Getting and Cleaning Data Course Notes

Xing Su

Contents

Overview

- finding and extracting raw data
- today any how to make data tiny
- Raw data → processing script → tidy data → data analysis → data communication

Raw and processed data

- **Data** = values of qualitative/quantitative, variables, belonging to a set of items
  - **variables** = measurement of characteristic of an item
- **Raw data** = original source of data, often hard to use, processing must be done before analysis, may need to be processed only once
- **Processed data** = ready for analysis, processing done (merging, transforming, etc.), all steps should be recorded
- Sequencing DNA: $1B for Human Genome Project → $10,000 in a week with Illumina

Tidy Data

1. **Raw Data**
   - no software processing has been done
   - did not manipulate, remove, or summarize in anyway
2. **Tidy data set**
   - end goal of cleaning data process
   - each variable should be in one column
   - each observation of that variable should be in a different row
   - one table for each kind of variable
     - if there are multiple tables, there should be a column to link them
   - include a row at the top of each file with variable names (variable names should make sense)
   - in general data should be save in one file per table
3. **Code book describing each variable and its values in the tidy data set**
   - information about the variables (w/ units) in dataset ***NOT*** contained in tidy data
   - information about the summary choice that were made (median/mean)
   - information about experimental study design (data collection methods)
   - common format for this document = markdown/Word/text
     - ***"study design"*** section = thorough description of how data was collected
     - ***"code book"*** section = describes each variable and units
4. **Explicit steps and exact recipe to get through 1 - 3 (instruction list)**
   - ideally a computer script (no parameters)
   - output = processed tidy data
   - in addition to script, possibly may need steps to run files, how script is run, and explicit instructions

Download files

- **Set working directory**
  - ***Relative***: `setwd("./data")`, `setwd("../")` = move up in directory
  - ***Absolute***: `setwd("/User/Name/data")`
- **Check if file exists and download file**
  - `if(!file.exists("./data")) {dir.create("./data")}`
- **Download file**
  - `download.file(url, destfile= "directory/filname.extension", method = "curl")`
    - `method = "curl"` [mac only for https]
  - `dateDownloaded <- date()` = record the download date
- **Read file and load data**
  - `read.table()` = need to specify `file`, `header`, `sep`, `row.names`, `nrows`
    - `read.csv()` = automatically set `sep = ","` and `header = TRUE`
  - `quote = ""` = no quotes (extremely helpful, common problem)
  - `na.strings` = set the character that represents missing value
  - `nrows` = how many rows to read
  - `skip` = how many lines to skip
  - `col.names` = specifies column names
  - `check.names = TRUE/FALSE` = If TRUE then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names and are not duplicated. If necessary they are adjusted appropriately to be syntactically valid names

Reading Excel files

- xlsx package: `read.xlsx(path, sheetIndex = 1, ...)`
  - `colIndex`, `rowIndex` = can be used to read certain rows and columns
- `write.xlsx()` = write out excel file
- `read.xlsx2()` = faster than `read.xlsx()` but unstable for reading subset of rows
- `XLConnect` package has more options for writing/manipulating Excel files
- generally good to store data in database/csv/tab separated files (.tab/.txt), easier to distribute

Reading XML

- **XML** = extensible markup language
- frequented used to store structured data, widely used in Internet apps
- extracting XML = basis for most of web scraping
- components
  - ***markup*** = labels that give text structure
  - ***content*** = actual text of document
- tags = `<section>`, `</section>`, `<line-break />`
- elements = `<Greeting>` test `</Greeting>`
- attributes = `<image src ="a.jpg" alt = "b">`
- reading file into R
  - `library(XML)`
  - `doc <- xmlTreeParse(fileUrl, useInternal = TRUE)` = loads data
  - `rootNode <- xmlRoot(doc)` = wrapper element for entire document
  - `xmlName(rootNode)` = returns name of the document
  - `names(rootNode)` = return names of elements
  - `rootNode[[1]]` = access first elements, similar to list
  - `rootNode[[1]][[1]]` = first sub component in the first element
  - `xmlSApply(rootNode, xmlValue)` = returns every single tagged element in the doc
- **XPath** (new language)
  - get specific elements of document
  - `/node` = top level node
  - `//node` = node at any level
  - `node[@attr-name = 'bob']` = node with attribute name
    - `xpathSApply(rootNode, "//name", xmlValue)` = get the values of all elements with tag "name"
    - `xpathSApply(rootNode, "//price", xmlValue)` = get the values of all elements with tag "price"
- **extract content by attributes**
  - `doc <- htmlTreeParse(url, useInternal = True)`
  - `scores <- xpathSApply(doc, "//li[@class='score']", xmlvalue)` = look for li elements with `class = "score"` and return their value

