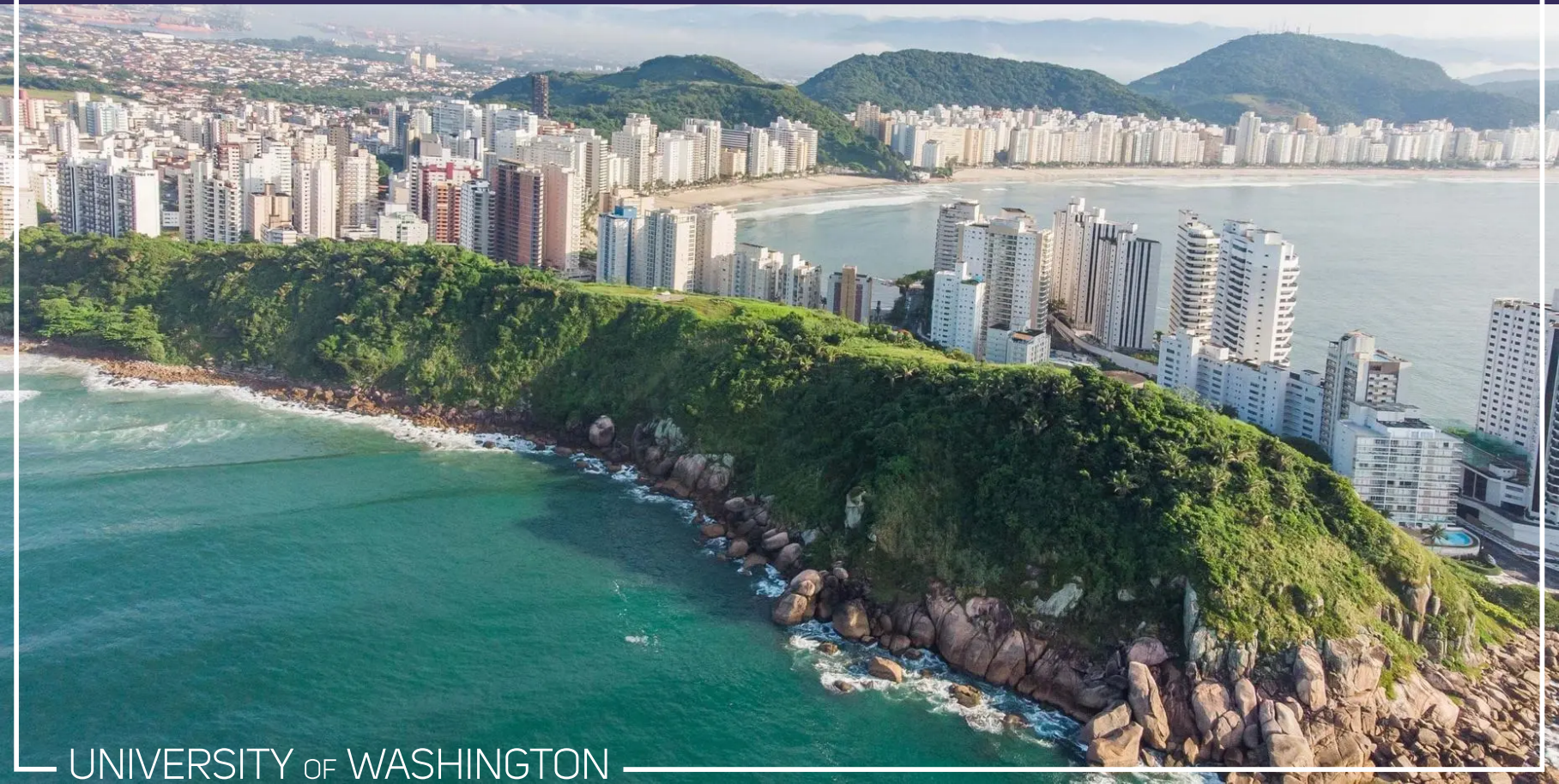
Importing, Exporting, and Cleaning Data

Adam Kuczynski



Today's Theme:

"Data Custodian Work"

Issues around getting data *in* and *out* of R and making it ready for your analyses:

- Working directories and projects
- Importing and Exporting data: `readr` and `haven`
- Cleaning and reshaping data: `tidyr`
- Dates and times: `lubridate`
- Controlling factor variables: `forcats`
- Working with strings: regular expressions and `stringr`

Working Directory

Your **working directory** is the folder on your computer where R will look for and save things by default

You can find your current working directory with `getwd()`

```
getwd()
```

```
## [1] "/home/adam/Documents/Class Materials/PSYCH 548 (Intro to R Programm
```

You can change your working directory using `setwd()`

```
setwd("/home/adam/Desktop")
```

R scripts are automatically run in the directory they are currently in. This means that, when you open a .R or a .Rmd file, your working directory is automatically set to that folder.[1]

[1] This only applies if RStudio is not already open.

Note: Windows users need to change back slashes (`\`) to forwarded slashes (`/`) for filepaths

Relative Paths

In your working directory, you can (and should!) refer to files using relative paths:

- `.` refers to your current working directory
- `..` refers to the folder your working directory is located in

Examples:

`./data/my_data.csv` refers to a file called "my_data.csv" located in the "data" subfolder of my working directory[1]

`../../figure1.png` refers to a file called "figure1.png" located two folders "up" from my working directory

---

The `here` package

The `here` package offers an easy and somewhat more reliable way of constructing relative file paths. When you load the `here` package at the top of your script, it looks for a `.here` file (which is automatically generated), allowing you to reference folders regardless of where the `.R` file is in the project structure. Check it out!

[1] The `./` is not strictly necessary. Often you will see it written as `data/my_data.csv`, which is perfectly fine!

