Lecture 4: Data Cleaning

James Sears*
AFRE 891/991 FS 25
Michigan State University

*Parts of these slides are adapted from **“Advanced Data Analytics”** by Nick Hagerty and **“Data Science for Economists”** by Grant McDermott.

Table of Contents

Prologue

Data Cleaning

No that we know how to wrangle data in R, it's time to talk more specifically about **Data Cleaning** - both the concerns to keep in mind while processing data and the "nitty gritty" of how to implement the necessary steps.

- **Importing data**
- **Keys and relational data**
- **Cleaning character strings**
- **Number Storage**

Packages we will use today:

- **stringr**
- **tidyverse**
- **nycflights13**

```
pacman::p_load(haven, sjlabelled, stringr, tidyverse, nycflights13)
```

Paths and Importing Data

Paths and Directories

The **working directory** is your **"current location"** in the filesystem.

What's your current working directory?

```
getwd()
```

```
## [1] "C:/Users/searsja1/OneDrive - Michigan State
University/Github/AFRE-891-991-FS25/Lecture Slides/04-Cleaning"
```

- This is an example of a full or **absolute path**.
  - Usually starts with `C:/` on Windows or `/` on Mac.
  - Defaults to the **folder containing your script/Rmd file**

Paths and Directories

In contrast, **relative paths** are defined **relative to** the full path of the working directory.

- Let's say your working directory is `"C:/Github/AFRE-891-991-FS25/"`
- You have a file saved at `"C:/Github/AFRE-891-991-FS25/assignment-1/assignment-1.Rmd"`
  - This is it's **absolute path**
- Its **relative path** would be **`"assignment-1/assignment-1.Rmd"`**
- Relies on the folder/directory nesting within your filesystem

In R you can use either an absolute or relative path in any given situation, but **relative paths** are usually **easier to work with**.

Paths and Directories: Best Practices

Option 1: use relative paths within GitHub repositories

- In the main folder (root directory), place
    - A **README** file that gives a basic overview of your project.
    - A **master script/R Markdown file** that lists and runs all other scripts
- Use paths **relative** to included folder structure specific to each type of input/output, a la...

Paths and Directories: Best Practices

```
/my_project
  /rawData
  /processedData
  /code
    /1_clean
    /2_process
    /3_results
  /output
    /tables
      /sumStats
      /regression
    /figures
    /estimates
```

The relative path to access a processed data file named `parcels.dta` is then

`"processedData/parcels.dta"`

Paths and Directories: Best Practices

Option 2: use relative paths within other Version Control

- Even if you're not using GitHub, you can use a **similar folder structure** for projects **saved locally**
- Back up files/sync across computers with cloud storage (a la OneDrive)

Download this data

Most data does not come nicely in R packages! You will download it from somewhere and load it in R.

Go here: NYTimes COVID-19 Data and click on the `colleges.csv` file, then the **Download raw file** link.

- This is a list of COVID-19 case counts reported at U.S. colleges and universities between July 2020 and May 2021.

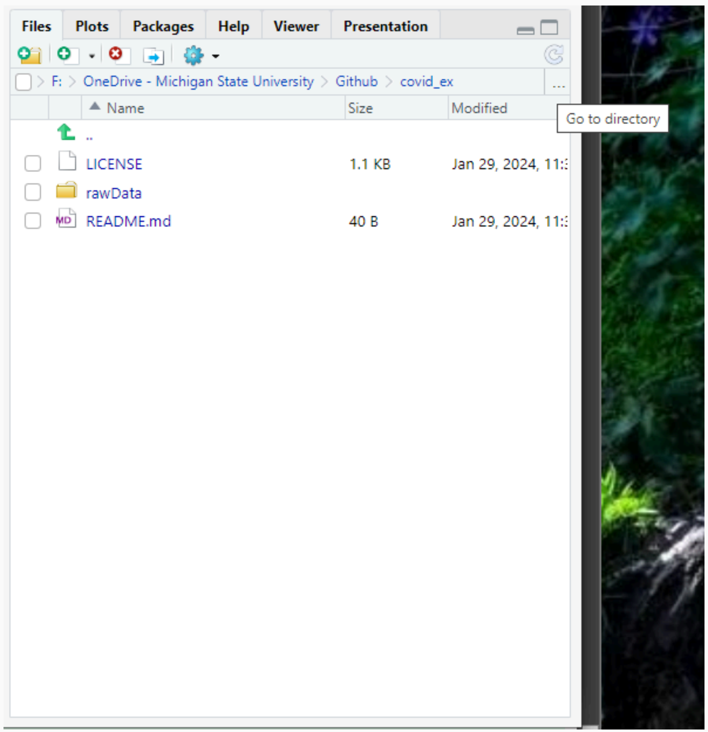
Save this file **somewhere sensible** on your computer

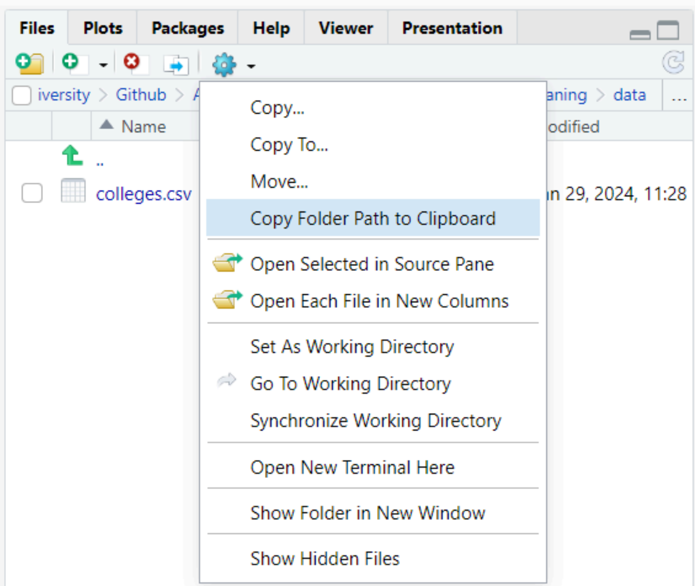
- Perhaps the "data" subfolder in your cloned Lecture 4 Slides folder?

Setting the Working Directory

You can change your working directory with `setwd(dir)`. Now:

1. **Find the location you saved your CSV file and copy the filepath.**
    - Manual navigation or use the **Files** window in bottom-right





Setting the Working Directory

You can change your working directory with `setwd(dir)`. Now:

1. **Find the location you saved your CSV file and copy the filepath.**
2. **Use the console in R to set your working directory to that location.**

For example:

```
setwd("F:/OneDrive - Michigan State University/Github/covid_ex/")
```

setwd Best Practices

Pro tip: minimize use of `setwd()` in scripts.

