Getting and Cleaning Data Week 1

Course Description

Before you can work with data you have to get some. This course will cover the basic ways that data can be obtained. The course will cover obtaining data from the web, from APIs, and from colleagues in various formats including raw text files, binary files, and databases. It will also cover the basics of data cleaning and how to make data tidy. Tidy data dramatically speed downstream data analysis tasks. The course will also cover the components of a complete data set including raw data, processing instructions, codebooks, and processed data. The course will cover the basics needed for collecting, cleaning, and sharing data.

- Data Collection
  - Raw files (.csv, .xlsx)
  - Databases (mySQL)
  - APIs

- Data Formats
  - Flat files (.csv, .txt)
  - XML
  - JSON

- Making Data Tidy

- Distributing data

- Scripting for data cleaning

Obtaining Data Motivation

- Basic ideas behind getting data ready for analysis
  - Finding and extracting raw data
  - tidy data principles and how to make data tidy
  - Practical implementation through a range of R packages

- Prerequisite courses:
  - The Data Scientist's Toolbox
  - R Programming

- Other useful courses:
  - Exploratory Analysis
  - Reporting Data and Reproducible Research

Goal of this course:
Focus on the first three stages:
raw data -> **Processing script** -> **tidy data** -> data analysis -> data communication

Raw and Processed Data

Raw data can be different, according to who you're speaking to.
"Data are values of qualitative or quantitative variables, belonging to a set of items."

Raw Data

- 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, depending on the field your work in.
- **All steps should be recorded.**

Components of tidy data

Four things you should have when you finished going from the raw data to a tidy data set:

- the raw data.
- a tidy data set
- a code book describing each variable and its values in the tidy data set (often called metadata)
- an explicit and exact recipe you used to go from 1 to 2 and 3. (in this case it will be recorded as an R script)

The Raw Data should be the rawest form of the data that you had access to.

- You ran no software on the data.
- You did not manipulate any numbers in the data set
- you did not remove any data from the data set
- you did not summarise the data set in any way

The tidy data is your objective

- each variable you measure is in one column
- each observation should be in a different row
- there should be one table for each "kind" of variable (eg: data from twitter, fb, etc, one table for each)
- if there are multiple tables, they should include a column in the table that allows them to be linked together.

include variable names if possible, and make them human-readable. in general, data should be saved in one file per table

The code book

- Information about the variables in the data set, not contained in the tidy data.
- Information about the summary choices made. (eg. mean or median)
- Information about the experimental study design used.

often written in Word or text file, or Rmarkdown
include a section called "Study design" that has a thorough description of how you collected the data
include a section called "code book" that describes all variables and units

Instruction list

- ideally a computer script (in R or python)
- the input for the script is the raw data
- the output is the processed, tidy data
- there are no parameters to the script -> exact recipe
  Needs to be reproducible

In some cases, it will not be possible to script every step. In that case you should provide instructions like:

- Step 1: take the raw file, run version 3.1.2 of *summarize software* with parameters a=1, b=2, c=3
- step 2: run software separately for each sample
- setp 3 - take column three of outputfile.out for each sample and that is the corresponding row in the output data set.
  very detailed instructions. **REPRODUCIBILITY**

Downloading files with R

You might use R to download files, so that the downloading process is included in the processing script.

- A basic component of working with data is knowing your working directory.

- the two main commands are `getwd()` and `setwd()`.

- Be aware of the relative versus absolute paths:
  - **relative** `setwd("../")`
  - **absolute** `setwd("C:/Users/RudyR/OneDrive/Desktop")`

- Important difference in Windows: `setwd(C:\\users\\RudyR\\Desktop)`

Checking for and creating directories

- `file.exists("directoryName")`

- `dir.create("directoryName")`

```
if(!file.exists("../data")) {
    dir.create("../data")
}  ## creates a folder called data, if it doesn't already exist.
```

Getting data from the internet

- `download.file()` command.

- Even if you could do this by hand, using the command helps with the reproducibility.

- important parameters are `url`, `destfile` and `method`.

- useful for downloading tab-delimited, csv and other files.

```
fileURL <- "https://www.stats.govt.nz/assets/Uploads/Household-expenditure-statistics/Household-expendi
destinationfile <- "C:/Users/RudyR/Desktop/R Projects/Rudys_First_Project/data/HouseholdExpenditure2019
## download.file(fileURL, destfile = destinationfile)
list.files("C:/Users/RudyR/Desktop/R Projects/Rudys_First_Project/data/")
```

```
## [1] "cameras.csv"                 "cameras.xlsx"
## [3] "EducationandEthnic.xls"      "HouseholdExpenditure2019.csv"
## [5] "ravens.html"                 "simple.xml"
```

```
dateDownloaded <- date()
dateDownloaded
```

```
## [1] "Wed Apr 28 13:43:33 2021"
```

- method `curl` needs to be specified for https websites when downloading on a Mac. Not needed on windows.

- it's important to keep track of the date in which it was downloaded, as the file online could be updated.

Notes

- if the url starts with *http*, you can use download.file()
- if it starts with *https*, on windows you may be ok.
- if it starts with *https*, on Mac you need to set `method = "curl"`
- if the file is big, this might take a while. You may set it up so the file doesn’t re-download every time you run the code.
- be sure to record the date when you downloaded it.