Importing and Exporting Data

Helpful Packages

R has the ability to read and write data in a number of formats. Although much of this functionality is built into Base R, several packages help as well:

- `haven` (SPSS, Stata, and SAS files)
- `foreign` (SPSS, Stata, SAS, and other files)[1]
- `readxl` (MS Excel files)
- `googlesheets4` (communicate directly with Google Sheets)
- `readr` (enhances base R functionality)

The most common way to read/write data in R is with a `.csv` file!

[1] I've found this package to be a bit buggy and prefer to use `haven`

.csv Files

```
"mpg","cyl","disp","hp","drat","wt","qsec","vs","am","gear","carb"
21,6,160,110,3.9,2.62,16.46,0,1,4,4
21,6,160,110,3.9,2.875,17.02,0,1,4,4
22.8,4,108,93,3.85,2.32,18.61,1,1,4,1
21.4,6,258,110,3.08,3.215,19.44,1,0,3,1
18.7,8,360,175,3.15,3.44,17.02,0,0,3,2
18.1,6,225,105,2.76,3.46,20.22,1,0,3,1
14.3,8,360,245,3.21,3.57,15.84,0,0,3,4
24.4,4,146.7,62,3.69,3.19,20,1,0,4,2
```

Read .csv files with: `read.csv("data/my_data.csv")`

Write .csv files with: `write.csv("data/my_data_cleaned.csv)`

Alternatively you can use `readr`...

Read with `readr::read_csv("data/my_data.csv")`

Write with: `readr::write_csv("data/my_data_cleaned.csv)`

Alternative Formats

The `read.table()` (and `write.table()`) function is a generic function that can read delimited files of any kind. In fact, `read.csv()` is a special case of `read.table()`!

For example:

- To read .tsv files:
  `read.table("data/my_data.tsv", sep = "\t")`[1]
- To write .tsv files:
  `write.table("data/my_data.tsv", sep = "\t)`
- To read in files with values separated by whitespace:
  `read.table("data/my_data.tsv", sep = " ")`
- To write files with values separated by whitespace:
  `write.table("data/my_data.tsv", sep = " ")`

[1] Or `read.delim("data/my_data.tsv", sep = "\t)`

.RData Files

.RData[1] files are specific to R and allow you to store as many objects as you would like into a single file

To **write** a .RData file, use the `save()` function:

```
save(my_data_raw, my_data_cleaned, fit_lm, fit_lm_quadratic,
     file =  "MyStudy.RData")
```

To **read** a .RData file, use the `load()` function:

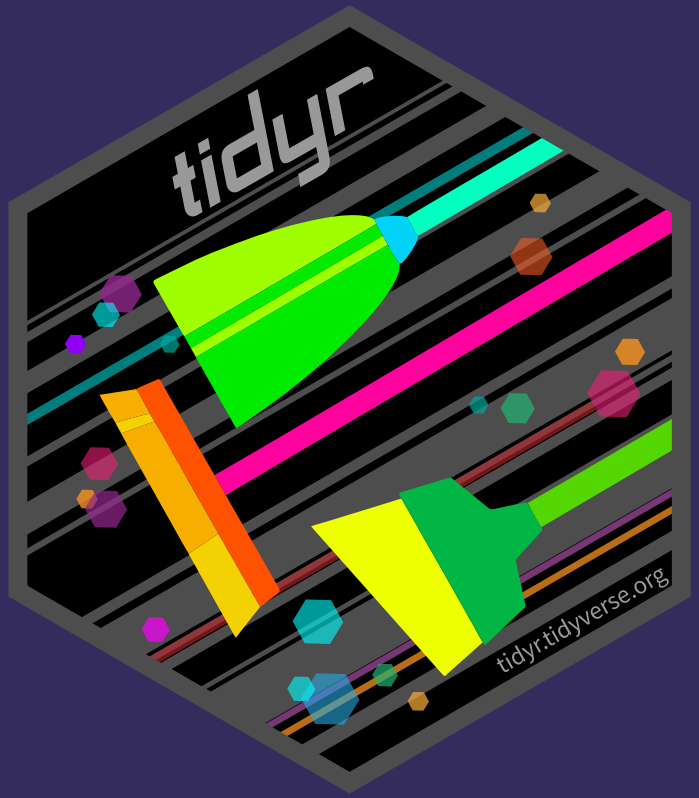
```
load(file = "MyStudy.RData")
```

☝ When you load .Rdata files, the objects *and their names* are loaded so you do not do this (like you do with `read.csv()`):

```
mydata <- load("MyStudy.RData") # 👈 Don't do this!
```

[1] The file extension can be whatever you want, but it defaults to .RData

Cleaning Data



Initial Spot Checks

First things to check after reading in data:

- Did *all* the data make it into R?
  - May need to use a different package or manually specify range
- Are the column names in good shape?
  - Modify `col_names=` or change with `colnames()` or `dplyr::rename()` function
- Are there "decorative" blank rows or columns to remove?
  - If possible, remove these from the source, otherwise do this with code (e.g., `data[-1, ]` to remove the first row)
- Are missing values (`""`, `NA`, `-999`, etc.) represented correctly?
  - Modify `na=` or change these after reading in the data
- Are the column classes correct? Numbers are numeric, strings are character, etc...
  - Modify with `col_types=` or change these after reading in the data

Slightly Messy Data

Area	FullProf	NotFullProf
Clinical	8	4
Social	3	1
Other	11	10

- What is an observation?
  - A group of faculty in a given area
- What are the variables
  - Area, Title
- What are the values?
  - Area: Clinical, Social, Other
  - Title: Full professor, Not full professor (as column names 👎)
  - Count: spread over two columns

Tidy Version

Area	Title	Count
Clinical	FullProf	8
Clinical	NotFullProf	4
Social	FullProf	3
Social	NotFullProf	1
Other	FullProf	11
Other	NotFullProf	10

✅ Each variable is a column

✅ Each observation is a row

✅ Ready for analyses and plotting!

