Getting and Cleaning Data - Notes

Tanner Prestegard

Course taken from 4/6/2015 - 5/3/2015

Motivation

- Real data can be messy and incomplete, or can be in a different format than you expect.
- Data can be in an environment where you need to extract it out (ex: mySQL or MongoDB).
- The goal of this course: raw data -> processing script -> tidy data.
- After you have tidy data, you can do the data analysis (but that is not the focus of this course).

Raw and processed data

- Definition of data: data are values of qualitative or quantitative variables, belonging to a set of items.
- Raw data
  - The original source of the data.
  - Often hard to use for data analyses.
  - Data analysis includes processing.
  - Raw data may only need to be processed once.
- Processed data
  - Data that is ready for analysis
  - Processing can include merging, subsetting, transforming, etc.
  - There may be standards for processing.
  - All processing steps should be recorded!

Components of Tidy Data

- You should have four things when you finish processing data.
  - The raw data.
  - A tidy data set.
  - A code book describing each variable and its value in the tidy data set (this is also referred to as metadata).
  - An explicit and exact recipe that you used to go from raw data to the tidy data and the code book (in our case, it will be an R script).
- Raw data is in the "right" (unadulterated) format if:
  - You ran no software on the data.

  - You did not manipulate or remove any of the data.
  - You did not summarize the data in any way.

- Tidy data description
  - Each variable that you measured should be in one column.
  - Each different observation should be in a different row.
  - There should be one table for each “kind” of variable.
  - If you have multiple tables, they should include a column in the tables that allows them to be linked.
  - Include a row at the top of each file with variable names. These variable names should be human readable.
  - In general, data should be saved in one file per table.

- The code book
  - Information about the variables (including units) in the data set.
  - Information about the summary choices you made.
  - Information about the experimental study design you used.
  - A common format is a Word/text file.
  - There should be a section called “study design” that has a thorough description of how you collected the data.
  - There must be a section called “code book” that describes each variable and its units.

- The instruction list
  - Ideally a computer script (an R script in our case).
  - The input for the script is the raw data.
  - The output is the processed, tidy data.
  - There are <u>no parameters</u> in the script. (the recipe should be exact, it shouldn’t need to be tweaked or modified by future users)
  - In some cases, it will not be possible to script every step. In that case you should provide instructions. (Ex: run the script on datasets 1, 2, 3 and then run another script to combine the results)

Downloading files

- The downloading process can be included in the processing script so that everything is automated and helps with reproducibility.
- Getting/setting your working directory:
  - `getwd()`
  - `setwd()`
  - Either can use absolute or relative paths.
- Checking for and creating directories:
  - `file.exists(“directoryName”)` will check to see if the directory exists.
  - `dir.create(“directoryName”)` will create a directory if it doesn’t exist.

- Getting data from the internet - `download.file()`.
  - Downloads a file from the internet.
  - Important parameters are `url, destfile, method`.
    - `url`: link to file.
    - `destfile`: filename to save the data as.
    - `method`: how to access the data. May need to use `method=”curl”`.
  - Useful for downloading tab-delimited, csv, and other files.
  - Good to keep track of the date that you downloaded the data because they may change.
    - Use the `date()` function to get this.
- Some notes:
  - If the URL starts with *http*, `download.file()` is OK.
  - If the URL starts with *http*s, `download.file()` is OK in Windows.
  - If the URL starts with *http*s, you may need to use `method=”curl”` on a Mac.

Reading local “flat” files

- Flat files are things like text files or csv files.
- Most common way to use this is with `read.table()`.
  - Flexible and robust but requires more parameters.
    - Important parameters: `file, header, sep, row.names, nrows`.
  - Reads the data into RAM - big data can cause problems.
  - Related: `read.csv(), read.csv2()`.
  - Other useful parameters:
    - quote: you can tell R whether there are any quoted values. `quote=’”’` means no quotes.
    - na.strings: sets the character that represents a missing value.
    - nrows: how many rows of the file to read.
    - skip: number of lines to skip before starting to read.
  - A big problem in reading flat files is that you see quotation marks like ‘ or “ in the data. Setting `quote=’”’` often resolves these problems.

