Data Wrangling

Joseph Nathan Cohen

2/1/2020

Contents

About Data Wrangling **2**
- What is Data Wrangling? 2
- Steps Involved 2
- Starting Our Session 2

Loading the Data **2**
- CSV Files 3
- Excel Spreadsheets 3
- From Another Analysis Program (Stata, SAS, SPSS) 3
- Pulling Data from the Web 4

Tidying the Data **4**
- What is "Tidy" Data 4
- Strategize Your Tidying Operations 5
- gather() 7
- spread() 8

Labeling **9**

Recoding **9**
- Rescaling Values 9
- Changing Values Based on Logical Tests 10
- Creating New Variables from Existing Data 10
- Creating Categories from Continuous Data 11
- When a Character Intrudes into Numeric Data 11

Reducing Data Tables **12**
- Dropping Variables 13
- Dropping Observations 14

Saving **14**

About Data Wrangling

What is Data Wrangling?

Data wrangling (a.k.a. "munging") is the process of preparing data for an analysis. Most data sets are messy and require preparation for analysis. Improperly cleaned data can distort your results. Many of the techniques that we will learn this year are premised on your data being organized according to the guidelines presented below.

Side note: Although wrangling is only one week of our semester, it typically takes up a considerably larger proportion of your analysis.

Steps Involved

We will assume that you have already acquired your data. Also, if you are using a secondary data set (i.e., one that you did not collect yourself), I will assume that you already read the codebook.

Once you have your data on hand, the wrangling process involve many kinds of operations

- Loading
- Reshaping / Tidying
- Labeling
- Recoding
- Trimming / Subsetting
- Saving
- Merging

These notes provide an overview of these steps.

Starting Our Session

Recall these lines of code to start your R session:

```
rm(list=ls())
gc()

directory <- "E:/Dropbox/Teaching/Data 712/05"
setwd(directory)
```

Let's get started.

Loading the Data

Most of you will probably be working with data that has been partly wrangled into a spreadsheet, comma-separated values file, or text file. It is also possible to download data through direct queries of online databases.

CSV Files

To import data from comma-separated values files, use the **read.csv()** command. Below, I am calling up sample murder data, which can be downloaded from the class Slack or Blackboard pages. Note that this is fake data.

```
homicides <- read.csv("Sample Homicide Data.csv")
homicides
```

```
##   Year Happy.City Funburgh Sadville Angry.Town Rageopolis
## 1 2000        688      284        7         11         45
## 2 2005        731      250        8         13         73
## 3 2010        733      357        8         13         33
## 4 2015       1200      388        8         15         45
## 5 2020       1929      434        8         17         61
```

Note that there is also a **write_csv()** command to write an object to a comma-separated values file:

```
write.csv(homicides, file = "Murder Data.csv")
```

Excel Spreadsheets

To import data from Excel to R, you can use the **read_xlsx()** command from the ***readxl*** package. Look at the Excel workbook first – you will find the data is on the second sheet:

```
library(readxl)  #Remember to load the library
population <- read_xlsx("Sample Population Data.xlsx", sheet = 2)
population
```

```
## # A tibble: 5 x 6
##   city        `2000` `2005`  `2010`  `2015`  `2020`
##   <chr>        <dbl>  <dbl>   <dbl>   <dbl>   <dbl>
## 1 Happy City 825000 950000 1100000 1200000 1350000
## 2 Funburgh   625000 700000  750000  775000  825000
## 3 Sadville    25000  26000   24000   26000   28000
## 4 Angry Town 125000 155000  165000  170000  165000
## 5 Rageopolis    -99  55000   80000  110000 1750000
```

Sure, the data is not yet clean. So far, we're only loading data into memory so that we can clean it.

From Another Analysis Program (Stata, SAS, SPSS)

These data can be read using commands from the ***foreign*** package. I call these objects "mydata", but remember that you can name them anything.

```
library(foreign)

#SAS transport
my.data <- read.xport("My SAS Data File.xport")

#Stata
my.data <- read.dta("My Stata Data dta")

#SPSS
my.data <- read.spss("My SPSS Data spss")
```

We won't be using these data here. They are presented here only for your future reference.