Billboard data

```
# Notice how read.csv() can also take a URL
bb <- read.csv("https://adamkucz.github.io/psych548/data/billboard.csv")
```

year	artist	track	time	date.entered	wk1	wk2	wk3	wk4	wk5
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	87	82	72	77	87
2000	2Ge+her	The Hardest Part Of ...	3:15	2000-09-02	91	87	92	NA	NA
2000	3 Doors Down	Kryptonite	3:53	2000-04-08	81	70	68	67	66
2000	3 Doors Down	Loser	4:24	2000-10-21	76	76	72	69	67
2000	504 Boyz	Wobble Wobble	3:35	2000-04-15	57	34	25	17	17
2000	98^0	Give Me Just One Nig...	3:24	2000-08-19	51	39	34	26	26
2000	A*Teens	Dancing Queen	3:44	2000-07-08	97	97	96	95	100
2000	Aaliyah	I Don't Wanna	4:15	2000-01-29	84	62	51	41	38
2000	Aaliyah	Try Again	4:03	2000-03-18	59	53	38	28	21
2000	Adams, Yolanda	Open My Heart	5:30	2000-08-26	76	76	74	69	68

Week columns continue up to `wk76`!

Billboard

- What are the **observations** in the data?
  - Week since entering the Billboard Hot 100 per song
- What are the **variables** in the data?
  - Year, artist, track, song length, date entered Hot 100, week since first entered Hot 100 (spread over 76 columns 👎), rank during week (also spread over 76 columns 👎)
- What are the **values** in the data?
  - e.g., 2 Pac, Baby Don't Cry (Keep…, 4 mins 22 secs, Feb 26, 2000, week 3, rank 72

Tidy Data

Tidy data (i.e., "long data") are organized such that:

1. The values for a single observation are in their own row
2. The values for a single variable are in their own column
3. There is only one value per cell

Why organize your data in this way?

- Easier to understand many rows than many columns
- Required for most types of analyses
- Required for creating figures
- Fewer confusing variable names

The `tidyr` Package

The `tidyr` package is part of the `tidyverse` and provides functions that help clean ("tidy up") data:

- `pivot_longer()`: takes data in **wide** format and pivots them down to make two new columns
  - a `name` column that stores the original column names
  - a `value` column with the values of the original columns
- `pivot_wider()`: takes data in **long** format and pivots them into multiple columns (inverts `pivot_longer()`)
- `separate()`: pulls apart one column into multiple columns (common after `pivot_longer()` where values are embedded in column names)
- `extract()` works like `separate()` but takes a regular expression to define groups rather than separation value
  - `extract_numeric()` does a simple version of this and just extracts the numeric part

pivot_longer()

Let's use `pivot_longer()` to get the week and rank variables out of their current layout into two columns (more rows, fewer columns):

```
library(tidyr)
library(magrittr)

bb_long <- bb %>%
  pivot_longer(cols = matches("^wk"),
               names_to = "week",
               values_to = "rank")
```

We could instead use: `pivot_longer(wk1:wk76, names_to = "week", values_to = "rank")` to pull out these contiguous columns, however it is safer to reference columns by column name!

```
dim(bb_long)
```

```
## [1] 24092     7
```

`pivot`ed Weeks

year	artist	track	time	date.entered	week	rank
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk1	87
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk2	82
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk3	72
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk4	77
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk5	87
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk6	94
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk7	99
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk8	NA
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk9	NA
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	wk10	NA

Now we have a single week column!

Can we `pivot()` better?

```
summary(bb_long$rank)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.00   26.00   51.00   51.05   76.00  100.00   18785
```

This is an improvement, but we might not want to keep the 18785 rows with missing ranks (i.e., observations for weeks since entering the Hot 100 that the song was no longer on the Hot 100)

Pivoting Better: `values_drop_na`

The argument `values_drop_na = TRUE` to `pivot_longer()` will remove rows with missing ranks

```
bb_long <- bb %>%
  pivot_longer(cols = matches("^wk"),
               names_to = "week",
               values_to = "rank",
               values_drop_na = TRUE)

summary(bb_long$rank)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00   26.00   51.00   51.05   76.00  100.00
```

```
dim(bb_long)
```

```
## [1] 5307    7
```

No more `NA`s and way fewer rows!

parse_number()

The `week` column is character type, but we want it to be numeric:

```
class(bb_long$week)
```

```
## [1] "character"
```

The `tidyr::parse_number()` function will extract the numeric information from a string:

```
bb_long$week <- parse_number(bb_long$week)
class(bb_long$week)
```

```
## [1] "numeric"
```

`parse_number()` only extract the *first* set of numeric information. For more complex pattern matching, you'll need to use **regular expressions** (covered later today)

```
parse_number("123abc456")
```

```
## [1] 123
```

```
parse_number("abc123456def789")
```

```
## [1] 123456
```

names_prefix=

You can also use the `names_prefix=` argument in this case:

```
bb_long <- bb %>%
  pivot_longer(cols = matches("^wk"),
               names_to = "week",
               values_to = "rank",
               values_drop_na = TRUE,
               names_prefix = "wk",
               names_transform = list("week" = as.integer))
kable(head(bb_long, 3))
```

year	artist	track	time	date.entered	week	rank
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	1	87
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	2	82
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	3	72

separate()

The track length (`time`) column isn't ready for analysis. Let's convert it to length in seconds:

```
bb_long <- bb_long %>%
  separate(time,
           into = c("mins", "secs"),
           sep = ":",
           convert = T) %>%
  relocate(mins, secs, .after = "rank")


bb_long$length <- (bb_long$mins*60) + bb_long$secs
```

year	artist	track	date.entered	week	rank	mins	secs	length
2000	2 Pac	Baby Don't Cry (Keep...	2000-02-26	1	87	4	22	262
2000	2 Pac	Baby Don't Cry (Keep...	2000-02-26	2	82	4	22	262
2000	2 Pac	Baby Don't Cry (Keep...	2000-02-26	3	72	4	22	262

`relocate()` is from the `dplyr` package (also in the Tidyverse) And allows you to manually shuffle around the order of columns. We will talk about `dplyr` next week!

pivot_wider()

`pivot_wider()` does the exact oppose of `pivot_longer()`, and is used when you want several columns that each represent one observation.