Reading JSON

- **JSON** = JavaScript Object Notation
- lightweight data storage, common format for data from application programming interfaces (API)
- similar to XML in structure but different in syntax/format
- data can be stored as:
  - numbers (double)
  - strings (double quoted)
  - boolean (true/false)
  - array (ordered, comma separated enclosed in `[]`)
  - object (unordered, comma separated collection of key/value pairs enclosed in `{}`)
- `jsonlite` package (json vignette can be found in help)
  - `library(jsonlite)` = loads package
  - `data <- fromJSON(url)` = strips data
    - `names(data$owner)` = returns list of names of all columns of owner data frame
    - `data$owner$login` = returns login instances
  - `data <- toJSON(dataframe, pretty = TRUE)` = converts data frame into JSON format
    - `pretty = TRUE` = formats the code nicely
  - `cat(data)` = prints out JSON code from the converted data frame
  - `fromJSON()` = converts from JSON object/code back to data frame

data.table

- inherits from `data.frame` (external package) → all functions that accept `data.frame` work on `data.table`
- can be much faster (written in C), ***much much faster*** at subsetting/grouping/updating
- **syntax**: `dt <- data.table(x = rnorm(9), y = rep(c("a","b","c"), each = 3), z = rnorm(9))`
- `tables()` = returns all data tables in memory
  - shows name, nrow, MB, cols, key
- some subset works like before = `dt[2, ], dt[dt$y=="a",]`
  - `dt[c(2, 3)]` = subset by rows, rows 2 and 3 in this case
- **column subsetting** (modified for `data.table`)
  - argument after comma is called an ***expression*** (collection of statements enclosed in `{}`)
  - `dt[, list(mean(x), sum(z))]` = returns mean of x column and sum of z column (no `""` needed to specify column names, x and z in example)
  - `dt[, table(y)]` = get table of y value (perform any functions)
- **add new columns**
  - `dt[, w:=z^2]`
    - when this is performed, a new data.table is created and data copied over (not good for large datasets)
  - `dt2 <- dt; dt[, y:= 2]`
    - when changes are made to dt, changes get translated to dt2
    - ***Note****: if copy must be made, use the* `copy()` *function instead*
- **multiple operations**
  - `dt[, m:= {temp <- (x+z); log2(temp +5)}]` → adds a column that equals log2(x+z + 5)
- **plyr like operations**
  - `dt[,a:=x>0]` = creates a new column a that returns TRUE if x > 0, and FALSE other wise
  - `dt[,b:=mean(x+w), by=a]` = creates a new column b that calculates the aggregated mean for `x + w` for when a = TRUE/FALSE, meaning every b value is gonna be the same for TRUE, and others are for FALSE
- **special variables**
  - `.N` = returns integer, length 1, containing the number (essentially count)
    - `dt <- data.table (x=sample(letters[1:3], 1E5, TRUE))` = generates data table
    - `dt[, .N, by =x]` = creates a table to count observations by the value of x
- **keys** (quickly filter/subset)
  - *example*: `dt <- data.table(x = rep(c("a", "b", "c"), each = 100), y = rnorm(300))` = generates data table
    - `setkey(dt, x)` = set the key to the x column
    - `dt['a']` = returns a data frame, where x = 'a' (effectively filter)
- **joins** (merging tables)
  - *example*: `dt1 <- data.table(x = c('a', 'b', ...), y = 1:4)` = generates data table
    - `dt2 <- data.table(x= c('a', 'd', ...), z = 5:7)` = generates data table
    - `setkey(dt1, x); setkey(dt2, x)` = sets the keys for both data tables to be column x
    - `merge(dt1, dt2)` = returns a table, combine the two tables using column x, filtering to only the values that match up between common elements the two x columns (i.e. 'a') and the data is merged together
- **fast reading of files**
  - *example*: `big_df <- data.frame(rnorm(1e6), rnorm(1e6))` = generates data table
    - `file <- tempfile()` = generates empty temp file
    - `write.table(big_df, file=file, row.names=FALSE, col.names = TRUE, sep = "\t", quote = FALSE)` = writes the generated data from big.df to the empty temp file
    - `fread(file)` = read file and load data = much faster than `read.table()` ## Reading from

MySQL

- `install.packages("RMySQL"); library(RMySQL)` = load MySQL package
- free/widely used open sources database software, widely used for Internet base applications
- each row = record
- data are structured in databases → series tables (dataset) → fields (columns in dataset)
- `dbConnect(MySQL(), user = "genome", db = "hg19", host = "genome-mysql.cse.ucsc.edu")` = open a connection to the database
  - `db = "hg19"` = select specific database
  - `MySQL()` can be replaced with other arguments to use other data structures
- `dbGetQuery(db, "show databases;")` = return the result from the specified SQL query executed through the connection
  - any SQL query can be substituted here
- `dbDisconnect(db)` = disconnects the open connection
  - crucial to disconnect as soon as all queries are performed
- `dbListFields(db, "name")` = returns the list of fields (columns) from the specified table
- `dbReadTable(db, "name")` = returns the the specified table
- `query <- dbSendQuery(db, "query")` = send query to MySQL database and store it remotely
- `fetch(query, n = 10)` = executes query and returns the result
  - `n = 10` = returns the first 10 rows
- `dbClearResult(query)` = clears query from remote database, important
- `sqldf` package example ## HDF5

