A crash course on R for data analysis

Clemens Schmid

TOC

The working environment

Getting started for this workshop

- Activate the relevant conda environment (don't forget to deactivate it later!)

```
conda activate r-python
```

- Navigate to

```
/vol/volume/3b-1-introduction-to-r-and-the-tidyverse/spaam_r_tidyverse_intro_2h
```

- Pull the latest changes in this Git repository

```
git pull
```

- Open RStudio
- Load the project with `File > Open Project...`
- Open this file `presentation.Rmd` in RStudio

R, RStudio and the tidyverse

- R is a fully featured programming language, but it excels as an environment for (statistical) data analysis (https://www.r-project.org)
- RStudio is an integrated development environment (IDE) for R (and other languages): (https://www.rstudio.com/products/rstudio)
- The tidyverse is a collection of packages with well-designed and consistent interfaces for the main steps of data analysis: loading, transforming and plotting data (https://www.tidyverse.org)
  - This introduction works with tidyverse ~v1.3.0
  - We will learn about `readr`, `tibble`, `ggplot2`, `dplyr`, `magrittr` and `tidyr`
  - `forcats` will be briefly mentioned
  - `purrr` and `stringr` are left out

Loading data into tibbles

Reading data with readr

- With R we usually operate on data in our computer's memory
- The tidyverse provides the package `readr` to read data from text files into the memory
- `readr` can read from our file system or the internet
- It provides functions to read data in almost any (text) format:

```
readr::read_csv()   # .csv files
readr::read_tsv()   # .tsv files
readr::read_delim() # tabular files with an arbitrary separator
readr::read_fwf()   # fixed width files
readr::read_lines() # read linewise to parse yourself
```

- `readr` automatically detects column types – but you can also define them manually

How does the interface of `read_csv` work?

- We can learn more about a function with `?`. To open a help file: `?readr::read_csv`
- `readr::read_csv` has many options to specify how to read a text file

```
read_csv(
  file,                         # The path to the file we want to read
  col_names = TRUE,             # Are there column names?
  col_types = NULL,             # Which types do the columns have? NULL -> auto
  locale = default_locale(),    # How is information encoded in this file?
  na = c("", "NA"),             # Which values mean "no data"
  trim_ws = TRUE,               # Should superfluous white-spaces be removed?
  skip = 0,                     # Skip X lines at the beginning of the file
  n_max = Inf,                  # Only read X lines
  skip_empty_rows = TRUE,       # Should empty lines be ignored?
  comment = "",                 # Should comment lines be ignored?
  name_repair = "unique",       # How should "broken" column names be fixed
  ...
)
```

What does readr produce? The tibble!

```
samples <- readr::read_tsv(sample_table_url)
```

- The tibble is a "data frame", a tabular data structure with rows and columns
- Unlike a simple array, each column can have another data type

```
print(samples, n = 3)
```

```
## # A tibble: 1,060 x 16
##   project_name publication_year publication_doi     site_name latitude longitude
##   <chr>                   <dbl> <chr>               <chr>        <dbl>     <dbl>
## 1 Warinner2014             2014 10.1038/ng.2906     Dalheim      51.6       8.84
## 2 Warinner2014             2014 10.1038/ng.2906     Dalheim      51.6       8.84
## 3 Weyrich2017              2017 10.1038/nature21674 Gola For~     7.66    -10.8
## # ... with 1,057 more rows, and 10 more variables: geo_loc_name <chr>,
## #   sample_name <chr>, sample_host <chr>, sample_age <dbl>,
## #   sample_age_doi <chr>, community_type <chr>, material <chr>, archive <chr>,
## #   archive_project <chr>, archive_accession <chr>
```

How to look at a tibble?

```
samples          # Typing the name of an object will print it to the console
str(samples)     # A structural overview of an object
summary(samples) # A human-readable summary of an object
View(samples)    # RStudio's interactive data browser
```

- R provides a very flexible indexing operation for `data.frames` and `tibbles`

```
samples[1,1]                             # Access the first row and column
samples[1,]                              # Access the first row
samples[,1]                              # Access the first column
samples[c(1,2,3),c(2,3,4)]               # Access a selection of rows and columns
samples[,-c(1,2)]                        # Remove the first two columns
samples[,c("site_name", "material")]     # Columns can be selected by name
```

- `tibbles` are mutable data structures, so their content can be overwritten

```
samples[1,1] <- "Cheesecake2015"         # replace the first value in the first column
```

Plotting data in `tibbles`

ggplot2 and the "grammar of graphics"