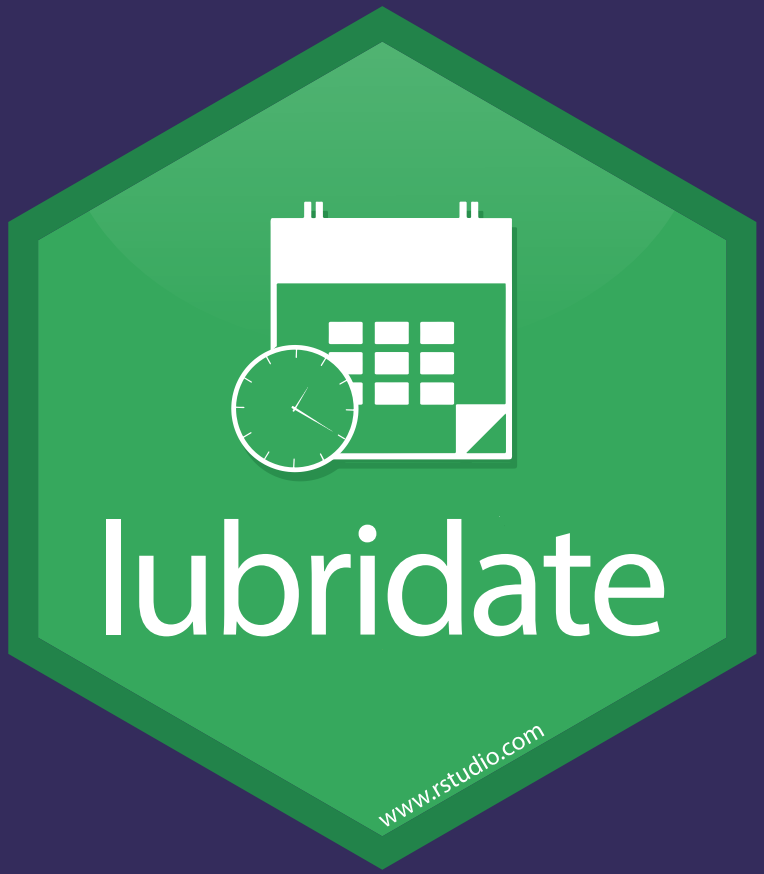
Although most regression analyses in R require your data to be in long format, some (e.g., SEM) require it to be in wide format.

Let's reshape `bb_long` back into wide format!

```
bb_wide <- bb_long %>%
  pivot_wider(names_from = "week",
              values_from = "rank",
              names_prefix = "wk")
```

year	artist	track	date.entered	mins	secs	length	wk1	wk2
2000	2 Pac	Baby Don't Cry (Keep...	2000-02-26	4	22	262	87	82
2000	2Ge+her	The Hardest Part Of ...	2000-09-02	3	15	195	91	87
2000	3 Doors Down	Kryptonite	2000-04-08	3	53	233	81	70
2000	3 Doors Down	Loser	2000-10-21	4	24	264	76	76

Dates and Times



Working with dates and times

Working with dates and times can be challenging, but fortunately R has nice built in capabilities and packages such as lubridate that are very helpful.

A note on Unix time (i.e., epoch time)

Unix time is the standard way computers keep track of date times, and you will likely encounter it at some point in your career. Unix time is the number of seconds since midnight UTC on Jan. 1, 1970.

For example, the exact time right now in Unix Time is: 1627414364

To convert epoch time in R, use the `as.POSIXct()` function. Here's the datetime in PDT when we hit 1 billion seconds after Jan 1, 1970:

```
as.POSIXct(1e9, origin = "1970-01-01")
```

```
## [1] "2001-09-08 18:46:40 PDT"
```

Datetime Conversion

Besides converting Unix Time into a more human-readable format, you will be converting datetimes from/into different formats

When converting date/datetime objects (regardless of the function), you will be using a set of **conversion specifications** which are technically unique to your OS but are widely shared across platforms. These specifications are represented as strings and start with a `%` followed by a letter. See `?strptime` for a full list of specifications

```
as.Date("20jul2021", "%d%b%Y")
```

```
## [1] "2021-07-20"
```

```
format(as.Date("2021-07-27"), "%B %d, 
```

```
## [1] "July 27, 2021"
```

```
as_datetime("July 27, 2021 01:00:00 PM",
            format = "%B %d, %Y %H:%M:%S %p",
            tz = "PDT")
```

```
## [1] "2021-07-27 01:00:00 PDT"
```

Math with Dates

All `Date` objects are in units of days:

```
as.Date("2021-07-27") + 1
```

```
## [1] "2021-07-28"
```

```
as.Date("2021-07-27") + 7
```

```
## [1] "2021-08-03"
```

The `lubridate` package offers some nice functions to help do this more clearly:

```
as.Date("2021-07-27") + days(1)
```

```
## [1] "2021-07-28"
```

```
as.Date("2021-07-27") + weeks(1)
```

```
## [1] "2021-08-03"
```