Reading local files

- repeat of content covered in R programming course

```
str(read.table)
```

```
## function (file, header = FALSE, sep = "", quote = "\"'", dec = ".", numerals = c("allow.loss",
##     "warn.loss", "no.loss"), row.names, col.names, as.is = !stringsAsFactors,
##     na.strings = "NA", colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
##     fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
##     comment.char = "#", allowEscapes = FALSE, flush = FALSE, stringsAsFactors = default.stringsAsFac
##     fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)
```

- `read.table()` is the main function, but requires a lot of parameters.
- there are faster methods
- reads the data into RAM, can cause problems for large sets.
- related to `read.csv()` and `read.csv2()`

```
destinationfile <- "C:/Users/RudyR/Desktop/R Projects/Rudys_First_Project/data/HouseholdExpenditure2019
ExpenditureData <- read.table(destinationfile, sep = ",", header = TRUE)
head(ExpenditureData)
```

```
##   MsCode     HECCode   Estimate RSE Flag
## 1   M003 Total Gross 2441552500  NA    R
## 2   M003   Total Net 2370610600  NA    R
## 3   M003          01  410549200  NA    R
## 4   M003        01.1   41365600  NA    R
## 5   M003     01.1.01   16868800  NA    R
## 6   M003   01.1.01.1    2116800  NA    R
```

- `read.csv()` has defaults `sep = ","` and `header = TRUE`.
- other parameters:
  - `quote` to tell R about quoted values. `quote=""` means no quotes.
  - `na.strings` sets the character that represents a missing value.
  - `nrows` how many rows to read the file.
  - `skip` how many lines to skip before starting to read.

Quotation marks ' or " in data values can cause trouble for reading the data. Setting `quote=""` often resolves this.

Reading Excel Files

Use packages to read excel files, such as *xlsx* (requires java), or *readxl* package.

```
## library(readxl)
str(read_xlsx)  ## many optional arguments
```

```
## function (path, sheet = NULL, range = NULL, col_names = TRUE, col_types = NULL,
##     na = "", trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(1000,
##         n_max), progress = readxl_progress(), .name_repair = "unique")
```

```
cameras <- read_xlsx("../data/cameras.xlsx")
head(cameras)
```

```
## # A tibble: 6 x 6
##   address        direction street   crossStreet   intersection     `Location 1`
##   <chr>          <chr>     <chr>    <chr>         <chr>            <chr>
## 1 S CATON AVE ~ N/B       Caton A~ Benson Ave    Caton Ave & Be~ (39.2693779962,~
## 2 S CATON AVE ~ S/B       Caton A~ Benson Ave    Caton Ave & Be~ (39.2693157898,~
## 3 WILKENS AVE ~ E/B       Wilkens~ Pine Heights Wilkens Ave & ~ (39.2720252302,~
## 4 THE ALAMEDA ~ S/B       The Ala~ 33rd St       The Alameda  &~ (39.3285013141,~
## 5 E 33RD ST & ~ E/B       E 33rd   The Alameda   E 33rd  & The ~ (39.3283410623,~
## 6 ERDMAN AVE &~ E/B       Erdman   Macon St      Erdman  & Maco~ (39.3068045671,~
```

```
 ## other
 ## write.xlsx function available in xlsx package
 ##
 ## XLConnect package has even more options for writing and manipulating
```

- *xlsx* package include options to read only specific columns. *readxl* can only select by rows.
- `write.xlsx` function available in *xlsx* package
- `read.xlsx2` is faster in large files.
- *XLConnect* package has even more options for writing and manipulating Excel files. Read the vignette if starting to use that package.
- in general it is advised to store your data in either a database, or in `.csv` or tab separated files `.tab` or `.txt`, as they are easier to distribute.

Reading XML

- *Extensible markup language*

- used to store stuctured data
- particularly used in internet applications
  - Markup: labels that give the text structure
  - Content: the actual text of the document

Tags, elements and attributes

- Tags are general labels. E.g.: `<section>` and `</section>` to start and end a section, or `<line-break />` is an empty tag.
- Elements are specific examples of tags. E.g.: `<Greeting>` *Hello* `</Greeting>`
- Attributes are components of the label. E.g.: ‘
- Very similar to html

Using the XML package

```
## library(XML)
fileURL <- "http://www.w3schools.com/xml/simple.xml"
## download.file(fileURL, destfile = "../data/simple.xml")
doc <- xmlTreeParse("../data/simple.xml", useInternalNodes = TRUE)
    ## loads the document into RAM in a way R can access it. Still a
    ## structured object, though.

rootNode <- xmlRoot(doc)  ## accesses that particular element
xmlName(rootNode)  ## accesses the name
```

```
## [1] "breakfast_menu"
```

```
names(rootNode) ## accesses all names in the structure
```

```
##   food   food   food   food   food
## "food" "food" "food" "food" "food"
```

```
rootNode[[1]]  ## access elements of the structured object, as if it's a list.
```

```
## <food>
##   <name>Belgian Waffles</name>
##   <price>$5.95</price>
##   <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
##   <calories>650</calories>
## </food>
```

```
rootNode[[1]][[1]]  ## first subcomponent of the first subcomponent.
```

```
## <name>Belgian Waffles</name>
```

```
xmlSApply(rootNode, xmlValue)
```

```
##
##                                "Belgian Waffles$5.95Two of our famous Belgian Waffles with plenty of
##
##                     "Strawberry Belgian Waffles$7.95Light Belgian waffles covered with strawberries
##
## "Berry-Berry Belgian Waffles$8.95Light Belgian waffles covered with an assortment of fresh berries
##
##                                                  "French Toast$4.50Thick slices made from our homemade
##
##                          "Homestyle Breakfast$6.95Two eggs, bacon or sausage, toast, and our ever-popu
```

```
    ## similar to sapply, loops through the parseXML document and applies the
    ## function xmlValue. Returns all the values separated only at the top level
    ## node: "food".
```

Using the ***`XPath`****, included in XML package*
Requires a whole new language, as well as learning XML, but a superficial understanding can already yield good results.
More information on this link.

