Working with Data Frames

Spring R ’24

Dominic Bordelon, Research Data Librarian, ULS

Agenda

1. What is working with data frames “about”?
2. The pipe operator
3. Manipulating cases and variables
    - Sidebar: missing data (NA)
    - Sidebar: factors (categorical variables)
4. Summarizing and grouping cases
5. Reshaping (pivoting) data
6. Joining (merging) related tables

About the trainer

Dominic Bordelon, Research Data Librarian
University Library System, University of Pittsburgh
dbordelon@pitt.edu

Services for the Pitt community:

- Consultations
- Training (on-request and via public workshops)
- Talks (on-request and publicly)
- Research collaboration

Support areas and interests:

- Computer programming fundamentals, esp. for data processing and analysis
- Open Science and Data Sharing
- Data stewardship/curation
- Research methods; science and technology studies

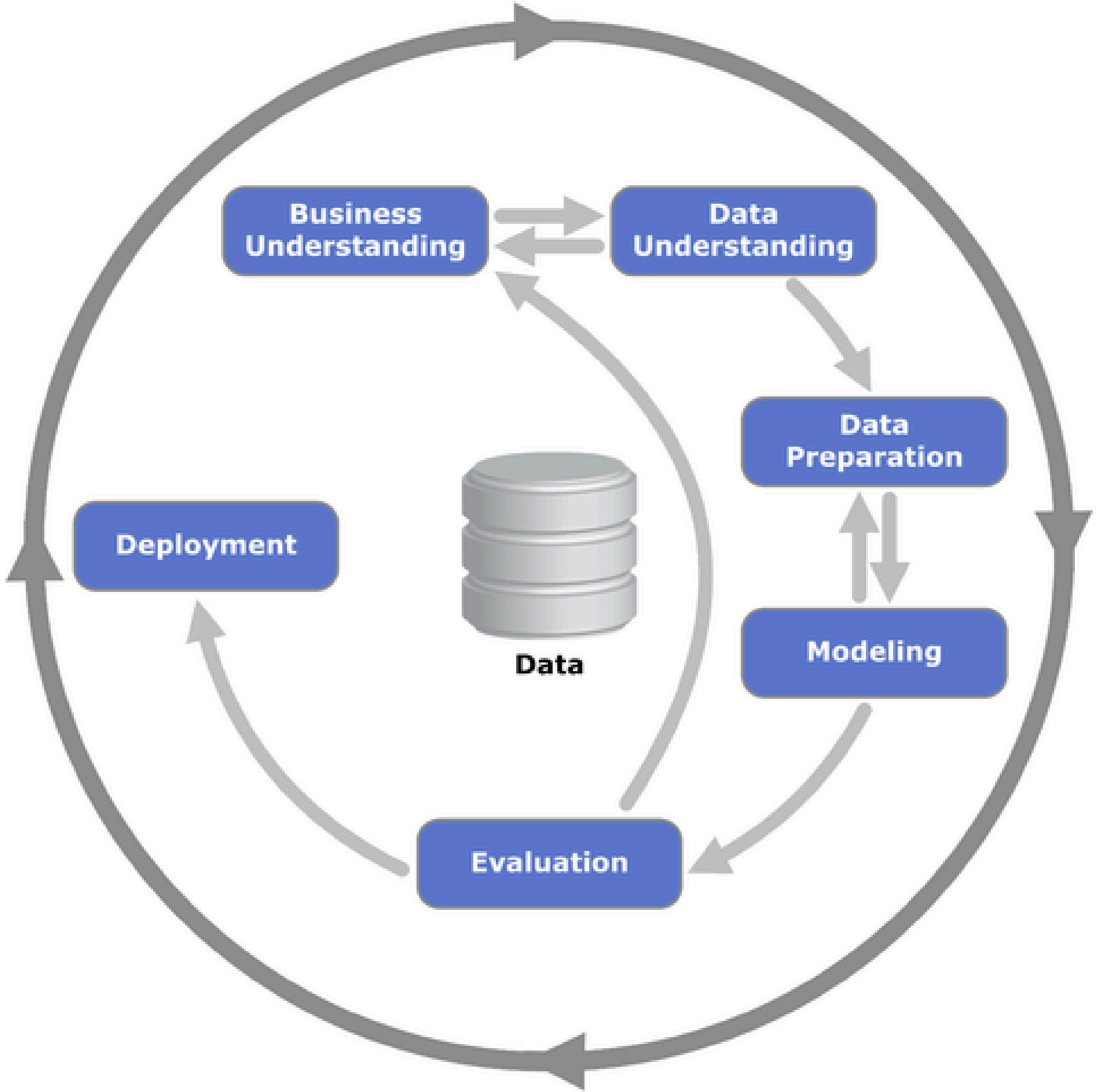
Spring R Series

#	Date	Title
1	2/22	Getting Started with Tabular Data
2 ⭐	**2/29**	**Working with Data Frames**
3	3/7	Data Visualization
4	3/21	Inference and Modeling Intro
5	3/28	Machine Learning Intro

What is working with data frames “about”?

Quick review: what is a data frame?

- A tabular format: rows (*observations*) and columns (*variables*)
- Analogous to a single table in an Excel worksheet
- Variables are *named* (e.g., patient ID, age, etc.), may be queried with `names(df)`, and may be accessed by `df$variable_name` (where `df` is the data frame of interest)
- **Each variable is a vector**
  - 💡 We can use vectorized operations/functions on any variable



Process diagram of the Cross-industry standard process for data mining (CRISP-DM).
Image credit: Kenneth Jensen, CC BY-SA 3.0, via Wikimedia Commons.

Data Understanding and Data Preparation

The functionality we're learning about today will enable us to:

- explore data (via summary statistics, filtering, and grouping) to enhance our Data Understanding