- `source("http://bioconductor.org/biocLite.R"); biocLite("rhdf5"); library(rhdf5)` (tutorial)
- used for storing large datasets, supports range of data types, can be used optimize reading/writing from disc to R
- **hierarchical format**
  - groups containing 0 or more datasets and metadata
    - each group has group header with group name and list of attributes
    - each group has group symbol table with a list of objection in the group
  - datasets = multidimensional array of data elements with metadata
    - each dataset has a header with name, datatype, data space, storage layout
    - each dataset has a data array (similar to data frame) with the data
- `h5createFile("filename")` = creates HDF5 file and returns TRUE/FALSE
- `h5createGroup("filename", "group1/subgroup1/...")` = creates group within the specified file
  - groups are created at the root `/` by default, but subgroups can be created by providing the path AFTER the parent group is created
- `h5ls("filename")` = prints out contents of the file by group, name, otype, etc
- `h5write(A, "filename", "groupname")` = writes A (could be array, matrix, etc.) to the file under the specific group
  - `h5write(A, "filename", "A")` = writes A directly at the top level
  - `h5write(values, "filename", "group/subgroupname/obj", index = list(1:3, 1))` = writes values in the specified obj at the specific location
    - *example*: `h5write(c(12, 13, 14), "ex.h5", "foo/A", index = list(1:3, 1))` = writes values 12, 13, 14 in the object A at the first 3 rows of the first column in the /foo group
- `h5read("filename", "groupname/A")` = reads A from specified group of the file
  - `h5read("filename", "A")` = reads A directly from the top level of the file
  - `h5read("filename", "group/subgroupname/obj", index = list(1:3, 1))` = writes specified values in the specified obj at the group within the file ## Web Scraping (tutorial)

- **webscraping** = programmatically extracting data from the HTML code of websites
- `con = url("website")` = opens connection from URL
- `htmlCode = readLines(con)` = reads the HTML code from the URL
  - always remember to `close(con)` after using it
  - the htmlCode return here is a bit unreadable
- **Parsing with XML**
  - `library(XML)`
  - `url <- "http://..."` = sets the desired URL as a character variable
  - `html <- htmlTreeParse(url, useInternalNodes = T)` = reads and parses the html code
  - `xpathSApply(html, "//title", xmlValue)` = returns the value of the `//title` node/element
  - `xpathSApply(html, "//td[@id='col-citedBy']", xmlValue)` = returns the value of the `//td` element where the `id = 'col-citedBy'` in the html code
- **Parsing with `httr` package** (tutorial)
  - `library(httr)`
  - `html2 <- GET(url)` = reads the HTML code from the URL
  - `cont = content(html2, as = "text")` = extracts the HTML code as a long string
  - `parsedHtml = htmlParse(cont, asText = TRUE)` = parses the text into HTML (same output as the XML package function htmlTreeParse)
  - `xpathSApply(html, "//title", xmlValue)` = returns the value of the //title node/element
  - accessing websites with passwords
    - `pg = GET("url")` = this would return a status 401 if the website requires log in without authenticating
    - `pg2 = GET("url", authenticate("username", "password"))` = this authenticates before attempting to access the website, and the result would return a status 200 if authentication was successful
    - `names2(pg2)` = returns names of different components
  - using handles (username/login information)
    - using handles allows you to save authentication across multiple parts of the website (only authenticate once for different requests)
    - *example*: `google = handle("http://google.com")`
    - `pg1 = GET(handle = google, path = "/")`
    - `pg2 = GET(handle = google, path = "search")` ## Working with API

- load `http` package first: `library(httr)`
  - allows GET, POST, PUT, DELETE requests if you are authorized
- `myapp = oath_app("app", key = "consumerKey", secret = "consumerSecret")` = start authorization process for the app
- `sig = sign_oauth1.0(myapp, token = "tokenGenerated", token_secret = "tokenSecret")` = login using the token information (sets up access so you can use it to get data)
- `homeTL = get("url", sig)` = use the established authentication (instead of username/password) to get the data (usually in JSON format)
  - use the url to specify what data you would like to get
  - use the documentation to get information and parameters for the url and data you have access to
- `json1 = content(homeTL)` = recognizes the data in JSON format and converts it to a structured R object [a bit hard to read]
- `json2 = jsonlite::fromJSON(toJSON(json1))` = converts data back into JSON format and then use the `fromJSON` function from the `jsonlite` package to read the data into a data frame
  - each row corresponds to a line of the data you received
- **GitHub example (tutorial)**:
  - `library(httr)`
  - `myapp <- oauth_app("github", key = "clientID", secret = "clientSecret")`
    - an application must be registered with GitHub first to generate the client ID and secrets
  - `github_token <- oauth2.0_token(oauth_endpoints("github"), myapp)`
    - `oauth_endpoints()` = returns the the authorize/access url/endpoints for some common web applications (GitHub, Facebook, google, etc)
    - `oauth2.0_token(endPoints, app)` = generates an oauth2.0 token with the credentials provided
  - `gtoken <- config(token = github_token)` = sets up the configuration with the token for authentication
  - `req <- with_config(gtoken, GET("https://api.github.com/rate_limit"))` = executes the configuration set to send a get request from the specified URL, and returns a response object
  - `library(jsonlite); json1 <- fromJSON(toJSON(content(req)))` = converts the content of the response object, to JSON format, and converts it again to data frame format
  - `names(json1)` = returns all the column names for the data frame
  - `json1[json1$name == "datasharing",]$created_at` = returns the create date for the data sharing repo ## Reading from Other Sources

