

# Introduction to Data Science

## Session 0: R and the tidyverse

---

Simon Munzert  
Hertie School

# Table of contents

1. Tidyverse basics<sup>1</sup>
2. Pipes
3. Data wrangling with dplyr
4. Data tidying with tidyr
5. Coding style
6. Summary

<sup>1</sup> Parts of this lecture draw on materials from Grant McDermott's excellent *Data Science for Economists* class.

# Today's session in a nutshell

DATA



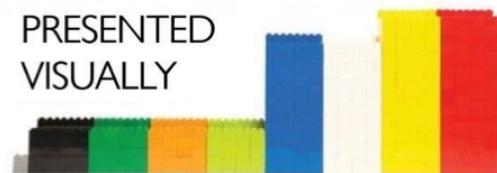
SORTED



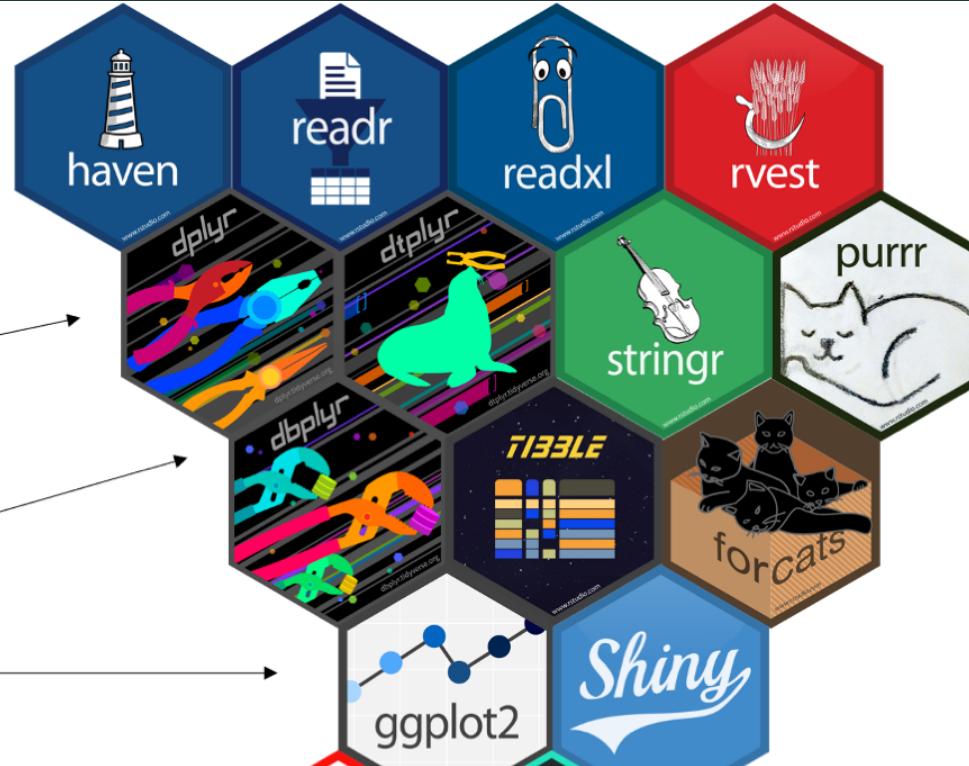
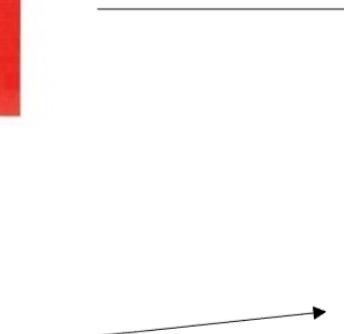
ARRANGED



PRESENTED VISUALLY



EXPLAINED WITH A STORY



# Tidyverse basics

---

# What is the tidyverse?

## R packages for data science

- Let's take it from the [tidyverse website](#):

**"The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures."**

- It's the contribution of many people of the R community.
- [Hadley Wickham](#) had a key role in shaping it by developing many of the core packages, such as `ggplot2`, `dplyr`, `tidyr`, `tibble`, and `stringr`.
- Install the complete tidyverse with:

```
R> install.packages("tidyverse")
```



Hadley Wickham

# A guide to the tidyverse

## Valuable resources

- Welcome to the [Tidyverse](#), a quick overview from many tidyverse contributors
- [Tidy data](#), a foundational paper on data wrangling and structuring, by Hadley Wickham, 2014, *Journal of Statistical Software*; check [here](#) for a hands-on vignette based on the `tidyverse` package
- [The tidyverse design guide](#), a (soon-to-be book) manifesto to promote design consistency across the tidyverse
- [R for Data Science](#), our main textbook for this course



*Journal of Statistical Software*  
August 2014, Volume 59, Issue 10.  
<http://www.jstatsoft.org/>