- ggplot2 offers an unusual, but powerful and logical interface
- The following example describes a stacked bar chart

```
library(ggplot2) # Loading a library to use its functions without ::
```

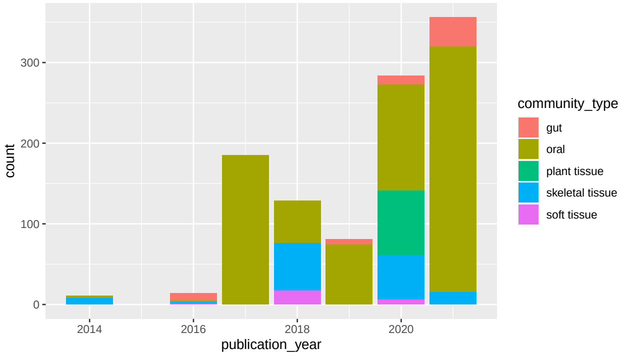
```
ggplot(            # Every plot starts with a call to the ggplot() function
  data = samples # This function can also take the input tibble
) +                # The plot consists of functions linked with +
geom_bar(          # "geoms" define the plot layers we want to draw
  mapping = aes( # The aes() function maps variables to visual properties
    x = publication_year, # publication_year -> x-axis
    fill = community_type # publication_year -> fill color
  )
)
```

- geom_*: data + geometry + statistical transformation + repositioning

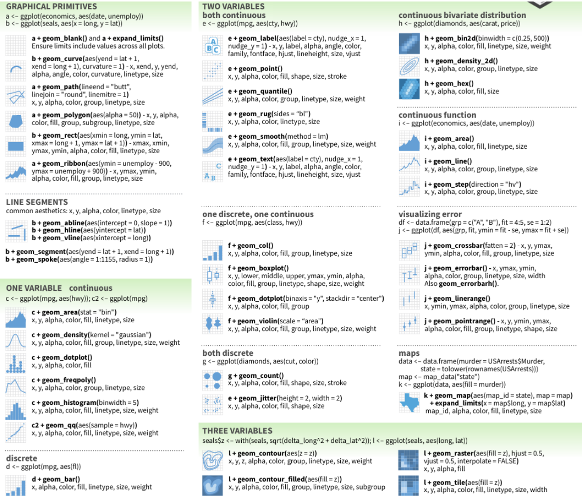
ggplot2 and the “grammar of graphics”

- This is the plot described above: number of samples per community type through time

```
ggplot(samples) +
geom_bar(aes(x = publication_year, fill = community_type))
```



ggplot2 features many geoms



GRAPHICAL PRIMITIVES
a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))

a + geom_blank() and a + expand_limits() Ensure limits include values across all plots.
b + geom_curve(aes(yend = lat + 1, xend = long + 1), curvature = 1) - x, xend, y, yend, alpha, angle, color, curvature, linetype, size
a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1) x, y, alpha, color, group, linetype, size
a + geom_polygon(aes(alpha = 50)) - x, y, alpha, color, fill, group, subgroup, linetype, size
b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1)) - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size
a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)) - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS
common aesthetics: x, y, alpha, color, linetype, size
b + geom_abline(aes(intercept = 0, slope = 1))
b + geom_hline(aes(yintercept = lat))
b + geom_vline(aes(xintercept = long))
b + geom_segment(aes(yend = lat + 1, xend = long + 1))
b + geom_spoke(aes(angle = 1:1155, radius = 1))

ONE VARIABLE continuous
c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)
c + geom_area(stat = "bin") x, y, alpha, color, fill, linetype, size
c + geom_density(kernel = "gaussian") x, y, alpha, color, fill, group, linetype, size, weight
c + geom_dotplot() x, y, alpha, color, fill
c + geom_freqpoly() x, y, alpha, color, group, linetype, size
c + geom_histogram(binwidth = 5) x, y, alpha, color, fill, linetype, size, weight
c2 + geom_qq(aes(sample = hwy)) x, y, alpha, color, fill, linetype, size, weight

discrete
d <- ggplot(mpg, aes(fl))
d + geom_bar() x, alpha, color, fill, linetype, size, weight