Reading Excel files

- May be the most widely used format for sharing data, but can be a little more difficult to handle with a scripting language like R.
- Need load the `xlsx` package. `XLConnect` can be useful too.
- Functions to use:
  - `read.xlsx()`
  - `read.xlsx2()` - this is much faster but may be slightly unstable for reading subsets of rows.
- You can read specific rows and/or columns using the `colIndex` and `rowIndex` parameters.
- `write.xlsx` will write out an Excel file (and has similar arguments).
- In general it is advised to store your data as flat files (csv or tsv) as they are easier to distribute.

Reading XML

- Extensible markup language.
- Frequently used to store structured data.
- Particularly widely used in internet applicatinos.
- Extracting XML is the basis for most web scraping.
- Components
  - Markup - labels that give the text structure.
  - Content - the actual text of the document.
- XML tags
  - Tags correspond to general labels.
    - Start tags - ex: `<section>`
    - End tags - ex: `</section>`
    - Empty tags - ex: `<line-break />`
  - Elements are specific examples of tags.
    - `<Greeting> Hello, world </Greeting>`
  - Attributes are components of the labels
    - `<img src=”jeff.jpg” alt=”instructor” />`
    - `<step number=”3”> Connect A to B. </step>`
- How to read XML files into R:
  - `library(XML)`
  - `doc <- xmlTreeParse(fileURL, useInternal=TRUE)`
  - After this, it's still a structured object, so we have to use different functions to get different elements.
  - `rootNode <- xmlRoot(doc)` to get root node.
  - Can use `names(rootNode)` to get names of root node elements.
- To directly access parts of the XML document:
  - Use double brackets: `rootNode[[1]]`, or `rootNode[[1]][[1]]` if you want to go into deeper subsets (if they exist).
- To programmatically extract parts of the file:
  - Use `xmlSApply(rootNode, xmlValue)` (`xmlValue` returns the value of the element).
- A better solution may be to use `XPath`.
  - Problem: it's a whole new language.
  - Get information from http://www.stat.berkeley.edu/~statcur/Workshop2/Presentations/XML.pdf
  - `/node` is the top-level node.
  - `//node` is a node at any level.
  - We probably only need to know a few different commands in `XPath`.

- `XPath` examples:
  - `xpathSApply(rootNode,"//name",xmlValue)` will return the names of all values.
  - `xpathSApply(rootNode,"//price,xmlValue)` will return the price nodes.
  - `xpathSApply(doc,"//li[@class='score']",xmlValue)` gets all list elements with a class equal to score.
- To parse HTML files, use `htmlTreeParse()`.

Reading JSON

- JSON = Javascript Object Notation
- Lightweight data storage.
- Common format for data from application programming interfaces (APIs).
- Similar structure to XML, but a very different syntax and format.
- Data stored as:
  - Numbers (double)
  - Strings (double quoted)
  - Boolean (true or false)
  - Array (ordered, comma-separated enclosed by [])
  - Object (unordered, comma-separated collection of key:value pairs enclosed by {})
- Reading data from JSON
  - `library(jsonlite)`
  - `jsonData <- fromJSON(fileURL)`
  - `names(jsonData)`
- You can also take data frames in R and turn it into a JSON data set.
  - `myjson <- toJSON(iris, pretty=TRUE)` (pretty=TRUE gives nice indentation)
  - We can take this and send it right back to a data table using `fromJSON()`.
    - `iris2 <- fromJSON(myjson)`

Using data.table

- Inherits from `data.frame` - all functions that accept data.frame work on data.table.
- Written in C so it is much faster in general, especially at subsetting, grouping, and updating.
- However, requires a new syntax.
- `data.table()` takes the same arguments as `data.frame()`
  - `DT <- data.frame(x=rnorm(9), y=rep(c("a","b","c"), each=3), z=rnorm(9))`
- See all the data tables in memory: `tables()`
  - Will tell you name of the data table, number of rows, memory used, column names, and if there is a "key."