`lubridate` also has some nice helper functions:

```
mday(as.Date("2021-07-27"))
```

```
## [1] 27
```

```
wday(as.Date("2021-07-27"))
```

```
## [1] 3
```

```
weekdays(as.Date("2021-07-27"))
```

```
## [1] "Tuesday"
```

```
yday(as.Date("2021-07-27"))
```

```
## [1] 208
```

Other Useful Date Functions

- `lubridate::is.Date()` returns `TRUE` if an object is `Date`, else `FALSE`
- `lubridate::NA_Date_` is `NA` with class of `Date`
- `lubridate` **duration objects**
  - create specified durations of time (in seconds) that are not bound by conventions such as leap year and daylight savings time.
  - See `?duration` for the list of functions, which operate intuitively

---

`seq.Date()`

Much like `seq()` and its variants, `seq.Date()` creates a `Date` vector:

```
seq(from = as.Date("2021-07-27"),
    to = as.Date("2021-08-05"),
    by = "days")
```

```
##  [1] "2021-07-27" "2021-07-28" "2021
##  [6] "2021-08-01" "2021-08-02" "2021
```

```
seq(from = as.Date("2021-07-27"),
    to = as.Date("2021-09-04"),
    length.out = 10)
```

```
##  [1] "2021-07-27" "2021-07-31" "2021
##  [6] "2021-08-17" "2021-08-22" "2021
```

Getting Usable Dates from Billboard

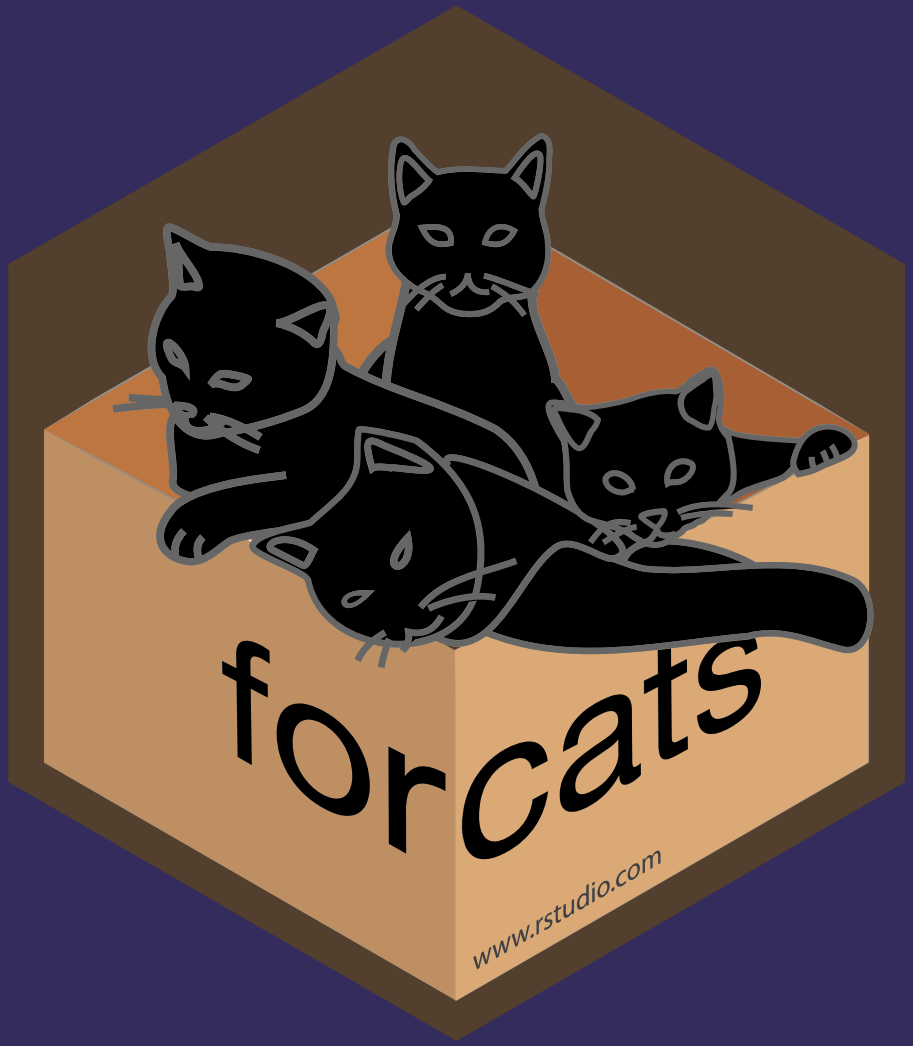
In `bb_long` we have the date the songs first hit the charts, but not the dates for the later weeks. However, we can calculate these dates (now that the data are in long form 🙌) from the `week` value

```
# bb_long$date <- as.Date(date.entered) + weeks(bb_long$week - 1)
# same as 👆 (we will discuss mutate() next week)
bb_long <- bb_long %>%
  mutate(date = as.Date(date.entered) + weeks(week - 1))
```

year	artist	track	date.entered	week	rank	mins	secs	length	date
2000	2 Pac	Baby Don't Cry (Keep...	2000-02-26	1	87	4	22	262	2000-02-26
2000	2 Pac	Baby Don't Cry (Keep...	2000-02-26	2	82	4	22	262	2000-03-04
2000	2 Pac	Baby Don't Cry (Keep...	2000-02-26	3	72	4	22	262	2000-03-11

First we had to convert `date.entered` from character to date. Then we added for each week after the song entered the top 100

Dealing with Factors



Factor Variables

Factors variables denote categorical data and are, of course, very common to work with in R, but they can be a bit fussy. The `forcats` package (part of the `tidyverse`) helps a lot!