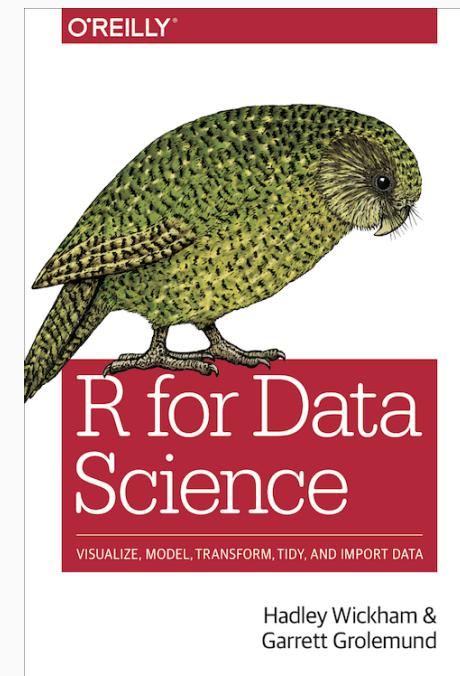
### Tidy Data

Hadley Wickham  
RStudio

#### Abstract

A huge amount of effort is spent cleaning data to get it ready for analysis, but there has been little research on how to make data cleaning as easy and effective as possible. This paper tackles a small, but important, component of data cleaning: data tidying. Tidy datasets are easy to manipulate, model and visualize, and have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table. This framework makes it easier to tidy data because it requires only a small set of tools to deal with a wide range of untidy datasets. The structure also makes it easier to develop tidy tools for data analysis, tools that both input and output tidy datasets. The advantages of a consistent data structure and matching tools are demonstrated with a case study free from mundane data manipulation chores.

*Keywords:* data cleaning, data tidying, relational databases, R.



# Tidyverse packages

## Loading the tidyverse

```
R> library(tidyverse)
```

```
## — Attaching core tidyverse packages ————— tidyverse 2.0.0 —
## ✓ dplyr     1.1.2    ✓ readr     2.1.4
## ✓forcats   1.0.0    ✓ stringr   1.5.0
## ✓ ggplot2   3.4.2    ✓ tibble    3.2.1
## ✓ lubridate 1.9.2    ✓ tidyr    1.3.0
## ✓ purrr    1.0.1
## — Conflicts ————— tidyverse_conflicts() —
## ✘ dplyr::filter() masks stats::filter()
## ✘ dplyr::lag()   masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

- We see that we have actually loaded a number of packages (which could also be loaded individually): `ggplot2`, `tibble`, `dplyr`, etc.
- We can also see information about the package versions and some namespace conflicts.

# Tidyverse packages *cont.*

- In addition to the currently 8 core packages, the tidyverse includes many others for more specialized usage.<sup>1</sup>
- See [here](#) for an overview, or just in R directly:

```
R> tidyverse_packages()
```

```
## [1] "broom"          "conflicted"     "cli"           "dbplyr"  
## [5] "dplyr"          "dtplyr"         "forcats"       "ggplot2"  
## [9] "googledrive"    "googlesheets4" "haven"        "hms"  
## [13] "httr"           "jsonlite"       "lubridate"     "magrittr"  
## [17] "modelr"         "pillar"         "purrr"        "ragg"  
## [21] "readr"          "readxl"         "reprex"        "rlang"  
## [25] "rstudioapi"    "rvest"          "stringr"       "tibble"  
## [29] "tidyverse"      "xml2"          "tidyverse"     "tidyverse"
```

- We'll use several of these additional packages during the remainder of this course (e.g., the `lubridate` package for working with dates and the `rvest` package for web scraping).
- However, bear in mind that these packages will have to be loaded separately.

<sup>1</sup> It also includes a *lot* of dependencies upon installation. This is a matter of some [controversy](#).

# The tidyverse philosophy

## Key philosophy for tidy data

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Basically, tidy data is more likely to be **long (i.e. narrow) format** than wide format.

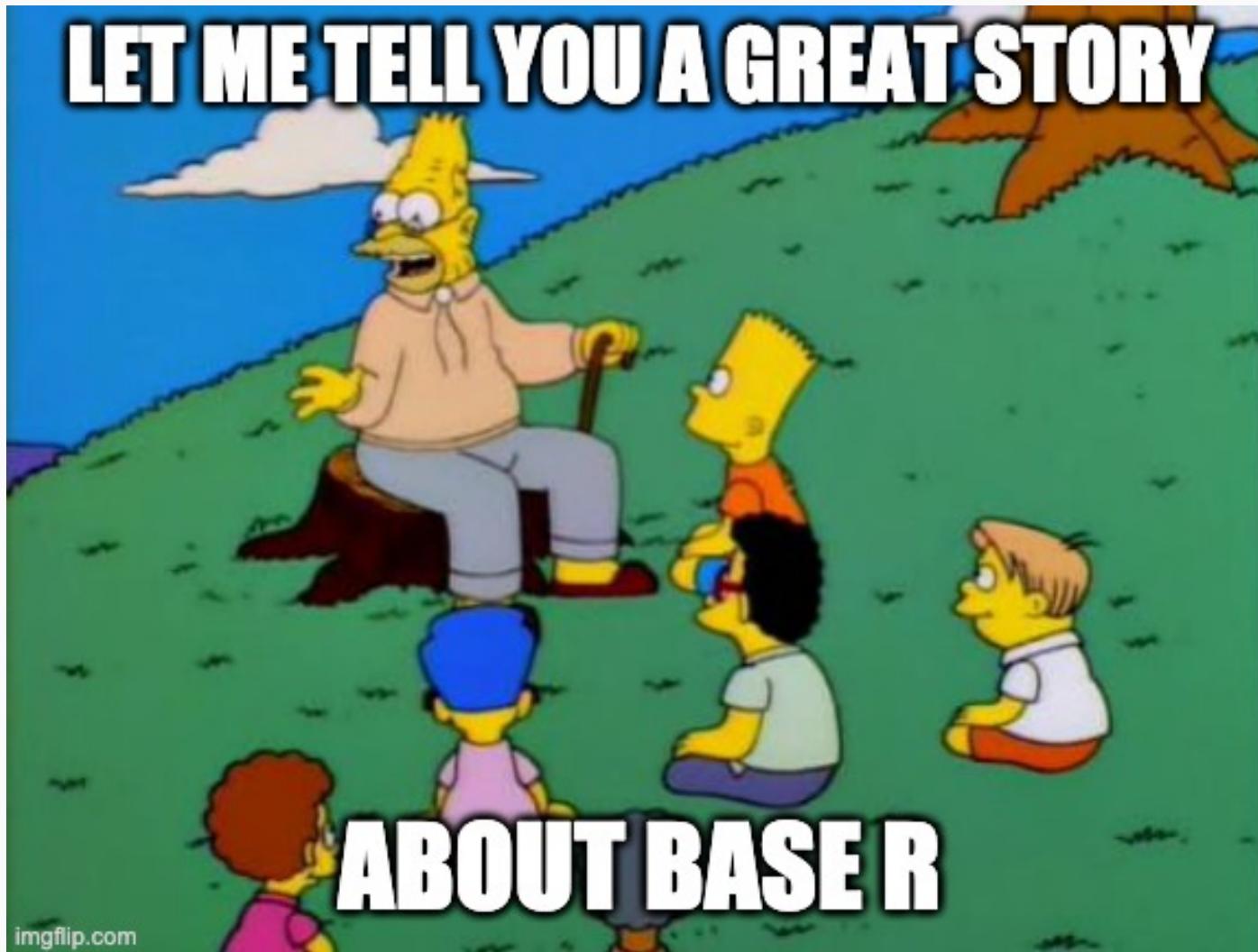
## More unifying principles

- Today, the tidyverse stands for more than just "tidy data".
- It is guided by the principles of being **human centered, consistent, composable**, and **inclusive**.
- We will learn about these **unifying principles** inductively when working with more and more tidyverse packages.
- Later today, we will learn about **tidyverse style principles** of low-level code formatting.

## Resources

Check out the [tidyverse design guide](#) for a comprehensive treatment of the tidyverse philosophy.

# Tidyverse vs. base R



# Tidyverse vs. base R: what's the difference?

- Both are compatible. You can wrangle your data with `dplyr`, plot it with `ggplot2`, and model it with yet another package.
- Ultimately, the tidyverse is just a bunch of (hugely popular!) packages that share design principles.
- Often, tidyverse packages don't reinvent the wheel. Instead, they offer more consistency in naming, arguments, and output (among other things).
- For instance, compare function naming principles (`tidyverse::snake_case` vs `base::period.case` rule; more on these conventions later) in these examples:

| <b>tidyverse</b>              | <b>base</b>                    |
|-------------------------------|--------------------------------|
| <code>?readr::read_csv</code> | <code>?utils::read.csv</code>  |
| <code>?dplyr::if_else</code>  | <code>?base::ifelse</code>     |
| <code>?tibble::tibble</code>  | <code>?base::data.frame</code> |

- If you call up the above examples, you'll see that the tidyverse alternative typically offers some enhancements or other useful options (and sometimes restrictions) over its base counterpart.
- And **remember:** There are (almost) always multiple ways to achieve a single goal in R.

# Tidyverse vs. base R: what's the difference? *cont.*

**Tidyverse**



Credit [sawiki.com](http://sawiki.com)

**Base R**



Credit [multimedialab.be](http://multimedialab.be)

# Tidyverse vs. base R: what to use?

## Stories from the past

- When I started to learn R ~14 years ago, there was no tidyverse. The learning curve felt much steeper. I often switched back to Stata for data wrangling.
- As the tidyverse grew, R became more convenient to use for the entire research pipeline.
- There's simply no need for you to live through the same pain.

## Why we start with the tidyverse

- Because [clever people think it's the right way](#).
- Documentation + community support are great.
- Having a consistent syntax makes it easier to learn.

## You still will want to check out base R alternatives later

- Base R is extremely flexible and powerful (and stable).
- There are some things that you'll have to venture outside of the tidyverse for.
- A combination of tidyverse and base R is often the best solution to a problem.
- Excellent base R data manipulation tutorials: [here](#) and [here](#).

# Now, let's get started with the tidyverse!

## R packages you'll need today

**tidyverse**

**nycflights13**

You can install/update them both with the following command.