TWO VARIABLES
both continuous
e <- ggplot(mpg, aes(cty, hwy))
e + geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust
e + geom_point() x, y, alpha, color, fill, shape, size, stroke
e + geom_quantile() x, y, alpha, color, group, linetype, size, weight
e + geom_rug(sides = "bl") x, y, alpha, color, linetype, size
e + geom_smooth(method = lm) x, y, alpha, color, fill, group, linetype, size, weight
e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

one discrete, one continuous
f <- ggplot(mpg, aes(class, hwy))
f + geom_col() x, y, alpha, color, fill, group, linetype, size
f + geom_boxplot() x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight
f + geom_dotplot(binaxis = "y", stackdir = "center") x, y, alpha, color, fill, group
f + geom_violin(scale = "area") x, y, alpha, color, fill, group, linetype, size, weight

both discrete
g <- ggplot(diamonds, aes(cut, color))
g + geom_count() x, y, alpha, color, fill, shape, size, stroke
e + geom_jitter(height = 2, width = 2) x, y, alpha, color, fill, shape, size

continuous bivariate distribution
h <- ggplot(diamonds, aes(carat, price))
h + geom_bin2d(binwidth = c(0.25, 500)) x, y, alpha, color, fill, linetype, size, weight
h + geom_density_2d() x, y, alpha, color, group, linetype, size
h + geom_hex() x, y, alpha, color, fill, size

continuous function
i <- ggplot(economics, aes(date, unemploy))
i + geom_area() x, y, alpha, color, fill, linetype, size
i + geom_line() x, y, alpha, color, group, linetype, size
i + geom_step(direction = "hv") x, y, alpha, color, group, linetype, size

visualizing error
df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))
j + geom_crossbar(fatten = 2) - x, y, ymax, ymin, alpha, color, fill, group, linetype, size
j + geom_errorbar() - x, ymax, ymin, alpha, color, group, linetype, size, width Also geom_errorbarh().
j + geom_linerange() x, ymin, ymax, alpha, color, group, linetype, size
j + geom_pointrange() - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

maps
data <- data.frame(murder = USArrests$Murder, state = tolower(rownames(USArrests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))
k + geom_map(aes(map_id = state), map = map) + expand_limits(x = map$long, y = map$lat) map_id, alpha, color, fill, linetype, size

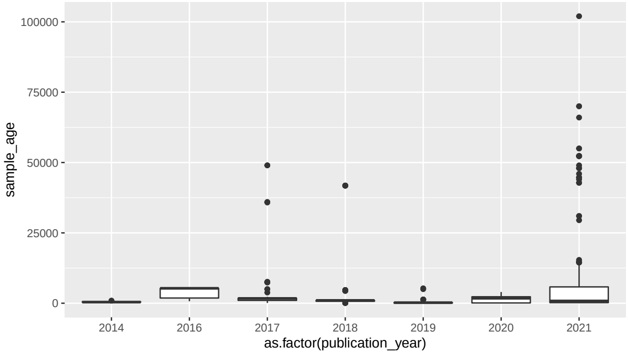
THREE VARIABLES
seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))
l + geom_contour(aes(z = z)) x, y, z, alpha, color, group, linetype, size, weight
l + geom_contour_filled(aes(fill = z)) x, y, alpha, color, fill, group, linetype, size, subgroup
l + geom_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE) x, y, alpha, fill
l + geom_tile(aes(fill = z)) x, y, alpha, color, fill, linetype, size, width

- RStudio shares helpful cheatsheets for the tidyverse and beyond: https://www.rstudio.com/resources/cheatsheets

scales control the behaviour of visual elements

- Another plot: Boxplots of sample age through time

```
ggplot(samples) +
geom_boxplot(aes(x = as.factor(publication_year), y = sample_age))
```

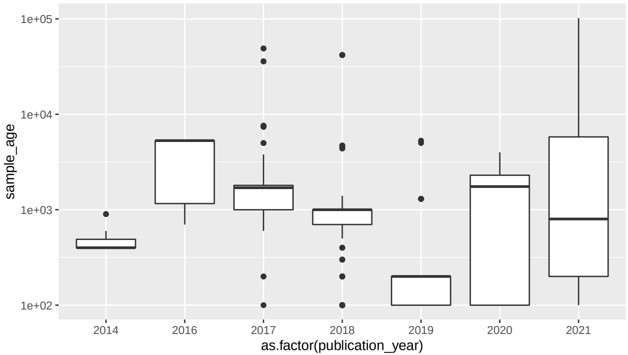


- This is not well readable, because extreme outliers dictate the scale

scales control the behaviour of visual elements

- We can change the **scale** of different visual elements - e.g. the y-axis

```
ggplot(samples) +
geom_boxplot(aes(x = as.factor(publication_year), y = sample_age)) +
scale_y_log10()
```