- Subsetting rows is exactly the same as for a data frame.
- If you subset with only one index, it subsets by rows, which is different from a data frame.
- You can't subset columns the same way as a data frame. (ex: `DT[,c(2,3)]`)
  - You have to use expressions to summarize the data in different ways.
  - Expressions are collections of statements enclosed in curly brackets.
  - Can pass it something like `DT[,list(mean(x),sum(z))]`
- To add a new column: `DT[,w:=z^2]`, where `w` is the new column.
- If you make a copy of a data table (`DT2 <- DT`), then modified the first table, the second one will also be modified because R doesn't actually copy it to save memory when working with big datasets.
  - Need to use the copy function to actually make a copy.
- Can perform multiple operations to create a new column.
  - Ex: `DT[,m:= {tmp <- (x+z); log2(tmp+5)}]`
  - The last line is returned as the column.
- Special variables:
  - `.N`: integer of length 1, containing the number of times that a particular group appears.
  - `DT[, .N, by=x]` returns a new column showing the number of occurrences of each element.
- Keys
  - Can be useful for subsetting.
    - `DT <- data.table(x=rep(c("a","b","c",each=100), y=rnorm(300))`
    - `setkey(DT, x)`
    - `DT['a']`
  - Can be used to facilitate joins for data tables.
    - `DT1 <- data.table(x=c('a','a','b','dt1'), y=1:4)`
    - `DT2 <- data.table(x=c('a','b','dt2'), z=5:7)`
    - `setkey(DT1, x); setkey(DT2, x)`
    - `merge(DT1, DT2)`
- Can be useful for fast reading of files.
  - If you save a data table in a file, you can use `fread()` to read it - this is a lot faster (about 10 times faster) than `read.table()`.

Reading from MySQL

- Free and widely used open source database software, widely used in internet-based applications.
- Data are structured in:
  - Databases
  - Tables within databases
  - Fields within tables

- Each row is called a record.
- Documentation at http://www.mysql.com
- Installing MySQL - go to http://dev.mysql.com/doc/reman/5.7/en/installing.html
- Install RMySQL on Windows:
  - Go to http://biostat.mc.vanderbilt.edu/wiki/Main/RMySQL or http://www.ahschulz.de/2013/07/23/installing-rmysql-under-windows/
- Connecting and listing databases:
  - `ucscDb <- dbConnect(MySQL(), user=”genome”, host =”genome-mysql.cse.ucsc.edu”)`
  - `result <- dbGetQuery(uscscDb,”show databases;”); dbDisconnect(ucscDb);`
  - result shows a list of all databases available in this MySQL server
  - To get a particular database: `hg19 <- dbConnect(MySQL(), user=”genome”, db=”hg19”, host =”genome-mysql.cse.ucsc.edu”)`
  - To get a list of all tables: `allTables <- dbListTables(hg19)`
- List field names of a table: `dbListFields(hg19,”tableName”)`
- Get dimensions of a specific table: `dbGetQuery(hg19, “select count(*) from tableName”)`
- To read from a table (example): `affyData <- dbReadTable(hg19, “affyU133Plus2”)`
  - To select a specific subset: `query <- dbSendquery(hg19, “select * from affyU133Plus2 where misMatches between 1 and 3”)`
  - `affyMis <- fetch(query); quantile(affyMis$misMatches)`
  - Can specify `n=10` as a 2nd argument in `query` to only get a few rows.
  - Note: `misMatches` is a column in the table.
  - Need to do `dbClearResult(query)` to clear the query from the remote server.
- Very important to close your connection! (do it as soon as you have the data you need)
- RMySQL vignette: http://cran.r-project.org/web/packages/RMySQL/RMySQL.pdf
- List of commands: http://www.pantz.org/software/mysql/mysqlcommands.html

Reading from HDF5

- Used for storing large data sets and structed data sets.
- Support storing a range of data types.
- HDF = Hierarchical Data Format
- Data is stored in groups which contain 0 or more groups with their metadata.
  - Have a group header with group name and list of attributes.
  - Have a group symbol table with a list of objects in the group.
- Datasets are multidimensional array of data elements along with metadata
  - Have a header with name, datatype, dataspace, and storage layout.
  - Have a data array with the data - like a data frame.