- `/node` Top level node
- `//node` node at any level
- `node[@attr-name]` Node with an attribute name
- `node[@attr-name='bob']` Node with attribute name attrname='bob'

```
## applies xmlValue to the nodes called "name"
xpathSApply(rootNode, "//name", xmlValue)
```

```
## [1] "Belgian Waffles"             "Strawberry Belgian Waffles"
## [3] "Berry-Berry Belgian Waffles" "French Toast"
## [5] "Homestyle Breakfast"
```

```
 ## turning it into a matrix
y <- xpathApply(rootNode,
                c("//name","//price","//description","//calories"),
                xmlValue)
matrix(y,nrow = 4, ncol = 5)
```

```
##      [,1]
## [1,] "Belgian Waffles"
## [2,] "$5.95"
## [3,] "Two of our famous Belgian Waffles with plenty of real maple syrup"
## [4,] "650"
##      [,2]
## [1,] "Strawberry Belgian Waffles"
## [2,] "$7.95"
## [3,] "Light Belgian waffles covered with strawberries and whipped cream"
## [4,] "900"
##      [,3]
## [1,] "Berry-Berry Belgian Waffles"
## [2,] "$8.95"
## [3,] "Light Belgian waffles covered with an assortment of fresh berries and whipped cream"
## [4,] "900"
##      [,4]
## [1,] "French Toast"
## [2,] "$4.50"
## [3,] "Thick slices made from our homemade sourdough bread"
## [4,] "600"
##      [,5]
## [1,] "Homestyle Breakfast"
## [2,] "$6.95"
## [3,] "Two eggs, bacon or sausage, toast, and our ever-popular hash browns"
## [4,] "950"
```

Reading an HTML file

```r
fileUrl <- "https://www.espn.com/nfl/team/schedule/_/name/bal/year/2016"
## download.file(fileUrl, destfile = "../data/ravens.html")
doc <- htmlTreeParse("../data/ravens.html", useInternal=TRUE)
scores <- xpathSApply(doc, "//span[@class='ml4']", xmlValue)
scores
```

```
##  [1] "20-13  "   "17-3  "    "38-6  "    "33-16  "   "34-20  "   "31-17  "  
##  [7] "27-3  "    "30-28  "   "28-24  "   "24-10  "   "23-17  "   "30-24 OT "
## [13] "19-14  "   "34-17  "   "47-42  "   "40-14  "   "27-13  "   "38-3  "   
```

```r
## getting the team names is harder as the <span> tag doesn't have any
## attributes to call it from. But you can select a more generic tag and build
## a matrix, then subsetting.
a <-  xpathSApply(doc, "//a[@class='AnchorLink']", xmlValue)
a ## all "a" tags with class = "AnchorLink", gives us the main table of results
```

```
##   [1] ""              "Tennessee "    "20-13  "       "Jackson "     
##   [5] "Jackson "      "Brown "        ""              "Buffalo "     
##   [9] "17-3  "        "Jackson "      "Edwards "      "Brown "       
##  [13] ""              "Cleveland "    "38-6  "        "Jackson "     
##  [17] "Jackson "      "Brown "        ""              "Houston "     
##  [21] "33-16  "       "Jackson "      "Edwards "      "Brown "       
##  [25] ""              "Kansas City "  "34-20  "       "Jackson "     
##  [29] "Jackson "      "Dobbins "      ""              "Washington "  
##  [33] "31-17  "       "Jackson "      "Jackson "      "Brown "       
##  [37] ""              "Cincinnati "   "27-3  "        "Jackson "     
##  [41] "Ingram II "    "Brown "        ""              "Philadelphia "
##  [45] "30-28  "       "Jackson "      "Jackson "      "Brown "       
##  [49] ""              "Pittsburgh "   "28-24  "       "Jackson "     
##  [53] "Dobbins "      "Snead IV "     ""              "Indianapolis "
##  [57] "24-10  "       "Jackson "      "Jackson "      "Boyle "       
##  [61] ""               "New England "  "23-17  "       "Jackson "
##  [65] "Jackson "       "Snead IV "     ""              "Tennessee "
##  [69] "30-24 OT "      "Jackson "      "Dobbins "      "Andrews "
##  [73] ""               "Pittsburgh "   "19-14  "       "McSorley "
##  [77] "Griffin III "   "Brown "        ""              "Dallas "
##  [81] "34-17  "        "Jackson "      "Edwards "      "Brown "
##  [85] ""               "Cleveland "    "47-42  "       "Jackson "
##  [89] "Jackson "       "Andrews "      ""              "Jacksonville "
##  [93] "40-14  "        "Jackson "      "Dobbins "      "Brown "
##  [97] ""               "New York "     "27-13  "       "Jackson "
## [101] "Edwards "       "Andrews "      ""              "Cincinnati "
## [105] "38-3  "         "Jackson "      "Dobbins "      "Boykin "
## [109] ""               "Buffalo "      ""              "Dallas "
## [113] ""               "Carolina "     ""              "Washington "
```

```
matx <- matrix(a[1:108], nrow = 18, ncol = 6, byrow = TRUE)
colnames(matx) <- c("", "Opponents", "Result", "HI PASS", "HI RUSH", "HI REC")
matx ## Column 1 is empty
```

```
##          Opponents       Result      HI PASS     HI RUSH        HI REC
##  [1,] "" "Tennessee "    "20-13  "   "Jackson "  "Jackson "     "Brown "
##  [2,] "" "Buffalo "      "17-3  "    "Jackson "  "Edwards "     "Brown "
##  [3,] "" "Cleveland "    "38-6  "    "Jackson "  "Jackson "     "Brown "
##  [4,] "" "Houston "      "33-16  "   "Jackson "  "Edwards "     "Brown "
##  [5,] "" "Kansas City "  "34-20  "   "Jackson "  "Jackson "     "Dobbins "
##  [6,] "" "Washington "   "31-17  "   "Jackson "  "Jackson "     "Brown "
##  [7,] "" "Cincinnati "   "27-3  "    "Jackson "  "Ingram II "   "Brown "
##  [8,] "" "Philadelphia " "30-28  "   "Jackson "  "Jackson "     "Brown "
##  [9,] "" "Pittsburgh "   "28-24  "   "Jackson "  "Dobbins "     "Snead IV "
## [10,] "" "Indianapolis " "24-10  "   "Jackson "  "Jackson "     "Boyle "
## [11,] "" "New England "  "23-17  "   "Jackson "  "Jackson "     "Snead IV "
## [12,] "" "Tennessee "    "30-24 OT " "Jackson "  "Dobbins "     "Andrews "
## [13,] "" "Pittsburgh "   "19-14  "   "McSorley " "Griffin III " "Brown "
## [14,] "" "Dallas "       "34-17  "   "Jackson "  "Edwards "     "Brown "
## [15,] "" "Cleveland "    "47-42  "   "Jackson "  "Jackson "     "Andrews "
## [16,] "" "Jacksonville " "40-14  "   "Jackson "  "Dobbins "     "Brown "
## [17,] "" "New York "     "27-13  "   "Jackson "  "Edwards "     "Andrews "
## [18,] "" "Cincinnati "   "38-3  "    "Jackson "  "Dobbins "     "Boykin "
```

```
matx[ ,2] ## Opponent names
```

```
##  [1] "Tennessee "    "Buffalo "      "Cleveland "    "Houston "
##  [5] "Kansas City "  "Washington "   "Cincinnati "   "Philadelphia "
##  [9] "Pittsburgh "   "Indianapolis " "New England "  "Tennessee "
## [13] "Pittsburgh "   "Dallas "       "Cleveland "    "Jacksonville "
## [17] "New York "     "Cincinnati "
```