- clean (standardize), restructure, and combine data in Preparation for analysis

Anecdotally, "everyone is interested in modeling, but 90% of the work is in the prerequisite Business Understanding, Data Understanding, and Data Preparation."

The dplyr and tidyr packages

> dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges.

> The goal of tidyr is to help you create **tidy data**. Tidy data is data where:
>
> 1. Every column is variable.
> 2. Every row is an observation.
> 3. Every cell is a single value.

…and using penguins examples

```r
install.packages("palmerpenguins")
```

```r
library(palmerpenguins)

# load palmerpenguins' data into your environment:
data(penguins)
names(penguins)
```

```
[1] "species"           "island"            "bill_length_mm"
[4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
[7] "sex"               "year"
```

palmerpenguins is one of many examples of an R package which functions as a downloadable data set.

The pipe operator

Pipe operator, `%>%`

- A tidyverse feature (magrittr package)
- Written as: `%>%` (percent greater-than percent)
- `exprA %>% exprB` evaluates expression A, and then sends its output to expression B as input

```
my_values <- c(1.33, 1.66, 2.33)

mean(my_values) %>% round(1)

# is the same as:

round(mean(my_values), 1)
```

Note that the first argument of `round()` has disappeared in the piped version, because it is filled by the mean just calculated. `1` is the `digits` argument, i.e., one decimal place.

Pipelines

We can string together as many piped expressions as we want. For example, this code calculates temperature changes from three experimental trials, averages and rounds them, then prints a short statement:

```
t_initial <- c(25.04, 24.88, 25.23)
t_final <- c(35.82, 35.88, 35.67)

(t_final - t_initial) %>%
  mean() %>%
  signif(4) %>%
  paste("°C avg. ΔT")
```

```
[1] "10.74 °C avg. ΔT"
```

Pipeline object assignment

Like other expressions, a pipeline can have its result assigned to an object.

