brackets (including the square brackets) and that should work.

```
plot(TMAX~NewDate, climate_data[1:1835,], ty='l')
```



With these data, you can see we have a very strange pattern – lots of missing data. At this point, I might need to find a better dataset – however, because

I truncated the data, the more recent data might be fine. This is part of the scientific process – deciding on what qualifies as high quality data!

6 Preparing Records for Analysis

Our next task is to create a template to automatically run our code, so we can regenerate the pre-processed data within a Rmd file, which will then produce a figure.

6.1 Summarizing wat your code is doing

Thus, far, we have already completed the following...

1. Find the path for the csv file (`file.choose()`).
2. Read the file into R (`read.csv(filepathfilename.csv`)).
3. Replace missing values with NA.
4. Fix the date formats.

If you have been successful with these steps, our nextg action items include the following:

1. Create a figure of Tmax versus time (`plot()`).
2. Overlay the best fit line (`abline()`, `coef()`, and `lm()`).
3. Go to "Step 3: Evaluating Monthly Trends using CHCN-Daily" to learn about how to analyze data.

Note, I have not outlined how to do this here, but it is outlined in the next guide.