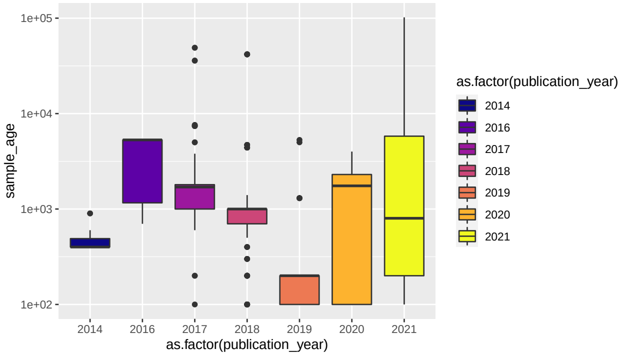


- The log-scale improves readability

scales control the behaviour of visual elements

- (Fill) color is a visual element of the plot and its scaling can be adjusted

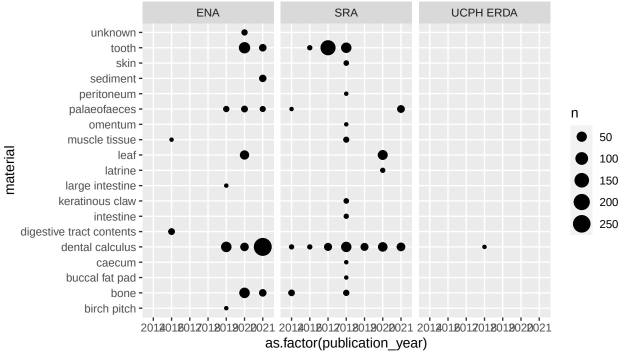
```
ggplot(samples) +
geom_boxplot(aes(x = as.factor(publication_year), y = sample_age,
                 fill = as.factor(publication_year))) +
scale_y_log10() + scale_fill_viridis_d(option = "C")
```



Defining plot matrices via `facets`

- Splitting up the plot by categories into **facets** is another way to visualize more variables at once

```
ggplot(samples) +
geom_count(aes(x = as.factor(publication_year), y = material)) +
facet_wrap(~archive)
```

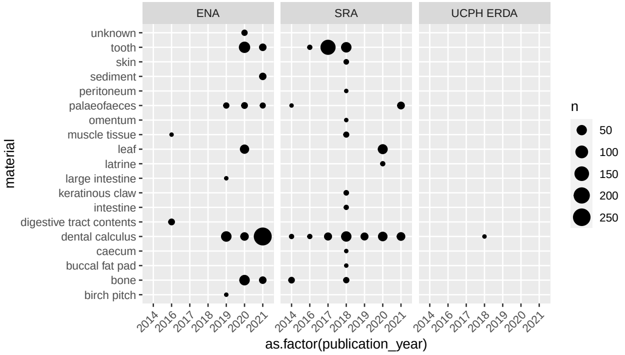


- Unfortunately the x-axis became unreadable

Setting purely aesthetic settings with theme

- Aesthetic changes like this can be applied as part of the theme

```
ggplot(samples) +
geom_count(aes(x = as.factor(publication_year), y = material)) +
facet_wrap(~archive) +
theme(axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1))
```



Exercise 1

1. Look at the `mtcars` dataset and read up on the meaning of its variables
2. Visualize the relationship between *Gross horsepower* and *1/4 mile time*
3. Integrate the *Number of cylinders* into your plot

Possible solutions 1

1. Look at the `mtcars` dataset and read up on the meaning of its variables

```
?mtcars
```

2. Visualize the relationship between *Gross horsepower* and *1/4 mile time*

```
ggplot(mtcars) + geom_point(aes(x = hp, y = qsec))
```

3. Integrate the *Number of cylinders* into your plot

```
ggplot(mtcars) + geom_point(aes(x = hp, y = qsec, color = as.factor(cyl)))
```

Conditional queries on tibbles

Selecting columns and filtering rows with select and filter

- The dplyr package includes powerful functions to subset data in tibbles based on conditions
- dplyr::select allows to select columns

```
dplyr::select(samples, project_name, sample_age)   # reduce to two columns
dplyr::select(samples, -project_name, -sample_age) # remove two columns
```

- dplyr::filter allows for conditional filtering of rows

```
dplyr::filter(samples, publication_year == 2014)  # samples published in 2014
dplyr::filter(samples, publication_year == 2014 |
                       publication_year == 2018)  # samples from 2015 OR 2018
dplyr::filter(samples, publication_year %in% c(2014, 2018)) # match operator: %in%
dplyr::filter(samples, sample_host == "Homo sapiens" &
                       community_type == "oral")  # oral samples from modern humans
```