Revising the same example: the value is calculated and assigned in its own pipe (as `delta_t_avg`), then combined with text afterwards.

```
delta_t_avg <- (t_final - t_initial) %>%
  mean() %>%
  signif(4)

paste(delta_t_avg, "°C avg. ΔT")
```

```
[1] "10.74 °C avg. ΔT"
```

Benefits of the pipe

1. Avoid function wrapping (which is hard to read)
2. Avoid storing too many intermediate results in the environment (using object assignment)
3. The pipeline is easy to read as a procedure
4. Pipelines are easy to modify, e.g., to add new intermediate calls, or to cut them short when a problem has appeared.

Keyboard shortcuts revisited

Function	Windows	macOS
Execute line	Ctrl-Enter	⌘-Enter
Assignment operator `<-`	Alt - (Alt-hyphen)	⌥ - (Option-hyphen)
Pipe operator `%>%`	Ctrl-Shift-M	⌘-Shift-M

Exercise 2.1

Pipe operator

`ex2.1-pipe-operator.qmd`

Practice…

- attaching packages
- using the pipe operator
- loading a CSV file and getting summary statistics

💡 Ctrl-Shift-M / ⌘-Shift-M inserts `%>%` (pipe)

Manipulating cases and variables

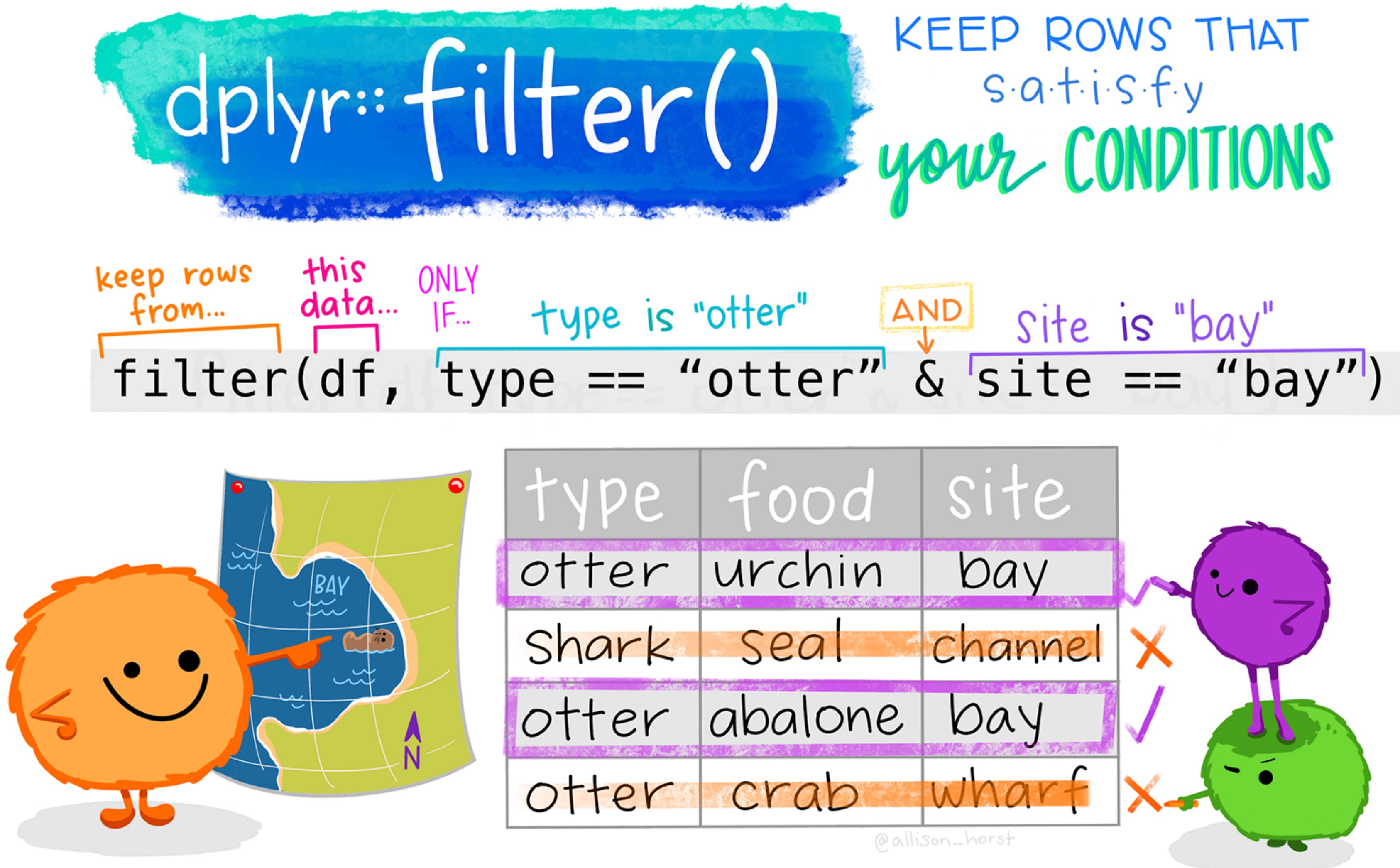
dplyr functions for manipulating cases and variables

