Lecture 4: Data-Cleaning and Workplace Management

Kelly and Gallego

Let's be honest. You've been meaning to get your shit together for years, and perhaps it hasn't happened yet. Your computer, like your desk at home – do you even have a desk at home? – is a mess.

Your computer has files named `final_FINAL_analysis_reallyThisTime_USEthisONE3.docx` , and if you had to re-do a plot that you did a year ago, there's no way you could get that done.

But now – *NOW* – is the time, my friends, to pull it together. The computer cannot think for you. The computer cannot guess what you really mean. The computer will do precisely what you tell it to, and nothing more. So part of learning a bit about coding is learning to think more like a computer. And part of this thinking, in turn, is learning to be more organized so that:

- you can point to particular files in particular places
- you can re-do analyses and get the exact same answer you did last time
- you can plot and re-plot graphs as necessary, with ease
- your future self can understand what your present self is doing

The *Pirate's Guide* Says:

Why object and file management is so important

> Your computer is a maze of folders, files, and selfies. Outside of R, when you want to open a specific file, you probably open up an explorer window that allows you to visually search through the folders on your computer. Or, maybe you select recent files, or type the name of the file in a search box to let your computer do the searching for you. While this system usually works for non-programming tasks, it is a no-go for R. Why? Well, the main problem is that all of these methods require you to *visually* scan your folders and move your mouse to select folders and files that match what you are looking for. When you are programming in R, you need to specify *all* steps in your analyses in a way that can be easily replicated by others and your future self. This means you can't just say: "Find this one file I emailed to myself a week ago" or "Look for a file that looks something like `experimentAversion3.txt`." Instead, need to be able to write R code that tells R *exactly* where to find critical files – either on your computer or on the web.
>
> To make this job easier, R uses *working directories*.

The working directory

> The **working directory** is just a file path on your computer that sets the default location of any files you read into R, or save out of R. In other words, a working directory is like a little flag somewhere on your computer which is tied to a specific analysis project. If you ask R to import a dataset from a text file, or save a dataframe as a text file, it will assume that the file is inside of your working directory.
>
> You can only have one working directory active at any given time. The active working directory is called your *current* working directory.
>
> To see your current working directory, use `getwd()`:

```
# Print my current working directory
getwd()
```

```
## [1] "/Volumes/GoogleDrive/My Drive/RPKDesktop/SMEA/SMEA_550C_Rcourse_2019/Lectures"
```

If you want to change your working directory, use the `setwd()` function. For example, if I wanted to change my working directory to an existing Dropbox folder called `yarrr`, I'd run the following code:

```
# Change my working directory to the following path
setwd(dir = "/Dropbox/yarrr")
```

[And back to us, away from the *Pirate's Guide*]...

A Suggestion

...So the working directory is like a "You Are Here" sign for **R**, pointing to the location of where things are happening – and where files are – as you're working right now.

To keep everything straight, for every project/assignment/class:

- create one main folder (= directory, e.g. `mainFolder`) containing no spaces in the title and not starting with a number (computers hate that).
- inside that folder, create subfolders that remain consistent from project to project. For example, `mainFolder/Data` , `mainFolder/Analysis` , `mainFolder/Writing` , and `mainFolder/Figures`.
- then, in **R**, you can keep your working analysis in the `/Analysis` folder, which uses raw data from the `/Data` folder, plots to the `/Figures` folder, etc. And this hierarchy doesn't change from project to project... only the name of the main folder changes. So you don't get confused.