- Someone else's working directory will be different from yours!
- You want your code to be portable.
- **Best:** build a repository/project folder and use a master script/Rmd that **automatically uses relative file paths**!
- **Better:** set working directory to the project folder **outside the script** (like we just did)
- **Good:** declare a `maindir` path to your project folder at the start of your script, set working directory to that path
- **Worst:** changing working directories more than once in a script (barf)

Read in Data (with readr)

The main reason we bother with working directories is to let us **read in and interact with data.**

The tidyverse package `readr` provides lots of options to read data into R. Read in the college COVID data using `read_csv` and the **relative filepath**:

```
col <- read_csv("data/colleges.csv")
```

`View` this data to take a look at it.

```
str(col)
```

```
## spc_tbl_ [1,948 × 9] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ date       : Date[1:1948], format: "2021-05-26" "2021-05-26" ...
##  $ state      : chr [1:1948] "Alabama" "Alabama" "Alabama" "Alabama" ...
##  $ county     : chr [1:1948] "Madison" "Montgomery" "Limestone" "Lee"
...
##  $ city       : chr [1:1948] "Huntsville" "Montgomery" "Athens" "Auburn"
...
```

Reading in Data with readr

readr can read a wide set of **plain text and delimited** files, as well as **R Data files**

File Type	Function	Use
CSV	`read_csv`	comma delimited[1]
CSV	`read_csv2`	semicolon delimited, comma decimal mark
TSV	`read_tsv`	tab delimited
Delimited Plain Text	`read_delim`	Any delimiter
Fixed Width Text	`read_fwf`	Fixed width
R Data	`read_rds`	storage-efficient native R files

More on readr options **here.**

File Types and Storage Size

Note that `.rds` files are considerably **more storage-efficient than STATA files**:

DTA File	dataAnalysis_Long.dta	1,915,291 KB
RDS File	dataAnalysis_Long_3_23	438,681 KB

Reading in Data with readr

readr will guess at column types, but often it makes sense to **manually specify what column types are**

- i.e. FIPS codes with leading zeroes as character strings

Include the `col_types = list()` argument to tell readr what the column types are before reading in:

```
col ← read_csv("data/colleges.csv",
                 col_types = list(
                   date = col_date(),
                   state = col_character(),
                   county = col_character(),
                   city = col_character(),
                   ipeds_id = col_character(),
                   college = col_character(),
                   cases = col_integer(),
                   cases_2021 = col_integer(),
                   notes = col_character()
                 ))
```

Reading in Data with readr

We can also use **string abbreviations** in the list to simplify:

```
col ← read_csv("data/colleges.csv",
                col_types = list(
                  date = "d", # can mix abbreviations/functions
                  state = col_character(),
                  county = "c",
                  city = "c",
                  ipeds_id = "c",
                  college = "c",
                  cases = "i",
                  cases_2021 = "i",
                  notes = "c"
                ))
```

Reading in Data with readr

Or if we don't want to specify variable names, we can use a **single string of abbreviations**:

```
col ← read_csv("data/colleges.csv",
                  col_types =  "dcccccciic"
                  )
```

Reading in Online Data

Note that we can read files in **directly from the internet** without downloading them first

- You **usually shouldn't** - this is terrible for **retention**!
  - What happens if the file location changes?
  - Or the host updates the year in the file name?

```
url ← "https://raw.githubusercontent.com/nytimes/covid-19-data/master/colleges/colleges.csv"
col ← read_csv(url)
```

Reading in Data: STATA Files

The **haven** package's `read_dta()` works great for reading in STATA-formatted `.dta` files.

- For example: hospital capacities by county for 2020-2021:

```
hosp_cap <- haven::read_dta("data/hospital_capacity.dta")
```

Note that, as in Stata, the variables **contain labels**. These are visible when viewing and are stored as a **variable attribute**:

```
attr(hosp_cap$fips_code, "label")
```

```
## [1] "County FIPS"
```

Labels

We *could* modify labels directly with `attr(df$var, "label") <- "Label"`

```
attr(hosp_cap$fips_code, "label")
```

```
## [1] "County FIPS"
```

```
attr(hosp_cap$fips_code, "label") <- "FIPS Code"
attr(hosp_cap$fips_code, "label")
```

```
## [1] "FIPS Code"
```

But that's a pain and doesn't fit into our tidy workflow.

Instead, let's us the **sjlabelled** package and its methods for labels

Labeled Data

sjlabelled includes methods for both **variable labels** *and* **data labels**

Variable Labels

- `get_label()`: retrieve variable labels
- `set_label()`: set variable labels
- `var_labels()`: pipeable way of setting labels

Data Labels

- `get_labels()`: retrieve data labels
- `set_labels()`: set data labels

Getting Variable Labels

Use `get_label()` to... get the variable labels.

- Returns a named character vector

One variable:

```
get_label(hosp_cap$fips_code)
```

```
## [1] "FIPS Code"
```

Or all at once:

```
get_label(hosp_cap)
```

```
##                  fips_code          collection_week                                   
n_hosp
##                "FIPS Code"        "Collection Week"                                "# 
Hospitals"
##                 beds_total           beds_inpatient
```

Setting Variable Labels

Use `set_label()` to assign labels.

For example, `beds_patient_inpatient` is missing its label. Let's add it:

```
set_label(hosp_cap$beds_ped_inpatient) <- "Pediatric Inpatient Beds"
get_label(hosp_cap$beds_ped_inpatient)
```

```
## [1] "Pediatric Inpatient Beds"
```

We could also use `set_label()` to set **all variable labels at once with the syntax**

```
hosp_cap <- set_label(hosp_cap, c("Var 1", "Var 2", ... , "Var K"))
```

Setting Variable Labels (Pipeable)

Alternatively, use the tidy-conforming function `var_labels()`.

- Syntax works like `rename(df, var1 = "Label 1", varj = "Label J")`

```
hosp_cap_lab <- hosp_cap %>%
  var_labels(fips_code = "County FIPS Code",
             n_hosp = "Hospital Count")
get_label(hosp_cap_lab)
```

```
##                   fips_code                 collection_week
##          "County FIPS Code"               "Collection Week"
##                      n_hosp                      beds_total
##            "Hospital Count"           "Total Hospital Beds"
##               beds_inpatient                        beds_icu
##  "Inpatient Hospital Beds"                      "ICU Beds"
##                  beds_adult            beds_adult_inpatient
##       "Adult Hospital Beds"          "Adult Inpatient Beds"
##          beds_ped_inpatient
## "Pediatric Inpatient Beds"
```

Getting Value Labels

We can use `get_labels()` in much the same way as `get_label()` to retrieve the **value labels**.