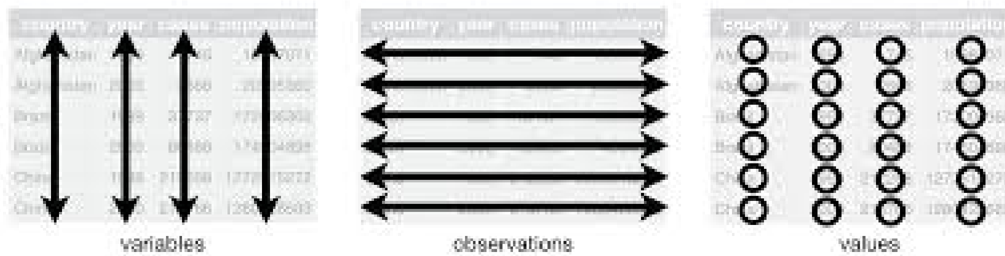

Figure 1: "Tidy Data Visualized"

Pulling Data from the Web

html Scraping

This involves downloading an html table from a web page. For a tutorial on this topic using the ***rvest*** package, visit https://blog.rstudio.org/2014/11/24/rvest-easy-web-scraping-with-r/

APIs

APIs are *Application Program Interfaces*, a generic term for software routines that allow computers to interface with each other. In practice, data scientists often use APIs to fetch data over the web. These routines are a bit more complicated. For a tutorial that uses the ***httr*** and ***jstonlite*** pacakges, see https://www.r-bloggers.com/accessing-apis-from-r-and-a-little-r-programming/

Want to Learn More?

If you wish to take a deeper dive into this topic, why not try the DataCamp course Intermediate Importing Data in R?

Tidying the Data

What is "Tidy" Data

At this step, you will "tidy" your data. Most of the methods we will study assume tidy data. "Tidy" data is a standard format for data storage in which:

- Rows correspond to subjects / units of analysis
- Columns correspond to variables

Figure 1 (below) depicts tidy data.

In circumstances in which you are dealing with longitudinal data, in which the same subject is measured over time, we treat time as another variable. I will illustrate this way of organizing data below. The data stored above in the objects **homicides** and **population** are *not* tidy.

Strategize Your Tidying Operations

Your first step is to make sense of how the data is organized in your raw set, and develop a sense of which reshaping operations are necessary to render tidy data. Take our data from the object **population**:

```
population
```

```
## # A tibble: 5 x 6
##   city       `2000` `2005`  `2010`  `2015`  `2020`
##   <chr>       <dbl>  <dbl>   <dbl>   <dbl>   <dbl>
## 1 Happy City 825000 950000 1100000 1200000 1350000
## 2 Funburgh   625000 700000  750000  775000  825000
## 3 Sadville    25000  26000   24000   26000   28000
## 4 Angry Town 125000 155000  165000  170000  165000
## 5 Rageopolis    -99  55000   80000  110000 1750000
```

I will look at a table like this and think of it like a puzzle. In this case, you can envision splitting the data table and stacking the columns. It's like a puzzle game! Here's a more drawn out illustration of this reshaping operation. As we go through the code, try to imagine splitting a data table up and refitting it to make it tidy.