For cases (rows):

- `filter()` returns cases which match 1+ logical condition(s)
- `arrange()` returns cases sorted according to 1+ variable(s)

For variables (columns):

- `select()` returns only certain variables of the data
- `rename()` renames variables
- `mutate()` creates new variables


dplyr::filter()
KEEP ROWS THAT
s·a·t·i·s·f·y
your CONDITIONS
keep rows from...
this data...
ONLY IF...
type is "otter"
AND
site is "bay"
filter(df, type == “otter” & site == “bay”)
BAY
N
type food site
otter urchin bay
shark seal channel
otter abalone bay
otter crab wharf
@allison_horst

`filter()` subsets cases

- `filter(df, ...)` where `df` is the data frame and `...` are 1+ logical expressions; each case that tests `TRUE` to the condition(s) will appear in the output
  - *Logical expression* means that the expression returns `TRUE` or `FALSE` when evaluated
- For the operations on the next slide, remember they are vectorized—the comparison is made to *each* value in the vector, and a vector of equal length is returned.

Useful logical operators and functions

Syntax	Example(s)	Name	Notes
`< <= >` `>= ==`	`age <= 34` `treatment_group == "A"`	Comparators	`==` means "equals" and works with both numeric and character data.
`%in%`	`age %in% 30:35` `treatment_group %in% c("A", "B")`	Membership (or "in") operator	Asks, "is the value found in vector y?"
`is.na()`	`is.na(age)` `is.na(treatment_group)`	is.na	Asks, "is the value missing?" Missingness is represented in R with NA (see next slide)
`& |`	`age < 34 & age >= 25`	Boolean AND and OR operators	Combine logical expressions

Sidebar: missing data (NA)

Missing data or the concept of missingness is represented in R with the symbol `NA` (*not* the string `"NA"` in quotation marks), for “Not Available”. It is equivalent to an empty cell in Excel.

What `NA` can mean (depending on context)

- Incomplete observation and/or mistake in data entry
- Chose not to answer
- Variable not relevant for this case

Consequences of `NA`

- Can’t do math with `NA` (`NA` $\neq 0$), or make number-line comparisons
- `NA` must be removed for arithmetic such as `mean()`; see `na.rm` argument and similar for many functions
- Cases having `NA` for a variable of interest may need to be dropped or imputed for analysis

Examples for `filter()`

🐧

```r
# fetch observations of Gentoo species
penguins %>% filter(species == "Gentoo")

# which female penguins have a bill longer than 40 mm?
penguins %>% filter(bill_length_mm > 40 & sex == "female")

# which penguins do not have a body mass recorded?
penguins %>% filter(is.na(body_mass_g))
```