Currently, we have no value labels on our numeric variables (say, `n_hosp`):

```
get_labels(hosp_cap$n_hosp)
```

```
## NULL
```

While the level of each character is counted as its label by default.

```
get_labels(hosp_cap$fips_code) %>% head(50)
```

```
##  [1] "01001" "01003" "01005" "01007" "01009" "01011" "01013" "01015"
"01017"
## [10] "01019" "01021" "01023" "01025" "01027" "01031" "01033" "01035"
"01039"
## [19] "01041" "01043" "01045" "01047" "01049" "01051" "01053" "01055"
"01057"
```

Setting Value Labels

We can use `set_labels()` to set the variable labels for a specific value(s) of a variable:

```
set_labels(df, var1, var2, ... , labels = c("label1" = value1, "label2" = value2, ... ))
```

```
hosp_cap_lab <- set_labels(hosp_cap_lab, fips_code,
                           labels = c("Autauga County, Alabama" = "01001")
                           )
get_labels(hosp_cap_lab$fips_code)
```

```
## [1] "Autauga County, Alabama"
```

Setting Value Labels

To set all levels of a variable, we can just pass a vector of labels

```
set_labels(df, var1, var2, ... , labels = c(label1, label2, ... ,labelJ))
```

Note that all this also works for vectors too!

```
vec <- c(10, 5, 4, 6, 8, 5, 4, 10, 10, 6)
get_labels(vec)
```

```
## NULL
```

Adding in text-based labels for each level

```
label_vec <- c(10, 5, 4, 6, 8)
names(label_vec) <- c("Ten", "Five", "Four", "Six", "Eight")
vec <- set_labels(vec, labels = label_vec)
vec
```

```
##  [1] 10  5  4  6  8  5  4 10 10  6
```

Reading in Data: Other Formats

Often we need to read in other data types, for which we'll need **other packages**

File Type	Function(s)	Package
CSV	`fread`	**data.table** (good for large files)
Excel (.xlsx, .xls)	`read_excel`	**readxl**
Google Sheets	`read_sheet`	**googlesheets4**
Stata, SAS, SPSS	`read_dta/read_sas/read_sav`	**haven**
R Data (.rds)	`readRDS`	**base R**

Challenge

Which state had the least total reported Covid-19 cases at colleges and universities?

- Was it Michigan?

Writing Out Data with readr

readr also makes it easy to write (save) out processed files with the `write_X(object, path)` functions.

File Type	Function	Use
CSV	`write_csv`	comma delimited[1]
CSV	`write_csv2`	semicolon delimited, comma decimal mark
TSV	`write_tsv`	tab delimited
Delimited Plain Text	`write_delim`	Any delimiter
Fixed Width Text	`write_fwf`	Fixed width
R Data	`write_rds`	storage-efficient native R files

Writing Out Data with readr

Let's say we want to produce a summary statistics table of **all cases** across **all colleges in a state**, sorted **most to least** for the **10 states with highest caseloads**

Using our data wrangling skills from last lecture, how could we write this?

Writing Out Data with readr

```
state_tab <- group_by(col, state) %>%
  summarise(total_cases = sum(cases, na.rm = T)) %>%
  arrange(desc(total_cases)) %>%
  ungroup() %>%
  filter(row_number() <= 10)
```

We can then save it out as a CSV with `write_csv(object, path)`

- Note that we only need to supply the **relative path**

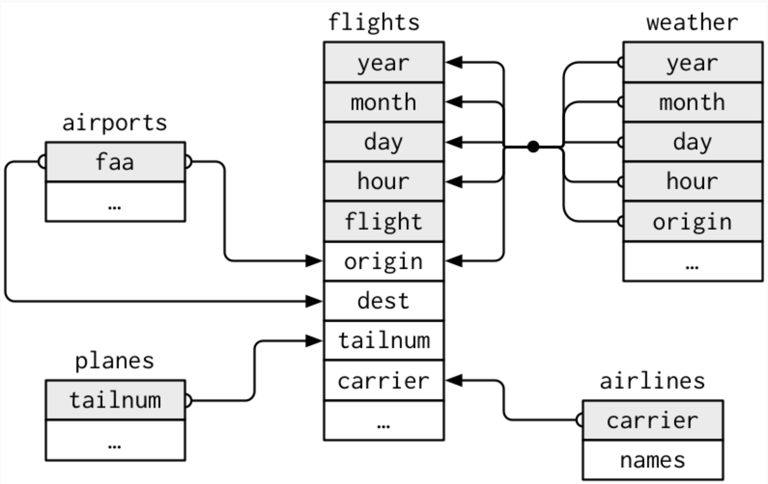
```
write_csv(state_tab, "data/state_top10.csv")
```

Keys and Relational Data

Images in this section are from **R for Data Science** by Wickham & Grolemund, used under **CC BY-NC-ND 3.0** and not included under this resource's overall CC license.

Relational data

More often than not, we'll be working with **relational data:** multiple tables of data that have relations to each other



- `flights` connects to `planes` via a single variable, `tailnum`.
- `flights` connects to `airlines` through the `carrier` variable.
- `flights` connects to `airports` in two ways: via the `origin` and `dest` variables.
- `flights` connects to `weather` via `origin` (the location), and `year`, `month`, `day` and `hour` (the time).

Keys

To join relation data, we need **key variable(s)** that **uniquely identifies an observation**.

- In `planes`, the key is `tailnum`.
- In `weather`, the key consists of 5 variables: (`year`, `month`, `day`, `hour`, `origin`).

Keys

There are two types of keys:

1. A **primary key** uniquely identifies an observation in **its own data frame**.
   - `planes$tailnum` is a **primary key** because it uniquely identifies each plane in the `planes` data frame.
2. A **foreign key** uniquely identifies an observation in **another data frame**.
   - `flights$tailnum` is a **foreign key** because it appears in the `flights` data frame where it matches each flight to a unique plane. A variable can be **both a primary key *and* a foreign key**.

- For example, `origin` is part of the `weather` primary key, and is also a foreign key for the `airports` data frame.

Keys

The **primary key** is the **first thing** you need to know about a new data frame.

Once you think you know the primary key, **verify it**. Here's one way to do that:

```
planes_dup <- rbind(planes, planes[1:100,]) %>% arrange(tailnum) %>%
mutate(n = row_number())

planes_dist <- distinct(planes_dup, .keep_all = T)

planes_dist2 <- group_by(planes_dup, tailnum) %>%
  mutate(count = row_number()) %>%
  filter(count == 1) %>%
  ungroup()

View(flights)
flights %>%
  count(month, day, dep_time, carrier, flight, tailnum) %>%
  filter(n > 1)
```

Keys

You can write a **unit test** into your code to make sure uniqueness is true before proceeding:

```
dups_planes <- planes %>%
  count(tailnum) %>%
  filter(n > 1)

stopifnot(nrow(dups_planes) == 0)

dups_weather <- weather |> # same thing using base R pipe
  count(year, month, day, hour, origin) |>
  filter(n > 1)

stopifnot(nrow(dups_weather) == 0)
```

```
## Error: nrow(dups_weather) == 0 is not TRUE
```

Surrogate Keys

What's the primary key in the `flights` data frame? Take a minute to investigate/verify.

