Data Cleansing I've Tried Scrubbing Even Soaking

Getting Started with R for Laboratory Medicine
Sunday Aug 04, 2019
AACC 2019, Anaheim CA

Dennis Orton

7/23/2019

Contents

1 Lesson 3: Cleansing and beautifying data - garbage to gold

Lets face it, a real-life data set is never going to be as clean as the examples we will give you in a controlled environment such as a workshop with a time limit, so we're not likely going to be able to cover every situation you may find yourself in. What we can do is try to give you some examples of the types of data formatting issues we have faced in our experiences and show you some examples of how to deal with troublesome data in general.

The main data formats that we encounter are factor, numeric, character, and date. The purpose of this section is to review these formats, how to convert between them, and how to "clean" imperfections out of the data to generate usable datasets. we will start with general data structure and work out from there.

1.1 Setting up and formatting your data

The most common form of data structure in the Clinical Lab is likely going to be a `dataframe`. Whether you are importing `.csv` files to plot patient comparison data, or want to summarize your QC running mean and SD, this is likely going to be your format of choice. For this section, we will start by synthesizing a "practice" data set.

Note that if you ever post questions online, you will need to know how to generate a "representative" data set, so knowing how to generate a dataframe can be helpfull as well!

```
# Start by constructing a dataframe
date <- c("June 28, 2019", "June 29, 2019",
          "June 30, 2019", "July 1, 2019",
          "July 2, 2019")
time <- c("18:45", "16:36", "7:30", "14:22", "12:36")
patient.ID <- c("A", "B", "C", "D", "E") # Here is a vector of patient IDs
result <- c(5, 6, 4, 7, 5) # Here is a vector of results
df <- data.frame(patient.ID, result, date, time)
# This generates a dataframe with two columns (variables) and five rows (observations)
df
```

```
##   patient.ID result          date  time
## 1          A      5 June 28, 2019 18:45
## 2          B      6 June 29, 2019 16:36
## 3          C      4 June 30, 2019  7:30
## 4          D      7  July 1, 2019 14:22
## 5          E      5  July 2, 2019 12:36
```

Never assume that R has interpreted what you plan to do with this dataframe correctly. It's good practice to always check the format of your data before moving on!

```
str(df)
```

```
## 'data.frame':    5 obs. of  4 variables:
##  $ patient.ID: Factor w/ 5 levels "A","B","C","D",..: 1 2 3 4 5
##  $ result    : num  5 6 4 7 5
##  $ date      : Factor w/ 5 levels "July 1, 2019",..: 3 4 5 1 2
##  $ time      : Factor w/ 5 levels "12:36","14:22",..: 4 3 5 2 1
```

Notice that the function `data.frame()` has interpreted the date and patient.ID variables as Factors! This is an issue if we are going to try to summarize these results by date or by patient ID. To change these results to character variables, we can use `as.character()` or just tell R that we want to import them as characters to begin with.

```
df <- data.frame(patient.ID, result, date, time, stringsAsFactors = FALSE)
str(df)
```

```
## 'data.frame':    5 obs. of  4 variables:
##  $ patient.ID: chr  "A" "B" "C" "D" ...
##  $ result    : num  5 6 4 7 5
##  $ date      : chr  "June 28, 2019" "June 29, 2019" "June 30, 2019" "July 1, 2019" ...
##  $ time      : chr  "18:45" "16:36" "7:30" "14:22" ...
```

Okay, now we have something to work with. From here we can change the formats for each variable as necessary.

```
df$patient.ID <- as.character(df$patient.ID) # Change to a character variable
df$result <- as.numeric(df$result) # This was already numeric, but I wanted to show you
```

Dates are a little more complicated . . .

1.2 A sign of the times:`lubridate()`

The `lubridate()` package allows us to deal with dates and times and do algebra on them as we would with other vectors. This represents a major advantage over the handling of dates using base R packages.

Lubridate makes logical assumptions about what you probably mean based on typical date formats.