Read also the XML package tutorials, in the *Reading XML guides* folder.

Reading JSON files

- **Javascript Object Notation**

- Lightweight way of storing data

- commonly format for data from application programming interfaces.

- Similar to XML structure, but different sintax.

- Data is stored as :
  - Numbers (double)
  - Strings (double quoted)
  - Boolean
  - array (comma separated enclosed in quare brackets)
  - object (unordered, comma separated collection of key:value pairs in curly brackets)

best place to start is wikipedia.

Use the `jsonlite` package

```
## library(jsonlite)
jsonData <- fromJSON("https://api.github.com/users/jtleek/repos")
 ## object is a structured data frame.
names(jsonData) ## top level variables
```

```
##  [1] "id"                "node_id"           "name"
##  [4] "full_name"         "private"           "owner"
##  [7] "html_url"          "description"       "fork"
## [10] "url"               "forks_url"         "keys_url"
## [13] "collaborators_url" "teams_url"         "hooks_url"
## [16] "issue_events_url"  "events_url"        "assignees_url"
## [19] "branches_url"      "tags_url"          "blobs_url"
## [22] "git_tags_url"      "git_refs_url"      "trees_url"
## [25] "statuses_url"      "languages_url"     "stargazers_url"
## [28] "contributors_url"  "subscribers_url"   "subscription_url"
## [31] "commits_url"       "git_commits_url"   "comments_url"
## [34] "issue_comment_url" "contents_url"      "compare_url"
## [37] "merges_url"        "archive_url"       "downloads_url"
## [40] "issues_url"        "pulls_url"         "milestones_url"
## [43] "notifications_url" "labels_url"        "releases_url"
## [46] "deployments_url"   "created_at"        "updated_at"
## [49] "pushed_at"         "git_url"           "ssh_url"
## [52] "clone_url"         "svn_url"           "homepage"
## [55] "size"              "stargazers_count"  "watchers_count"
## [58] "language"          "has_issues"        "has_projects"
## [61] "has_downloads"     "has_wiki"          "has_pages"
## [64] "forks_count"       "mirror_url"        "archived"
## [67] "disabled"          "open_issues_count" "license"
## [70] "forks"             "open_issues"       "watchers"
## [73] "default_branch"
```

```
names(jsonData$owner) ## within the element "owner" was stored another data
```

```
##  [1] "login"               "id"                  "node_id"
##  [4] "avatar_url"          "gravatar_id"         "url"
##  [7] "html_url"            "followers_url"       "following_url"
## [10] "gists_url"           "starred_url"         "subscriptions_url"
## [13] "organizations_url"   "repos_url"           "events_url"
## [16] "received_events_url" "type"                "site_admin"
```

```
                      ## frame with these top level names
```

Writing JSON files

```
## Using the Iris dataset as an example
myjson <- toJSON(iris, pretty = TRUE)
## pretty = TRUE gives it readable indentation.
class(myjson)
```

```
## [1] "json"
```

```
iris2 <- fromJSON(myjson) ## getting it back as a data frame
head(iris2)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
head(iris)  ## data frame is identical to the original one
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

This link contains a good tutorial on jsonlight. Also, check the jsonlite vignette and www.json.org for further resources.

The data.table Package

Often faster and more memory efficient than data frames. All functions that accept data.frame work on data.table. Faster at subsetting, group and updating variables. Updated often.
Requires its own syntax.