Most regression analyses in R will take factor variables and do the dummy coding for you. When doing so, R takes the lowest level of a factor and uses it as the reference for a regression, which is often not desirable

The order of factor levels controls the order of categories in tables, on axes, in legends, and facets (subplots; discussed in a few weeks) in `ggplot2`

`forcats` functions all start with `fct_`, which helps a lot with RStudio autocomplete

To see all the functionality built into `forcats`, see the manual

fct_relevel()

`fct_relevel()` (similar to base R `relevel()`) allows you to reorder factor levels to any location

```
f <- factor(x = c("a", "b", "c", "d"),
            levels = c("b", "c", "d", "a"))
print(f)
```

```
## [1] a b c d
## Levels: b c d a
```

```
fct_relevel(f, "a")
```

```
## [1] a b c d
## Levels: a b c d
```

```
fct_relevel(f, "b", "a")
```

```
## [1] a b c d
## Levels: b a c d
```

```
fct_relevel(f, "a", after = 2)
```

```
## [1] a b c d
## Levels: b c a d
```

fct_recode()

`fct_recode()` allows you to change factor levels by hand. Non-mentioned factors will remain in the data:

```
x <- factor(c("apple", "apple", "cat", "banana", "dog"))
fct_recode(x, apples = "apple", bananas = "banana")
```

```
## [1] apples  apples  cat     bananas dog
## Levels: apples bananas cat dog
```

If you name the level `NULL` it will be replaced with `NA` in the data

```
fct_recode(x, NULL = "apple", bananas = "bananas")
```

```
## Warning: Unknown levels in `f`: bananas
```

```
## [1] <NA>   <NA>   cat    banana dog
## Levels: banana cat dog
```

fct_collapse()

`fct_collapse()` allows you to collapse factor levels into a smaller number of groups

```
party <- factor(c("strong dem", "dem", "ind", "rep", "strong rep",

fct_collapse(party,
             democrat = c("strong dem", "dem"),
             republican = c("strong rep", "rep"),
             other_level = "third_party")
```

```
## [1] democrat    democrat    third_party republican  republican  t
## [7] third_party
## Levels: democrat republican third_party
```

The `other_level_` argument allows you to collapse all non-mentioned levels into one level

fct_lump

The `fct_lump` function provides methods to lump levels together programatically:

```
x <- factor(rep(LETTERS[1:9], times = c(40, 10, 5, 27, 1, 1, 1, 1, 1)))
table(x)
```

```
## x
##  A  B  C  D  E  F  G  H  I
## 40 10  5 27  1  1  1  1  1
```

Lump together all levels except 3 most frequent

```
levels(
  fct_lump_n(x, 3)
)
```

```
## [1] "A"     "B"     "D"     "Other"
```

Lump together all levels less than `prop`

```
levels(
  fct_lump_prop(x, .20)
)
```

```
## [1] "A"     "D"     "Other"
```

Lump together the least frequent levels, ensuring that `other` remains the smallest level

```
levels(
  fct_lump_lowfreq(x)
)
```

```
## [1] "A"     "D"     "Other"
```

`fct_drop()` and `fct_expand()`

`fct_drop()` (similar to base R `droplevels()`) removes levels of the factor that are not in the data

```
x <- factor(c("a", "b"), levels = c("a", "b", "c", "d"))
fct_drop(x)
```

```
## [1] a b
## Levels: a b
```

```
fct_drop(x, only = "c")
```

```
## [1] a b
## Levels: a b d
```

`fct_expand()` does the opposite and *adds* levels to a factor

```
fct_expand(x, "e", "f", "g")
```

```
## [1] a b
## Levels: a b c d e f g
```

`fct_infreq()` and `fct_inorder()`

`fct_inorder()` orders the levels of factor by the order in which they appear in the data

```
x <- factor(c("c", "a", "b", "b", "c", "a", "b"))
levels(x)
```

```
## [1] "a" "b" "c"
```

```
levels(
  fct_inorder(x)
)
```

```
## [1] "c" "a" "b"
```

`fct_infreq()` orders the levels of a factor by the count of each level in the data (descending)

```
levels(
  fct_infreq(x)
)
```

```
## [1] "b" "a" "c"
```

fct_explicit_na

`fct_explicit_na()` gives NA values an explicit factor level, which makes sure they appear in summary information (e.g., tables) and plots

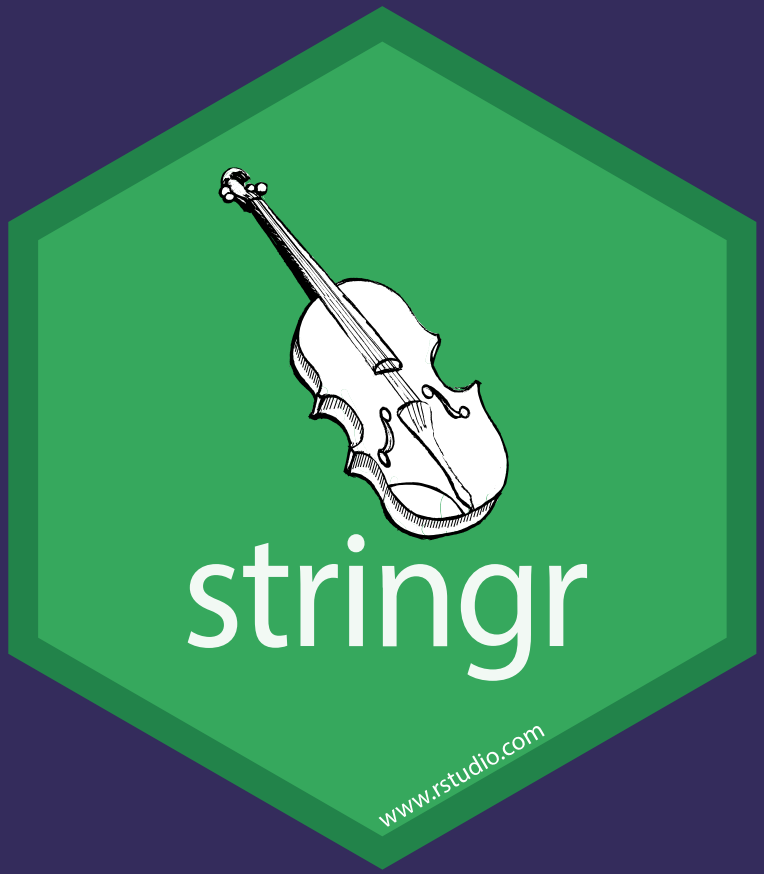
```
f1 <- factor(c("a", "a", NA, NA, "a", "b", NA, "c", "a", "c", "b"))
table(f1)
```

```
## f1
## a b c
## 4 2 2
```

```
f1 <- fct_explicit_na(f1)
table(f1)
```

```
## f1
##         a         b         c (Missing)
##         4         2         2         3
```