Surrogate Keys

What's the primary key in the `flights` data frame? Take a minute to investigate/verify.

You might think it would be the date + the carrier + the flight or tail number, but neither of those are unique:

```
flights %>%
  count(year, month, day, carrier, flight) %>%
  filter(n > 1)
```

```
## # A tibble: 24 × 6
##     year month   day carrier flight     n
##    <int> <int> <int> <chr>    <int> <int>
##  1  2013     6     8 WN        2269     2
##  2  2013     6    15 WN        2269     2
##  3  2013     6    22 WN        2269     2
##  4  2013     6    29 WN        2269     2
##  5  2013     7     6 WN        2269     2
##  6  2013     7    13 WN        2269     2
```

Surrogate keys

If a data frame lacks a primary key but it is tidy (each row is an observation), it's often useful to add in a **surrogate key**:

```
flights2 = flights %>%
  arrange(year, month, day, carrier, flight, sched_dep_time) %>%
  mutate(id = row_number()) %>%
  relocate(id) %>%
  head(8)
```

Relations

A **primary key** and the corresponding **foreign key** in another data frame form a **relation**.

In general, relations are **one-to-many**: Each flight has one plane, but each plane has many flights.

- Sometimes you'll see a **one-to-one** relation, but you can think of this as a special case of one-to-many.
- You can also find **many-to-many** relations, but you can think of these as two one-to-many relations going in each direction.
- There's a many-to-many relationship between airlines and airports: each airline flies to many airports; each airport hosts many airlines.

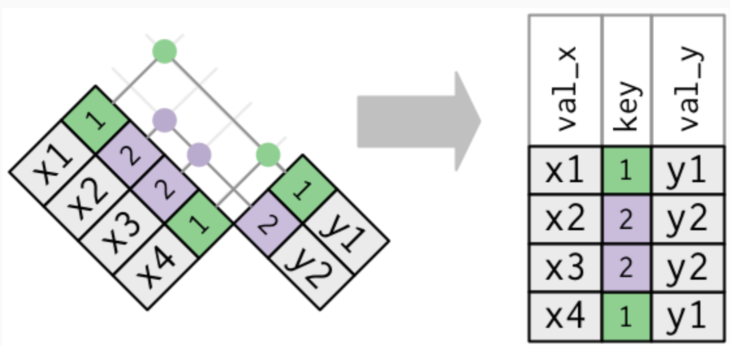
Relations

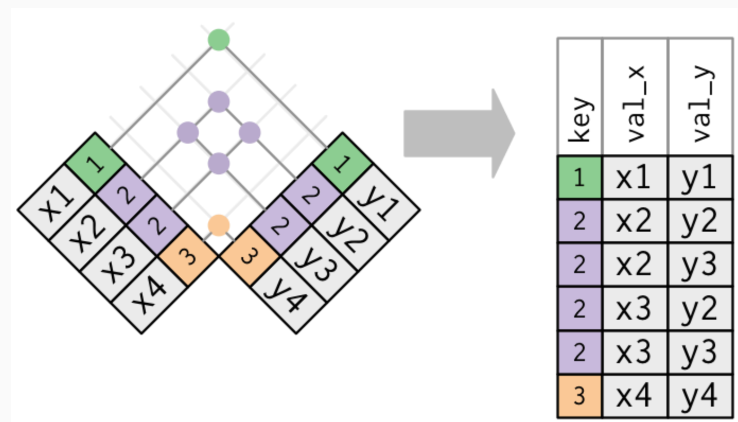
Note on Stata: NEVER USE `merge m:m`. JUST DON'T DO IT. There is no scenario in which it will give you what you want. This syntax should not exist. If you are tempted, you are probably either confused or looking for `joinby`.

Relations

`join` does **not** think about whether your key is unique, or what type of relation you have.

- Instead, it simply returns all possible combinations of observations in your two dataframes:





Duplicate Keys

What if you join by a key that is not actually unique, when you think it is?

You'll get **extra rows with incorrect matches**:

```
flights_weather <- flights %>%
  left_join(weather, by=c("year", "month", "day", "origin"))
nrow(flights_weather)
```

```
## [1] 8036575
```

Now you no longer have a dataframe of unique flights.

```
nrow(flights)
```

```
## [1] 336776
```

Best Practice: Joins

Here's an example of a good (safe) way to join `flights` and `planes`:

1. Confirm the primary key in `planes` is unique

```
# Confirm that tailnum is the primary key (unique ID) of planes
dups_planes <- planes %>%
  count(tailnum) %>%
  filter(n > 1)

stopifnot(nrow(dups_planes) == 0)
```

Best Practice: Joins

Here's an example of a good (safe) way to join `flights` and `planes`:

2. Join, keeping the original join keys from both datasets

```
# Join, keeping the join keys from both datasets
flights_planes <- flights %>%
  left_join(planes %>% rename(year_built = year), by="tailnum",
keep=TRUE) %>%
  rename(tailnum = tailnum.x, tailnum_planes = tailnum.y)
```

Best Practice: Joins

Here's an example of a good (safe) way to join `flights` and `planes`:

3. Confirm the join was one-to-many

```
# Confirm the join was 1:many
stopifnot(nrow(flights) == nrow(flights_planes))
```

String Cleaning

Parts of this section are adapted from **“Introduction to Data Science”** by Rafael A. Irizarry, used under **CC BY-NC-SA 4.0**.

String Cleaning

Regardless of where we get them from, **character strings** often require a lot of cleaning work to get them into our desired formats

- **Surveys:** report agricultural yields in a mix of bushels, pounds, hundred weight, tons, etc.
- **Admin records:** manual entry of information prone to typos/inconsistencies

Whether we want to convert to numeric values/dates or find matching info, we will likely need to do some pre-processing on our strings.

Let's practice some **key string cleaning steps**.