```r
## library(data.table)
## can be used in the same way as data.frame
DF <- data.frame(x=round(rnorm(9),3),
                 y=rep(c("a","b","c"),each=3),
                 z=round(rnorm(9),3))
head(DF,3)
```

```
##        x y      z
## 1 -0.442 a  1.399
## 2  0.709 a  0.088
## 3  0.017 a -0.502
```

```r
DT <- data.table(x=round(rnorm(9),3),
                 y=rep(c("a","b","c"),each=3),
                 z=round(rnorm(9),3))
head(DT,3)
```

```
##         x y      z
## 1:  1.313 a -0.060
## 2: -1.157 a  0.914
## 3: -1.208 a -1.731
```

```r
tables() ## allows you to check all data tables in the memory
```

```
##    NAME NROW NCOL MB  COLS KEY
## 1:   DT    9    3  0 x,y,z
## Total: 0MB
```

Subsetting a data table

```r
## similar to data.frame
DT[2,] ## subset rows by index
```

```
##         x y     z
## 1: -1.157 a 0.914
```

```r
DT[DT$y=="b",] ## subset rows by logical vector
```

```
##         x y      z
## 1: -0.844 b -1.792
## 2:  1.127 b  1.250
## 3: -0.015 b -0.384
```

```r
## different:
## when using only one index, it subsets based on rows instead of columns
DT[c(2,3)]
```

```
##         x y      z
## 1: -1.157 a  0.914
## 2: -1.208 a -1.731
```

```
DF[c(2,3)]
```

```
##   y      z
## 1 a  1.399
## 2 a  0.088
## 3 a -0.502
## 4 b  0.818
## 5 b -0.351
## 6 b  0.311
## 7 c -0.799
## 8 c  0.353
## 9 c  0.408
```

Subsetting columns is different:

- the argument you pass after the comma is called an "expression".
- In R an expression is a collection of statements enclosed in curly brackets.
- example:

```
{
  x <- 1
  y <- 2
} ## one expression
k <- {print(10);5} ## print is 10 but value is 5
```

```
## [1] 10
```

```
print(k)
```

```
## [1] 5
```

```
## So for example, instead of putting an index in the subsetting, you can pass
## a list of functions you want to perform.
DT[,list(mean(x),sum(z))]
```

```
##             V1     V2
## 1: 0.02144444 -0.461
```

```
DT[,table(y)]
```

```
## y
## a b c
## 3 3 3
```

Adding new columns

```
DT[, w:=round(z^2,3)] ## use := to add the variable w to the data frame.
## Doing this with data frame can be very memory consuming, but it's not a
## problem in data.table
##
## But be careful if making a copy of the data table, as changes to the
## original will change the copies, too.
DT2 <- DT
DT[ , y:= 2]
head(DT2,3) ## DT2 is changed with it.
```

```
##          x y      z     w
## 1:  1.313 2 -0.060 0.004
## 2: -1.157 2  0.914 0.835
## 3: -1.208 2 -1.731 2.996
```

```
## overcome this by using the copy() function
DT3 <- copy(DT)
DT[ , y:= 4] ## change y to 4
head(DT3,3) ## not changed
```

```
##          x y      z     w
## 1:  1.313 2 -0.060 0.004
## 2: -1.157 2  0.914 0.835
## 3: -1.208 2 -1.731 2.996
```

Multiple step functions to create new variables

- each statement in an expression is followed by a “;”.

```
DT[ ,m:= {tmp <- (x+z); tmp2 <- log2(tmp+5); round(tmp2,2)}]
## assign temporary variable tmp, then use it.
head(DT, 3)
```

```
##          x y      z     w    m
## 1:  1.313 4 -0.060 0.004 2.64
## 2: -1.157 4  0.914 0.835 2.25
## 3: -1.208 4 -1.731 2.996 1.04
```

“plyr” like operations

- plyr is an R package that makes it simple to split data apart, do stuff to it, and mash it back together.

```
DT[ ,a:=x>0] ## column a is logical
head(DT,3)
```

```
##          x y      z     w    m     a
## 1:  1.313 4 -0.060 0.004 2.64  TRUE
## 2: -1.157 4  0.914 0.835 2.25 FALSE
## 3: -1.208 4 -1.731 2.996 1.04 FALSE
```

```
DT[, b:= mean(x+w),by=a] ## mean of rows where a == True is separated from
                         ## mean of rows where a == False
head(DT,3) ## this is pretty cool!
```

```
##          x y      z     w    m     a       b
## 1:  1.313 4 -0.060 0.004 2.64  TRUE 1.59125
## 2: -1.157 4  0.914 0.835 2.25 FALSE 0.67820
## 3: -1.208 4 -1.731 2.996 1.04 FALSE 0.67820
```

Special variables in data.table

- .N An integer, length 1, containing the number of rows
  - contains the number of times that a specific group appears

```
set.seed(123)
DT2 <- data.table(x=sample(letters[1:3], 1E5, replace = TRUE))
DT2[ , .N, by=x] ## count number of times / rows by x variable
```

```
##    x     N
## 1: c 33294
## 2: b 33305
## 3: a 33401
```

```
## calculates it faster than table(DT2$x)
```

Keys

Setting a key allows you to sort and subsett the data table much faster than you'd be able to on a data frame.