- **interacting directly with files**
  - `file` = open a connection to a text file
  - `url` = opens a connection to a URL
  - `gzfile/bzfile` = opens a connection to a .gz/.bz2 file
  - `?connections` = for more information about opening/closing connections in R
- `foreign` **package**
  - loads data from Minitab/S/SAS/SPSS/Stat/Systat
  - basic functions
    - `read.arff` (Weka)
    - `read.dta` (Stata)
    - `read.mtp` (Minitab)
    - `read.octave` (Octave)
    - `read.spss` (SPSS)
    - `read.xport` (SAS)
    - `read.fwf` (fixed width files, [.for])
    - *example*: `data <- read.fwf(file = "quiz02q5.for", skip = 4, widths = c(-1, 9,-5, 4, 4, -5, 4, 4,-5, 4, 4,-5, 4, 4))`
    - `widths = c()` = specifies the width of each variable
    - the negative numbers indicate the space to disregard/take out
- **Other packages/functions**
  - `RPostresSQL` = provides DBI-compliant database connection from R
  - `RODBC` = provides interfaces to multiple databases including PostgreQL, MySQL, Microsoft Access, SQLite
  - `RMongo/rmongodb` = provides interfaces to MongoDb
    - similar to MySQL, except send queries in the database's syntax
  - reading images (functions)
    - `jpeg`, `readbitmap`, `png`, `EBImage` (Bioconductor)
  - reading (GIS Geographical Information Systems) data (packages)
    - `rdgal`, `rgeos`, `raster`
  - reading music data (packages)
    - `tuneR`, `seewave` ## dplyr

- external package, load by `library(dplyr)`
  - developed by Hadley Wickham of RStudio
  - optimized/distilled version of the `plyr` package, does not provide new functionality but greatly simplifies existing R functionality
  - very fast, many key operations coded in C++
  - `dplyr` package also works on data.table and SQL interface for relational databases (DBI package)
- load data into `tbl_df` (data frame table) by `data <- tbl_df(rawData)`
  - main advantage to using a `tbl_df` over a regular data frame is printing
  - more compact output/informative = similar to a combination of `head`/`str`
    - displays class, dimension, preview of data (10 rows and as many columns as it can fit), undisplayed variables and their class
- **functions**
  - ***Note****: for all functions, first argument always the data frame, and result is always a data frame*
  - `select()`
    - *example*: `select(dataFrameTable, var1, var2, var3)` = returns a table (similar in format as calling the actual data frame table)
    - no need to use `$` as we would normally, since `select()` understands that the variables are from the dataFrameTable
    - columns are returns in order specified
    - `:` operator (normally reserved for numbers) can be used to select a range of columns (from this column to that column), works in reverse order as well = `select(dataFrameTable, var1:var5)`
    - `"-column"` can be used to specify columns to throw away = `select(dataFrameTable, -var1)` = but this does not modify original dataFrameTable
    - `-(var1:size)` = eliminate all columns
    - normally this can be accomplished by finding the indices of names using the `match("value", vector)` function
  - `filter()`
    - *example*: `filter(cran, package == "swirl")` = returns a table (similar in format as calling the actual data frame table)
    - returns all rows where the condition evaluates to TRUE
    - automatically recognized that package is a column without `$`
    - able to specify as many conditions as you want, separated by `,`, `|` and `&` work here as well
    - multiple conditions specified by `,` is equivalent to `&`
    - `is.na(var1)` also works here
    - ***Note****: `?Comparison` brings up relevant documentation for relational comparators*
  - `arrange()`
    - *example*: `arrange(dataFrameTable, var)` = order the data frame table by specified column/variable
    - `desc(var)` = arrange in descending order by column value
    - can specify multiple values to sort by by using `,`
    - order listed in the call will be the order that the data is sorted by (can use in conjunction with `desc()`)
  - `rename()`
    - *example*: `rename(dataFrameTable, newColName = colName)` = renames the specified column with new name
    - capable of renaming multiple columns at the same time, no quotes needed
  - `mutate()`
    - create a new variable based on the value of one or more existing variables in the dataset
    - capable of modifying existing columns/variables as well
    - *example*: `mutate(dataFrameTable, newColumn = size / 2^20)` = create a new column with specified name and the method of calculating
    - multiple columns can be created at the same time by using `,` as separator, new variables can even reference each other in terms of calculation