`arrange()` sorts cases

- `arrange(df, ...)` returns `df` with cases sorted according to one or more variables
- To sort in reverse (descending) order, wrap the variable name in `desc()`

```
# sort penguins with longest bill first
penguins %>% arrange(desc(bill_length_mm))

# sort penguins by island and then by species (alphabetically)
penguins %>% arrange(island, species)
```

`select()` and `rename()` work on variables

- `select(df, v1, v2, v3)`, where `v1` etc. are variable names, is for selecting which variables you want to retain in the data frame.
  - Numbers also work as an alternative to names
  - Use a negative sign (`-`) to *negate* a column (i.e., “all variables except…”)
  - Note: dropped variables are removed from the data frame, not merely hidden!
- `rename(df, new_name = old_name)` renames a variable `old_name` to `new_name`

```
# select the species column, and cols 4 through 8 except for 5:
penguins %>% select(species, 4:8, -5)

# rename species to "common_name"
penguins %>% rename(common_name = species)
```

`mutate()` creates a new variable in the data frame

- `mutate(df, new_variable = expr)` creates `new_variable` in `df`
- `expr` may be: a mathematical expression, a function call, a vector of appropriate length, or a fixed value. `expr` may also implement "if-then" logic, using one or more variables of the same case (e.g., "If temperature is above 90, then heat category is High").

```r
# isolate body masses, then convert penguin mass from g to kg and lbs:
penguins %>%
  select(body_mass_g) %>%
  mutate(body_mass_kg = body_mass_g / 1000,
         body_mass_lbs = body_mass_g / 453.6)

# to store the result back to penguins:
penguins <- penguins %>%
  mutate(body_mass_kg = body_mass_g / 1000,
         body_mass_lbs = body_mass_g / 453.6)
```

Sidebar: factors

A common `mutate()` task is to convert an existing variable's type or measurement units. For example, our `penguins_raw$Island` variable is encoded as character data, but we would like to convert it to a categorical variable. In R, a categorical variable is encoded in a **factor**, a vector which accepts only certain values. Factors may be ordered.

```
# penguins_raw is a version of penguins without factors encoded

summary(penguins_raw$Island) # no level counts

# convert Island to factor, and assign result back to penguins_raw:
penguins_raw <- penguins_raw %>%
  mutate(Island = as_factor(Island))

summary(penguins_raw$Island) # levels with counts!
```

Summarizing and grouping cases

`summarize()` returns a single row of summary calculations

- Aggregation functions
  - center: `mean()`, `median()`
  - spread: `sd()`, `IQR()`
  - range: `min()`, `max()`
  - position: `first()`, `last()`, `nth()`
  - count: `n()`, `n_distinct()`
  - logical: `any()`, `all()`
- To name the columns in the output data frame, use named arguments.
- Usually we will summarize after grouping (next section)

Examples for `summarize()`

The `glimpse()` function is used because its compact vertical view is perfect for a single-row table.

```
# median, mean, and SD of body mass:
penguins %>%
  summarize(count = n(),
            median_mass = median(body_mass_g, na.rm = TRUE),
            mean_mass = mean(body_mass_g, na.rm = TRUE),
            sd_mass = sd(body_mass_g, na.rm = TRUE)) %>%
  glimpse()

# mean of each numeric variable:
penguins %>%
  summarize(across(is.numeric, mean, na.rm=TRUE)) %>%
  glimpse()
```

`group_by()` creates groupings using a variable

- `group_by(v1)` groups cases in the data according to their value for the variable (factor) `v1`.
- Grouping information is appended to the data frame as metadata
- `summarize()` understands these groups and applies function calls to the groups, rather than the whole data set, returning one row for each group
- `group_by(v1, v2)` groups cases by `v1` and then `v2` (order does matter)