The functions that lubridate uses are `mdy()`, `ymd()` and `dmy()`. They have very predictable behavior.

```
library(lubridate)

mdy("Aug-20,1755")
```

```
## [1] "1755-08-20"
```

```
mdy("Aug/20/1755")
```

```
## [1] "1755-08-20"
```

```
mdy("08-20-1755")
```

```
## [1] "1755-08-20"
```

```
mdy("08201755")
```

```
## [1] "1755-08-20"
```

```
mdy("August 20, 1755")
```

```
## [1] "1755-08-20"
```

```
mdy("August 20 1755")
```

```
## [1] "1755-08-20"
```

```
#all work perfectly without any explicit statements about format.
```

The real gold with lubridate is how it can handle times too!

```
#calculating the number of days since the Declaration of Independence
then <- mdy_hms("July 04,  1776 14:32:45")
then
```

```
## [1] "1776-07-04 14:32:45 UTC"
```

```
now <- ymd_hms(Sys.time())
now
```

```
## [1] "2019-07-24 16:51:02 UTC"
```

```
delta <- difftime(now, then, units = "days")
delta #wow
```

```
## Time difference of 88773.1 days
```

```
#calculating the number of days Canada has been a country
then <- mdy_hms("July 01,  1867 11:17:21")
then
```

```
## [1] "1867-07-01 11:17:21 UTC"
```

```
now <- ymd_hms(Sys.time())
now
```

```
## [1] "2019-07-24 16:51:02 UTC"
```

```
delta <- difftime(now, then, units = "days")
delta
```

```
## Time difference of 55540.23 days
```

Now lets apply the `dmy()` and `dmy_hm()` formats to our `df` dataframe.

```
df$date <- mdy(df$date) # note that I had to change "dmy()" to "mdy()".
```

It's fairly common for dates and times to not be listed in the same column. We can deal with that in a dataframe by using `paste()`. This function is fairly intuitive, then we can apply the `dmy_hm()` function to format the new column.

```
# paste the date and time using a space as a separator into a new
# column called "date.time" in the "df" dataframe.
date.time <- paste(date, time, sep = " ")
df <- data.frame(patient.ID, result, date.time, stringsAsFactors = FALSE)
```

```
df$date.time <- mdy_hm(date.time)
str(df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ patient.ID: chr  "A" "B" "C" "D" ...
##  $ result    : num  5 6 4 7 5
##  $ date.time : POSIXct, format: "2019-06-28 18:45:00" "2019-06-29 16:36:00" ...
```

1.2.1 Exercise

1. Read the file Potassium.csv into a variable called `K.data`. This file contains real K^+ data extracted from SunQuest from Dan's lab for one month from the two ER bays. It contains order time, collection time, receive time and result time. Use the `head()` function to get an idea of how the dates and times are formatted. All the K^+ results presented were run on an ABL800 whole blood analyzer directly from an unspun PST tube.
   - Convert the order, receive, and result times into dates
   - Calculate the order-to-result times and store it a column called TAT ("turn around time"). Use the function `difftime()` to calculate the time difference.
     - What is the median and IQR of the TAT?
     - What is the 90th and 99th percentiles of the TAT?
     - What is the maximum value of TAT?
   - Calculate the receive-to-result times and store it a column called lab.TAT (the analysis time).
     - What is the median and IQR of the lab.TAT?
     - What is the 90th and 99th percentiles of the lab.TAT?
     - What is the maximum value of lab.TAT? What day did it occur?
     - What strange finding have you discovered?

1.3 Pulling the Strings

1.3.1 Manipulating data with grep(), gsub() and regular expressions

In programming, the word 'string' refers to anything treated as text. Strings may be one word or several words, and can include other characters as well. For example, the word "computer" is a string, as is the sentence "I have 4 computers." There are an array of tools in R specifically for working with strings. In general lab data, a string may be a sample identifier, a patient name or hospital admission number, a gender, or a comment on a sample result. We can use strings to build a structure from which to extract information.

Here we have a list of Sample Identifiers where the prefix "C" indicates a chemistry sample while "H" indicates a hematology sample and "Z" indicates a

QC material :

```
sample.ids <- c("C001", "C002", "H001", "H002", "ZC001", "ZC002", "ZH001", "ZH002")
```

identify which samples correspond to QC Samples using grep().

```
library(stringr)
grepl("Z", sample.ids)
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
# returns a TRUE or FALSE to the question "does the string contain the pattern ZC?"
grep("Z", sample.ids)
```

```
## [1] 5 6 7 8
```

```
# returns the location of strings that contain the patern ZC
```

We often want to filter out only the QC or patient data from our datasets, so lets look at how to rename them using `gsub()`

```
gsub("Z", "QC", sample.ids) #what did this do?
```

```
## [1] "C001"   "C002"   "H001"   "H002"   "QCC001" "QCC002" "QCH001" "QCH002"
```

```
gsub("ZC001", "QC", sample.ids) #okay, now we're getting somewhere
```

```
## [1] "C001"  "C002"  "H001"  "H002"  "QC"    "ZC002" "ZH001" "ZH002"
```