```
R> install.packages(  
+   c('tidyverse', 'nycflights13'),  
+   repos = 'https://cran.rstudio.com',  
+   dependencies = TRUE  
+ )
```



# Pipes

---



*Ceci n'est pas un gif.*

Credit [likestowastetime/imgur](#)

# The pipe

%>%

# Example

## The pipe way

```
R> Alex %>%  
+   wake_up(7) %>%  
+   shower(temp = 38) %>%  
+   breakfast(c("coffee", "croissant")) %>%  
+   walk(step_function()) %>%  
+   bvg(  
+     train = "U2",  
+     destination = "Stadtmitte"  
+   ) %>%  
+   hertie(course = "Intro to DS")
```

## The classic way

```
R> hertie(  
+   bvg(  
+     walk(  
+       breakfast(  
+         shower(  
+           wake_up(  
+             Alex, 7  
+           ),  
+             temp = 38  
+           ),  
+             c("coffee", "croissant")  
+           ),  
+             step_function()  
+           ),  
+             train = "U2",  
+             destination = "Stadtmitte"  
+           ),  
+             course = "Intro to DS"  
+           )
```

# Example

## The pipe way

```
R> Alex %>%
+   wake_up(7) %>%
+   shower(temp = 38) %>%
+   breakfast(c("coffee", "croissant")) %>%
+   walk(step_function()) %>%
+   bvg(
+     train = "U2",
+     destination = "Stadtmitte"
+   ) %>%
+   hertie(course = "Intro to DS")
```

## The classic way, nightmare edition

```
R> alex_awake ← wake_up(Alex, 7)
R> alex_showered ← shower(alex_awake,
+                           temp = 38)
R> alex_replete ← breakfast(alex_showered,
+                            c("coffee", "croissant"))
R> alex_underway ← walk(alex_replete,
+                        step_function())
R> alex_on_train ← bvg(alex_underway,
+                       train = "U2",
+                       destination = "Stadtmitte")
R> alex_hertie ← hertie(alex_on_train,
+                        course = "Intro to DS")
```

# The beauty of pipes

## A simple but powerful tool

- The forward-pipe operator `%>%` pipes the left-hand side values forward into expressions on the right-hand side.
- We replace `f(x)` with `x %>% f()`.

## Why piping is cool

- It structures sequences of data operations as pipes, i.e. left-to-right (as opposed to from the inside and out).
- It serves the natural way of reading ("do this, then this, then this, ...").
- It avoids nested function calls.
- It improves cognitive performance of code writers and readers.
- It minimizes the need for local variables and function definitions.

## Background

- The pipe was originally created in 2014 by [Stefan Milton Bache](#) and published with the `magrittr` package.
- Magrittr? [Get it?](#) 
- The basics come with the tidyverse by default, but `magrittr` can do more (watch out for the "tee" pipe, `%T>%`, the "exposition" pipe, `%%%`, and the "assignment" pipe, `%>=%`). Also, be sure to check out [aliases](#).

# Piping etiquette

## When to avoid the pipe

- Pipes are not very handy when you need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
- Don't use the pipe when there are meaningful intermediate objects that can be given informative names (and that are used later on).

## Instead, here's how to use it

- `%>%` should always have a space before it, and should usually be followed by a new line.
- A one-step pipe can stay on one line, but unless you plan to expand it later on, you should consider rewriting it to a regular function call.
- `magrittr` allows you to omit `()` on functions that don't have arguments (as in `mydata %>% summary`). Avoid this feature.

# The base R pipe: |>

The magrittr pipe has proven so successful and popular that the R core team **recently added** a "native" pipe operator to base R (version 4.1), denoted `|>.`<sup>1</sup>

- Here's how it works:

```
mtcars |> subset(cyl = 4) |> head()  
mtcars |> subset(cyl = 4) |> (\(x) lm(mpg ~ disp, data = x))()
```

- This illustrates how the popularity of the tidyverse has repercussions on the development of base R.
- Note that with the native pipe, the RHS function has to be written out together with the brackets (i.e., `... |> head()` instead of `... |> head`).
- Also note the use of the new shorthand inline function syntax, `\(x)`, to pass content to the RHS but not to the first argument.
- Now, should we use the "magrittr" pipe or the native pipe? The native pipe might make more sense in the long term, since it avoids dependencies and might be more efficient. Check out [this Stackoverflow post](#) for a discussion of differences.

<sup>1</sup>That's actually a `|` followed by a `>`. The default font on these slides just makes it look extra fancy.

# The tidyverse core developer team



# Data wrangling with dplyr

---

# Key dplyr verbs

There are five key `dplyr` verbs that you need to learn.<sup>1</sup>

1. `filter()`: Filter (i.e. subset) rows based on their values.
2. `arrange()`: Arrange (i.e. reorder) rows based on their values.
3. `select()`: Select (i.e. subset) columns by their names.
4. `mutate()`: Create new columns.
5. `summarize()`: Collapse multiple rows into a single summary value.<sup>2</sup>



But let's start with studying the key commands using the `starwars` dataset that comes pre-packaged with `dplyr`.

<sup>1</sup> There is much, much more in `dplyr`, and we will look beyond these core functions later. Have a glimpse at the [overview at tidyverse.org](#) and at this excellent [cheat sheet](#).

<sup>2</sup> `summarize()` with an "s" works too. I slightly prefer the barbarian version though.

# 1) dplyr::filter()

We can chain multiple filter commands with the pipe (`%>%`), or just separate them within a single filter command using commas.

```
R> starwars %>%
+   filter(
+     species == "Human",
+     height >= 190
+   )

## # A tibble: 4 × 14
##   name      height   mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr>
## 1 Darth Va...     202    136 none        white       yellow       41.9 male   mascul...
## 2 Qui-Gon ...     193     89 brown       fair        blue        92     male   mascul...
## 3 Dooku        193     80 white       fair        brown       102    male   mascul...
## 4 Bail Pre...     191     NA black      tan         brown       67     male   mascul...
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

# 1) dplyr::filter() *cont.*

Regular expressions work well, too.

```
R> starwars %>%
+   filter(stringr::str_detect(name, "Skywalker"))

## # A tibble: 3 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>     <int> <dbl> <chr>       <chr>       <dbl> <chr> <chr>
## 1 Luke Skywalker    172     77 blond      fair        blue         19 male   masculin...
## 2 Anakin Skywalker   188     84 blond      fair        blue        41.9 male   masculin...
## 3 Shmi Skywalker    163     NA black      fair        brown        72 female feminin...
```

# 1) dplyr::filter() *cont.*

A very common `filter()` use case is identifying (or removing) missing data cases.

```
R> starwars %>%  
+   filter(is.na(height))  
  
## # A tibble: 6 × 14  
##   name      height   mass hair_color skin_color eye_color birth_year sex   gender  
##   <chr>     <int> <dbl> <chr>       <chr>       <dbl> <chr> <chr>  
## 1 Arvel Cr...     NA     NA brown      fair       brown          NA male  masculin...  
## 2 Finn           NA     NA black      dark       dark          NA male  masculin...  
## 3 Rey            NA     NA brown      light      hazel          NA femal... feminin...  
## 4 Poe Dame...    NA     NA brown      light      brown          NA male  masculin...  
## 5 BB8             NA     NA none       none      black          NA none  masculin...  
## 6 Captain ...    NA     NA unknown   unknown   unknown          NA <NA> <NA>  
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,  
## #   vehicles <list>, starships <list>
```

To remove missing observations, simply use negation: `filter(!is.na(height))`.

# 1) dplyr::filter() cont.

Importantly, when we list several filter conditions, `filter()` interprets them as a Boolean "AND".

```
R> starwars %>%  
+   filter(str_detect(name, "Skywalker"),  
+           eye_color == "blue")  
  
## # A tibble: 2 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender  
##   <chr>     <int> <dbl> <chr>       <chr>       <dbl> <chr> <chr>  
## 1 Luke Skywalker 172     77 blond       fair        blue         19 male   masculin...  
## 2 Anakin Skywalker 188     84 blond       fair        blue        41.9 male   masculin...  
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,  
## #   vehicles <list>, starships <list>
```

We can work with operators `|` ("OR") and `&` ("AND") and combine them with parentheses to specify more complex filter commands, as in:

```
R> starwars %>%  
+   filter(species == "Wookiee" | (species == "Human" & height ≥ 200))
```