- To install and load:
  - `source(“http://bioconductor.org/biocLite.R”)`
  - `biocLite(“rhdf5”)`
  - `library(rhdf5)`
- To create a file: `created = h5createFile(“example.h5”)`
- To create groups within the file:
  - `created = h5createGroup(“example.h5”,”foo”)`
  - `created = h5createGroup(“example.h5”,”foo/foobaa”)`
- To list groups: `h5ls(“example.h5”)`
- Write to groups:
  - `A = matrix(1:10,nr=5,nc=2)`
  - `h5write(a, “example.h5”, “foo/A”)`
  - `B = array(seq(0.1,2.0,by=0.1),dim=c(5,2,2))`
  - `attr(B, “scale”) <- “liter”`
  - `h5write(B, “example.h5”, “foo/foobaa/B”)`
- Write a data set directly:
  - `df = data.frame(1L:5L, seq(0,1,length.out=5), c(“ab”,”cde”,”fghi”,”a”,”s”), stringAsFactors=FA`
  - `h5write(df, “example.h5”,”df”)`
- Reading data
  - `readA = h5read(“example.h5”, “foo/A”)`
  - `readB = h5read(“example.h5”, “foo/foobaa/B”)`
  - `readdf = h5read(“example.h5”, “df”)`
- Writing and reading chunks
  - `h5write(c(12,13,14),”example.h5”,”foo/A”,index=list(1:3,1))`
  - This writes to the first 3 rows in the first column of this dataset.
  - Can do the same kind of indexing with `h5read`.
- hdf5 can be used to opimize reading/writing from disk in R.

Reading data from the web

- Web scraping is programmatically extracting data from the HTML code of websites.
  - It can be a great way to get data.
  - Many websites have information you may want to programmatically read.
  - In some cases this is against the terms of service for the website.
  - Attempting to read too many pages too quickly can get your IP address blocked.
- Getting data off webpages with `readLines()`

  - `con = url(“web address”)`
  - `htmlCode = readLines(con)`
  - `close(con)`
- Can parse with XML instead:
  - `html = htmlTreeParse(url, useInternalNodes=TRUE)`
  - `xpathSApply(html, “//title”, xmlValue)`
- Using GET from the httr package
  - `html2 = GET(url)`
  - `content2 = content(html2, as=”test”)`
  - `parsedHtml = htmlParse(content2, asText=TRUE)`
  - Then you can use `xpathSApply` as you would have otherwise.
- Accessing website with passwords
  - `pg2 = GET(url,authenticate(“user”,”passwd”)`
- Using handles
  - `google = handle(“http://google.com”)`
  - `pg1 = GET(handle=google, path=”/”)`
  - Only need to authenticate this handle one time.

Reading data from APIs

- API - Application Programming Interface
- Example: https://dev.twitter.com/docs/api/1/get/blocks/blocking
- Usually need to create an account with the development team: https://dev.twitter.com/apps
- Need to create a new application, fill in details.
- Accessing Twitter from R
  - `myapp = oauth_app(“twitter”, key=”yourConsumerKey”, secret=”yourConsumerSecret”)`
  - `sig = sign_oauth1.0(myapp, token=”yourToken”, token_secret=“yourTokenSecret”)`
  - `homeTL = GET(“https://api.twitter.com/1.1/statuses/home_timeline.json”, sig)`
- Converting the json object
  - `json1 = content(homeTL)`
  - `json2 = jsonlite::fromJSON(toJSON(json1))`
- How do we know what URL to use?
  - Documentation for Twitter API: https://dev.twitter.com/docs/api/1.1/get/search/tweets
  - Lots of other useful information there.
- httr allows GET, POST, PUT, DELETE requests if you are authorized.
- You can authenticate with a user name or password, but most modern APIs use something like oauth.
- httr works well with Facebook, Google, Twitter, Github, etc.