Cleaning Your Data

It's quite possible that alongside your magnum opus `final_FINAL_analysis_reallyThisTime_USEthisONE3.docx` you also have `revised_Data_RAW_processed_normalized2_rpk.xlsx`, which contains something like this.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Sample	Well_in_	**Extract me**	Tag	Lib	Combo	Random	Random_	Final_lib	Qubit PC	Qubit391	ul templa
2	SA_201706A.1	A1	1:50	01	A	01A	12B	12	B	3.73		3
3	SA_201706B.1	B1	1:50	02	A	02A	04D	04	D	3.22		3
4	SA_201706C.1	C1	1:50	03	A	03A	05C	05	C	4.38		3
5	TR_201706A.1	D1	1:50	04	A	04A	12A	12	A	2.82		3
6	TR_201706B.1	E1	1:50	05	A	05A	10B	10	B	1.52		5
7	TR_201706C.1	F1	1:50	06	A	06A	08B	08	B	2.76		3
8	LL_201706A.1	G1	1:50	07	A	07A	15B	15	B	0.85		5
9	LL_201706B.1	H1	1:100	08	A	08A	10C	10	C	1.81		5
10	LL_201706C.1	A2	1:10	09	A	09A	01A	01	A	1.82		5
11	PO_201706A.1	B2	1:200	10	A	10A	05B	05	B	0.4		5
12	PO_201706B.1	C2	1:200	11	A	11A	04B	04	B	0.24		5
13	PO_201706C.1	D2	1:200	12	A	12A	14A	14	A	0.59		5
14	TW_201706A.1	E2	1:100	13	A	13A	10E	10	E	0.69		5
15	**TW_201706B.1**	**F2**	**1:200**	**14**	**A**	**14A**	**03D**	**03**	**D**	**0.37**	**0.41**	**5**
16	TW_201706C.1	G2	1:100	15	A	15A	01E	01	E	1.52		5
17	K+.1	H2	1:100	16	A	16A	16B	16	B	9.32		1
18	**SA_201708A.1**	**A3**	**1:10**	**01**	**B**	**01B**	**12F**	**12**	**F**	**0.32**	**0.32**	**5**
19	SA_201708B.1	B3	1:10	02	B	02B	05E	05	E	2.98		3
20	SA_201708C.1	C3	1:10	03	B	03B	15C	15	C	1.77		5
21	TR_201708A.1	D3	1:10	04	B	04B	06C	06	C	0.59		5
22	TR_201708B.1	E3	VOID	VOID	VOID	VOID	VOID	VOID	VOID	0		5
23	TR_201708C.1	F3	VOID	VOID	VOID	VOID	VOID	VOID	VOID	0		5
24	LL_201708A.1	G3	1:10	07	B	07B	12D	12	D	0.82		5
25	LL_201708B.1	H3	1:10	08	B	08B	03C	03	C	1.73		5

You'll notice all kinds of things here... many of which are problematic for processing data. For example:

- a mix of numeric and non-numeric data within a column (should the computer treat this as a number,

or not?)
- formatting as data: what are bold values? what about those with colored backgrounds? (this formatting encodes information of some kind... but it will be lost if the formatting changes, and furthermore, my future self won't know what it means.)
- a mix of missing-data values (do "VOID" and mean the same thing here?)
- some values have notes attached to them (how should we handle this when importing data?)

So, let's do some practice cleaning a dataset so we can work with it. A "clean" dataset is one that is ready for analysis, without problematic formatting or punctuation or craziness. For example, a computer sees these values differently: `" 0"`, `"0"`, `0`, `zero`, `Zero`, `none`. In Excel, you (a human), might interpret those the same way, but the computer has no idea what to do with them. (Note the first and second examples differ only by a space... which, when it's in a character string, is a character, even if it's invisible to the human eye. And the third element is a number, when the first two are characters. See what I mean? Data cleaning is important).

As a first step, I've opened a file in Excel, and saved as a .csv text file. [There are other ways to do this, including reading directly into **R** from Excel using the package `readxl` , but for simplicity and universality, let's do it this way for now.]

```
rawdata <- read.csv("../Data/field_samples.csv")

names(rawdata)  #give the names of the columns
```

```
##  [1] "date_collected"   "time"             "site_name"
##  [4] "site_code"        "subsite_name"     "lon"
##  [7] "lat"              "salinity"         "temp"
## [10] "depth_sample..m." "depth_bottom_m"   "collector"
## [13] "field_label"      "lab_label"        "notes_field"
```

```
#View(rawdata)  #take a look at the resulting dataframe
```

So... we've got a mix of things here, and some of the potential problems we identified in the screenshot earlier.

Looking at the columns `date_collected` and `time`, should those be factors? They should probably be in some more useful format. And what about the temperature field `temp`? Shouldn't that be numeric? Because it's not right now.

factor

For example, say we want to know the change in water temperature over the days sampled. We can't easily do that right now.

```
plot(temp~date_collected, data=rawdata)
```

This gives us kind of a nonsense plot. That's because we haven't cleaned our data properly, and we have the temperature variable being read as a factor, rather than as a number.