## 2) dplyr::arrange()

arrange() sorts observations in increasing order by default.

```
R> starwars %>%  
+   arrange(birth_year)  
  
## # A tibble: 87 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender  
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr>  
## 1 Wicket ...     88    20  brown      brown      brown          8  male  masculin...  
## 2 IG-88        200   140  none       metal      red            15  none  masculin...  
## 3 Luke Sk...     172    77  blond      fair       blue           19  male  masculin...  
## 4 Leia Or...     150    49  brown      light      brown           19  femal... feminin...  
## 5 Wedge A...     170    77  brown      fair       hazel          21  male  masculin...  
## 6 Plo Koon      188    80  none       orange     black           22  male  masculin...  
## 7 Biggs D...     183    84  black      light      brown           24  male  masculin...  
## 8 Han Solo      180    80  brown      fair       brown          29  male  masculin...  
## 9 Lando C...     177    79  black      dark       brown          31  male  masculin...  
## 10 Boba Fe...    183   78.2 black     fair       brown         31.5 male  masculin...  
## # i 77 more rows  
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,  
## #   vehicles <list>, starships <list>
```

Note: Arranging on a character-based column (i.e. strings) will sort alphabetically.

## 2) dplyr::arrange() cont.

We can also arrange items in descending order using `arrange(desc())`.

```
R> starwars %>%  
+   arrange(desc(birth_year))  
  
## # A tibble: 87 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender  
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr>  
## 1 Yoda        66    17  white       green       brown        896 male   masculin...  
## 2 Jabba D...   175   1358 <NA>       green-tan... orange       600 herm... masculin...  
## 3 Chewbac...   228   112  brown       unknown      blue         200 male   masculin...  
## 4 C-3PO       167    75 <NA>       gold        yellow       112 none   masculin...  
## 5 Dooku       193    80  white       fair        brown        102 male   masculin...  
## 6 Qui-Gon...   193    89  brown       fair        blue         92 male   masculin...  
## 7 Ki-Adi-...   198    82  white       pale        yellow       92 male   masculin...  
## 8 Finis V...   170    NA  blond       fair        blue         91 male   masculin...  
## 9 Palpati...   170    75  grey        pale        yellow       82 male   masculin...  
## 10 Cliegg ...  183    NA  brown       fair        blue         82 male   masculin...  
## # i 77 more rows  
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,  
## #   vehicles <list>, starships <list>
```

### 3) dplyr::select()

Use commas to select multiple columns out of a data frame. (You can also use `<first>:<last>` for consecutive columns). Deselect a column with "-".

```
R> starwars %>%
+   select(name:skin_color, species, -height)

## # A tibble: 87 × 5
##       name           mass hair_color    skin_color  species
##       <chr>        <dbl> <chr>        <chr>      <chr>
## 1 Luke Skywalker     77  blond        fair       Human
## 2 C-3PO              75  <NA>         gold      Droid 
## 3 R2-D2              32  <NA>         white, blue Droid 
## 4 Darth Vader        136 none         white      Human
## 5 Leia Organa         49  brown        light      Human
## 6 Owen Lars          120 brown, grey  light      Human
## 7 Beru Whitesun lars 75  brown        light      Human
## 8 R5-D4              32  <NA>         white, red Droid 
## 9 Biggs Darklighter  84  black        light      Human
## 10 Obi-Wan Kenobi    77  auburn, white fair      Human
## # i 77 more rows
```

### 3) dplyr::select() cont.

You can also rename some (or all) of your selected variables in place.

```
R> starwars %>%
+   select(alias = name, crib = homeworld, sex = gender)

## # A tibble: 87 × 3
##       alias          crib     sex
##       <chr>        <chr>    <chr>
## 1 Luke Skywalker Tatooine masculine
## 2 C-3PO          Tatooine masculine
## 3 R2-D2          Naboo    masculine
## 4 Darth Vader   Tatooine masculine
## 5 Leia Organa   Alderaan feminine
## 6 Owen Lars     Tatooine masculine
## 7 Beru Whitesun lars Tatooine feminine
## 8 R5-D4          Tatooine masculine
## 9 Biggs Darklighter Tatooine masculine
## 10 Obi-Wan Kenobi Stewjon  masculine
## # i 77 more rows
```

If you just want to rename columns without subsetting them, you can use `rename()`.

### 3) dplyr::select() cont.

The `select(contains(<PATTERN>))` option provides a nice shortcut in relevant cases.

```
R> starwars %>%  
+   select(name, contains("color"))  
  
## # A tibble: 87 × 4  
##   name          hair_color    skin_color eye_color  
##   <chr>         <chr>        <chr>      <chr>  
## 1 Luke Skywalker  blond       fair        blue  
## 2 C-3PO           <NA>        gold        yellow  
## 3 R2-D2           <NA>        white, blue red  
## 4 Darth Vader    none        white        yellow  
## 5 Leia Organa    brown       light        brown  
## 6 Owen Lars      brown, grey light        blue  
## 7 Beru Whitesun lars brown       light        blue  
## 8 R5-D4           <NA>        white, red  red  
## 9 Biggs Darklighter black       light        brown  
## 10 Obi-Wan Kenobi auburn, white fair        blue-gray  
## # i 77 more rows
```

There are many more useful selection helpers, such as `starts_with()`, `ends_with()`, and `matches()`. See [here](#) for an overview.