String Cleaning Example

Let's load in the raw data output from a web form asking students to report their height in inches:

```
library(dslabs)
data(reported_heights)
str(reported_heights)
```

```
## 'data.frame':    1095 obs. of  3 variables:
##  $ time_stamp: chr  "2014-09-02 13:40:36" "2014-09-02 13:46:59" "2014-09-02 13:59:20" "2014-09-02 14:51:53" ...
##  $ sex       : chr  "Male" "Male" "Male" "Male" ...
##  $ height    : chr  "75" "70" "68" "74" ...
```

Unfortunately `height` is not numeric. Can we coerce it to numeric?

```
heights2 <- reported_heights %>%
  mutate(height_num = as.numeric(height))
sum(is.na(heights2$height_num))
```

String Cleaning Example

Yes, but we **lose a lot of information** because there are plenty of **non-numeric entries:**

```
heights_probs ← filter(heights2, is.na(height_num))
View(heights_probs)
heights_probs$height
```

```
##  [1] "5' 4\""                  "165cm"                    "5'7"
##  [4] ">9000"                   "5'7\""                    "5'3\""
##  [7] "5 feet and 8.11 inches" "5'11"                     "5'9''"
## [10] "5'10''"                  "5,3"                      "6''"
## [13] "6,8"                     "5' 10"                    "Five foot eight
inches"
## [16] "5'5\""                   "5'2\""                    "5,4"
## [19] "5'3"                     "5'10''"                   "5'3''"
## [22] "5'7''"                   "5'12"                     "2'33"
## [25] "5'11"                    "5'3\""                    "5,8"
## [28] "5'6''"                   "5'4"                      "1,70"
## [31] "5'7.5''"                 "5'7.5''"                  "5'2\""
## [34] "5' 7.78\""               "yyy"                      "5'5"
```

String Cleaning Workflow

Many of these entries have valuable information, so let's try to salvage as much as we can.

The general way to proceed is:

1. Identify the **most common patterns** among the problematic entries.
2. **Write an algorithm** to correct these.
3. **Review results** to make sure your algorithm worked correctly.
4. Look at the **remaining problematic entries**. Tweak your algorithm or add another one.
5. **Stop** when all useful information is **corrected** (or when MB < MC).

What are the **most common patterns?**

String Cleaning Workflow

Many of these entries have valuable information, so let's try to salvage as much as we can.

The general way to proceed is:

1. Identify the **most common patterns** among the problematic entries.
2. **Write an algorithm** to correct these.
3. **Review results** to make sure your algorithm worked correctly.
4. Look at the **remaining problematic entries**. Tweak your algorithm or add another one.
5. **Stop** when all useful information is **corrected** (or when MB < MC).

What are the **most common patterns?**

- Strings of the form `x'y` or `x'y"` where `x` is feet and `y` is inches.
- Strings of the form `x ft y inches`, except that "ft" and "inches" are inconsistent.

String Cleaning Workflow

Many of these entries have valuable information, so let's try to salvage as much as we can.

The general way to proceed is:

1. Identify the **most common patterns** among the problematic entries.
2. **Write an algorithm** to correct these.
3. **Review results** to make sure your algorithm worked correctly.
4. Look at the **remaining problematic entries**. Tweak your algorithm or add another one.
5. **Stop** when all useful information is **corrected** (or when MB < MC).

My suggested approach:

1. Try to convert everything to the pattern `x y`.
2. `separate` the feet and inches values.
3. Calculate total inches from feet and inches.

String Cleaning Functions

To implement this, we'll use a subset of **stringr**'s **string cleaning functions**:

Function	Description
`str_replace(_all)(df, str, pattern, replacement)`	Replace the first (all) matches of `pattern` with `replacement` within the string `str`
`str_trim(df, str)`	Remove all whitespace from start and end of string `str`
`str_squish(df, str)`	Remove all whitespace from start and end of string `str` *and* replace all internal whitespace with a single space

1. Replace Punctuation

Start by replacing 3 different punctuation marks with spaces (note we have to **escape** the " with `\"`):

```
heights2 <- reported_heights %>%
  mutate(height_clean = str_replace_all(height, "'", " "),
  height_clean = str_replace_all(height_clean, ",", " "),
  height_clean = str_replace_all(height_clean, "\"", " "))
heights2$height_clean
```

```
##     [1] "75"                          "70"
##     [3] "68"                          "74"
##     [5] "61"                          "65"
##     [7] "66"                          "62"
##     [9] "66"                          "67"
##    [11] "72"                          "6"
##    [13] "69"                          "68"
##    [15] "69"                          "66"
##    [17] "75"                          "64"
##    [19] "60"                          "67"
##    [21] "66"                          "5   4 "
```

1. Replace Punctuation

We can make this more concise with the **"or" operator (`|`)**:

- Rather than three iterations, write one call that replaces all matches of ' or " or , with " " with a single argument

```
heights2 ← reported_heights %>%
   mutate(height_clean =  str_replace_all(height, "'|,|\"|,", " "))
heights2$height_clean
```

```
##    [1] "75"                     "70"
##    [3] "68"                     "74"
##    [5] "61"                     "65"
##    [7] "66"                     "62"
##    [9] "66"                     "67"
##   [11] "72"                     "6"
##   [13] "69"                     "68"
##   [15] "69"                     "66"
##   [17] "75"                     "64"
##   [19] "60"                     "67"
##   [21] "66"                     "5  4 "
```

2. Remove Common Words + Extra

Next, get rid of some common words and **trim extra spaces**:

```
heights2 <- reported_heights %>%
     mutate(height_clean =  str_replace_all(height,
              "'|,|\"|,|ft|feet|inches|and|cm", " "),
            height_clean = str_trim(height_clean))
heights2$height_clean
```

```
##    [1] "75"                 "70"                 "68"                 "74"
##    [5] "61"                 "65"                 "66"                 "62"
##    [9] "66"                 "67"                 "72"                 "6"
##   [13] "69"                 "68"                 "69"                 "66"
##   [17] "75"                 "64"                 "60"                 "67"
##   [21] "66"                 "5   4"              "70"                 "73"
##   [25] "72"                 "69"                 "69"                 "72"
##   [29] "64"                 "72"                 "75"                 "71"
##   [33] "67"                 "66"                 "67"                 "69"
##   [37] "68"                 "66.75"              "72"                 "5.3"
```

2. Remove Common Words + Extra

Additionally remove extra spaces **before/after/within strings**:

```
heights2 <- reported_heights %>%
     mutate(height_clean =  str_replace_all(height,
              "'|,|\"|,|ft|feet|inches|and|cm", " "),
              height_clean = str_squish(height_clean))
heights2$height_clean
```

```
##    [1] "75"                "70"                "68"                "74"
##    [5] "61"                "65"                "66"                "62"
##    [9] "66"                "67"                "72"                "6"
##   [13] "69"                "68"                "69"                "66"
##   [17] "75"                "64"                "60"                "67"
##   [21] "66"                "5 4"               "70"                "73"
##   [25] "72"                "69"                "69"                "72"
##   [29] "64"                "72"                "75"                "71"
##   [33] "67"                "66"                "67"                "69"
##   [37] "68"                "66.75"             "72"                "5.3"
```

3. Remove Punctuation (Periods)

```r
heights2 <- reported_heights %>%
     mutate(height_clean =  str_replace_all(height,
                "'|,|\"|,|ft|feet|inches|and|cm", " "),
            height_clean = str_squish(height_clean),
            height_clean = str_replace(height_clean, " \\.", " "))
heights2$height_clean
```

```
##    [1] "75"                 "70"                 "68"                 "74"
##    [5] "61"                 "65"                 "66"                 "62"
##    [9] "66"                 "67"                 "72"                 "6"
##   [13] "69"                 "68"                 "69"                 "66"
##   [17] "75"                 "64"                 "60"                 "67"
##   [21] "66"                 "5 4"                "70"                 "73"
##   [25] "72"                 "69"                 "69"                 "72"
##   [29] "64"                 "72"                 "75"                 "71"
##   [33] "67"                 "66"                 "67"                 "69"
##   [37] "68"                 "66.75"              "72"                 "5.3"
##   [41] "69"                 "68"                 "63"                 "60"
##   [45] "73"                 "74"                 "74"                 "66"
##   [49] "68"                 "73"                 "70"                 "68"
##   [53] "73"                 "70.5"               "165"                "71"
```

4. Calculate Total Inches

Now **separate** the cleaned height into feet and inch variables.