One way of cleaning your data would be to work in Excel and just fix the things you can see that are wrong. For example, it's obvious that one of the numbers in the temperature variable isn't written as a number, but rather as a character string (it's in quotes), and that there are other issues with that same variable (it's got a mix of numbers and words):

```
rawdata$temp
```

```
##  [1] 14         14         12         12         <NA>       <NA>
##  [7] HOBO broke HOBO broke HOBO broke HOBO broke 13         13.5
## [13] 13.5       13.5       13.5       13         13         14
## [19] 13         13         14         13.7       13         13
## [25] 13         "13"       12         12         14         13.5
## [31] 13         13         13         13         13         13
## [37] 12         12.5       13         12         12
## Levels: "13" 12 12.5 13 13.5 13.7 14 HOBO broke
```

But you can perhaps more easily (and more repeatably) do this in **R**, like so:

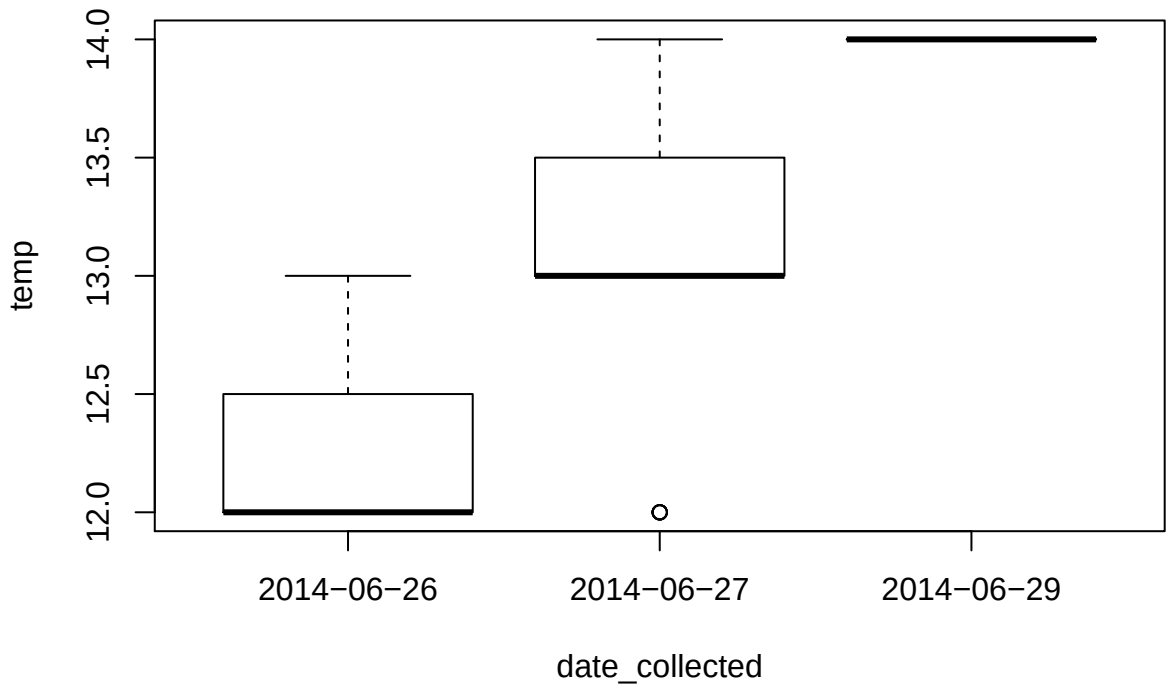
```
cleandata <- rawdata  #first, we might want to create a duplicate dataframe to work from, calling this

as.numeric(cleandata$temp) #this doesn't work, because the computer has no idea how to turn a mix of fa
```

```
##  [1]  7  7  2  2 NA NA  8  8  8  8  4  5  5  5  5  4  4  7  4  4  7  6  4
## [24]  4  4  1  2  2  7  5  4  4  4  4  4  4  2  3  4  2  2
```

Also, notice that the values that **R** gave to the numeric version of your factor are the order in which the factor levels were ordered on the first place. On the first two instances, the value "14" was converted into 7, because "14" was the 7th level of the factor `temp`.

```
cleandata$temp <- gsub("HOBO broke", NA, cleandata$temp) #replace all instances of "HOBO broke" with NA

cleandata$temp <- gsub("\"", "", cleandata$temp) #replace all quotation marks with nothing.  [Here, we
```

```
cleandata$temp <- as.numeric(cleandata$temp)  #now this works, because the data are all in a single, pr

plot(temp~date_collected, data=cleandata) #this now makes more sense
```



Interesting side note: regular expressions

If you think this is cool, you are right. But you can think of a file of 1000s of rows with different mistakes - It will take a fair amount of time to go through all of them. It might be too much. If we assume that all letters have no place in our `temp` variable.

```
gsub("[A-Z]", NA, rawdata$temp)
```

```
##  [1] "14"     "14"     "12"     "12"     NA       NA       NA
##  [8] NA       NA       NA       "13"     "13.5"   "13.5"   "13.5"
## [15] "13.5"   "13"     "13"     "14"     "13"     "13"     "14"
## [22] "13.7"   "13"     "13"     "13"     "\"13\"" "12"     "12"
## [29] "14"     "13.5"   "13"     "13"     "13"     "13"     "13"
## [36] "13"     "12"     "12.5"   "13"     "12"     "12"
```

`[A-Z]` is what we call a `regular expression`. In this case, it matches all capital letters and replaces them with NA

There are many different regular expressions, and they are tricky to understand when reading code, but useful in the way they search and match around big datasets. If you use them in your code, add a comment explaining what do you want to search or replace.