Reading from other sources

- There is a package in R for almost everything!
- The best way to find packages - Google "MySQL R package" for example.
- Interacting more directly with files
  - file - open a connection to a text file.
  - url - open a connection to a url.
  - gzfile - open a connection to a .gz file.
  - bzfile - open a connection to a .bz2 file.
  - ?connection for more information.
  - Remember to close connections.
- The `foreign` packages is useful for working with data from other statistical analysis programs.
  - Loads data from Minitab, S, SAS, SPSS, Stata, Systat
  - Basic functions:
    - read.arff (Weka)
    - read.dta (Stata)
    - read.mtp (Minitab)
    - read.octave (Octave)
    - read.spss (SPSS)
    - read.xport (SAS)
- Reading images
  - jpeg - https://cran.r-project.org/web/packages/jpeg/index.html
  - readbitmap - http://cran.r-project.org/web/packages/readbitmap/index.html
  - png - http://cran.r-project.org/web/packages/png/index.html
- Reading GIS data (Geographic Information Systems)
  - rdgal, rgeos, raster
- Reading music data (directly from mp3)
  - tuneR, seewave
  - Lots of nice musical processing packages in R.

Subsetting and sorting

- Subsetting a column: `x[,1]` or `x[,"var1"]` where `var1` is the variable name.
- Subsetting rows and columns: `x[1:2,"var1"]`
- Subsetting with logical statements:
  - `x[(x$var1 <= 3 & x$var3 > 11),]`
- Dealing with missing values:
  - `x[which(x$var2 > 8),]`. The which command doesn't return indices of NA values.

- Sorting: `sort(x$var1, decreasing=TRUE, na.last=TRUE)`
- Ordering: sorting a data frame by a particular column's values.
  - `x[order(x$var1),]`
- Ordering with `plyr` package:
  - `library(plyr)`
  - `arrange(x,var1)` does the same thing as the `order` examples from above.
  - `arrange(x,desc(var1))` puts it in descending order.
- Adding rows and columns to data frames:
  - `x$var4 <- rnorm(5)` where `var4` was not previously a column of `x`.
  - `y <- cbind(X,rnorm(5))` does a column binding to add a column to the right side of `x`.
  - `y <- cbind(rnorm(5),x)` binds to the left side.
  - Can use `rbind` to add rows.

Summarizing data

- Look at first three rows of data frame: `head(data, n=3)`
- Look at last three rows of data frame: `tail(data, n=3)`
- To summarize: `summary(data)`.
  - Tells you minimum, median, mean, max, quartiles, etc.
- More in-depth information: `str(data)`
  - Tells you the class and size of the object (and lots of other information).
- Can look at quantiles: `quantile(data$var1, na.rm=TRUE, probs=c(0.5,0.75,0.9)`
- Make table: `table(data$var2, useNA=”ifany”)`
  - This tells you about the number of unique values and how often they occur.
  - If any missing values, there will be an extra table column that will tell you about the number of NAs.
  - To make a 2D table based on other variables: `table(data$var1, data$var2)`
- Check for missing values:
  - Total number of NAs: `sum(is.na(data$var1))`
  - Check if there are any NAs: `any(is.na(data$var1))`
  - Check if all elements of an array satisfies a condition: `all(data$var1 > 0)`
- Row and column sums:
  - `colSums(is.na(data))`
  - `rowSums(is.na(data))`
  - `all(colSums(is.na(data))==0)`
- Values with specific characteristics:

    - `table(data$var1 %in% c(“21212”,”21213”))`
    - This returns the number of cases where this is true or false.
    - Subsetting with this method: `data[data$var1 %in% c(“21212”,”21213”),]`

- Cross tabs: identify where relationships exist in a data set.
    - `data(UCBAdmissions)`
    - `DF = as.data.frame(UCBAdmissions)`
    - `xt <- xtabs(Freq ~ Gender + Admit, data=DF)`
- Flat tables
    - `warpbreaks$replicate <- rep(1:9, len = 54)`
    - `xt = xtabs(breaks ~., data=warpbreaks)`
    - This creates multiple 2D tables.
    - We can summarize these tables in a larger, more compact form using `ftable(xt)`.
- Size of a data set
    - `object.size(data)` gives size in bytes.
    - `print(object.size(data), units=”Mb”)` gives size in MB.