To do this, we'll need one of the `separate_` functions. There are two types:

1. **separate_wider_X**: separate one variable into **multiple columns**
   - `separate_wider_delim()`: split on a **delimiter**
   - `separate_wider_position()`: split on **position**
   - `separate_wider_regex()`: split on a **regular expression**
2. **separate_longer_X**: separate one variable into **multiple rows**
   - `separate_longer_delim()`: split on a **delimiter**
   - `separate_longer_position()`: split on **position**

Separate Wider

1. separate_wider_X: separate one variable into **multiple columns**

- `separate_wider_delim()`: split on a **delimiter**
- `separate_wider_position()`: split on **position**
- `separate_wider_regex()`: split on a **regular expression**

```
df_wide <- select(col, state, county) %>% slice(83:86)
df_wide
```

```
## # A tibble: 4 × 2
##   state      county
##   <chr>      <chr>
## 1 California Los Angeles
## 2 California Ventura
## 3 California San Luis Obispo
## 4 California Los Angeles
```

Separate Wider

1. separate_wider_X: separate one variable into **multiple columns**

- `separate_wider_delim()`: split on a **delimiter**
- `separate_wider_position()`: split on **position**
- `separate_wider_regex()`: split on a **regular expression**

```
df_wide %>% separate_wider_delim(county, # variable to split
              delim = " ",  # delimiter to use
              names = c("cty_1", "cty_2"), # new var names
              too_few = "align_end", # grab the end if too few pieces
              too_many = "debug") # add debug col if too many pieces
```

```
## # A tibble: 4 × 7
##   state      cty_1 cty_2   county       county_ok county_pieces
county_remainder
##   <chr>      <chr> <chr>   <chr>        <lgl>             <int> <chr>
## 1 California Los   Angeles Los Angeles  TRUE                  2 ""
## 2 California <NA>  Ventura Ventura      TRUE                  1 ""
## 3 California San   Luis    San Luis Ob… FALSE                 3 " Obispo"
## 4 California Los   Angeles Los Angeles  TRUE                  2 ""
```

Separate Longer

2. separate_longer_X: separate one variable into **multiple rows**

- `separate_longer_delim()`: split on a **delimiter**
- `separate_longer_position()`: split on **position**

```
## # A tibble: 10 × 2
##    state         college
##    <chr>         <chr>
##  1 Oregon        Reed College
##  2 Minnesota     St. Paul College
##  3 Michigan      Jackson College
##  4 Michigan      Central Michigan University
##  5 Louisiana     Southern University Law Center
##  6 Pennsylvania  Moravian College
##  7 Texas         Austin College
##  8 Florida       Keiser University at Jacksonville
##  9 Minnesota     Bethany Lutheran College
## 10 West Virginia West Virginia University
```

Separate Longer

2. separate_longer_X: separate one variable into **multiple rows**

- `separate_longer_delim()`: split on a **delimiter**
- `separate_longer_position()`: split on **position**

```
df_long %>%
  separate_longer_delim(college, delim = " ")
```

```
## # A tibble: 28 × 2
##    state     college
##    <chr>     <chr>
##  1 Oregon    Reed
##  2 Oregon    College
##  3 Minnesota St.
##  4 Minnesota Paul
##  5 Minnesota College
##  6 Michigan  Jackson
##  7 Michigan  College
##  8 Michigan  Central
##  9 Michigan  Michigan
## 10 Michigan  University
## # ℹ 18 more rows
```

4. Calculate Total Inches

```
heights2 <- reported_heights %>%
     mutate(height_clean =  str_replace_all(height,
              "'|,|\"|,|ft|feet|inches|and|cm", " "),
            height_clean = str_squish(height_clean),
            height_clean = str_replace(height_clean, " \\.", " ")) %>%
    separate_wider_delim(height_clean, # variable to split
                         delim = " ",   # delimiter to split on
                         names = c("feet", "inches"), # new var names
                         too_few = "align_end", # add just inches if too
few
                         too_many = "debug")  %>% # add col to diagnose
too many
  arrange(height_clean_ok)
```

4. Calculate Total Inches

What new variables do we have?

- `feet/inches`: the split variables we requested
- `height_clean_ok`: boolean of whether we got the "right" number of pieces (1-2)
- `height_clean_pieces`: numeric number of split pieces
- `height_clean_remainder`: the extra string pieces when > 2

5. Deal with Extra Pieces

Look at the top of the data. We can see there are a few values with extra pieces that are erroneous entries:

```
head(heights2, 4)
```

```
## # A tibble: 4 × 9
##   time_stamp          sex    height    feet  inches height_clean
height_clean_ok
##   <chr>               <chr>  <chr>     <chr> <chr>  <chr>             <lgl>
## 1 2014-10-08 19:19:33 Female Five foo… Five  foot   Five foot e… FALSE
## 2 2017-08-09 12:16:38 Male   7,283,465 7     283    7 283 465    FALSE
## 3 2014-09-02 13:40:36 Male   75        <NA>  75     75           TRUE
## 4 2014-09-02 13:46:59 Male   70        <NA>  70     70           TRUE
## # ℹ 2 more variables: height_clean_pieces <int>, height_clean_remainder
<chr>
```

5. Deal with Extra Pieces

Let's remove those (I'm starting a new dataframe to iterate further)

- Use an anonymous lambda function (add `~` before function and `.x` for argument)

```
heights3 <- heights2 %>%
    # replace NA
  mutate(across(c(feet, inches),
         ~ifelse(is.na(.x), 0, .x))) %>%
  # apply ifelse across both feet/inches variables
  mutate(across(c(feet, inches),
         ~ifelse(height_clean_pieces ==3, NA, as.numeric(.x)))) %>%
  # drop variables we no longer need
  select(-height_clean_pieces, -height_clean_remainder, -
height_clean_ok)
head(heights3, 2)
```

```
## # A tibble: 2 × 6
##   time_stamp          sex    height                        feet inches
height_clean
```

6. Make Combined Inch Measurement

Now add a "clean" combined inch measurement:

```
heights3 <- heights3 %>%
  mutate(inches_clean = feet * 12 + inches)%>%
  arrange(inches_clean)
```

If you `View` the data, you'll find:

- Many values between 5 and 7 which are clearly in **feet** instead of inches.
- Many values between 150 and 214 which are clearly in **cm** instead of inches.