  - `summarize()`
    * collapses the dataset into a single row
    * *example*: `summarize(dataFrameTable, avg = mean(size))` = returns the mean from the column in a single variable with the specified name
    * summarize(can return the requested value for each group in the dataset
  - `group_by()`
    * *example*: `by_package <- group_by(cran, package)` = creates a grouped data frame table by specified variable
    * `summarize(by_package, mean(size))` = returns the mean size of each group (instead of 1 value from the `summarize()` example above)
    * ***Note**: `n()` = counts number of observation in the current group*
    * ***Note**: `n_distinct()` = efficiently count the number of unique values in a vector*
    * ***Note**: `quantile(variable, probs = 0.99)` = returns the 99% percentile from the data*
    * ***Note**: by default, `dplyr` prints the first 10 rows of data if there are more than 100 rows; if there are not, it will print everything*
  - `rbind_list()`
    * bind multiple data frames by row and column
    * *example*: `rbind_list(passed, failed)`
- **Chaining/Piping**
  - allows stringing together multiple function calls in a way that is compact and readable, while still accomplishing the desired result
    * ***Note**: all variable calls refer to the tbl_df specified at the same level of the call*
  - `%>%` = chaining operator
    * ***Note**: `?chain` brings up relevant documentation for the chaining operator*
    * Code on the right of the operator operates on the result from the code on the left
    * `exp1 %>% exp2 %>% exp3 ...`
      · `exp1` is calculated first
      · `exp2` is then applied on exp1 to achieve a result
      · `exp3` is then applied to the result of that operation, etc.
    * ***Note**: the chaining aspect is done with the data frame table that is being passed from one call to the next*
    * ***Note**: if the last call has no additional arguments, `print()` for example, then it is possible to `leave()` off* ## tidyr

- `gather()`
  - gather columns into key value pairs
  - *example*: `gather(students, sex, count, -grade)` = gather each key (in this case named sex), value (in this case count) pair into one row
    - effectively translates to (columnName, value) with new names imposed on both = all combinations of column name and value
    - `-grade` = signifies that the column does not need to be remapped, so that column is preserved
    - `class1:class5` = can be used instead to specify where to gather the key values
- `separate()`
  - separate one column into multiple column
  - *example*: `separate(data = res, col = sex_class, into = c("sex", "class")` = split the specified column in the data frame into two columns
    - ***Note**: the new columns are created in place, and the other columns are pushed to the right*
    - ***Note**: `separate()` is able to automatically split non-alphanumeric values by finding the logical separator; it is also possible to specify the separator by using the `sep` argument*
- `spread()`
  - spread key-value pairs across multiple columns = turn values of a column into column headers/variables/new columns
  - *example*: `spread(students3, test, grade)` = splits "test" column into variables by using it as a key, and "grade" as values
    - ***Note**: no need to specify what the columns are going to be called, since they are going to be generated using the values in the specified column*
    - ***Note**: the value will be matched and split up according their alignment with the key ("test") = midterm, A*
- `extract_numeric()`
  - extract numeric component of variable
  - *example*: `extract_numeric("class5")` = returns 5
  - *example*: `mutate(class = extract_numeric(class))` = changes the class name to numbers only
- `unique()` = general R function, not specific to tidyr
  - returns a vector with the duplicates removed
- ***Note**: when there are redundant information, it's better to split up the info into multiple tables; however, each table should also contain primary keys, which identify observations and link data from one table to the next* ## lubridate

- consistent, memorable syntax for working with dates
- `wday(date, label = TRUE)` = returns number 1 - 7 representing Sunday - Saturday, or returns three letter day of the week if label = TRUE
- `today()`, `now()` = returns the current date and time, with extractable parts (hour(), month())
  - `tzone = "America/New_York"` = used to specify time zones (list here)
- `ymd("string")` = converts string in to year month day format to a POSIXct time variable
  - `mdy("string")` = parses date in month day year format
  - `dmy(2508195)` = parses date in day month year format using a number
  - `ymd_hms("string")` = parses the year month day, hour minute second
  - `hms("string")` = parses hour minute second
    - `tz = ""` = can use the "tz" argument to specify time zones (list here)
  - ***Note**: there are a variety of functions that are available to parse different formats, all of them are capable of converting the correct information if the order of month year day is correct*
  - ***Note**: when necessary, `//` or `-` should be added to provide clarity in date formatting*
- `update(POSIXct, hours = 8, minutes = 34, seconds = 55)` = updates components of a date time
  - ***Note**: does not alter the date time passed in unless explicitly assigned*
- arithmetic can be performed on date times by using the `days()` `hours()` `minutes()`, etc. functions
  - *example*: `now() + hours(5) + minutes(2)` = returns the date time for 5 hours and 2 minutes from now
- `with_tz(time, tone ="")` = return date-time in a different time zone
- `as.period(new_interval(last_time, arrive))` = return the properly formatted difference between the two date times ## Subsetting and Sorting

- **subsetting**
  - `x <- data.frame("var1" = sample(1:5), "var2" = sample(6:10), "var3" = (11:15))` = initiates a data frame with three names columns
  - `x <- x[sample(1:5),]` = this scrambles the rows
  - `x$var2[c(2,3)] = NA` = setting the 2nd and 3rd element of the second column to NA
  - `x[1:2, "var2"]` = subsetting the first two row of the the second column
  - `x[(x$var1 <= 3 | x$var3 > 15), ]` = return all rows of x where the first column is less than or equal to three or where the third column is bigger than 15
  - `x[which(x$var2 >8), ]` = returns the rows where the second column value is larger than 8
    - ***Note**: `which(condition)` = useful in dealing with NA values as it returns the indices of the values where the condition holds true (returns FALSE for NA)*
- **sorting/ordering**
  - `sort(x$var1)` = sort the vector in increasing/alphabetical order
    - `decreasing = TRUE` = use decreasing argument to sort vector in decreasing order
    - `na.last = TRUE` = use na.last argument to sort the vector such that all the NA values will be listed last
  - `x[order(x$var1, x$var2), ]` = order the x data frame according to var1 first and var2 second
  - `plyr` package: `arrange(data.farme, var1, desc(var2)` = see dplyr sections
- **adding row/columns**
  - `x$var4 <-rnorm(5)` = adds a new column to the end called var4
  - `cbind(X, rnorm(5))` = combines data frame with vector (as a column on the right)
    - `rbind()` = combines two objects by putting them on top of each other (as a row on the bottom)
    - ***Note**: order specified in the argument is the order in which the operation is performed ##*