Working With Strings



Regular Expressions

A **regular expression** (i.e, **regex**) is a string that allows you to match, locate, and manage text data extremely flexibly. You write an **expression**, apply it to text input, and then do things with the **matches** you find.

- **Literal characters** are defined text snippets to search for (e.g., `Seattle`, `206`)
- **Metacharacters** provide flexibility in describing patterns:
  - backslash `\`, caret `^`, dollar sign `$`, period `.`, pipe `|`, question mark `?`, asterisk `*`, plus sign `+`, parentheses `(` and `)`, square brackets `[` and `]`, curly braces `{` and `}`
  - To search for a metacharacter as a literal character, you need to **escape** it with two preceding backslashes `\\`. For example to match `($1.50)` you'd need to write `\\(\\$1\\.50\\)`

Always test your regular expressions! Here is a helpful web app to make sure your regular expression is working the way you intend.

`grep()`, `gsub()`, et al.

`grep()` takes a regular expression (`pattern`) and a vector of text (`x`) and returns the index (`value = F`) or actual text (`value = T`) of the match. `grepl()` returns a logical vector specifiying where matches occurred:

```
text <- c("a", "b", "a b", "b a", "bac", "aba")
```

```
grep("a", text); grep("a", text, value
```

```
## [1] 1 3 4 5 6
```

```
## [1] "a"   "a b" "b a" "bac" "aba"
```

```
grepl("a", text)
```

```
## [1]  TRUE FALSE  TRUE  TRUE  TRUE  T
```

`gsub()` finds all matches and replaces them with specified text. `sub()` replaces just the first

```
gsub("a", "z", text)
```

```
## [1] "z"   "b"   "z b" "b z" "bzc" "z
```

```
sub("a", "z", text)
```

```
## [1] "z"   "b"   "z b" "b z" "bzc" "z
```

Regex Metacharacters

When placed in your search string, metacharacters help you evaluate strings flexibly (i.e., without hardcoding exactly what you're searching for)

```
text <- c("a", "b", "ab", "a b", "b a", "bac", "aba", "wxyz")
```

- `^` specifies the *beginning* of a string

```
grep("^a", text, value = T)
```

```
## [1] "a"   "ab"  "a b" "aba"
```

- `$` specifies the *end* of a string

```
grep("a$", text, value = T)
```

```
## [1] "a"   "b a" "aba"
```

- `.` matches *anything*

```
grep(".a$", text, value = T)
```

```
## [1] "b a" "aba"
```

- `?` means optionally match (i.e., "it might be there")

```
grep("^..?.$", text, value = T)
```

```
## [1] "ab"  "a b" "b a" "bac" "aba"
```

```
text <- c("a", "b", "ab","aba", "wxyz", "123", "1b34", "aaxy52", "aaabb4c")
```

- `|` means match this *or* that

```
grep("^a|4$", text, value = T)
```

```
## [1] "a"       "ab"      "aba"     "1b34"
```

- `{` and `}` are used to specify the number of characters to match. If 2 numbers, specifies minimum and maximum.

```
grep("^a{2,3}", text, value = T)
```

```
## [1] "aaxy52"  "aaabb4c"
```

- `[` and `]` specify a *group* of characters to match. `[a-z]` matches *all* lowercase letters, `[0-9]` matches *all* numbers. `[a-zA-Z0-9@]` matches all letters, numbers, and the @ symbol.

```
grep("[0-9]{2}$", text, value = T)
```

```
## [1] "123"    "1b34"   "aaxy52"
```

- `(` and `)` are used to create groups that you can reference in `gsub()`

```
gsub("^([a-z])(.*)$", "\\U\\1\\2_letter", t
```

```
## [1] "A_letter"       "B_letter"       "AB
## [5] "WXYZ_letter"    "123"            "1b
## [9] "AAABB4C_letter"
```

```
text <- c("a", "b", "ab", "aba", "wxyz", "123", "1b34", "aaxy52", "aaabb4c")
```

- `*` means match 0+ times.

```
grep("^a*b", text, value = T)
```

```
## [1] "b"       "ab"      "aba"     "a
```

- `+` means match 1+ times.

```
grep("^a+b", text, value = T)
```

```
## [1] "ab"      "aba"     "aaabb4c"
```

`+` and `*` are particularly useful after groups of characters. For example:

```
# Match anything that ends in 1 or more numbers
grep("[0-9]+$", text, value = T)
```

```
## [1] "123"    "1b34"   "aaxy52"
```

```
# Match anything that has at least 1 number surrounded by letters
grep("^[a-zA-Z]+[0-9]+.*[a-zA-Z]$", text, value = T)
```

```
## [1] "aaabb4c"
```