7. Fix Units

This is a good use case for `case_when()` and `between()`:

```
heights3 <- heights3 %>%
  mutate(inches_clean = case_when(
    # First convert values in feet
    inches_clean >= 5 & inches_clean <= 7 ~ inches_clean*12,
    # Next convert cm values if between 150 and 214
    between(inches_clean, 150, 214) ~ inches_clean / 2.54,
    # Otherwise, keep same value
    TRUE ~ inches_clean)
  )
```

8. Check Plausible Range

How many values are still outside a plausible range?

```
heights3 %>%
  mutate(ok = between(inches_clean, 3.5*12, 7.5*12)) %>%
  count(ok)
```

```
## # A tibble: 3 × 2
##   ok        n
##   <lgl> <int>
## 1 FALSE    29
## 2 TRUE   1061
## 3 NA        5
```

9. Deal with Implausible Values

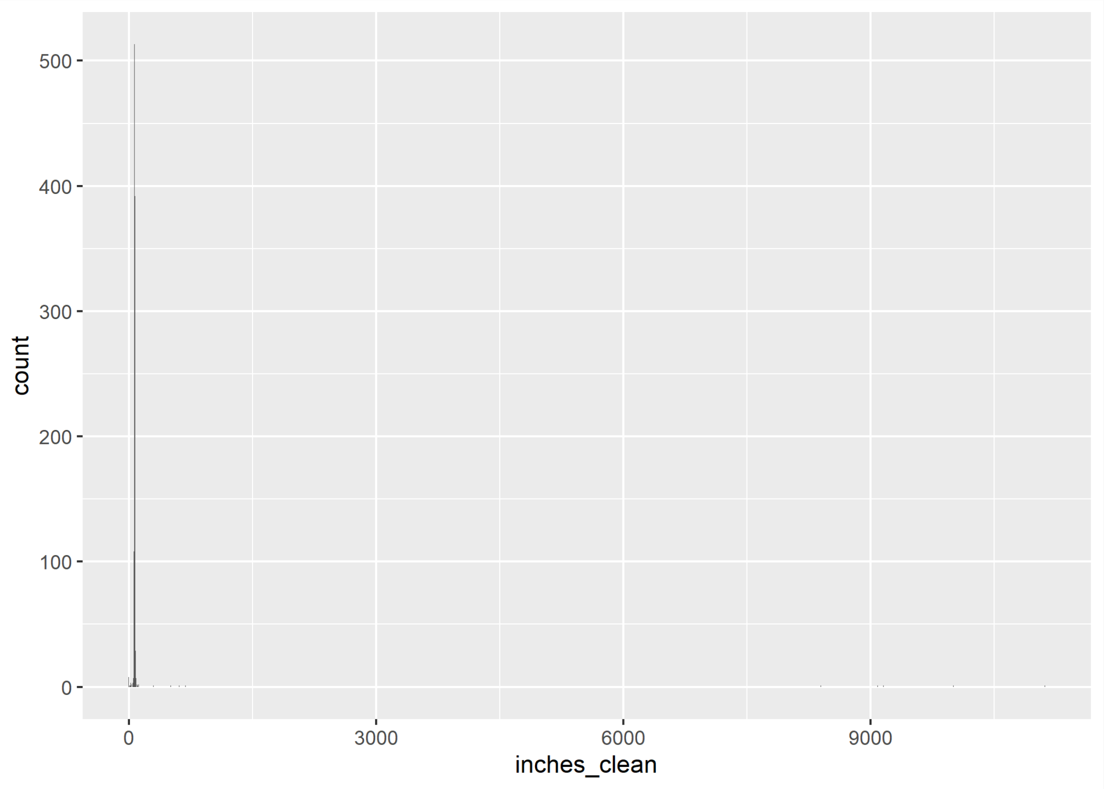
What should we do with our implausible values?

1. Some of these may still contain interpretable information. **There may be more cleaning to do.**
2. Some of them may not, in which case we probably won't use them for analysis.
   - Don't discard them yet! We'll come back to extreme values (aka outliers) in a couple of weeks.
3. You'll find there are also a few instances where our cleaned value appears sensible, but the original value does not.
   - You may need to tweak the algorithm further.

LOOK AT THE DISTRIBUTIONS!

Pro Tip: always look at distributions of numeric variables!

```r
ggplot(heights3) +
  geom_histogram(aes(x = inches_clean), binwidth = 6)
```



Aside: Regular Expressions

Regular expressions are code to **describe patterns in strings** that are common acros basically all programming languages

```
names <- c("Python", "SPSS", "Stata", "Julia")

# Match strings that CONTAIN a lowercase "t"
str_view_all(names, "t")
```

```
## [1] │ Py<t>hon
## [2] │ SPSS
## [3] │ S<t>a<t>a
## [4] │ Julia
```

Common Regular Expressions

Common regular expression operators include

Match strings that **start** with a capital "S":

```
str_view_all(names, "^S")
```

```
## [1] │ Python
## [2] │ <S>PSS
## [3] │ <S>tata
## [4] │ Julia
```

Match strings that **end** with a lowercase "a":

```
str_view_all(names, "a$")
```

```
## [1] │ Python
## [2] │ SPSS
## [3] │ Stat<a>
## [4] │ Juli<a>
```

`^` and `$` are called **anchors**.

Common Regular Expressions

Match all lowercase vowels:

```
str_view_all(names, "[aeiou]")
```

```
## [1] │ Pyth<o>n
## [2] │ SPSS
## [3] │ St<a>t<a>
## [4] │ J<u>l<i><a>
```

Match everything BUT lowercase vowels:

```
str_view_all(names, "[^aeiou]")
```

```
## [1] │ <P><y><t><h>o<n>
## [2] │ <S><P><S><S>
## [3] │ <S><t>a<t>a
## [4] │ <J>u<l>ia
```

Common Regular Expressions

Use a vertical bar (`|`) for "or":

```
str_view_all(names, "Stata|SPSS")
```

```
## [1] │ Python
## [2] │ <SPSS>
## [3] │ <Stata>
## [4] │ Julia
```

And parentheses to clarify:

```
str_view_all(names,
"S(tata|PSS)")
```

```
## [1] │ Python
## [2] │ <SPSS>
## [3] │ <Stata>
## [4] │ Julia
```

Last Remarks on Regular Expressions

All kinds of regex cheat sheets and interactive testers are available via a quick Google.

Regexps are hard to read and troubleshoot. Try not to get too deep into them -- you can often accomplish the same goal by breaking it up into smaller chunks.