Summarizing Data

- `head(data.frame, n = 10) / tail(data.frame, n = 10)` = prints top/bottom 10 rows of data
- `summary(data.frame)` = displays summary information
  - for factors variables, the summary table will display count of the top 6 values
  - for numeric variables, the summary table will display min, 1st quantile, median, mean, 3rd quantile, max
- `str(data.frame)` = displays class of the object, dimensions, variables (name, class, preview of data)
- `quantile(variable, na.rm = TRUE, probs = c(0.5, 0.75, 0.9))` = displays the specified quantile of the variable
  - default returns 0, .25, .5, .75, 1 quantiles
- `table(variable, useNA = "ifany")` = tabulates the values of the variable by listing all possible values and the number of occurrences
  - `useNA = "ifany"` = this will add another column if there are any NA values in the variable and displays how many as well
  - `table(var1, var2)` = tabulate the data against each other to see if theres an relationship between them
- **check for missing values**
  - `sum(is.na(variable))` = TRUE = 1, FALSE = 0, so if this sum = 0, then there's no missing values
  - `any(is.na(variable))` = returns TRUE/FALSE of if there is any NA values in the variable
  - `all(variable >0)` = check all values of a variable against some condition and return TRUE/FALSE
- **row/column sums**
  - `colSums/rowSums(is.na(data.frame))` = column sum of is.na check for every column; works the exact same way with rowSums
- **values with specific characteristics**
  - `table(data.frame$var1 %in% c("str1", "str2"))` = returns a FALSE/TRUE table that counts how many values from the data frame variable contains the specified values in the subsequent vector
  - `x[x$var1 %in% c("str1", "str2"), ]` = subsets rows from the data frame where the var1 == str1 or str2
- **cross tabs**
  - `xt <- xtabs(Freq ~ Gender + Admit, data = data.frame)` = displays a cross table of Gender and Admit variables, where the values of frequency is displayed
  - `xt2 <- xtabs(var1 ~ ., data = data.frame)` = cross-tabulate variable 1 with all other variables, creates multiple two dimensional tables
  - `ftable(xt2)` = compacts the different tables and prints out a more compacted version | | Admitted | Rejected | ———|———-|———-| | Male | 1198 | 1493 | | Female | 557 | 1278 |
- **size of data set**
  - `object.size(obj)` = returns size of object in bytes
  - `print(object.size(obj), units = "Mb"` = prints size of object in Mb ## Creating New

Variables

- **sequences**
  - `s <- seq(1, 10, by = 2)` = creates a sequence from 1 to 10 by intervals of 2
  - `length = 3` = use the length argument to specify how many numbers to generate
  - `seq(along = x)` = create as many elements as vector x
- **subsetting variables**
  - `restData$nearMe = restData$neighborhood %in% c("Roland", "Homeland")` = creates a new variable `nearMe` that returns TRUE if the neighborhood value is Roland or Homeland, and FALSE otherwise
- **binary variables**
  - `restData$zipWrong = ifelse(restData$zipCode<0, TRUE, FALSE)` = creates a new variable `zipWrong` that returns TRUE if the `zipcode` is less than 0, and FALSE otherwise
  - `ifelse(condition, result1, result2)` = this function is the same as a if-else statement
- **categorical variables**
  - `restData$zipGroups = cut(restData$zipCode, breaks = quantile(restData$zipCode)` = creates new variable `zipGroups` that specify ranges for the zip code data such that the observations are divided into groups created by the quantile function
    - `cut(variable, breaks)` = cuts a variable/vector into groups at the specified breaks
    - ***Note**: class of resultant variable = factor*
    - `quantile(variable)` = returns 0, .25, .5, .75, 1 by default and thus provides for ranges/groups for the data to be divided in
  - using `Hmisc` package
    - `library(Hmisc)`
    - `restData$zipGroups = cut2(restData$zipCode, g = 4)`
    - `cut2(variable, g=4)` = automatically divides the variable values into 4 groups according the quantiles
    - ***Note**: class of resultant variable = factor*
- **factor variables**
  - `restData$zcf <- factor(restData$zipCode)` = coverts an existing vector to factor variable
    - `levels = c("yes", "no")` = use the levels argument to specify the order of the different factors
    - ***Note**: by default, converting variables to the factor class, the levels will be structured alphabetically unless otherwise specified*
  - `as.numeric(factorVariable)` = converts factor variable values into numeric by assigning the lowest (first) level 1, the second lowest level 2, . . . , etc.
- **category + factor split**
  - using `plyr` and `Hmisc` packages
  - `library(plyr); library(Hmisc)`
  - `readData2 <- mutate(restData, zipGroups = cut2(zipCode, g = 4)`
    - this creates `zipGroups` and splits the data from `zipCode` all at the same time
- **common transforms**
  - `abs(x)` = absolute value
  - `sqrt(x)` = square root
  - `ceiling(x)`, `floor()` = round up/down to integer
  - `round(x, digits = n)` = round to the number of digits after the decimal point
  - `signif(x, digits = n)` = round to the number of significant digits
  - `cos(x)`, `sin(x)`, `tan(x)` ... etc = trigonometric functions
  - `log(x)`, `log2(x)`, `log10(x)` = natural log, log 2, log 10
  - `exp(x)` = exponential of x ## Reshaping Data