```
DT2 <- data.table(x=rep(letters[1:3],each=100), y=round(rnorm(300),3))
setkey(DT2,x) ## column x is set as the key
head(DT2['a'],6) ## subsetting automatically knows to search for the key
```

```
##    x      y
## 1: a  0.886
## 2: a  2.829
## 3: a  2.031
## 4: a  1.907
## 5: a  0.215
## 6: a -0.863
```

Joining data tables

- Use keys to facilitate joins

```
DT1 <- data.table(x=c("a","a","b","dt1"), y=1:4)
DT2 <- data.table(x=c("a","b","dt2"), z=5:7)
DT1
```

```
##      x y
## 1:   a 1
## 2:   a 2
## 3:   b 3
## 4: dt1 4
```

```
DT2
```

```
##      x z
## 1:   a 5
## 2:   b 6
## 3: dt2 7
```

```
setkey(DT1, x); setkey(DT2, x)
merge(DT1, DT2)  ## merge function uses the keys to organise the rows. Rows
```

```
##    x y z
## 1: a 1 5
## 2: a 2 5
## 3: b 3 6
```

```
##unique to each data table are excluded.
## much faster than merging with data frame.
```

Faster reading from file

- fread function is provided by the data.table package.
- stands for "fast read"
- substitute for read.table

```
big_df <- data.frame(x=round(rnorm(1E6),3),y=round(rnorm(1E6),3))
head(big_df, 3)
```

```
##        x      y
## 1 -0.047 -1.253
## 2 -0.765 -0.315
## 3  1.525 -1.177
```

```
file <- tempfile()  ## location of a new temporary file in temp files folder.
write.table(big_df, file=file, row.names = FALSE, col.names = TRUE,
            sep = "\t", quote = FALSE) ## writes the data frame in the temp file
system.time(fread(file))  ## time required to read the file
```

```
##    user  system elapsed
##    0.08    0.00    0.03
```

```
system.time(read.table(file, header=TRUE, sep="\t"))
```

```
##    user  system elapsed 
##    0.61    0.00    0.61
```

New functionalities are frequently being added to the data.table package.

- The latest version can be found here
- Here is a list of differences between `data.table` and `data.frame`
- Credit to Raphael Gottardo and Kevin Ushey

Swirl

Install Getting and Cleaning Data course with the command `install_from_swirl("Getting and Cleaning Data")`.

Manipulating Data with `dplyr` package

```
## The  advantage to using a tbl_df over a regular data frame is the printing.
mytibble
```

```
## # A tibble: 50 x 9
##    Name  Height  Time Confidence   Age Score    IQ Date       Country
##    <chr>  <dbl> <dbl>      <dbl> <dbl> <dbl> <dbl> <date>     <chr>  
##  1 Rbgc    1.87  37            6    51   4.8    93 2021-04-28 CAN    
##  2 Hrqw    1.25  36.2          3    44   8      92 2021-04-28 CAN    
##  3 Atio    1.42  24.5          8    63   7.6   107 2021-04-28 UK     
##  4 Lmsy    1.69  20.3         10    34   5.4    86 2021-04-28 AUS    
##  5 Onpo    1.44  31.3          2    51   4.6   100 2021-04-28 CAN    
##  6 Wksv    1.52  22            9    59   7.5   102 2021-04-28 AUS    
##  7 Glze    1.58  30.4          2    33   6.9   111 2021-04-28 CAN    
##  8 Mjxz    1.68  24            3    45   7.3   113 2021-04-28 UK     
##  9 Gimr    1.42  42.2          7    43   6     104 2021-04-28 UK     
## 10 Jhzw    1.8   30.4          8    32   5.1   105 2021-04-28 NZ     
## # ... with 40 more rows
```

First, we are shown the class and dimensions of the dataset. Just below that, we get a preview of the data. Instead of attempting to print the entire dataset, dplyr just shows us the first 10 rows of data and only as many columns as fit neatly in our console. At the bottom, we see the names and classes for any variables that didn't fit on our screen.

“The dplyr philosophy is to have small functions that each do one thing well.” Specifically, dplyr supplies five ‘verbs’ that cover most fundamental data manipulation tasks: `select()`, `filter()`, `arrange()`, `mutate()`, and `summarize()`.

select()

Select (and optionally rename) variables in a data frame, using a concise mini-language that makes it easy to refer to variables based on their name (e.g. a:f selects all columns from a on the left to f on the right).

```
y <- select(mytibble, Name, Country, Score)
## can rearrange the order of columns, no need to type datfratbl$Name, etc.
head(y,5)
```

```
## # A tibble: 5 x 3
##   Name  Country Score
##   <chr> <chr>   <dbl>
## 1 Rbgc  CAN       4.8
## 2 Hrqw  CAN       8
## 3 Atio  UK        7.6
## 4 Lmsy  AUS       5.4
## 5 Onpo  CAN       4.6
```

You can also use predicate functions like `is.numeric` to select variables based on their properties. Tidyverse selections implement a dialect of R where operators make it easy to select variables:

- `:` for selecting a range of consecutive variables.
- `-` to omit a column.
- `!` for taking the complement of a set of variables.
- `&` and | for selecting the intersection or the union of two sets of variables.
- `c()` for combining selections.
- `everything()` Matches all variables.
- `last_col()` Select last variable, possibly with an offset.
- `starts_with()` Starts with a prefix.
- `ends_with()` Ends with a suffix.
- `contains()` Contains a literal string.
- `matches()` Matches a regular expression.
- `num_range()` Matches a numerical range like x01, x02, x03.
- `all_of()` Matches variable names in a character vector. All names must be present, otherwise an out-of-bounds error is thrown.
- `any_of()` Same as all_of(), except that no error is thrown for names that don't exist.
- `where()` Applies a function to all variables and selects those for which the function returns TRUE.