Creating new variables

- Often the raw data won’t have a value you are looking for, so you may need to transform the data to get what you want.
- Common variables to create:
    - “Missingness” indicators
    - “Cutting up” quantitative variables.
    - Applying transforms.
- Creating sequences (sometimes you need an index for your data set)
    - `s1 <- seq(1,10, by=2)` gives `1 3 5 7 9`
    - `s2 <- seq(1,10, length=3)` gives `1.0 5.5 10.0`
    - `x <- c(1,3,8,15,100); seq(along = x)` gives `1 2 3 4 5`
- Subsetting variables
    - `restData$nearMe = restData$neighborhood %in% c(“Roland Park”, “Homeland”)` adds a new subset to the data set.
- Creating binary variables
    - `restData$zipWrong = ifelse(restData$zipCode < 0, TRUE, FALSE)` gives a value of `TRUE` if the zip code is negative.
- Creating categorical variables out of quantitative variables
    - `restData$zipGroups = cut(restData$zipCode, breaks=quantile(restData$zipCode))`
    - Puts zip codes into four groups based on quantiles of all of the zip codes.

  - Easier cutting:
    - `library(Hmisc)`
    - `restData$zipGroups = cut2(restData$zipCode, g=4)`
    - Finds quantiles automatically, you don't have to set the breaks in advance.
  - Cutting produces factor variables.

- Creating factor variables
  - `restData$zcf <- factor(restData$zipCode)`

- Levels of factor variables
  - `yesno <- sample(c("yes","no"), size=10, replace=TRUE)`
  - `yesnofac = factor(yesno, levels=c("yes","no"))`
  - `relevel(yesnofac, ref="yes")`

- Convert factor to numeric: `as.numeric(yesnofac)`

- Using the `mutate` function - creates a new version of a variable and simultaneously adds it to the data set.
  - `library(Hmisc); library(plyr)`
  - `restData2 = mutate(restData, zipGroups=cut2(zipCode, g=4))`
  - Creates a new data frame `restData2` based on `restData` and adds a new variable `zipGroups`.

- Common data transforms:
  - `abs()`, `sqrt()`, `ceiling()`, `floor()`, `round(x, digits=n)`, `signif(x, digits=n)`, `cos()`, `sin()`, `log()`, `log2()`, `log10()`, `exp()`.

Reshaping data

- Putting data into the format that you want.
- Tidy data principles (reminder)
  - Each variable forms a column.
  - Each observation forms a row
  - Each table/file stores data about one kind of observation (e.g. people/hospitals)
- Use the `reshape2` library. Examples use `mtcars` data set.
- Melting data frames
  - `mtcars$carname <- rownames(mtcars)`
  - `carMelt <- melt(mtcars, id=c("carname","gear","cyl"), measure.vars=c("mpg","hp"))`
    - We tell it which variables are measurements and which are IDs.
- Casting data frames - now we want to re-cast the data set in a particular form.
  - `cylData <- dcast(carMelt, cyl ~ variable)`
  - This gives the number of measures of each variable as a function of the number of cylinders.
  - `cylData <- dcast(carMelt, cyl ~ variable, mean)` gives the mean of each variable as a function of the number of cylinders.

- Averaging values:
  - Use `tapply(InsectSprays$count, InsectSprays$spray, mean)`
  - Another way:
    - `spIns = split(InsectSprays$count, InsectSprays$spray)`
    - `sprCount = lapply(spIns, mean)`
    - Combine back into a vector: `unlist(sprCount)` or `sapply(spIns, mean)`
  - Another way: `plyr` package
    - `ddply(Insect.Sprays,.(spray), summarize, mean=mean(count))`
- Creating a new variable:
  - `spraySums <- ddply(InsectSprays,.(spray), summarize, sum=ave(count, FUN=sum))`
- `plyr` tutorials:
  - http://plyr.had.co.nz/0-user/
  - http://www.r-bloggers.com/a-quick-primer-on-split-apply-combine-problems/
- See also the functions:
  - `acast` - for casting as multi-dimensional arrays.
  - `arrange` - for faster reordering without using order() commands.
  - `mutate` - for adding new variables.