- **melting data frames**
  - `library(reshape2)` = loads the reshape2 package
  - `mtcars$carname <- rownames(mtcars)` = takes the row names/name of each observation and makes a new variable called "carname"
  - `carMelt <- melt(mtcars, id=c("carname", "gear", "cyl"), measure.vars = c("mpg", "hp"))` = converts dataframe into a castable melted data frame by reshaping the data
    - ID variables and measured variables are defined separately through `id` and `measure.vars` arguments
    - ID variables (identifiers) are kept in rows, while all measured variables have been split into variable and value columns
    - variable column = `mpg`, `hp` to qualify for the corresponding value column
    - value column = contains the numeric value from previous measured variable columns like `mpg` `hp`
    - ID variables are repeated when a new measured variable begins such that each row is an unique observation (long/tall table)
- **casting data frames**
  - `cylData <- dcast(carMetl, cyl ~ variable)` = tabulate the data by rows (left hand side variable, cyl in this case) by columns (right hand side variable, variable in this case), so this is a table of cylinder vs mpg and hp
    - by default, `dcast()` summarizes the data set by providing the length argument (count)
    - can add argument (mean) to specify the value of the table produced
- **calculating factor sums**
  - **method 1**: `tapply(InsectSprays$count, InsectSpray$spray, sum)` = splits the InsectSpray count values by spray groups and calculates the sum of each group
  - **method 2**: split-apply-combine
    - `s <- split(InsectSprays$count, InsectSpray$spray)` = splits InsectSpray count values into groups by spray
    - `sprCount <- lapply(s, sum)` = apply sum for all of the groups and return a list
    - `unlist(sprCount)` = converts a list into a vector with names
    - `sapply(s, sum)` = apply sum for all of the groups and return a vector
  - **method 3**: plyr package
    - `ddply(dataframe, .(variables), method, function)` = for each subset of a data frame, apply function then combine results into a data frame
    - `dataframe` = data being processed
    - `.(variables)` = variables to group/summarize by
    - `method` = can be a variety of different functions defined within the ply package, mutate, summarize, arrange, filter, select, etc.
    - `function` = how the data is going to be calculated
    - *example*: `ddply(InsectSprays, .(spray), summarize, sum = sum(count))` = summarize spray groups by providing the same of counts for each group
    - `spraySums<- ddply(InsectSprays, .(spray), summarize, sum = ave(count, FUN = sum))` = creates a data frame (2 columns) where each row is filled with the corresponding spray and sum (repeated multiple times for each group)
    - the result can then be used and added to the dataset for analysis ## Merging Data

- `merge(df1, df2, by/by.x/by.y, all = TRUE)` = merges two data frames
  - `df1`, `df2` = data frames to be merged
  - `by = "col1"/c("col1", "col2")` = merge the two data frames by columns of the specified names
    - ***Note**: if `by` argument is not specified, the two data frames will be merged with all columns with the same name [default behavior]*
    - ***Note**: columns must have the same names for this to work*
  - `by.x/by.y = "col.x"/"col.y"` = specifies which columns from x and y should be used to perform the merge operation
  - `all = TRUE` = if there are values that exist in one but not the other, new rows should be created with NA as values for the missing data
- `plyr` package: `join(df1, df2)` = merges the columns by columns in common
  - faster but only works with columns that have the same name
  - `dfList = list(df1, df2, df3); join_all(dfList)` = joins together a list of data frames using the common columns ## Editing Text Variables

- `tolower()` = make all character values lowercase letters
- `toupper()` = make all character values uppercase letters
- `strsplit(value, "\\.")` = splits string into character vector by specified separator
  - ***Note**: `\\` must be added for the reserved operators in R*
- `sapply(list, function)` = can specify custom functions to return the part of the character desired
  - *example*: `fElement <- function(x){x[1]}; sapply(vector, fElement)`
- `sub(pattern, replacement, object)` = replaces the first occurrence of pattern and replaces it with the replacement string
  - *example*: `sub("_", "", nameVector)` = removes first "_" from the character vector
- `gsub(pattern, replacement, object)` = replaces all occurrences of the specified pattern and replaces it with the replacement string
- `grep("text", object, value = FALSE)` = searches through object to return the indices of where the text is found
  - `value = TRUE` = returns the values instead of the indices
  - ***Note**: `grep()` returns `integer(0)` if no value appears (length of the result = 0)*
- `grepl("text", object)` = searches through object and returns the T/F logical vector for if the text has been found
  - *example*: `data2 <- data1![grepl("test", data$intersection), ]`
- `string` package [`library(stringr)`]
  - `nchar(object/string)` = returns number of characters in each element of object/string, works on matrix/data frames as well
  - `substr("text", 1, 7)` = returns a substring of the specified beginning and ending characters
    - ***Note**: R is uses 1 based indexing system, which means the first character of the string = 1*
    - ***Note**: substring returns a string that includes both first AND last letters and everything inbetween*
  - `paste("str1", "str2", sep = " ")` = combines two strings together into one by using the specified separator (`default = " "`)
  - `paste0("str1", "str2")` = combines series of strings together with no separator
  - `str_trim("  text   ")` = trims off whitespace from start and end of string
- **General Rules**
  - name of variables should be
    - all lowercase when possible
    - descriptive
    - unique
    - contains no underscores/dots/space
  - variables with character values
    - made into factor variables
    - descriptive
    - use TRUE/FALSE instead of 1/0
    - use Male/Female instead of 1/0 or M/F ## Regular Expressions