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. - Jamie Zawinski

Last Remarks on Regular Expressions

This is (the start of) a real regular expression that checks whether an email address is valid:

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:
(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()
<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()
<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\
[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:
[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?
[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
```

Useful Functions for Cleaning Data

stringr functions we've used here:

- `str_replace` and `str_replace_all`: Replace parts of strings.
- `str_trim` and `str_squish`: Remove extra spaces.
- `str_view_all`: Illustrates matches, to help develop regular expressions.

Other **tidyverse** functions we've used:

- `between`: Test whether values fall within a numerical range.
- `case_when`: Multiple conditional expressions.

Useful Functions for Cleaning Data

Other useful **stringr** functions:

- `str_sub`: Subset strings by position of characters.
- `str_detect`: Test whether a string matches a pattern.
- `str_extract` and `str_extract_all`: extract matching portion(s) of a string

Other useful **tidyverse** functions:

- `na_if`: Set a certain value to missing.
- `bind_rows`: Append two datasets that have the same variable structure.
- `replace_na`: Set missing values to a certain value.

Number Storage

Floating Point Problems

Simplify this expression: $1 - \frac{1}{49} * 49$

It's obviously 0. Now ask R:

```
1 - (1/49)*49
```

```
## [1] 1.110223e-16
```

This is called a **floating point** problem. It arises from the way computers store numbers.

Floating Point Problems

R doesn't notice that $49/49$ simplifies to 1. It just follows the order of operations. So the first thing it does is calculate:

```
(1/49)
```

```
## [1] 0.02040816
```

Which is an irrational number. So R rounds it to 53 significant digits before multiplying by 49.

Floating Point Problems

Most of the time, 53 digits is plenty of precision. But sometimes it creates problems.

Note: This explanation is actually too simple. The floating-point issue goes **deeper than just irrational numbers**. Here's another example:

```
1 - 0.9 - 0.1
```

```
## [1] -2.775558e-17
```

In 1996, a floating-point error caused a European Space Agency rocket to **self-destruct 37 seconds after it was launched.**

Avoiding Floating Point Errors

Pay attention to the data type of your variables.

Avoid using logical conditions like `height==180` for numeric variables.

- `height` may even read as `180` in the `View` window
- But under the hood, it might still be stored as `180.000000000000173…`.

What you can do instead:

- **Best option:** `dplyr::near` compares numbers with a built-in tolerance.
- Use `>` and `<` comparisons, or `between(height, 179.9, 180.1)`.
- Convert in place: `as.integer(height) == 180`
- Or with finer control: `round(height, digits=6) == 180`
- If all values are integers, store the variable as an integer in the first place.

How to Store a Number?

Numeric variables are stored in scientific notation.

- Use to represent a single value, for which digits decrease in importance from left to right.
- Example: My height is `172.962469405113283` cm.

Integer variables lack decimal places.

- Saves memory relative to numeric variables.
- Stores values exactly, avoiding some floating-point problems.

Character variables store the full sequence of digits literally.

- Use when digits lack quantitative information, and each digit is equally important.
- Phone numbers, credit card numbers, etc.
- No chance of the right-most digits getting lost or corrupted.

More Variable Formats

Dates and times allow you to easily do math and logic on dates and times.

- See tidyverse package `lubridate`.

Factors allow you to store values as numbers, but *display* them as strings.

- This is useful for sorting things like month names: "Jan", "Feb", "Mar", "Apr"....
- See tidyverse packages `forcats`.

Memory Space

Memory space quickly becomes a problem when you work with large datasets.

- But R does a reasonably good job of handling storage efficiently.

Logical variables are smaller than integers, which are smaller than numeric.

Does it save memory to store a variable as a factor instead of a string?

- This **used to be true:** factor variables only store the factor labels once.
- But **no longer:** R uses a global string pool - each unique string is only stored once.

`pryr::object_size()` will tell you how much memory an object takes up (accounting for shared elements within an object).

Data Cleaning Checklist

Data Cleaning Checklist

Part A. Get to know your data frame.

---

1. **Convert file formats**, as necessary.
2. **Import data and wrangle into a tidy layout.**
3. **Remove irrelevant, garbage, or empty** columns and rows.
4. **Identify the primary key**, or define a surrogate key.
5. **Resolve duplicates** (remove true duplicates, or redefine the primary key).
6. **Understand the definition, origin, and units** of each variable, and document as necessary.
7. **Rename variables** as necessary, to be succinct and descriptive.

Data Cleaning Checklist

Part B. Check your variables.

---

1. Understand patterns of missing values.

- Find out why they're missing.
- Make sure they are not more widespread than you expect.
- Convert other intended designations (i.e., -1 or -999) to NA.
- Distinguish between missing values and true zeros.

2. Convert to numeric when variables are inappropriately stored as strings. Correct typos as necessary.

3.Convert to date/time format where appropriate.

Data Cleaning Checklist

Part B. Check your variables.

---

1. Recode binary variables as 0/1 as necessary. (Often stored as "Yes"/"No" or 1/2.)

2. Convert to factors when strings take a limited set of possible values.

Data Cleaning Checklist

Part C. Check the values of your quantitative variables.

---

1. Make units and scales consistent. Avoid having in the same variable:

- Some values in meters and others in feet.
- Some values in USD and others in GBP.
- Some percentages as 40% and others as 0.4.
- Some values as millions and others as billions.

2. Perform logical checks on quantitative variables:

- Define any range restrictions each variable should satisfy, and check them (graphically too!).
- Correct any violations that are indisputable data entry mistakes.
- Create a flag variable to mark remaining violations.

Data Cleaning Checklist

Part D. Check the rest of your values.

---

1. Clean string variables. Some common operations:

- Make entirely uppercase or lowercase
- Remove punctuation
- Trim spaces (extra, starting, ending)
- Ensure order of names is consistent
- Remove uninformative words like "the" and "a"
- Correct spelling inconsistencies (consider text clustering packages)

2. Literally look at your data tables every step of the way, to spot issues you haven't thought of, and to make sure you're actually doing what you think you're doing.

Data Cleaning Checklist

Part E. Finish up the cleaning phase.

---

1. Save your clean data to disk before further manipulation (merging dataframes, transforming variables, restricting the sample). Think of the whole wrangling/cleaning/analysis pipeline as 2 big phases:

- Taking messy data from external sources and making a nice, neat table that you are likely to use for multiple purposes in analysis.
- Taking that nice, neat table and doing all kinds of new things with it.

2. Record all steps in a script.

3. Never overwrite the original raw data file.

Data Cleaning Tips

Whenever possible, make changes to values **only by logical conditions** on one or more substantive variables - ***not*** by observation ID or (even worse) row number.

You want the changes you make to be rule-based, for 2 reasons:

- So that they're **general** -- able to handle upstream changes to the data.
- So that they're **principled** -- no one can accuse you of cherry-picking.

Table of Contents