I'm going to take the -population- object above, and create five objects, each of which has the data covering one year. I keep the city name column for each object, and add the corresponding year as an additional column. The operations that I'm using include:

- **cbind()** which binds two tables with the same number of rows across columns. It's like taking one table and adding it as columns to the right of another table.
- **rbind()**, or "row bind". This command stacks data tables with similar numbers of columns on top of each other
- **rep()** which creates repeated number series

```
#I accomplish the operation using this code:
pop.1 <- cbind( population[,c(1,2)], rep(2000,5))

#The resulting data object looks like this:
pop.1
```

```
##         city   2000 rep(2000, 5)
## 1 Happy City 825000         2000
## 2   Funburgh 625000         2000
## 3   Sadville  25000         2000
## 4 Angry Town 125000         2000
## 5 Rageopolis    -99         2000
```

```
#Do it for the other columns
#The command cbind() binds vectors or tables together by columns
pop.2 <- cbind(population[,c(1,3)], rep(2005,5))
pop.3 <- cbind(population[,c(1,4)], rep(2010,5))
pop.4 <- cbind(population[,c(1,5)], rep(2015,5))
pop.5 <- cbind(population[,c(1,6)], rep(2020,5))
```

I'm going to use a loop to give all of these data objects the same column names. This is necessary to avoid an error with **rbind()**. The commands I use below include:

- **for**, which initiates loops. More on that later.
- **get()** to call a text object as if it were an object name
- **paste()** and **paste0** to concatenate multiple elements into a single, concatenated character set
- **names()** to call out the names of an object's constituent elements (e.g., variable names on a data table)
- **assign()** to assign a value to a name in an environment.

```r
for (i in 1:5){
  temp <- get(paste0("pop.", i))
  names(temp) <- paste(c("city", "population", "year"))
  assign(paste0("pop.", i), temp)
  }
```

We then stack these individual frames using **rbind()**

```r
pop.data <- rbind(pop.1, pop.2, pop.3, pop.4, pop.5)

#Forgive my nit pickiness, but I like my data tables
#to have unit identifiers first and then time variables (if present) second.
pop.data <- pop.data[,c(1,3,2)]
```

And our data is tidy!

```r
pop.data
```

```
##          city year population
## 1  Happy City 2000     825000
## 2    Funburgh 2000     625000
## 3    Sadville 2000      25000
## 4  Angry Town 2000     125000
## 5  Rageopolis 2000        -99
## 6  Happy City 2005     950000
## 7    Funburgh 2005     700000
## 8    Sadville 2005      26000
## 9  Angry Town 2005     155000
## 10 Rageopolis 2005      55000
## 11 Happy City 2010    1100000
## 12   Funburgh 2010     750000
## 13   Sadville 2010      24000
## 14 Angry Town 2010     165000
## 15 Rageopolis 2010      80000
## 16 Happy City 2015    1200000
## 17   Funburgh 2015     775000
## 18   Sadville 2015      26000
## 19 Angry Town 2015     170000
## 20 Rageopolis 2015     110000
## 21 Happy City 2020    1350000
## 22   Funburgh 2020     825000
## 23   Sadville 2020      28000
## 24 Angry Town 2020     165000
## 25 Rageopolis 2020    1750000
```

Now in the above example, my intent was to reshape the data in a way that allowed you to "watch" a data frame be reshaped. Luckily, there are functions in the ***tidyr*** package that make these kinds of operations much easier.

gather()

For data like **population**:

```
population
```

```
## # A tibble: 5 x 6
##   city       `2000` `2005`  `2010`  `2015`  `2020`
##   <chr>       <dbl>  <dbl>   <dbl>   <dbl>   <dbl>
## 1 Happy City 825000 950000 1100000 1200000 1350000
## 2 Funburgh   625000 700000  750000  775000  825000
## 3 Sadville    25000  26000   24000   26000   28000
## 4 Angry Town 125000 155000  165000  170000  165000
## 5 Rageopolis    -99  55000   80000  110000 1750000
```

The **gather()** operation requires that you specify, in this order:

- *data object*, the name of the object with the data
- *key's name*, the name of the variable that captures what is being conveyed in the column labels
- *values' name*, the name of the variable that captures what is being measured in the data table's cells
- *unit variable* demarcated by a minus sign. This should correspond to the variable name in the original data table that contains the unit identifier.

```
library(tidyr) #Don't forget to load the package
pop.tidy <- gather(population, year, population, -city)
pop.tidy
```

```
## # A tibble: 25 x 3
##    city       year  population
##    <chr>      <chr>      <dbl>
##  1 Happy City 2000      825000
##  2 Funburgh   2000      625000
##  3 Sadville   2000       25000
##  4 Angry Town 2000      125000
##  5 Rageopolis 2000         -99
##  6 Happy City 2005      950000
##  7 Funburgh   2005      700000
##  8 Sadville   2005       26000
##  9 Angry Town 2005      155000
## 10 Rageopolis 2005       55000
## # ... with 15 more rows
```

spread()

Sometimes, your units are spread over multiple rows, with different rows representing different variable scores. For example, consider this data table:

```
schools
```

```
##                school      variable year score
## 1  George Washington       students 2015  1233
## 2  George Washington       teachers 2015    94
## 3  George Washington administrators 2015    21
## 4  George Washington       students 2016  1310
## 5  George Washington       teachers 2016    93
## 6  George Washington administrators 2016    23
## 7                 JFK       students 2015   725
## 8                 JFK       teachers 2015    15
## 9                 JFK administrators 2015     9
## 10                JFK       students 2016   771
## 11                JFK       teachers 2016    17
## 12                JFK administrators 2016     8
## 13   Abraham Lincoln       students 2015   301
## 14   Abraham Lincoln       teachers 2015    14
## 15   Abraham Lincoln administrators 2015     9
## 16   Abraham Lincoln       students 2016   291
## 17   Abraham Lincoln       teachers 2016    11
## 18   Abraham Lincoln administrators 2016     6
## 19      Harry Truman       students 2015  3200
## 20      Harry Truman       teachers 2015   130
## 21      Harry Truman administrators 2015    27
## 22      Harry Truman       students 2016  3290
## 23      Harry Truman       teachers 2016   141
## 24      Harry Truman administrators 2016    28
```

We want to tidy the data so that each row represents all of the data associated with one unit-time combination (here, school and year), with a separate column for the student and teacher counts. Input the following options:

- *data object*
- *key column*, the column with the variable labels
- *values column*, column with the values

```
school.tidy <- spread(schools, variable, score)
school.tidy
```

```
##               school year administrators students teachers
## 1   Abraham Lincoln 2015              9      301       14
## 2   Abraham Lincoln 2016              6      291       11
## 3 George Washington 2015             21     1233       94
## 4 George Washington 2016             23     1310       93
## 5      Harry Truman 2015             27     3200      130
## 6      Harry Truman 2016             28     3290      141
## 7               JFK 2015              9      725       15
## 8               JFK 2016              8      771       17
```

Labeling

Labelling involves changing the column names on a data table. We did it above, but present it formally here.

Recall that the **names()** operation calls up the names of an object's elements. For example, with the object **school**:

```
names(schools)
```

```
## [1] "school"   "variable" "year"     "score"
```

If we want to change the column names:

```
names(schools) <- paste(c("schools", "person", "year", "value"))
```

And now the table has new names:

```
head(schools,3)
```

```
##             schools         person year value
## 1 George Washington       students 2015  1233
## 2 George Washington       teachers 2015    94
## 3 George Washington administrators 2015    21
```

To rename a specific variable in a table:

```
names(schools)[2] <- paste("variable")
head(schools, 3)
```

```
##             schools       variable year value
## 1 George Washington       students 2015  1233
## 2 George Washington       teachers 2015    94
## 3 George Washington administrators 2015    21
```

Recoding

Rescaling Values

One task in data management is to recode errors in the data For example, imagine we had a data set measuring the heights and weights of five men. Height is measured in centimeters, and weight is measured in pounds:

```
measures <- data.frame(names=c("Jim", "Billy", "Tom", "Joe", "Tony"),
                       height=c(181, 190, 190, 178, 165),
                       weight=c(179, 145, 230, 156, 5))
measures
```

```
##   names height weight
## 1   Jim    181    179
## 2 Billy    190    145
## 3   Tom    190    230
## 4   Joe    178    156
## 5  Tony    165      5
```