- **RegEx** = combination of literals and metacharacters
- used with `grep/grepl/sub/gsub` functions or any other that involve searching for strings in character objects/variables
- `^` = start of the line (metacharacter)
  - *example*: `^text` matches lines such as "text ..."
- `$` = end of the line (metacharacter)
  - *example*: `text$` matches lines such as "... text"
- `[]` = set of characters that will be accepted in the match (character class)
  - *example*: `^[Ii]` matches lines such as "I ..." or "i ..."
- `[0-9]` = searches for a a range of characters (character class)
  - *example*: `[a-zA-Z]` will match any letter in upper or lower case
- `[^?.]` = when used at beginning of character class, "^" means not (metacharacter)
  - *example*: `[^?.]$` matches any line that does end in "." or "?"
- `.` = any character (metacharacter)
  - *example*: `9.11` matches 9/11, 9911, 9-11, etc
- `|` = or, used to combine subexpressions called alternatives (metacharacter)
  - *example*: `^([Gg]ood | [Bb]ad)` matches any lines that start with lower/upper "Good..." and "Bad ..."
  - ***Note**: () limits the scope of alternatives divided by "|" here*
- `?` = expression is optional = 0/1 of some character/expression (metacharacter)
  - *example*: `[Gg]eorge( [Ww]\.)?` [Bb]ush matches "george bush", "George W. Bush"
  - ***Note**: "" was added before"." because "." is a metacharacter, "." called escape dot, tells the expression to read it as an actual period instead of an operator*
- `*` = any number of repetition, including none = 0 or more of some character/expression (metacharacter)
  - *example*: `.*` matches anything combination of characters
  - ***Note**: `*` is greedy = always matches the longest possible string that satisfies the regular expression*
    - greediness of `*` can be turned off with the `?`
    - *example*: `s.*?s` matches the shortest "s...s" text
- `+` = 1 or more repetitions = 1 or more of some character/expression (metacharacter)
  - *example*: `[0-9]+` matches matches many at least digit 1 numbers such as "0", "90", or "021442132"
- `{m, n}` = interval quantifier, allows specifying the minimum and maximum number of matches (metacharacter)
  - `m` = at least, `n` = not more than
  - `{m}` = exactly `m` matches
  - `{m, }` = at least m matches
  - *example*: `Bush( +[^ ]+ +){1, 5}` debates matches "Bush + (at least one space + any word that doesn't contain space + at least one space) this pattern repeated between 1 and 5 times + debates"
- `()` = define group as the the text in parentheses, groups will be remembered and can be referred to by `\1`, `\2`, etc.
  - *example*: `([a-zA-Z]+) +\1 +` matches "any word + at least one space + the same word repeated + at least one space" = "night night", "so so", etc. ## Working with Dates

- `date()` = returns current date in character format
- `Sys.Date()` = returns the current date in Date format
- `format(object, "format")` = formats object in specified format
  - when object = Date, then we can use any combination of the following
    - `%d` = day as number (0-31)
    - `%a` = abbreviated weekday
    - `%A` = unabbreviated weekday
    - `%m` = month (00-12
    - `%b` = abbreviated month
    - `%B` = unabbreviated month
    - `%y` = 2 digit year
    - `%Y` = 4 digit year
  - *example*: `format(Sys.Date(), "%a %b %d")` = returns "Sun Jan 18"
- `as.Date("character", "format")` = converts character vector/variable into Date format by using the codes above
  - *example*: `z <- as.Date("1jan1960", "%d%b%Y")` = creates a Date of "1960-01-01"
- `Date1 - Date2` = prints the difference between the dates in this format "Time difference of n days"
  - `as.numeric()` on this result will print/store n, the numeric difference
- `weekdays(Date)`, `months(Date)` = returns the weekday/month of the given Date object
- `julian(Date)` = converts the Date, which is the number of days since the origin
  - `attr(, "origin")` = prints out the origin for the julian date format, which is 1970-01-01
- `lubridate` package [`library(lubridate)`] = see lubridate section
  - `?Sys.timezone` = documentation on how to determine/set timezones ## Data Sources

- `quantmod` package = get historical stock prices for publicly traded companies on NASDAQ or NYSE