Chaining functions together with the pipe %>%

- The pipe %>% in the `magrittr` package is a clever infix operator to chain data and operations

```
library(magrittr)
samples %>% dplyr::filter(publication_year == 2014)
```

- It *pipes* the LHS *in* as the first argument of the function appearing on the RHS
- That allows for sequences of functions ("tidyverse style")

```
samples %>%
  dplyr::select(sample_host, community_type) %>%
  dplyr::filter(sample_host == "Homo sapiens" & community_type == "oral") %>%
  nrow() # count the rows
```

- `magrittr` also offers some more operators, among which the extraction %$% is particularly useful

```
samples %>%
  dplyr::filter(material == "tooth") %$%
  sample_age %>% # extract the sample_age column as a vector
  max()          # get the maximum of said vector
```

Summary statistics in base R

- Summarising and counting data is indispensable and R offers all operations you would expect in its base package

```
nrow(samples)                # number of rows in a tibble
length(samples$site_name)    # length/size of a vector
unique(samples$material)     # unique elements of a vector

min(samples$sample_age)      # minimum
max(samples$sample_age)      # maximum

mean(samples$sample_age)     # mean
median(samples$sample_age)   # median

var(samples$sample_age)      # variance
sd(samples$sample_age)       # standard deviation
quantile(samples$sample_age, probs = 0.75) # sample quantiles for the given probs
```

- many of these functions can ignore missing values with an option `na.rm = TRUE`

Group-wise summaries with group_by and summarise

- These summary statistics are particular useful when applied to conditional subsets of a dataset
- dplyr allows such summary operations with a combination of group_by and summarise

```
samples %>%
  dplyr::group_by(material) %>%  # group the tibble by the material column
  dplyr::summarise(
    min_age = min(sample_age),    # a new column: min age for each group
    median_age = median(sample_age), # a new column: median age for each group
    max_age = max(sample_age)     # a new column: max age for each group
  )
```

- grouping can be applied across multiple columns

```
samples %>%
  dplyr::group_by(material, sample_host) %>% # group by material and host
  dplyr::summarise(
    n = dplyr::n(),  # a new column: number of samples for each group
    .groups = "drop" # drop the grouping after this summary operation
  )
```

Sorting and slicing tibbles with arrange and slice

- dplyr allows to arrange tibbles by one or multiple columns

```
samples %>% dplyr::arrange(publication_year)         # sort by publication year
samples %>% dplyr::arrange(publication_year,
                           sample_age)               # ... and sample age
samples %>% dplyr::arrange(dplyr::desc(sample_age)) # sort descending on sample age
```

- Sorting also works within groups and can be paired with slice to extract extreme values per group

```
samples %>%
  dplyr::group_by(publication_year) %>%         # group by publication year
  dplyr::arrange(dplyr::desc(sample_age)) %>% # sort by age within (!) groups
  dplyr::slice_head(n = 2) %>%                 # keep the first two samples per group
  dplyr::ungroup()                             # remove the still lingering grouping
```

- Slicing is also the relevant operation to take random samples from the observations in a tibble

```
samples %>% dplyr::slice_sample(n = 20)
```

Exercise 2

1. Determine the number of cars with four *forward gears* (`gear`) in the `mtcars` dataset
2. Determine the mean *1/4 mile time* (`qsec`) per *Number of cylinders* (`cyl`) group
3. Identify the least efficient cars for both *transmission types* (`am`)

Possible solutions 2

1. Determine the number of cars with four *forward gears* (gear) in the mtcars dataset

```
mtcars %>% dplyr::filter(gear == 4) %>% nrow()
```

2. Determine the mean *1/4 mile time* (qsec) per *Number of cylinders* (cyl) group

```
mtcars %>% dplyr::group_by(cyl) %>% dplyr::summarise(qsec_mean = mean(qsec))
```

3. Identify the least efficient cars for both *transmission types* (am)

```
#mtcars3 <- tibble::rownames_to_column(mtcars, var = "car") %>% tibble::as_tibble()
mtcars %>% dplyr::group_by(am) %>% dplyr::arrange(mpg) %>% dplyr::slice_head()
```

Transforming and manipulating tibbles

Renaming and reordering columns and values with `rename`, `relocate` and `recode`

- Columns in tibbles can be renamed with `dplyr::rename` and reordered with `dplyr::relocate`

```
samples %>% dplyr::rename(country = geo_loc_name) # rename a column
samples %>% dplyr::relocate(site_name, .before = project_name) # reorder columns
```