Managing data frames with dplyr

- Data frame properties:
  - One observation per row.
  - Each column represents a variable or measure or characteristic.
- `dplyr` does not necessarily provide any new functionality, but greatly simplifies a lot of other operations and makes them much faster.
- `dplyr` function properties
  - First argument is a data frame.
  - Subsequent arguments describe what to do with it.
  - You can refer to columns in the data frame directly without using the $ operator.
  - The result is a new data frame.
  - Data frames must be properly formatted and annotated for this to be useful.
- `select` function
  - Allows you access a set of columns within the data frame.
  - Example: `head(select(data, city:dptp))` selects all columns between those labeled city and dptp
  - `head(select(chicago,-(city:dptp)))` selects all columns except those in the specified range.
- `filter` function
  - Used to subset rows based on conditions

  - `filter(chicago, pm25tmean2 > 30 & tmpd > 80)` selects all rows where these conditions are satisfied.

- `arrange` function
  - Used to re-order the rows of a data frame based on the values of a particular column.
- `rename` function
  - Used to rename a variable.
  - `chicago <- rename(chicago, pm25 = pm25tmean2, dewpoint = dptp)`
    - New variable name is on the left!
- `mutate` function
  - Creates a new variable and adds it to the data frame.
  - `chicago <- mutate(chicago, pm25detrend = pm25-mean(pm25, na.rm = TRUE))`
- `group_by` function
  - Allows you to essentially split a data frame according to categorical variables.
  - `chicago <- mutate(chicago, tempcat = factor(1*(tmpd > 80), label=c("cold","hot"))`
  - `hotcold <- group_by(chicago, tempcat)`
  - `summarize(hotcold, pm25 = mean(pm25), o3 = max(o3tmean2), no2 = median(no2tmean2))` gives the resuling information as grouped by the "cold" and "hot" categories.
- Can use the pipeline operator `%>%` to send the output of one function directly to another function.
- `dplyr` can be useful with other data frame "backends" like data.table and the SQL interface for relational databases via the DBI package.

Merging data

- Merging more than one data set together, usually want to use some type of ID to match them up.
- Examples include two datasets: one called `reviews` and one called `solutions`.
- The `merge()` command merges data frames.
  - `x`: data frame 1
  - `y`: data frame 2
  - Use `by`, `by.x`, or `by.y` to merge.
  - Example: `merged_data = merge(reviews, solutions, by.x="solution_id", by.y="id", all=TRUE)`
  - Default - merge all common column names (can do `intersect(names(solutions), names(reviews))` to get common names).
- Can also use `join()` in the `plyr` package - faster, but fewer features.
  - Will only merge on the basis of a common ID.
  - Can use `join_all(list(df1,df2,df3))` to merge more than one data frame (must use data frame list syntax).

Editing text variables

- Fixing character vectors
  - Convert case: `tolower()`, `toupper()`
  - Splitting strings: `strsplit(str_vect,"separator")`.
    - Example: `strsplit("Location.1","\\.")` gives `"Location" "1"`.
  - Using `sapply()`
    - `firstElement <- function(x){x[1]}`
    - `sapply(splitNames,firstElement)`
  - Using `sub()`
    - Replace underscores with nothing: `sub("_","",names(reviews))`
  - `gsub()` - replaces multiple instances. `sub()` just replaces the first occurrence.
- Finding values
  - `grep(search_str,str_vect)`, returns vector indices of matches.
    - Can use `value=TRUE`, this returns the value rather than the index.
  - `grepl(search_str,str_vect)`, returns logical vector of TRUE or FALSE.
- Other useful functions in the `stringr` and base packages
  - `nchar(str)` tells you the number of characters in the string.
  - `substr(str,n1,n2)` gets the sub-string between the indices specified.
  - `paste(str1,str2,str3,sep=" ")` pastes the strings together using the specified separator.
  - `paste0(str1,str2,str3)` pastes together without any space.
  - `str_trim("Jeff    ")` trims of all space at the end.
- Names of variables should be:
  - All lower case when possible.
  - Descriptive.
  - Not duplicated.
  - Not have underscores, dots, or white spaces.