This is great, but replacing each string one by one is a little silly. What if we can look for the pattern "Z" followed by any number of other characters and replace them all with "QC"? You can. It's called a "Regular Expression", and just to show you the magic:

```
gsub("Z.*", "QC", sample.ids)
```

```
## [1] "C001" "C002" "H001" "H002" "QC"   "QC"   "QC"   "QC"
```

Included in this course package is a "Cheat Sheet" which shows you all of the regular expressions and what their uses are. In this case the "." represents "Any Character" while the "*" symbol represents "matches at least 0 times". So in essence, the above script is saying to match any string that contains the pattern "Z" and has any number of characters following it.

Other useful notations include "^" meaning "begins with,"|" meaning "or", and "&" meaning "and". So few other ways to do the same thing we already did:

```
gsub("^Z.*", "QC", sample.ids) # begins with "Z" and has text after
```

```
## [1] "C001" "C002" "H001" "H002" "QC"   "QC"   "QC"   "QC"
```

```
# useful to restrict the pattern to look only at strings that start with Z
gsub("ZC.*|ZH.*", "QC", sample.ids) #change ZC or ZH containing string to "QC"
```

```
## [1] "C001" "C002" "H001" "H002" "QC"   "QC"   "QC"   "QC"
```

if you want to specify Chemistry or Hematology QC, you will need to do multiple replacements and define the varaible each time.

```
sample.ids <- gsub("ZC.*", "Chemistry QC", sample.ids)
sample.ids <- gsub("ZH.*", "Hematology QC", sample.ids)
sample.ids
```

```
## [1] "C001"          "C002"          "H001"          "H002"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

Now to identify patient samples as well. Samples starting with "C" are chemistry specimens and those starting with "H" are hematology, then each letter is followed by some series of numbers from 0 to 9. The regular expression for this would be to use "" which means "one of".

```
# works if there is always a zero after the "C" in the patient result
gsub("^C0.*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "H001"          "H002"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
#works for any number after the "C"
gsub("^C[0-9].*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "H001"          "H002"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
# now we'll include an "or" so we can convert chem and heme samples to "Patient"
gsub("^C[0-9].*|^H[0-9].*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "Patient"       "Patient"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
# side note that this works using letters as well as numbers
gsub("^[CH]0.*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "Patient"       "Patient"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
# this works because your string
# 1) "^" starts with,
# 2) "one of C or H",
# 3) followed by a zero and
# 4) "." any character, 5) which appears "*" at least 0 times

gsub("^[A-Z]0.*", "Patient", sample.ids)
```

```
## [1] "Patient"       "Patient"       "Patient"       "Patient"
## [5] "Chemistry QC"  "Chemistry QC"  "Hematology QC" "Hematology QC"
```

```
# this will work with any letters from A to Z
```

1.4 Coping with Non-numeric Data

Now that we know how to fix our data, we will want to process the numeric data. To do so, lets use another dataset. This is a made-up QC dataset containing data for liver panel tests Albumin, ALT, Ammonia, Total Bilirubin, and Direct Bilirubin. There are 8 columns including identifiers for Test, Specimen ID, Test Site, QC Mnemonic, QC Lot number, Analyzer Name, Result, and Date/time of result. There are 7 days of QC data in here for three sites from six analyzers, so we're looking at lots of QC data. R can handle much much more than that, but this is a good start.

First, use `read.csv()` to import the data file "QCData_Jun2019.csv" to a dataframe called `qc.data`. Start by surveying the data using `head()` and `str()`

```
qc.data <- read.csv(file = "Data_Files/QCData_Jun2019.csv", sep = ",")
str(qc.data)
```

```
## 'data.frame':    570 obs. of  8 variables:
##  $ Test       : Factor w/ 5 levels "ALB","ALT","AMM",..: 5 5 5 5 5 5 5 5 5 5 ...
##  $ Spec_Num   : Factor w/ 277 levels "0106:C00046Q",..: 1 2 3 6 8 9 10 11 12 15 ...
##  $ Site       : Factor w/ 3 levels "A","B","C": 1 1 1 3 3 3 2 2 2 2 ...
##  $ QC_Mnemonic: Factor w/ 7 levels "AMML1","BIL3",..: 7 5 6 2 6 5 7 5 6 7 ...
##  $ QC_Lot     : Factor w/ 7 levels "31851","31852",..: 5 1 2 5 2 1 6 1 2 7 ...
##  $ Analyzer   : Factor w/ 6 levels "Byfuglien","Conner",..: 3 3 3 2 2 2 6 5 5 5 ...
##  $ Result     : Factor w/ 308 levels "105.4","105.7",..: 143 306 253 182 294 80 164 107 3
##  $ Result_Date: Factor w/ 232 levels "2018-06-01 10:41",..: 7 8 9 10 12 13 14 15 16 17 ..
```