```
y <- select(mytibble, Name:Confidence) ## using ":" to select columns
head(y,5)
```

```
## # A tibble: 5 x 4
##   Name  Height  Time Confidence
##   <chr>  <dbl> <dbl>      <dbl>
## 1 Rbgc    1.87  37            6
## 2 Hrqw    1.25  36.2          3
## 3 Atio    1.42  24.5          8
## 4 Lmsy    1.69  20.3         10
## 5 Onpo    1.44  31.3          2
```

```
y <- select(mytibble, -IQ)  ## using "-" to omit a column
head(y,5)
```

```
## # A tibble: 5 x 8
##   Name  Height  Time Confidence   Age Score Date       Country
##   <chr>  <dbl> <dbl>      <dbl> <dbl> <dbl> <date>     <chr>
## 1 Rbgc    1.87  37            6    51   4.8 2021-04-28 CAN
## 2 Hrqw    1.25  36.2          3    44   8   2021-04-28 CAN
## 3 Atio    1.42  24.5          8    63   7.6 2021-04-28 UK
## 4 Lmsy    1.69  20.3         10    34   5.4 2021-04-28 AUS
## 5 Onpo    1.44  31.3          2    51   4.6 2021-04-28 CAN
```

```
y <- select(mytibble, -(Score:Country))  ## mixing different strategies
head(y,5)
```

```
## # A tibble: 5 x 5
##   Name  Height  Time Confidence   Age
##   <chr>  <dbl> <dbl>      <dbl> <dbl>
## 1 Rbgc    1.87  37            6    51
## 2 Hrqw    1.25  36.2          3    44
## 3 Atio    1.42  24.5          8    63
## 4 Lmsy    1.69  20.3         10    34
## 5 Onpo    1.44  31.3          2    51
```

filter()

The `filter()` function is used to subset a data frame, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of `TRUE` for all conditions. Note that when a condition evaluates to `NA` the row will be dropped, unlike base subsetting with `[`.

```
y <- filter(mytibble, Height >= 1.8)  ## no need to use datfratbl$Height
head(y,5)
```

```
## # A tibble: 5 x 9
##   Name  Height  Time Confidence   Age Score    IQ Date       Country
##   <chr>  <dbl> <dbl>      <dbl> <dbl> <dbl> <dbl> <date>     <chr>
## 1 Rbgc    1.87  37            6    51   4.8    93 2021-04-28 CAN
## 2 Jhzw    1.8   30.4          8    32   5.1   105 2021-04-28 NZ
## 3 Pedh    1.91  23.7          1    42   5.7    96 2021-04-28 UK
## 4 Pimq    1.87  25.2          6    22   3.5    73 2021-04-28 AUS
## 5 Cdkb    1.83  13.6          6    19   6.8   116 2021-04-28 AUS
```

arrange()

Orders the rows of a data frame by the values of selected columns.

```
y <- arrange(mytibble, Name)
head(y,5)
```

```
## # A tibble: 5 x 9
##   Name  Height  Time Confidence   Age Score    IQ Date       Country
##   <chr>  <dbl> <dbl>      <dbl> <dbl> <dbl> <dbl> <date>     <chr>  
## 1 Andb    1.5   37.8         10    59   6.8   100 2021-04-28 UK     
## 2 Asma    1.64  33.1          5    48   3.4    85 2021-04-28 CAN    
## 3 Atio    1.42  24.5          8    63   7.6   107 2021-04-28 UK     
## 4 Cdkb    1.83  13.6          6    19   6.8   116 2021-04-28 AUS    
## 5 Daqa    1.5   44.4          8    45   6.7    99 2021-04-28 US     
```

```
y <- arrange(mytibble, Country, Name)
## arrange by Country, and on identical groups, arrange by name
head(y,5)
```

```
## # A tibble: 5 x 9
##   Name  Height  Time Confidence   Age Score    IQ Date       Country
##   <chr>  <dbl> <dbl>      <dbl> <dbl> <dbl> <dbl> <date>     <chr>  
## 1 Cdkb    1.83  13.6          6    19   6.8   116 2021-04-28 AUS    
## 2 Ebto    1.62  33.6          4    41   8.2   113 2021-04-28 AUS    
## 3 Hsnz    1.99  34.5          2    42   3.2    97 2021-04-28 AUS    
## 4 Lmsy    1.69  20.3         10    34   5.4    86 2021-04-28 AUS    
## 5 Okhj    1.44  29.2          6    61   6.6   109 2021-04-28 AUS    
```

mutate()

mutate() adds new variables and preserves existing ones; transmute() adds new variables and drops existing ones. New variables overwrite existing variables of the same name. Variables can be removed by setting their value to NULL.

```
y <- mutate(mytibble, Correct_Time = Time + 10) ## add 10 to Time in new column
head(select(y, -(Confidence:Date)),5)
```

```
## # A tibble: 5 x 5
##   Name  Height  Time Country Correct_Time
##   <chr>  <dbl> <dbl> <chr>          <dbl>
## 1 Rbgc    1.87  37   CAN             47  
## 2 Hrqw    1.25  36.2 CAN             46.2
## 3 Atio    1.42  24.5 UK              34.5
## 4 Lmsy    1.69  20.3 AUS             30.3
## 5 Onpo    1.44  31.3 CAN             41.3
```

```
y <- mutate(mytibble, Date = NULL) ## do not display Date column
head(y,5)
```

```
## # A tibble: 5 x 8
##   Name  Height  Time Confidence   Age Score    IQ Country
##   <chr>  <dbl> <dbl>      <dbl> <dbl> <dbl> <dbl> <chr>
## 1 Rbgc    1.87  37            6    51   4.8    93 CAN
## 2 Hrqw    1.25  36.2          3    44   8      92 CAN
## 3 Atio    1.42  24.5          8    63   7.6   107 UK
## 4 Lmsy    1.69  20.3         10    34   5.4    86 AUS
## 5 Onpo    1.44  31.3          2    51   4.6   100 CAN
```