- Values in columns can also be changed with `dplyr::recode`

```
samples$sample_host %>% dplyr::recode(`Homo sapiens` = "modern human")
```

- R supports explicitly ordinal data with `factors`, which can be reordered as well
- `factors` can be handeld more easily with the `forcats` package

```
ggplot(samples) + geom_bar(aes(x = community_type)) # bars are alphabetically ordered
```

```
sa2 <- samples
sa2$cto <- forcats::fct_reorder(sa2$community_type, sa2$community_type, length)
# fct_reorder: reorder the input factor by a summary statistic on an other vector
ggplot(sa2) + geom_bar(aes(x = community_type)) # bars are ordered by size
```

Adding columns to tibbles with mutate and transmute

- A common application of data manipulation is adding derived columns. `dplyr` offers that with `mutate`

```
samples %>%
  dplyr::mutate(                                                  # add a column that
    archive_summary = paste0(archive, ": ", archive_accession) # combines two other
  ) %$% archive_summary                                           # columns
```

- `dplyr::transmute` removes all columns but the newly created ones

```
samples %>%
  dplyr::transmute(
    sample_name = tolower(sample_name), # overwrite this columns
    publication_doi                     # select this column
)
```

- `tibble::add_column` behaves as `dplyr::mutate`, but gives more control over column position

```
samples %>% tibble::add_column(., id = 1:nrow(.), .before = "project_name")
```

Conditional operations with `ifelse` and `case_when`

- `ifelse` allows to implement conditional `mutate` operations, that consider information from other columns, but that gets cumbersome easily

```
samples %>% dplyr::mutate(hemi = ifelse(latitude >= 0, "North", "South")) %$% hemi
```

```
samples %>% dplyr::mutate(
  hemi = ifelse(is.na(latitude), "unknown", ifelse(latitude >= 0, "North", "South"))
) %$% hemi
```

- `dplyr::case_when` is a much more readable solution for this application

```
samples %>% dplyr::mutate(
  hemi = dplyr::case_when(
    latitude >= 0 ~ "North",
    latitude < 0  ~ "South",
    TRUE          ~ "unknown" # TRUE catches all remaining cases
  )
) %$% hemi
```

Long and wide data formats

- For different applications or to simplify certain analysis or plotting operations data often has to be transformed from a **wide** to a **long** format or vice versa

I	A	B
a	1	4
b	2	5

$\xrightarrow{\text{longer}}$

I	N	V
a	A	1
b	A	2
a	B	4
b	B	5

$\xrightarrow{\text{wider}}$

I	A	B
a	1	4
b	2	5

- A table in **wide** format has N key columns and N value columns
- A table in **long** format has N key columns, one descriptor column and one value column

A wide dataset

```
carsales <- tibble::tribble(
  ~brand, ~`2014`, ~`2015`, ~`2016`, ~`2017`,
  "BMW",  20,      25,      30,      45,
  "VW",   67,      40,     120,      55
)
```

```
## # A tibble: 2 x 5
##   brand `2014` `2015` `2016` `2017`
##   <chr>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 BMW       20     25     30     45
## 2 VW        67     40    120     55
```

- Wide format becomes a problem, when the columns are semantically identical. This dataset is in wide format and we can not easily plot it
- We generally prefer data in long format, although it is more verbose with more duplication. "Long" format data is more "tidy"

Making a wide dataset long with pivot_longer

```
carsales_long <- carsales %>% tidyr::pivot_longer(
  cols = tidyselect::num_range("", range = 2014:2017), # set of columns to transform
  names_to = "year",              # the name of the descriptor column we want
  names_transform = as.integer, # a transformation function to apply to the names
  values_to = "sales"             # the name of the value column we want
)
```

```
## # A tibble: 8 x 3
##   brand  year sales
##   <chr> <int> <dbl>
## 1 BMW    2014    20
## 2 BMW    2015    25
## 3 BMW    2016    30
## 4 BMW    2017    45
## 5 VW     2014    67
## 6 VW     2015    40
## 7 VW     2016   120
## 8 VW     2017    55
```

Making a long dataset wide with pivot_wider

```
carsales_wide <- carsales_long %>% tidyr::pivot_wider(
  id_cols = "brand",  # the set of id columns that should not be changed
  names_from = year,  # the descriptor column with the names of the new columns
  values_from = sales # the value column from which the values should be extracted
)
```

```
## # A tibble: 2 x 5
##   brand `2014` `2015` `2016` `2017`
##   <chr>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 BMW       20     25     30     45
## 2 VW        67     40    120     55
```