```
# notice all of the columns are being imported as Factors we didn't set
# stringsAsFactors to FALSE

head(qc.data)
```

```
##   Test     Spec_Num Site QC_Mnemonic  QC_Lot   Analyzer Result
## 1 BILT 0106:C00046Q    A      CMBIL3 BC18093 Hellebuyck  363.4
## 2 BILT 0106:C00064Q    A      CBLIQ1   31851 Hellebuyck    ERR
## 3 BILT 0106:C00066Q    A      CBLIQ2   31852 Hellebuyck   78.7
## 4 BILT 0106:C00109Q    C        BIL3 BC18093     Conner  387.7
## 5 BILT 0106:C00116Q    C      CBLIQ2   31852     Conner   90.8
## 6 BILT 0106:C00120Q    C      CBLIQ1   31851     Conner   16.8
##       Result_Date
## 1 2018-06-01 2:36
## 2 2018-06-01 3:02
## 3 2018-06-01 3:05
## 4 2018-06-01 7:06
```

```
## 5 2018-06-01 7:14
## 6 2018-06-01 7:15
```

```
# gives you an idea of the type of data you're dealing with
```

Okay, now we just have to format the columns as we wish.

```
as.numeric(qc.data$Result)
```

Wait–what just happened here? This makes no sense at all. These are all integers between 1 and 308. Do you see what has happened?

R's default handling of converting of a factor variable to a numeric variable is to convert it to the number of the factor (1 through 308 in this case). We personally find this irritating, but it is evidently working as intended.

So the correct thing to do is:

```
as.numeric(as.character(qc.data$Result))
```

```
## Warning: NAs introduced by coercion
```

The default R behaviour of treating strings as factors can be over–ridden when the csv file is read by specifying `stringsAsFactors = FALSE`, same as the `data.frame()` function we used before. When one does this, the `as.character` part of the expression above is unnecessary.

```
qc.data <- read.csv(file = "Data_Files/QCData_Jun2019.csv", stringsAsFactors = FALSE)
# re-read the data using stringAsFactors = FALSE
qc.data$Result <- as.numeric(qc.data$Result)
```

```
## Warning: NAs introduced by coercion
```

```
str(qc.data) # results are numeric
```

```
## 'data.frame':    570 obs. of  8 variables:
##  $ Test       : chr  "BILT" "BILT" "BILT" "BILT" ...
##  $ Spec_Num   : chr  "0106:C00046Q" "0106:C00064Q" "0106:C00066Q" "0106:C00109Q" ...
##  $ Site       : chr  "A" "A" "A" "C" ...
##  $ QC_Mnemonic: chr  "CMBIL3" "CBLIQ1" "CBLIQ2" "BIL3" ...
##  $ QC_Lot     : chr  "BC18093" "31851" "31852" "BC18093" ...
##  $ Analyzer   : chr  "Hellebuyck" "Hellebuyck" "Hellebuyck" "Conner" ...
##  $ Result     : num  363.4 NA 78.7 387.7 90.8 ...
##  $ Result_Date: chr  "2018-06-01 2:36" "2018-06-01 3:02" "2018-06-01 3:05" "2018-06-01 7:
```

```
head(qc.data) # data looks good
```

```
##   Test     Spec_Num Site QC_Mnemonic  QC_Lot   Analyzer Result
## 1 BILT 0106:C00046Q    A      CMBIL3 BC18093 Hellebuyck  363.4
## 2 BILT 0106:C00064Q    A      CBLIQ1   31851 Hellebuyck     NA
## 3 BILT 0106:C00066Q    A      CBLIQ2   31852 Hellebuyck   78.7
```