```
## merge data frames
head(newcols,5)
```

```
## # A tibble: 5 x 4
##   Test_Version Weather Gender Blood_Type
##          <dbl> <chr>   <chr>  <chr>
## 1          1.1 Rain    Male   0
## 2          1.1 Sunny   Other  A
## 3          1.1 Windy   Male   A
## 4          1.1 Rain    Other  0
## 5          1.1 Rain    Other  AB
```

```
y <- mutate(select(mytibble, -IQ, -Confidence, -Date), newcols)
head(y,5)
```

```
## # A tibble: 5 x 10
##   Name  Height  Time   Age Score Country Test_Version Weather Gender Blood_Type
##   <chr>  <dbl> <dbl> <dbl> <dbl> <chr>          <dbl> <chr>   <chr>  <chr>
## 1 Rbgc    1.87  37      51   4.8 CAN              1.1 Rain    Male   0
## 2 Hrqw    1.25  36.2    44   8   CAN              1.1 Sunny   Other  A
## 3 Atio    1.42  24.5    63   7.6 UK               1.1 Windy   Male   A
## 4 Lmsy    1.69  20.3    34   5.4 AUS              1.1 Rain    Other  0
## 5 Onpo    1.44  31.3    51   4.6 CAN              1.1 Rain    Other  AB
```

`summarize()`

`summarise()`creates a new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarising all observations in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.
`summarise()` and `summarize()` are synonyms.

```
summarize(mytibble, mean(Time)) ## mean of the Time column
```

```
## # A tibble: 1 x 1
##   `mean(Time)`
##          <dbl>
## 1         30.5
```

```
## this function becomes more useful once we introduce grouping to the
## data frame
```

Quiz Week 1

Question 1 The American Community Survey distributes downloadable data about United States communities. Download the 2006 microdata survey about housing for the state of Idaho using download.file() from here:
https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Fss06hid.csv
and load the data into R. The code book, describing the variable names is here:
https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2FPUMSDataDict06.pdf
How many properties are worth $1,000,000 or more?

```
housing <- read.csv("./Quiz week 1/getdata_data_ss06hid.csv")
VAL <- housing$VAL[!is.na(housing$VAL)]
sum(VAL == 24)
```

```
## [1] 53
```

Question 2
Use the data you loaded from Question 1. Consider the variable FES in the code book. Which of the "tidy data" principles does this variable violate?

```
## Numeric values in tidy data can not represent categories.
```

Question 3
Download the Excel spreadsheet on Natural Gas Aquisition Program here:
https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2FDATA.gov_NGAP.xlsx
Read rows 18-23 and columns 7-15 into R and assign the result to a variable called `dat`.
What is the value of `sum(dat$Zip*dat$Ext,na.rm=T)`

```
dat <- read_xlsx("./Quiz week 1/getdata_data_DATA.gov_NGAP.xlsx",
                 range = "G18:O23")
dat
```

```
## # A tibble: 5 x 9
##     Zip CuCurrent PaCurrent PoCurrent Contact        Ext Fax         email Status
##   <dbl>     <dbl>     <dbl>     <dbl> <chr>        <dbl> <chr>       <lgl>  <dbl>
## 1 74136         0         1         0 918-491-6998     0 918-491-6~ NA          1
## 2 30329         1         0         0 404-321-5711    NA <NA>        NA          1
## 3 74136         1         0         0 918-523-2516     0 918-523-2~ NA          1
## 4 80203         0         1         0 303-864-1919     0 <NA>        NA          1
## 5 80120         1         0         0 345-098-8890   456 <NA>        NA          1
```

```
sum(dat$Zip*dat$Ext,na.rm=T)
```

```
## [1] 36534720
```

Question 4
Read the XML data on Baltimore restaurants from here:
https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Frestaurants.xml
How many restaurants have zipcode 21231?

```
rest <- xmlTreeParse("./Quiz week 1/getdata_data_restaurants.xml",
                     useInternalNodes = TRUE)
datrest <- xmlRoot(rest)
zipcode <- xpathApply(datrest, "//zipcode", xmlValue)
sum(zipcode == 21231)
```

```
## [1] 127
```

Question 5
The American Community Survey distributes downloadable data about United States communities. Download the 2006 microdata survey about housing for the state of Idaho using download.file() from here:
https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Fss06pid.csv
using the fread() command load the data into an R object called `DT`
The following are ways to calculate the average value of the variable `pwgtp15`
broken down by sex. Using the data.table package, which will deliver the fastest user time?

```
DT <- fread(file = "./Quiz week 1/getdata_data_ss06pid.csv")

system.time(DT[,mean(pwgtp15),by=SEX])
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(sapply(split(DT$pwgtp15,DT$SEX),mean))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(tapply(DT$pwgtp15,DT$SEX,mean))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(mean(DT[DT$SEX==1,]$pwgtp15),
         mean(DT[DT$SEX==2,]$pwgtp15))
```

```
##    user  system elapsed
##    0.00    0.02    0.02
```

```
mean(DT$pwgtp15,by=DT$SEX) ## incorrect answer
```

```
## [1] 98.21613
```

```
rowMeans(DT)[DT$SEX==1]; rowMeans(DT)[DT$SEX==2]   ## returns error
```

```
## Error in rowMeans(DT): 'x' must be numeric
```

```
## Error in rowMeans(DT): 'x' must be numeric
```