Regular expressions

- Thought of as a combination of literals (exact matches) and metacharacters.
- Have a rich set of metacharacters.
- Used to search through strings and match patterns.
- We need a way to express whitespace word boundaries, sets of literals, the beginning/end of a line, and other things.
- Metacharacters (seem to be the same as Perl for the most part):
  - `^`: start of a line.
  - `$`: end of a line.
  - `[Bb][Uu]`: matches BU or Bu or bU or bu.

  - `[0-9][a-zA-Z]`: matches any combination of number and uppercase or lowercase letters.
  - `[^?.]$`: caret here indicates "NOT". This will match any line not ending in a question mark or period.
  - `.`: used to refer to any character. `9.10` will match `9/10, 9110, 911`.
  - `str1|str2|str3`: matches `str1` OR `str2` OR `str3`.
  - `?`: the indicated expression is optional.
    - Ex: `[Gg]eorge( [Ww]\.)? [Bb]ush` will optionally match a W or w in the middle.
    - Note: have to escape the `.` here so it is not interpreted as a metacharacter.
  - `*`: repeats any character any number of times, including none.
  - `+`: at least one of the item.
  - `{m}`: match expression exactly `m` times.
  - `{m,}`: match expression at least `m` times.
  - `{1,5}`: match expression at least 1 time but no more than 5.

- Parentheses can also be used to return the matched expressions.
- The matches can be accessed using `\1, \2`, etc.
  - Example: `+([a-zA-Z]+) +\1 +` is looking for repetition of a phrase.
- The `*` is greedy so it always matches the longest possible string that satisfies the regular expression.
  - To make it less greedy, we can use the question mark.
  - `^s(.*?)s$` is less greedy than `^s(.*)s$`.
- Regular expressions are typically used with `grep, grepl, sub, gsub`.

Working with dates

- `date()` function gives a character variable containing something like "`Sun Jan 12 17:48:33 2014`".
- `Sys.Date()` gives Date variable containing something like "2014-01-12".
  - `format(d2, "%a %b %d")` gives "`Sun Jan 12`".
  - Useful characters:
    - %d - day as a number.
    - %a - abbreviated weekday.
    - %A - unabbreviated weekday.
    - %m - month (01-12).
    - %b = abbreviated month.
    - %B = unabbreviated month.
    - %y - 2 digit year.
    - %Y - 4 digit year.
- Creating dates:
  - `x <- c("1jan1960", "2jan1960", "30jul1960")`
  - `z <- as.Date(x, "%d%b%Y")`
  - Difference: `as.numeric(z[1] - z[2])` gives -1
- `weekdays(date1)` gives day of week name as a character vector.

- `months(date1)` gives full month name as a character vector.
- `julian(date1)` gives the number of days since 1/1/1970.
  - `attr(, “origin”)` gives the origin of the Julian calendar.
- lubridate packages for dates:
  - `ymd(“20140108”)` gives `“2014-01-08 UTC”`
  - `mdy(“08/04/2013”)` gives `“2013-08-04 UTC”`
  - `dmy(“03-04-2013”)` gives `“2013-04-03 UTC”`
  - Can also use things like `ymd_hms(str, tz=timezone)` on a character vector of year-monthh-day hour-minute-second.
    - To learn more about how to handle timezones look at ?Sys.timezone.
  - `wday()` gives the weekday as a number; specifying `label=TRUE` gives the day of the week abbreviation.
- `POSIXct` and `POSIXlt` are also useful date and time classes.

Data resources

- Open government sites
  - United Nations - http://data.un.org
  - United States - http://www.data.gov
  - Many more: http://www.data.gov/opendatasites
- Gapminder: development and human health. http://www.gapminder.org
- Survey data from United States: http://www.asdfree.com
- Infochimps Marketplace: http://www.infochimps.com/marketplace
- Kaggle: company that offers data science competitions. http://www.kaggle.com
- UCI machine learning
- Some APIs with R interfaces
  - twitter and twitteR package
  - figshar and rfigshare package
  - PLoS and rplos package
  - rOpenSci - academic focu.
  - Facebook and RFacebook
  - Google maps and RGoogleMaps