```
## 4 BILT 0106:C00109Q    C        BIL3 BC18093     Conner  387.7
## 5 BILT 0106:C00116Q    C      CBLIQ2   31852     Conner   90.8
## 6 BILT 0106:C00120Q    C      CBLIQ1   31851     Conner   16.8
##       Result_Date
## 1 2018-06-01 2:36
## 2 2018-06-01 3:02
## 3 2018-06-01 3:05
## 4 2018-06-01 7:06
## 5 2018-06-01 7:14
## 6 2018-06-01 7:15
```

Something else somewhat surprising has happened. What do you notice? See the NA? This is what all non-numerics become, even if one is < 40 and another is > 200 ("NA" stands for "not available"). In this case, some text is present within the QC data. So, be careful. If we want to preserve something of what was actually in the data file, we will need another approach. If we are happy to have all non-numerics display NAs, then what we have here is fine.

But if we now want to look at the statistics of the Results column, the NA results cannot contribute. It would be good to figure out what is going on. In this case, all we want to do is remove the lines of data that are non-numeric because they contain nonsense information, but keep in mind that this may not be the case if you are looking at $>$ or $<$ values in your patient data. You can use `gsub()` to remove or replace these symbols with whatever you desire.

Identify rows of data that contain NA using `is.na()`

```
is.na(qc.data$Result)
# this is a logical variable that asks "is this value NA?" for each row of data.
# Neat, but not useful here.
```

This uses the function `is.na()` to generate a dataframe containing lines that have NA in the Result column. It's a good idea to use this to check what data you're removing before you actually do it.

```
# there are four values in the dataset that are now NA, corresponding to
# specific sample ID's
qc.data[is.na(qc.data$Result),]
```

```
##     Test     Spec_Num Site QC_Mnemonic QC_Lot   Analyzer Result
## 2   BILT 0106:C00064Q    A      CBLIQ1  31851 Hellebuyck     NA
## 9   BILT 0106:C00183Q    B      CBLIQ2  31852  Scheifele     NA
## 503  ALB 0306:C00242Q    A      CBLIQ1  31851  Byfuglien     NA
## 534  ALB 0506:C00118Q    C      CBLIQ1  31851     Conner     NA
##         Result_Date
## 2   2018-06-01 3:02
## 9   2018-06-01 8:13
## 503 2018-06-03 8:51
## 534 2018-06-05 7:11
```

So what we want is to use an ! (represents "is not") ahead of the function `is.na()` to result the rows of data that "are not NA"

```
num.qcdata <- qc.data[!is.na(qc.data$Result),]
head(num.qcdata)
```

```
##   Test     Spec_Num Site QC_Mnemonic   QC_Lot   Analyzer Result
## 1 BILT 0106:C00046Q    A      CMBIL3  BC18093 Hellebuyck  363.4
## 3 BILT 0106:C00066Q    A      CBLIQ2    31852 Hellebuyck   78.7
## 4 BILT 0106:C00109Q    C        BIL3  BC18093     Conner  387.7
## 5 BILT 0106:C00116Q    C      CBLIQ2    31852     Conner   90.8
## 6 BILT 0106:C00120Q    C      CBLIQ1    31851     Conner   16.8
## 7 BILT 0106:C00127Q    B      CMBIL3 BC18093A    Wheeler  376.3
##       Result_Date
## 1 2018-06-01 2:36
## 3 2018-06-01 3:05
## 4 2018-06-01 7:06
## 5 2018-06-01 7:14
## 6 2018-06-01 7:15
## 7 2018-06-01 7:18
```