- Applications of wide datasets are adjacency matrices to represent graphs, covariance matrices or other pairwise statistics
- When data gets big, then wide formats can be significantly more efficient (e.g. for spatial data)

Exercise 3

1. Move the column `gear` to the first position of the mtcars dataset
2. Make a new dataset `mtcars2` with the column `mpg` and an additional column `am_v`, which encodes the *transmission type* (`am`) as either `"manual"` or `"automatic"`
3. Count the number of cars per *transmission type* (`am_v`) and *number of gears* (`gear`). Then transform the result to a wide format, with one column per *transmission type*.

Possible solutions 3

1. Move the column gear to the first position of the mtcars dataset

```
mtcars %>% dplyr::relocate(gear, .before = mpg)
```

2. Make a new dataset `mtcars2` with the column gear and an additional column `am_v`, which encodes the *transmission type* (am) as either `"manual"` or `"automatic"`

```
mtcars2 <- mtcars %>% dplyr::mutate(
  gear, am_v = dplyr::case_when(am == 0 ~ "automatic", am == 1 ~ "manual")
)
```

3. Count the number of cars in `mtcars2` per *transmission type* (`am_v`) and *number of gears* (`gear`). Then transform the result to a wide format, with one column per *transmission type*.

```
mtcars2 %>% dplyr::group_by(am_v, gear) %>% dplyr::tally() %>%
  tidyr::pivot_wider(names_from = am_v, values_from = n)
```

Combining tibbles with join operations

Types of joins

Joins combine two datasets x and y based on key columns

- Mutating joins add columns from one dataset to the other
  - Left join: Take observations from x and add fitting information from y
  - Right join: Take observations from y and add fitting information from x
  - Inner join: Join the overlapping observations from x and y
  - Full join: Join all observations from x and y, even if information is missing
- Filtering joins remove observations from x based on their presence in y
  - Semi join: Keep every observation in x that is in y
  - Anti join: Keep every observation in x that is not in y

A second dataset

```
libraries <- readr::read_tsv(library_table_url)
print(libraries, n = 3)
```

```
## # A tibble: 1,657 x 20
##   project_name publication_year data_publication_doi sample_name archive
##   <chr>                   <dbl> <chr>                <chr>       <chr>
## 1 Warinner2014             2014 10.1038/ng.2906      B61         SRA
## 2 Warinner2014             2014 10.1038/ng.2906      B61         SRA
## 3 Warinner2014             2014 10.1038/ng.2906      B61         SRA
## # ... with 1,654 more rows, and 15 more variables: archive_project <chr>,
## #   archive_sample_accession <chr>, library_name <chr>, strand_type <chr>,
## #   library_polymerase <chr>, library_treatment <chr>,
## #   library_concentration <dbl>, instrument_model <chr>, library_layout <chr>,
## #   library_strategy <chr>, read_count <dbl>, archive_data_accession <chr>,
## #   download_links <chr>, download_md5s <chr>, download_sizes <chr>
```

Meaningful subsets

```r
print(samsub, n = 3)
```

```
## # A tibble: 1,060 x 3
##   project_name sample_name sample_age
##   <chr>        <chr>            <dbl>
## 1 Warinner2014 B61                900
## 2 Warinner2014 G12                900
## 3 Weyrich2017  Chimp              100
## # ... with 1,057 more rows
```

```r
print(libsub, n = 3)
```

```
## # A tibble: 1,657 x 4
##   project_name sample_name library_name     read_count
##   <chr>        <chr>       <chr>                 <dbl>
## 1 Warinner2014 B61         S1-Shot-B61-calc   13228381
## 2 Warinner2014 B61         S2-Shot-B61-calc   13260566
## 3 Warinner2014 B61         S3-Shot-B61-calc    8869866
## # ... with 1,654 more rows
```

Left join

Take observations from x and add fitting information from y

A	B	C
a	t	1
b	u	2
c	v	3

+

A	B	D
a	t	3
b	u	2
d	w	1

=

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	-

```
left <- dplyr::left_join(
  x = samsub,                              # 1060 observations
  y = libsub,                              # 1657 observations
  by = c("project_name", "sample_name") # the key columns by which to join
)
```

```
## # A tibble: 1,881 x 5
##   project_name sample_name sample_age library_name     read_count
##   <chr>        <chr>            <dbl> <chr>                 <dbl>
## 1 Warinner2014 B61                900 S1-Shot-B61-calc   13228381
## # ... with 1,880 more rows
```