Adding a new level to a factor

We looked at the dataset, and turns out we want to change the subsite_name for those with a "NA" value for center2. Let's try.

```
cleandata$subsite_name[is.na(cleandata$subsite_name)] <- "center2"
```

```
## Warning in `[<-.factor`(`*tmp*`, is.na(cleandata$subsite_name), value =
```

```
## structure(c(NA, : invalid factor level, NA generated
```

And this is why factors are awful. They don't like new levels to be added to them.

As it happens in **R**, there are many ways to solve this. One would be to convert the factor into a column of type `character`, and then manipulate it freely

```
cleandata$subsite_name.test <- as.character(cleandata$subsite_name)

cleandata$subsite_name.test[is.na(cleandata$subsite_name.test)] <- "center2"

cleandata$subsite_name.test <-NULL
```

But you maybe want to keep the column as a factor. Then you can use `levels()` to find out what are the *allowed* values of your factor, and then add a new value or subsitute one current value

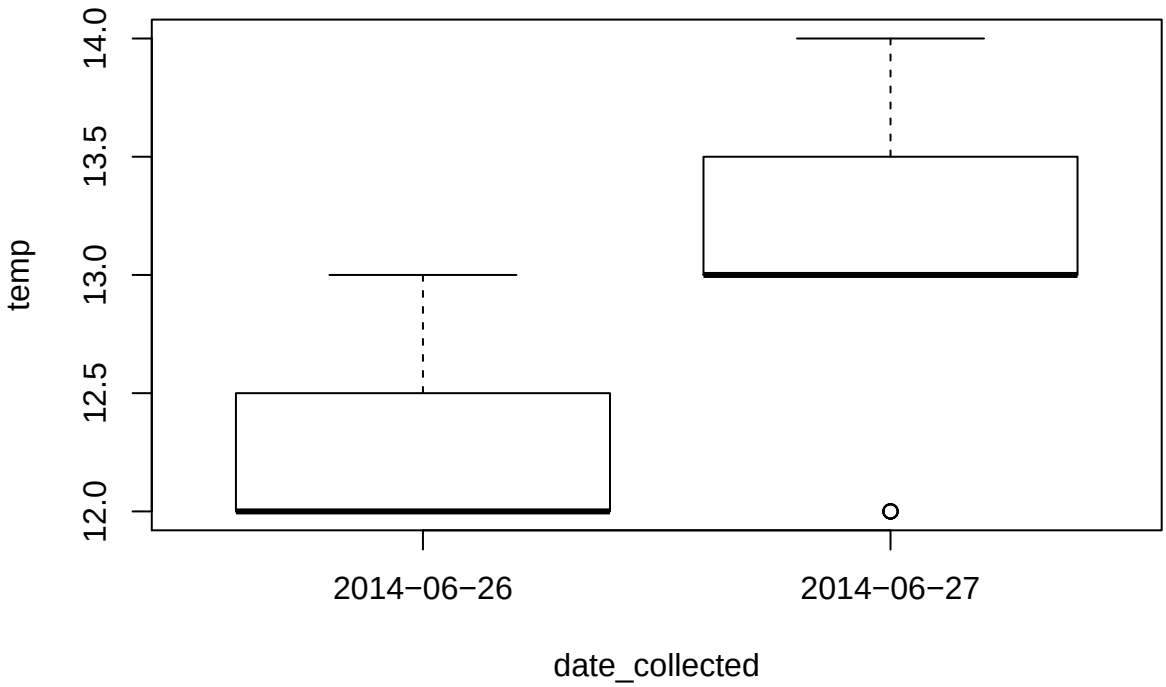
```
levels(cleandata$subsite_name) # Four valid values
```

```
## [1] "center"  "central" "north"   "south"
```

```
levels(cleandata$subsite_name) <- c(levels(cleandata$subsite_name), "center2") # Add a fifth value

cleandata$subsite_name[is.na(cleandata$subsite_name)] <- "center2" #
```

Date

```
#we can also tell R that the date_collected column is a date, rather than an unordered factor
cleandata$date_collected <- as.Date(cleandata$date_collected)

#which is useful for subsetting, putting things in order, etc

boxplot(temp~date_collected,
     data=cleandata[cleandata$date_collected<"2014-06-29",]) #just plot dates before June 29
```



###Never Repeat Yourself

Humans are really good at being lazy. So we have invented tools to avoid doing repetitive work. And sure, typing the same thing over and over again isn't as much work as, say, coal mining, but still.