```
str(num.qcdata) # perfect, no more NAs!
```

```
## 'data.frame':    566 obs. of  8 variables:
##  $ Test       : chr  "BILT" "BILT" "BILT" "BILT" ...
##  $ Spec_Num   : chr  "0106:C00046Q" "0106:C00066Q" "0106:C00109Q" "0106:C00116Q" ...
##  $ Site       : chr  "A" "A" "C" "C" ...
##  $ QC_Mnemonic: chr  "CMBIL3" "CBLIQ2" "BIL3" "CBLIQ2" ...
##  $ QC_Lot     : chr  "BC18093" "31852" "BC18093" "31852" ...
##  $ Analyzer   : chr  "Hellebuyck" "Hellebuyck" "Conner" "Conner" ...
##  $ Result     : num  363.4 78.7 387.7 90.8 16.8 ...
##  $ Result_Date: chr  "2018-06-01 2:36" "2018-06-01 3:05" "2018-06-01 7:06" "2018-06-01 7:
```

1.4.1 Exercise

Let's apply what we just learned using `gsub()` our QC data set. * Replace the short names in the "Test" column "ALB", "ALT", "AMM", "BILD", and "BILT" with their full names. * Format the Result_Date column as a date using `lubridate()` as we did before.

2 Lesson 4: Meet the 'tidyverse' - Gather, Join, Filter, and Clean

There is a paradigm of R programming referred to as the "tidyverse" that is particularly useful for certain types of data summary and data visualization. It allows for rapid-high level commands accomplishing a great deal in few lines of highly readable code. The challenge of using tidyverse packages is that they are under very rapid development and this can mean that updates in the packages can cause your code to stop functioning. Additionally, many statistical packages use traditional "base R" and they may not cooperate with tidyverse output. However, in the context of our work in health care, we are usually doing fairly simple one-off reports for research or answering a specific question so using the tidyverse makes sense.

2.1 Packages

Before we can start using these functions we will have to install and load them. This is because the 'tidyverse' does not exist in 'base' R (that is, the pre-loaded set of function which are installed when you install the R program). Rather, the tidyverse is a set of packages. Packages are themselves sets of functions which have been written by whomever authoured the package. These user-created functions have been 'packaged' and made available for anyone to download and use.

Packages are what makes R great. Lots can be done in base R but often to acheive what you want, you end up having to write your own functions. This is fine if you like programming and find coding your own functions enjoyable. Sometimes, though, we just want to 'get it done' - and whatever 'it' is that you are trying to do, chances are that someone out there has done it before and written a package for you to make it easier.

A final word about packages - they are only as good as the person who wrote them. The tidyverse is a group of packages which have been written by Hadley Wickham (who created RStudio) and his team - which means that we can rely on them being functional, updated and usually bug-free. The same cannot always be said of smaller packages, so if you are ever poking around for a package to fill a niche need, read the documentation and do some google searches to see how others have been able to work with the package before spending too much time trying to code with it.

2.1.1 Exercise

- Hopefully you installed the tidyverse when you first installed R as per the pre-course instructions. Just in case, the following code will install the

tidyverse package if it is not already there.

```
if("tidyverse" %in% rownames(installed.packages()) == FALSE) {install.packages("tidyverse")}
```

This command asks R to download the most current version of the tidyverse from CRAN, which is an online package repository. Note that you need to be connected to the internet for this to work.

Since quite a few packages are being downloaded, this may take a few minutes. Perhaps a good time for a coffee break.

Once tidyverse is installed, we need to load it by calling library():

```
library(tidyverse)
```

You only need to install a package once, but you will need to load any packages you need each time you restart R.

2.2 gather() and spread()

Now that we are in the tidyverse, the first thing we want to introduce is what is meant by "tidy data".

When humans prepare spreadsheet data, they typically have each row represent all the observations on single subject. This is usually pretty easy to look at but it happens to have a number of disadvantages from a statistical programming standpoint.

For example, in traditional "untidy" data, you might have each subject on a row and all of their lab tests represented as columns: Sodium, Potassium, Chloride, Bicarbonate, Creatinine, pH, Troponin etc. But in the tidy data paradigm, all of the blood tests should be factors under a single column labelled "Test" and then each row represents a single observation, not a single patient.

It is frequently necessary to jump back and forth between the traditional view, which we call "data wide" and the tidy view which we call "data long". This is accomplished with the functions `gather()` and `spread()`.

Let's starting exploring these funtions by reading in some data on "anthropometric" (for want of a better term) measurements on opossum.

```
possum <- read_csv("Data_Files/possum.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   X1 = col_character(),
##   case = col_double(),
##   site = col_double(),
```

```
##   Pop = col_character(),
##   sex = col_character(),
##   age = col_double(),
##   hdlngth = col_double(),
##   skullw = col_double(),
##   totlngth = col_double(),
##   taill = col_double(),
##   footlgth = col_double(),
##   earconch = col_double(),
##   eye = col_double(),
##   chest = col_double(),
##   belly = col_double()
## )
```

```
# we used read_csv here (a tidyverse function) instead of read.csv (base R)
# the anthropomorphic measurements of the possums are wide
head(possum)
```