- Left joins are the most common join operation: Add information from another dataset

Right join

Take observations from y and add fitting information from x

A	B	C
a	t	1
b	u	2
c	v	3

+

A	B	D
a	t	3
b	u	2
d	w	1

=

A	B	C	D
a	t	1	3
b	u	2	2
d	w	-	1

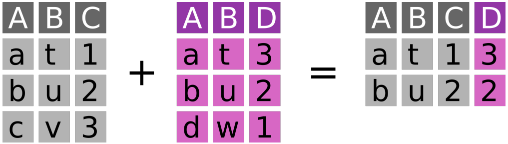
```r
right <- dplyr::right_join(
  x = samsub,                                 # 1060 observations
  y = libsub,                                 # 1657 observations
  by = c("project_name", "sample_name")
)
```

```
## # A tibble: 1,820 x 5
##   project_name sample_name sample_age library_name     read_count
##   <chr>        <chr>            <dbl> <chr>                 <dbl>
## 1 Warinner2014 B61                900 S1-Shot-B61-calc   13228381
## # ... with 1,819 more rows
```

- Right joins are almost identical to left joins – only x and y have reversed roles

Inner join

Join the overlapping observations from x and y



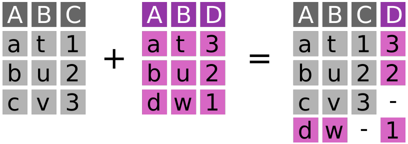

```r
inner <- dplyr::inner_join(
  x = samsub,                                       # 1060 observations
  y = libsub,                                       # 1657 observations
  by = c("project_name", "sample_name")
)
```

```
## # A tibble: 1,787 x 5
##   project_name sample_name sample_age library_name     read_count
##   <chr>        <chr>            <dbl> <chr>                 <dbl>
## 1 Warinner2014 B61                900 S1-Shot-B61-calc   13228381
## # ... with 1,786 more rows
```

- Inner joins are a fast and easy way to check, to which degree two dataset overlap

Full join

Join all observations from x and y, even if information is missing




```
full <- dplyr::full_join(
  x = samsub,                                   # 1060 observations
  y = libsub,                                   # 1657 observations
  by = c("project_name", "sample_name")
)
```

```
## # A tibble: 1,914 x 5
##   project_name sample_name sample_age library_name     read_count
##   <chr>        <chr>            <dbl> <chr>                 <dbl>
## 1 Warinner2014 B61                900 S1-Shot-B61-calc   13228381
## # ... with 1,913 more rows
```

- Full joins allow to preserve every bit of information

Semi join

Keep every observation in x that is in y

A	B	C
a	t	1
b	u	2
c	v	3

+

A	B	D
a	t	3
b	u	2
d	w	1

=

A	B	C
a	t	1
b	u	2

```r
semi <- dplyr::semi_join(
  x = samsub,                                     # 1060 observations
  y = libsub,                                     # 1657 observations
  by = c("project_name", "sample_name")
)
```

```
## # A tibble: 966 x 3
##   project_name sample_name sample_age
##   <chr>        <chr>            <dbl>
## 1 Warinner2014 B61                900
## # ... with 965 more rows
```

- Semi joins are underused operations to filter datasets

Anti join

Keep every observation in x that is not in y

A	B	C
a	t	1
b	u	2
c	v	3

+

A	B	D
a	t	3
b	u	2
d	w	1

=

A	B	C
c	v	3

```r
anti <- dplyr::anti_join(
  x = samsub,                                   # 1060 observations
  y = libsub,                                   # 1657 observations
  by = c("project_name", "sample_name")
)
```

```
## # A tibble: 94 x 3
##   project_name sample_name sample_age
##   <chr>        <chr>            <dbl>
## 1 Willman2018  213                200
## # ... with 93 more rows
```

- Anti joins allow to quickly specify incomplete datasets and missing information

Exercise 4

Consider the following additional dataset:

```
gear_opinions <- tibble::tibble(gear = c(3, 5), opinion = c("boring", "wow"))
```

1. Add my opinions about gears to the `mtcars` dataset
2. Remove all cars from the dataset for which I don't have an opinion

Possible Solutions 4

1. Add my opinions about gears to the mtcars dataset

```
dplyr::left_join(mtcars, gear_opinions, by = "gear")
```

2. Remove all cars from the dataset for which I don't have an opinion

```
dplyr::anti_join(mtcars, gear_opinions, by = "gear")
```