Let's say we want to convert the weight measurement from centimeters to inches. An inch is 2.54 cm. We recode the height mesaure using a simple arithmetic operation:

```
measures$height <- measures$height/2.54
measures
```

```
##   names   height weight
## 1   Jim 71.25984    179
## 2 Billy 74.80315    145
## 3   Tom 74.80315    230
## 4   Joe 70.07874    156
## 5  Tony 64.96063      5
```

Changing Values Based on Logical Tests

The data says Tony is 5 lbs heavy. That is most likely an error. We need to recode that variable as missing.

Let us create a rule that recodes anyone who is less than 60cm tall as missing. We can do that using the **ifelse()** command, which lists a logical test, then the variable's replacement value if the test is true, then the replacement value if it is false:

```
measures$weight <- ifelse(measures$weight < 80, NA, measures$weight)
measures
```

```
##   names   height weight
## 1   Jim 71.25984    179
## 2 Billy 74.80315    145
## 3   Tom 74.80315    230
## 4   Joe 70.07874    156
## 5  Tony 64.96063     NA
```

I use **ifelse()** to recode missing data.

Creating New Variables from Existing Data

We want to calculate our respondents' body mass index (BMI), which is their weight (in kg) over their height (in squared cm).

```
measures$bmi <- (measures$weight * 0.45) / ((measures$height*2.54)/100)^2
measures
```

```
##   names   height weight      bmi
## 1   Jim 71.25984    179 24.58716
## 2 Billy 74.80315    145 18.07479
## 3   Tom 74.80315    230 28.67036
## 4   Joe 70.07874    156 22.15629
## 5  Tony 64.96063     NA       NA
```

Creating Categories from Continuous Data

The **cut()** function can be used to create groupings based on continuous data. For example, if we wanted to classify our group of men into underweight (BMI<18.5) or overweight (BMI>25):

```
measures$weightstatus <- cut(measures$bmi, c(0,18.5,25,99),
                             labels=c("Underweight","Normal Weight","Overweight"))
measures
```

```
##   names   height weight      bmi  weightstatus
## 1   Jim 71.25984    179 24.58716 Normal Weight
## 2 Billy 74.80315    145 18.07479   Underweight
## 3   Tom 74.80315    230 28.67036    Overweight
## 4   Joe 70.07874    156 22.15629 Normal Weight
## 5  Tony 64.96063     NA       NA          <NA>
```

When a Character Intrudes into Numeric Data

Here's a problem that is often a pain for me. When you load a data table into R, the program guesses variable types: numbers, characters, or logical (TRUE/FALSE). Sometimes, miscoded data causes R to misread your data. For example, we load the CSV file "sales_data.csv" attached to this lesson:

```
sales <- read.csv("sales_data.csv")
sales
```

```
##          Team  Sales Profit
## 1    New York 850000  95000
## 2 Philadelphia 455000 100100
## 3      Boston 550000 134000
## 4     Toronto 315000 69300F
## 5    Montreal 275000  63900
```

If you summarize the variable, you find that R interpreted the **sales$Profit** as a factor (a multichotomous variable). One of the numbers in that variable had an alphabetical character.

```
summary(sales)
```

```
##            Team       Sales           Profit
##  Boston      :1   Min.   :275000   100100:1
##  Montreal    :1   1st Qu.:315000   134000:1
##  New York    :1   Median :455000   63900 :1
##  Philadelphia:1   Mean   :489000   69300F:1
##  Toronto     :1   3rd Qu.:550000   95000 :1
##                   Max.   :850000
```

If Data Entry Error

If this is primary data, then the alphabetical character may be a data entry mistake. This may also be the case with secondary data if the codebook does not mention the inclusion of alphabetical variables in numerical variables. Under such circumstances, you might choose to simply recode as missing all entries with alphabetical characters. This is done by:

```
sales$Profit <- as.numeric(as.character(sales$Profit))
sales
```

```
##          Team  Sales Profit
## 1    New York 850000  95000
## 2 Philadelphia 455000 100100
## 3      Boston 550000 134000
## 4     Toronto 315000     NA
## 5    Montreal 275000  63900
```