```
## # A tibble: 6 x 15
##   X1     case  site Pop   sex     age hdlngth skullw totlngth taill
##   <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>
## 1 C3        1     1 Vic   m         8    94.1   60.4     89    36
## 2 C5        2     1 Vic   f         6    92.5   57.6     91.5  36.5
## 3 C10       3     1 Vic   f         6    94     60       95.5  39
## 4 C15       4     1 Vic   f         6    93.2   57.1     92    38
## 5 C23       5     1 Vic   f         2    91.5   56.3     85.5  36
## 6 C24       6     1 Vic   f         1    93.1   54.8     90.5  35.5
## # ... with 5 more variables: footlgth <dbl>, earconch <dbl>, eye <dbl>,
## #   chest <dbl>, belly <dbl>
```

We can convert them to data long format with the `gather()` function. In the this function we must determine three things:

1. What columns are to be gathered together as factors of a similar type?– This parameter referred to as the "key" and in our output the column will be named "Possum_Metric".
2. What do you want to call the column that contains the associated values?– This parameter is referred to as the "value" and in our output the column will be named "Result".
3. Which columns (by name or number) are to be gathered?–In this case it is columns 7–15. You can also denote which columns you *don't* want gathered – so we could also write this as "-c(1:6)".

```
possum.long <- gather(possum, key = Possum_Metric, value = Result, 7:15)
head(possum.long,10)
```

```
## # A tibble: 10 x 8
##    X1     case  site Pop   sex     age Possum_Metric Result
```

```
##    <chr> <dbl> <dbl> <chr> <chr> <dbl> <chr>          <dbl>
##  1 C3        1     1 Vic   m         8 hdlngth         94.1
##  2 C5        2     1 Vic   f         6 hdlngth         92.5
##  3 C10       3     1 Vic   f         6 hdlngth         94
##  4 C15       4     1 Vic   f         6 hdlngth         93.2
##  5 C23       5     1 Vic   f         2 hdlngth         91.5
##  6 C24       6     1 Vic   f         1 hdlngth         93.1
##  7 C26       7     1 Vic   m         2 hdlngth         95.3
##  8 C27       8     1 Vic   f         6 hdlngth         94.8
##  9 C28       9     1 Vic   f         9 hdlngth         93.4
## 10 C31      10     1 Vic   f         6 hdlngth         91.8
```

As it turns out, this permits very slick calculations which allow you to rapidly generate summary statistics based on variables of your choosing.

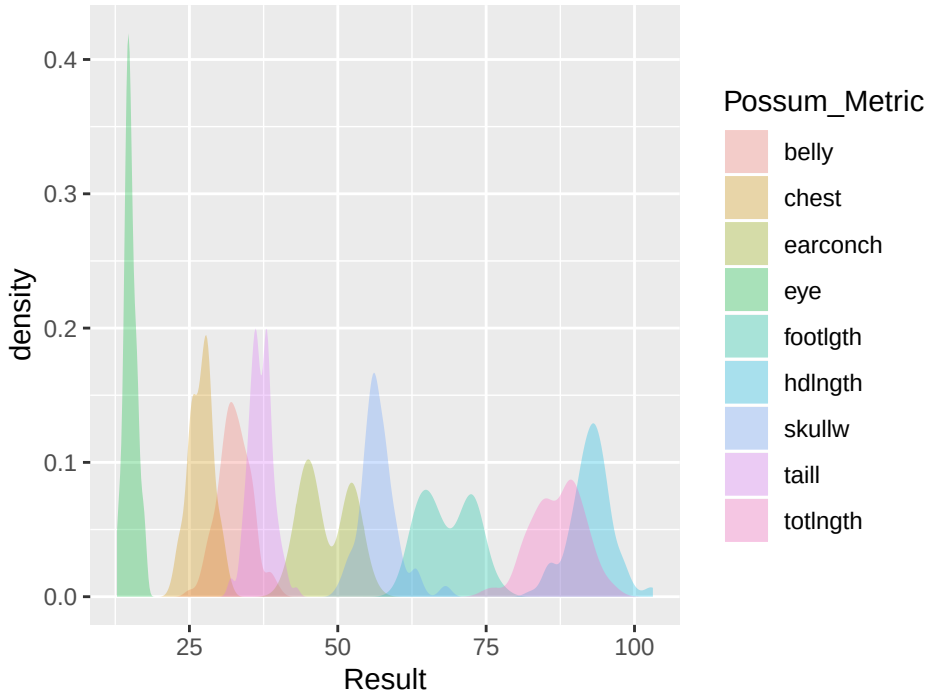
```
#long data permits rapid calculations
group_by(.data = possum.long, key = Possum_Metric) %>%
  summarise(Mean = mean(Result, na.rm = TRUE),
            SD = sd(Result, na.rm = TRUE),
            "%CV" = (100*sd(Result, na.rm = TRUE)/mean(Result, na.rm = TRUE)))
```