So the rule-of-thumb is: if you find yourself repeatedly doing some task, you are doing it wrong. There is a tool to help you.

####Find-and-Replace

We saw above the function `gsub` , which is extremely useful in this regard. [And note that the class of similar functions is called `regular expressions`, or `regexp` for short. This is true across many coding languages.]

```
?gsub
```

A huge benefit of automatically substituting values is that the computer doesn't make mistakes. People do.

####String Splitting

Perhaps you have a bunch of labels called `mylabel_123`, and you want to use just part of that label, such as the `123`. If you just have one, sure, you could just delete the first part of the label. But if you have hundreds, you'd be doing the same operation many times. Instead, you can use `strsplit` , which lets you split a string based upon a character you choose... here, we can choose the underscore:

```
strsplit("mylabel_123", split = "_")
```

```
## [[1]]
## [1] "mylabel" "123"
```

The output is a LIST with elements split by the character you've chosen.

Sometimes you will want to separate the values of a column using special characters. I, for instance, generate the sample names for my projects using the format **St_YYYYMMDD_A**. That contains the Site or Station (St), the date of sampling and the replicate measurement (usually A, B C...). In R, you can easily split the sample name and retrieve these values:

```
OA.Samples <- read.csv("../ProblemSets/OA_Samples.csv", stringsAsFactors = F)

head(strsplit(OA.Samples$SampleName, "_") )
```

```
## [[1]]
## [1] "PO"       "20170311" "A"
##
## [[2]]
## [1] "PO"       "20170311" "B"
##
## [[3]]
## [1] "PO"       "20170311" "C"
##
## [[4]]
## [1] "TW"       "20170311" "A"
##
## [[5]]
## [1] "TW"       "20170311" "B"
##
## [[6]]
## [1] "TW"       "20170311" "C"
```

But, the output is a list with a variable number of elements. And what I had in mind is to create three new columns in the same dataframe named Site, Date and Replicate.

There is another tool to do this. Is called `separate`, and is part of the tidyverse, so we will dive into them later on. But as a taste, see how nicely it does things

```
head( separate(data = OA.Samples,# From the dataframe OA_Sampes
         col = SampleName, # Separate the column Sample
         into = c("Site", "Date", "Replicate"), # into 3 columns (It will fill columns with NAs if ther
         sep = "_", # Separate with underscore
         remove = F) )# and Keep the original
```

```
##      SampleName Site     Date Replicate                  Area SampleShort
## 1 PO_20170311_A   PO 20170311         A Hood Canal Intertidal  PO_201703A
## 2 PO_20170311_B   PO 20170311         B Hood Canal Intertidal  PO_201703B
## 3 PO_20170311_C   PO 20170311         C Hood Canal Intertidal  PO_201703C
## 4 TW_20170311_A   TW 20170311         A Hood Canal Intertidal  TW_201703A
## 5 TW_20170311_B   TW 20170311         B Hood Canal Intertidal  TW_201703B
## 6 TW_20170311_C   TW 20170311         C Hood Canal Intertidal  TW_201703C
##     Station Latitude Longitude     Time     Tide Month
## 1 PO_201703 47.35981 -123.1556 12:30:00 Incoming March
## 2 PO_201703 47.35981 -123.1556 12:30:00 Incoming March
## 3 PO_201703 47.35981 -123.1556 12:30:00 Incoming March
## 4 TW_201703 47.37843 -122.9748 13:30:00 Incoming March
## 5 TW_201703 47.37843 -122.9748 13:30:00 Incoming March
## 6 TW_201703 47.37843 -122.9748 13:30:00 Incoming March
```

```
# I used head() wrapping the command separate() to avoid the enormous dataframe that results
```

####Coercion

Often in data-cleaning you have a value in one format and you want the computer to see it as a different format. So, you've got `"0"` – which the computer sees as a character – and you want `0` – which the computer sees as a number.

```
?as.numeric  #to make numeric
?as.factor  #to make a categorical variable
```

```
## Help on topic 'as.factor' was found in the following packages:
##
##   Package               Library
##   generics              /usr/local/Cellar/r/3.6.0_3/lib/R/library
##   base                  /usr/local/Cellar/r/3.6.0_3/lib/R/library
##
##
## Using the first match ...
```

```
?as.character #to make a character string / word
```