### 3) dplyr::select() cont.

The `select( ... , everything())` option is another useful shortcut if you only want to bring some variable(s) to the "front" of a data frame.

```
R> starwars %>%
+   select(species, homeworld, everything()) %>%
+   head(5)

## # A tibble: 5 × 14
##   species homeworld name           height  mass hair_color skin_color eye_color
##   <chr>     <chr>    <chr>        <int>  <dbl>   <chr>       <chr>      <chr>
## 1 Human     Tatooine  Luke Skywalker  172     77  blond       fair       blue
## 2 Droid     Tatooine  C-3PO          167     75 <NA>        gold       yellow
## 3 Droid     Naboo     R2-D2          96      32 <NA>       white, blue red
## 4 Human     Tatooine  Darth Vader  202    136  none        white       yellow
## 5 Human     Alderaan  Leia Organa  150     49  brown       light       brown
## # i 6 more variables: birth_year <dbl>, sex <chr>, gender <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Note: The new `relocate()` function available in dplyr 1.0.0 has brought a lot more functionality to the ordering of columns. See [here](#).

## 4) dplyr::mutate()

You can create new columns from scratch with `mutate()`, or (more commonly) as transformations of existing columns.

```
R> starwars %>%  
+   select(name, birth_year) %>%  
+   mutate(  
+     dog_years = birth_year * 7, ## Separate with a comma  
+     comment = paste0(name, " is ", dog_years, " in dog years.")  
+   ) %>%  
+   slice(1:6) # Just show first six observations  
  
## # A tibble: 6 × 4  
##   name      birth_year  dog_years comment  
##   <chr>        <dbl>      <dbl> <chr>  
## 1 Luke Skywalker    19        133  Luke Skywalker is 133 in dog years.  
## 2 C-3PO            112       784  C-3PO is 784 in dog years.  
## 3 R2-D2             33        231  R2-D2 is 231 in dog years.  
## 4 Darth Vader      41.9      293. Darth Vader is 293.3 in dog years.  
## 5 Leia Organa       19        133  Leia Organa is 133 in dog years.  
## 6 Owen Lars          52       364  Owen Lars is 364 in dog years.
```

Note: `mutate()` is order aware. So you can chain multiple mutates in a single call.

## 4) dplyr::mutate() cont.

Boolean, logical and conditional operators all work well with `mutate()` too.

```
R> starwars %>%
+   select(name, height) %>%
+   filter(name %in% c("Luke Skywalker", "Anakin Skywalker")) %>%
+   mutate(tall1 = height > 180) %>%
+   mutate(tall2 = ifelse(height > 180, "Tall", "Short")) ## Same effect, but can choose labels

## # A tibble: 2 × 4
##   name           height tall1 tall2
##   <chr>          <int>  <lgl> <chr>
## 1 Luke Skywalker     172 FALSE Short
## 2 Anakin Skywalker    188 TRUE  Tall
```

## 4) dplyr::mutate() cont.

Lastly, combining `mutate()` with the `across()` feature allows you to easily work on a subset of variables. For example:

```
R> starwars %>%  
+   select(name:eye_color) %>%  
+   mutate(across(where(is.character), toupper)) %>%  
+   head(5)  
  
## # A tibble: 5 × 6  
##   name           height  mass hair_color skin_color eye_color  
##   <chr>         <int> <dbl> <chr>       <chr>      <chr>  
## 1 LUKE SKYWALKER     172    77 BLOND       FAIR       BLUE  
## 2 C-3PO              167    75 <NA>        GOLD       YELLOW  
## 3 R2-D2                96    32 <NA>      WHITE, BLUE  RED  
## 4 DARTH VADER        202   136 NONE        WHITE      YELLOW  
## 5 LEIA ORGANA         150    49 BROWN      LIGHT      BROWN
```

Note: More on `across()` and `where()` later!

# 5) dplyr::summarize()

You can summarize variables with all sorts of operations (e.g., `mean()`, `median()`, `n()`, `n_distinct()`, `sum()`, `first()`, `last()`, ...).

```
R> starwars %>%
+   group_by(species, gender) %>%
+   summarize(mean_height = mean(height, na.rm = TRUE)) %>%
+   head(5)

## `summarise()` has grouped output by 'species'. You can override using the
## `.`groups` argument.

## # A tibble: 5 × 3
## # Groups:   species [5]
##   species   gender   mean_height
##   <chr>     <chr>        <dbl>
## 1 Aleena    masculine      79
## 2 Besalisk  masculine     198
## 3 Cerean    masculine     198
## 4 Chagrian  masculine     196
## 5 Clawdite  feminine     168
```

Note: This is particularly useful in combination with the `group_by()` command. Again, more on this later!

## 5) dplyr::summarize() cont.

Note that including `na.rm = TRUE` is usually a good idea with the functions fed into `summarize()`. Otherwise, any missing value will propagate to the summarized value too.