Examples for group_by()

```
# mean and sd body mass of each observed species:
penguins %>%
  group_by(species) %>%
  summarize(n = n(),
            mean_mass = mean(body_mass_g, na.rm=TRUE),
            sd_mass = sd(body_mass_g, na.rm=TRUE)) %>%
  glimpse()

# mean of each numeric variable for each species and sex:
penguins %>%
  group_by(species, sex) %>%
  summarize(n = n(),
            across(is.numeric, mean, na.rm=TRUE))
# note groups for sex == NA
```

Exercise 2.2

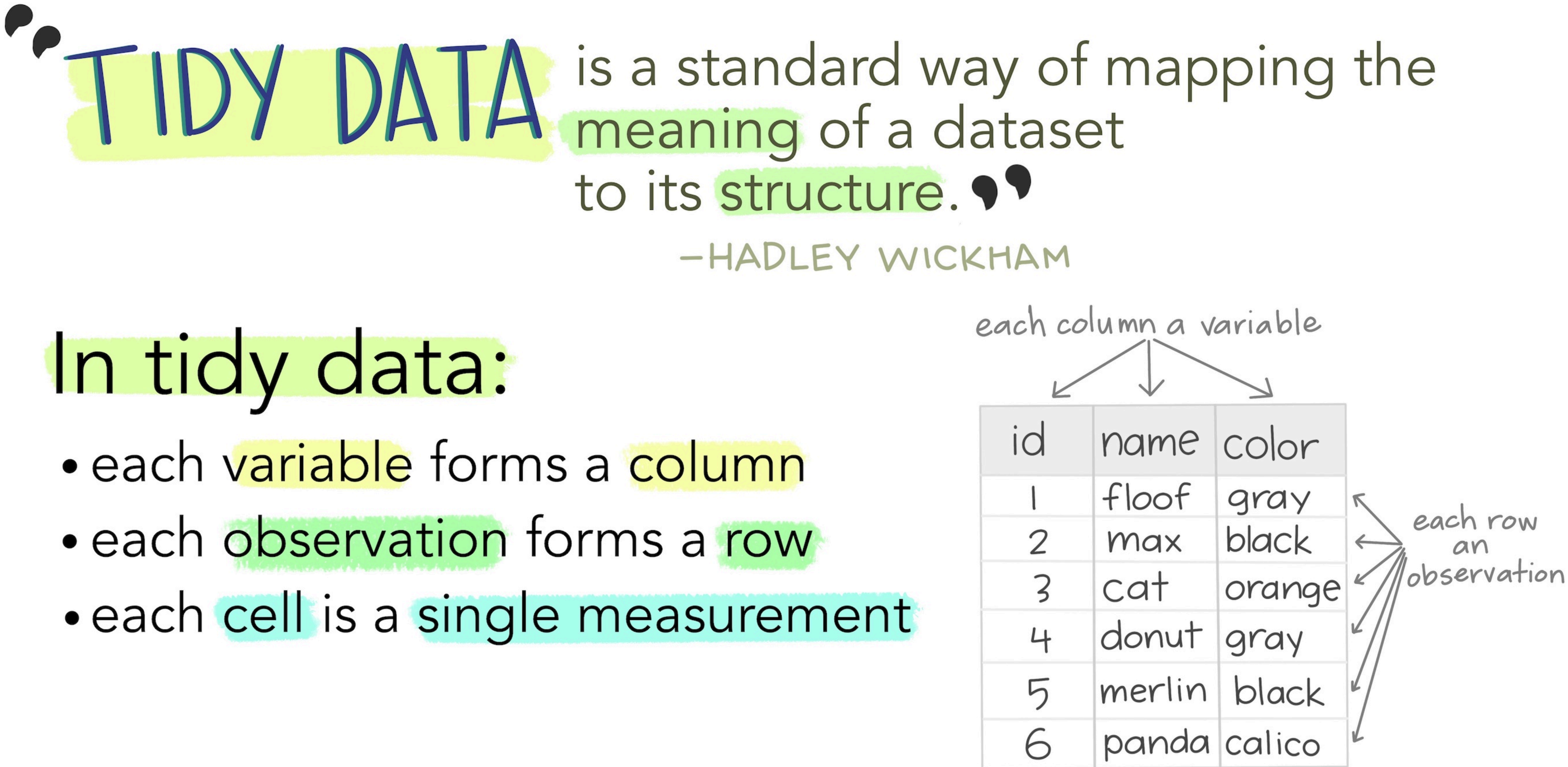
Data frame manipulation

`ex2.2-data-frame.qmd`

Practice…

- using dplyr functions to interact with a data frame
- using multiple functions together in a pipeline
- grouping and summarizing observations

Reshaping (pivoting) data



Wickham, H. (2014). Tidy Data. Journal of Statistical Software *59* (10). DOI: 10.18637/jss.v059.i10

Illustration from the Openscapes blog *Tidy Data for reproducibility, efficiency, and collaboration* by Julia Lowndes and Allison Horst

`pivot_longer()` collapses 3+ columns into two

…creating more rows (a “longer” data frame)

`pivot_longer(cols, names_to, values_to)` where

- `cols` is a vector of variable names (or selection such as using `across()`), whose columns you want to collapse
- `names_to` is the name of a new variable which will receive the *names* of the collapsing columns
- `values_to` is the name of a new variable which will receive the *values* of the collapsing columns

`pivot_longer()` example: religion and income

tidyr's `relig_income` dataset, from a Pew religion and income survey, has 1 row per *religion* and a column for the *count* of people in each income category. Let's treat each count as its own observation. (This and following example are from the tidyr Pivoting vignette, also available by running `vignette("pivot")`.)

```
data(relig_income)
names(relig_income)
# sample 5 random rows:
slice_sample(relig_income, n = 5)

relig_income_long <- relig_income %>%
  pivot_longer(cols = 2:11,
               names_to = "income_category",
               values_to = "count")

slice_sample(relig_income_long, n = 5)
```

`pivot_wider()` expands two columns into more

…creating more variables (a “wider” data frame)

`pivot_wider(names_from, values_from)` where

- `names_from` is the name of a variable whose values will form the *names* of the new variables
- `values_from` is the name of a variable whose values will form the *values* of the new variables, corresponding to the appropriate name