```
## # A tibble: 9 x 4
##   key       Mean    SD `%CV`
##   <chr>    <dbl> <dbl> <dbl>
## 1 belly     32.6  2.76  8.48
## 2 chest     27    2.05  7.58
## 3 earconch  48.1  4.11  8.54
## 4 eye       15.0  1.05  6.98
## 5 footlgth  68.5  4.40  6.42
## 6 hdlngth   92.6  3.57  3.86
## 7 skullw    56.9  3.11  5.47
## 8 taill     37.0  1.96  5.29
## 9 totlngth  87.1  4.31  4.95
```

and plots:

```
#long data permits rapid visualizations (which we will cover later)
library(ggplot2)
p <- ggplot(possum.long, aes(x = Result, fill = Possum_Metric)) +
  geom_density(alpha = 0.3, color = NA)
p
```

```
## Warning: Removed 1 rows containing non-finite values (stat_density).
```



If we want to convert back to 'untidy' data, this is accomplished by `spread()`. For this function again we need to determine our "key" and "value" columns:

1. The "key" column will be "spread" across the top of the table (this column becomes the column names).
2. The "value" column contains the values we want to populate these new columns.

```
possum.wide <- spread(possum.long, key = Possum_Metric, value = Result)
head(possum.wide, 10)
```

```
## # A tibble: 10 x 15
##    X1     case  site Pop   sex     age belly chest earconch   eye footlgth
##    <chr> <dbl> <dbl> <chr> <chr> <dbl> <dbl> <dbl>    <dbl> <dbl>    <dbl>
##  1 A1       29     1 Vic   f         3  32    24       51.8  14       74.9
##  2 A2       30     1 Vic   f         2  33    24.5     50.8  14.5     70.6
##  3 A3       31     1 Vic   m         3  31    27       52.5  14.5     68
##  4 A4       32     1 Vic   f         4  34    28       52    14.9     74.8
##  5 AD1      33     1 Vic   m         3  30    24       51.8  14.8     70.8
##  6 BB13     35     2 Vic   m         4  35.5  28       55.5  16.4     71.2
##  7 BB15     36     2 Vic   m         7  36    25.5     52    14.9     74.3
##  8 BB17     37     2 Vic   f         2  31.5  28       52    13.6     71.2
##  9 BB25     38     2 Vic   m         7  30    27       49.5  15.9     68.4
## 10 BB31     39     2 Vic   f         1  25    25       53.4  13       68.7
## # ... with 4 more variables: hdlngth <dbl>, skullw <dbl>, taill <dbl>,
```

```
## #   totlngth <dbl>
```

2.2.1 Exercise

Using the `num.qc` dataset you generated earlier, summarise the QC data running means and sds according to test, qc mnemonic, and site to a new dataframe called `qc.means`.

2.3 `arrange()`, `filter()`, and `select()`

There are some more very useful and simple functions in the tidyverse which we will need before we get too much further.

The first is `arrange()` and it does pretty much what you think it would. Let's try it out on the QC data:

```
arrange(num.qcdata, Site)
```

If we want to arrange from largest to smallest, we would ask for the ages in descending `desc()` order:

```
arrange(num.qcdata, desc(Site))
```

What happens if we `arrange()` something that is not a character?

```
arrange(num.qcdata, Result) # numeric variable
arrange(num.qcdata, Result_Date) # date variable
```

Next up, `filter()` and `select()`. Filtering means choosing *rows* while selecting means choosing *columns*. You can filter rows based on what the rows contain, but you can only select columns by name or position.

```
filter(num.qcdata, Site == "B") # all results from site B
filter(num.qcdata, Result < 10) # all results less than 10

# if you want to filter by multiple criteria, simply join the
# filter criteria using an & symbol
filter(num.qcdata, Site == "A" & Test == "AMM" & QC_Mnemonic == "CBALC1")

# can also filter by date range
filter(num.qcdata, Result_Date >= "2018-06-01" & Result_Date < "2018-06-04")
```

```
select(num.qcdata, Test, Site, Result) # returns the named columns
select(num.qcdata, 1:7) # returns columns 1 to 7
```

2.3.1 Exercise

- From the num.qcdata data, create a table which only has the specimen number, site and result for Albumin CBLIQ1 QC run on June 3, 2018. Arrange this table by site.

2.4 Acknowledgements

- Dan Holmes, Stephen Master, Will Slade & Janet Simons's Intro to R Workshop