The data now reads as numeric:

```
summary(sales)
```

```
##           Team       Sales            Profit
##  Boston      :1   Min.   :275000   Min.   : 63900
##  Montreal    :1   1st Qu.:315000   1st Qu.: 87225
##  New York    :1   Median :455000   Median : 97550
##  Philadelphia:1   Mean   :489000   Mean   : 98250
##  Toronto     :1   3rd Qu.:550000   3rd Qu.:108575
##                   Max.   :850000   Max.   :134000
##                                    NA's   :1
```

If It's a Footnote

If the data is a footnote, and you are assured that you can treat the numeric portion of that alphanumeric entry as the true variable value, then you may wish to extricate it by using **gsub()**, a function that replaces characters within a cell. Recall that our issue is a stray "F" in **sales$Profit**:

```
sales <- read.csv("sales_data.csv")
sales$Profit <- as.numeric(gsub("F", "", as.character(sales$Profit)))
summary(sales)
```

```
##           Team       Sales            Profit
##  Boston      :1   Min.   :275000   Min.   : 63900
##  Montreal    :1   1st Qu.:315000   1st Qu.: 69300
##  New York    :1   Median :455000   Median : 95000
##  Philadelphia:1   Mean   :489000   Mean   : 92460
##  Toronto     :1   3rd Qu.:550000   3rd Qu.:100100
##                   Max.   :850000   Max.   :134000
```

Reducing Data Tables

There may be times in which you want to remove data from a table to make it easier to use. *Dropping variables* involves removing entire variables from a data set. *Subsetting* occurs when we remove observations with particular values on a variable.

Dropping Variables

There are two useful ways to trim variables off a data set. I usually do it by column numbers. For example, if I wanted to remove the **sales$Sales** variable from the **sales** set. I can do it by asking R to retain columns:

```
sales[,c(1,3)]
```

```
##          Team Profit
## 1    New York  95000
## 2 Philadelphia 100100
## 3      Boston 134000
## 4     Toronto  69300
## 5    Montreal  63900
```

Or I can ask it to *remove* a column by adding a minus sign before the column number:

```
sales[,-c(2)]
```

```
##          Team Profit
## 1    New York  95000
## 2 Philadelphia 100100
## 3      Boston 134000
## 4     Toronto  69300
## 5    Montreal  63900
```

You can also use the **select()** function from the ***dplyr*** function. With that command, you first list the data object, then column names you want to retain:

```
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.5.3
```

```
select(sales, Team, Profit)
```

```
##          Team Profit
## 1    New York  95000
## 2 Philadelphia 100100
## 3      Boston 134000
## 4     Toronto  69300
## 5    Montreal  63900
```

Or you can use the minus sign to have observations dropped:

```
select(sales, -Sales)
```

```
##          Team Profit
## 1    New York  95000
## 2 Philadelphia 100100
## 3      Boston 134000
## 4     Toronto  69300
## 5    Montreal  63900
```

Dropping Observations

In situations in which you want to remove observations from a data set, use the **subset()** operation. In this command, you specify the data frame you want to use, and the criteria for *keeping* an observation in the subset. So, for example, to keep only observations with profits over $90,000 from the object **sales**:

```
subset(sales, Profit > 90000)
```

```
##          Team  Sales Profit
## 1    New York 850000  95000
## 2 Philadelphia 455000 100100
## 3      Boston 550000 134000
```

Saving

Above, we showed that you can write data as a CSV using the **read.csv()** function. R also has some proprietary data formats, which allow you to store entire R objects. You can save it using the **saveRDS()** function:

```
saveRDS(sales, file = "sales data.RDS")
```

And call it up with **readRDS()**.

```
sales <- readRDS("sales data.RDS")
```