Metacharacter Shortcuts

- `[:alpha:]` matches all lowercase(`[:lower:]`) and uppercase (`[:upper:]`) letters (same as `[a-zA-Z]`)
- `[:digit:]` matches digits 0 through 9 (same as `[0-9]`)
- `[:alnum:]` matches all letters and digits (same as `[a-zA-Z0-9]`)
- `[:blank:]` matches all blank characters (spaces, tabs; same as `[\\s\\t]`
- `[:space:]` matches all space characters (tab, newline, etc.)
- `[:punct:]` matches all punctuation characters (`!` `"` `#` `$` `%` `&` `'` `(` `)` `*` `+` `,` `-` `.` `/` `:` `;` `<` `=` `>` `?` `@` `[` `\` `]` `^` `_` `` ` `` `{` `|` `}` `~`)
- `[:graph:]` matches all graphical characters (`[:alnum:]` and `[:punct:]`)

Lookarounds

A **lookahead** allows you to match characters that are (**positive lookahead** `(?=)`) or are not (**negative lookahead** `(?!)`) followed by certain characters

```
text <- c("1a", "2a", "3b", "4b", "5c")
grep("[0-9](?=[bc])", text, value = T, perl = T)
```

```
## [1] "3b" "4b" "5c"
```

```
grep("[0-9](?!a)", text, value = T, perl = T)
```

```
## [1] "3b" "4b" "5c"
```

A **lookbehind** allows you to match characters that are (**positive lookbehind** `(?<=)`) or are not (**negative lookbehind** `(?<!)`) preceded by certain characters

```
text <- c("1one", "2one", "3one", "4one")
grep("(?<=[1-3])one", text, value = T, perl = T)
```

```
## [1] "1one" "2one" "3one"
```

```
grep("(?<![14])one", text, value = T, perl = T)
```

```
## [1] "2one" "3one"
```

stringr

The `stringr` package from the `tidyverse` offers several useful functions when working with strings (including upgraded base R functions)

Functions in `stringr` begin with `str_`, which makes RStudio's autocomplete helpful

- `str_detect()` is equivalent to `grepl()` and can be `!grepl()` with `negate = T`
- `str_sub()` takes a substring based on `start` and `end` values. Negative values specify placement from the end of the string.

```
str_sub("Washington", 1, -3)
```

```
## [1] "Washingt"
```

- `str_length()` is equivalent to `nchar()` and returns the number of characters in a string
- `str_to_upper()`, `str_to_lower()`, and `str_to_title()` convert cases

```
str_to_upper("washington"); str_to_lower("WASHINGTON"); str_to_title(c("this is a title"))
```

```
## [1] "WASHINGTON"
```

```
## [1] "washington"
```

```
## [1] "This Is A Title"
```

- `str_trim()` removes extra leading or trailing whitespace

```
text <- c("  string       ", "mystring  ")
str_trim(text)
```

```
## [1] "string"   "mystring"
```

- `str_pad()` to pad a string with characters (default is whitespace)

```
str_pad("hadley", 30, "left")
```

```
## [1] "                        hadley"
```

```
str_pad("hadley", 30, "right")
```

```
## [1] "hadley                        "
```

```
str_pad("hadley", 30, "both")
```

```
## [1] "           hadley            "
```

- `str_count()` counts the number of matches in a string

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_count(fruit, "a")
```

```
## [1] 1 3 1 1
```

Food inspection data for practice

We will be using the "Food Establishment Inspection Data" from King County Dept. of Health.

```
restaurants <- read_csv("https://adamkucz.github.io/psych548/data/restaurants.csv",
                        col_types = "cccccccnncciclcciccc")
```

```
## Rows: 257,387
## Columns: 22
## $ Name                          <chr> "+MAS CAFE", "100 LB CLAM", "100 LB CLAM"…
## $ `Program Identifier`          <chr> "+MAS CAFE", "100 LB CLAM", "100 LB CLAM"…
## $ `Inspection Date`             <chr> "07/29/2020", "09/12/2019", "07/24/2017",…
## $ Description                   <chr> "Seating 0-12 - Risk Category III", "Seat…
## $ Address                       <chr> "1906 N 34TH ST", "1001 FAIRVIEW AVE N Un…
## $ City                          <chr> "SEATTLE", "SEATTLE", "SEATTLE", "SEATTLE…
## $ `Zip Code`                    <chr> "98103", "98109", "98109", "98109", "9810…
## $ Phone                         <chr> "(206) 491-4694", "(206) 369-2978", "(206…
## $ Longitude                     <dbl> -122.3346, -122.3317, -122.3317, -122.331…
## $ Latitude                      <dbl> 47.64818, 47.62902, 47.62902, 47.62902, 4…
## $ `Inspection Business Name`    <chr> "+MAS CAFE", "100 LB CLAM", "100 LB CLAM"…
## $ `Inspection Type`             <chr> "Consultation/Education - Field", "Routin…
## $ `Inspection Score`            <int> 0, 0, 25, 25, 25, 25, 0, 0, 25, 25, 25, 2…
## $ `Inspection Result`           <chr> "Satisfactory", "Incomplete", "Unsatisfac…
## $ `Inspection Closed Business` <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,…
## $ `Violation Type`              <chr> NA, NA, "BLUE", "RED", "RED", "RED", NA, …
## $ `Violation Description`       <chr> NA, NA, "3300 - Potential food contaminat…
## $ `Violation Points`            <int> 0, 0, 5, 5, 10, 5, 0, 0, 5, 5, 10, 5, 0, …
## $ Business_ID                   <chr> "PR0046367", "PR0085848", "PR0085848", "P…
## $ Inspection_Serial_Num         <chr> "DAZMP7KTI", "DAISVPYB0", "DAYYFZ1IJ", "D…
## $ Violation_Record_ID           <chr> NA, NA, "IV7PVOPQG", "IVYAWAZOU", "IVMPY7…
## $ Grade                         <chr> "1", "2", "2", "2", "2", "2", "2", "2", "…
```