`pivot_wider()` example: fish encounters

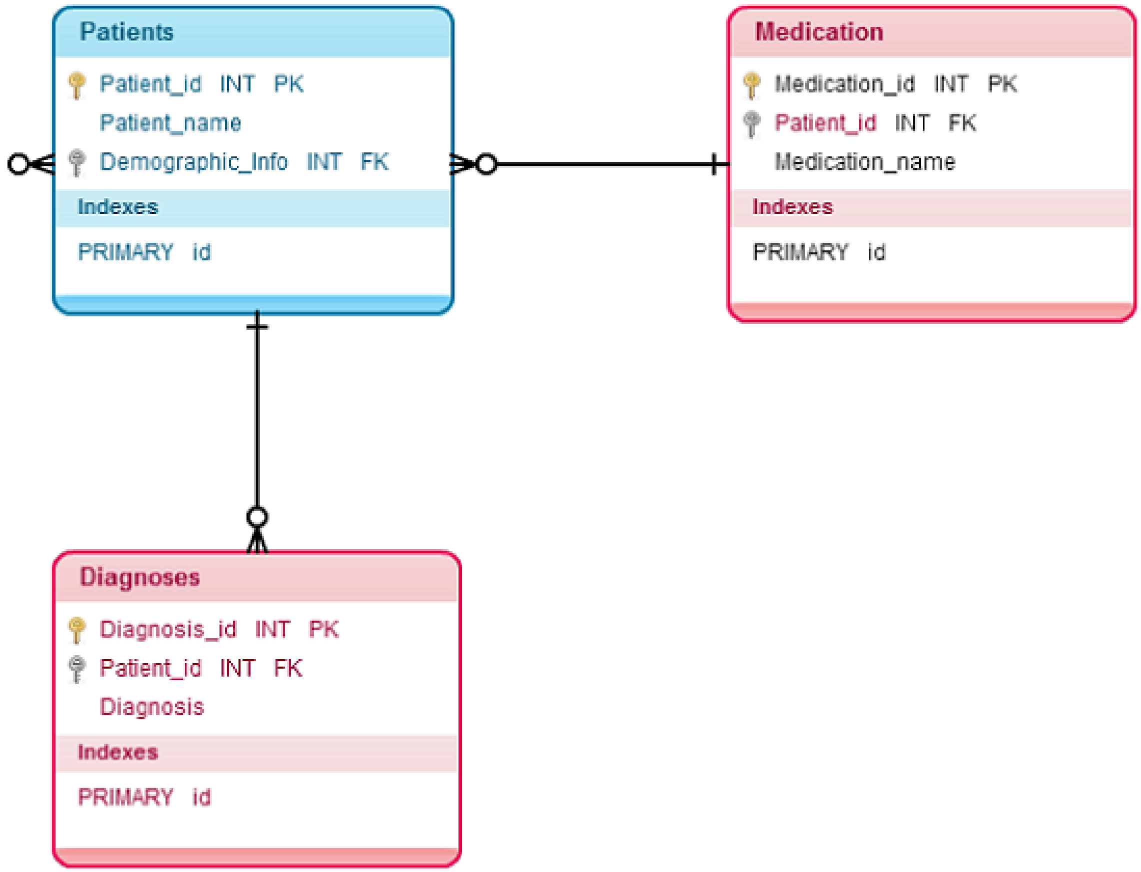
> It's relatively rare to need `pivot_wider()` to make tidy data, but it's often useful for creating summary tables for presentation, or data in a format needed by other tools. The `fish_encounters` dataset, contributed by Myfanwy Johnston, describes when fish swimming down a river are detected by automatic monitoring stations. Many tools used to analyse this data need it in a form where each station is a column.

```r
data(fish_encounters)
names(fish_encounters)
# sample 5 random rows:
slice_sample(fish_encounters, n = 5)

fish_encounters_wide <- fish_encounters %>%
  pivot_wider(names_from = station,
              values_from = seen)

# use glimpse() for previewing a wide table
slice_sample(fish_encounters_wide, n = 5) %>%
  glimpse()
```

Joining (merging) related tables

The relational model

Data of interest often “live” in more than one table.



Detail of a relational database diagram, relating Patients records to their Diagnoses and Medications. Image credit: Tsedenjav.Sh, CC BY-SA 4.0, via Wikimedia Commons.

`left_join()` treats the *left* table as primary

- `left_join(x, y, by)` where `x` is the left table, `y` is the right table, and `by` is the variable name by/on which to join
- All rows in `x` will be retained in the output, whether they match in `y` or not
- The number of rows in the output will depend on how many matches there are in `y` for each row of `x`

`left_join()` example: penguin species

Let us supplement our penguin observations with a table of information about the penguin species. Note that not all of our observed species in x exist in y. Note also that there is a species in y that we have not observed in x.

```
species_info <- read_excel("data/penguin-species.xlsx")
species_info
penguins %>%
  left_join(y=species_info, by="species")
```

Other tabular combinations

- More joins
  - `right_join()` treats the *right* df as the primary table, keeping all its rows
  - `inner_join()` returns the minimal set, because it requires values in *both* tables
  - `full_join()` returns the maximal set, keeping all rows from both df's
- Row- and column-binding: combining tables non-relationally (bind_rows(), bind_cols())
- Set operations
  - union
  - intersect
  - setdiff

Wrap up

Session in review

Today we learned about:

- The pipe operator
- Manipulating data frames
- Grouping and summarizing data to better understand it
- Reshaping and joining of tables

Join us next week for data visualization!