```
R> ## Probably not what we want
R> starwars %>%
+   summarize(mean_height = mean(height))

## # A tibble: 1 × 1
##   mean_height
##       <dbl>
## 1        NA

R> ## Much better
R> starwars %>%
+   summarize(mean_height = mean(height, na.rm = TRUE))

## # A tibble: 1 × 1
##   mean_height
##       <dbl>
## 1     174.
```

# 5) dplyr::summarize() cont.

The same `across()`-based workflow that we saw with `mutate()` a few slides back also works with `summarize()`. For example:

```
R> starwars %>%
+   group_by(species) %>%
+   summarize(across(where(is.numeric), mean, na.rm = TRUE)) %>%
+   head(5)

## Warning: There was 1 warning in `summarize()` .
## i In argument: `across(where(is.numeric), mean, na.rm = TRUE)` .
## i In group 1: `species = "Aleena"` .
## Caused by warning:
## ! The ` ... ` argument of `across()` is deprecated as of dplyr 1.1.0.
## Supply arguments directly to ` .fns` through an anonymous function instead.
##
## # Previously
## across(a:b, mean, na.rm = TRUE)
##
## # Now
## across(a:b, \((x) mean(x, na.rm = TRUE)))

## # A tibble: 5 × 4
##   species    height     mass birth year
```

# Grouping with dplyr::group\_by()

With `group_by()`, you can create a "grouped" copy of a table grouped by unique values of a column. If multiple columns are specified, the function groups by all available combinations of values.

```
R> by_species_gender ← starwars %>% group_by(species, gender)
R> by_species_gender

## # A tibble: 87 × 14
## # Groups:   species, gender [42]
##   name    height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr>
## 1 Luke Sk...     172     77 blond      fair        blue         19 male   mascul...
## 2 C-3PO        167     75 <NA>       gold        yellow      112 none   mascul...
## 3 R2-D2         96      32 <NA>       white, bl... red          33 none   mascul...
## 4 Darth V...     202     136 none       white       yellow      41.9 male   mascul...
## 5 Leia Or...     150      49 brown      light       brown        19 fema... femin...
## 6 Owen La...     178     120 brown, gr... light       blue         52 male   mascul...
## 7 Beru Wh...     165      75 brown      light       blue         47 fema... femin...
## 8 R5-D4         97      32 <NA>       white, red red          NA none   mascul...
## 9 Biggs D...     183      84 black      light       brown        24 male   mascul...
## 10 Obi-Wan...    182      77 auburn, w... fair        blue-gray     57 male   mascul...
## # i 77 more rows
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

# Grouping with dplyr::group\_by() cont.

## More notes on grouping

- Grouping doesn't change how the data looks (apart from listing how it's grouped).
- Grouping changes how it acts with other dplyr verbs such as `summarize()` and `mutate()`, as we've already seen.
- By default, `group_by()` overrides existing grouping. Use `.add = TRUE` to append instead.
- By default, groups formed by factor levels that don't appear in the data are dropped. Set `.drop = FALSE` if you want to keep them.
- `ungroup()` removes existing grouping.
- `dplyr` notifies you about grouping variables every time you do operations on or with them. If you find these messages annoying, **switch them off** with `options(dplyr.summarise.inform = FALSE)`.

```
R> options(dplyr.summarise.inform = FALSE)
R> by_species_gender %>%
+   summarize(mean(height, na.rm = TRUE)) %>%
+   filter(n_distinct(gender) == 2)

## # A tibble: 8 × 3
## # Groups:   species [4]
##   species   gender   `mean(height, na.rm = TRUE)`<dbl>
##   <chr>     <chr>                96
## 1 Droid     feminine            140
## 2 Droid     masculine           160.
## 3 Human    feminine            182.
## 4 Human    masculine           213
## 5 Kaminoan feminine           229
## 6 Kaminoan masculine          178
## 7 Twi'lek  feminine           180
## 8 Twi'lek  masculine
```

# Other dplyr goodies

`slice()`: Subset rows by position rather than filtering by values. There's also `slice_sample()` to randomly select rows, `slice_head()` and `slice_tail()` to select first or last rows, and more.

```
R> starwars %>% slice(c(1, 5))

## # A tibble: 2 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>     <int> <dbl> <chr>       <chr>       <dbl> <chr> <chr>
## 1 Luke Skywalker    172     77 blond      fair        blue         19 male   masculin...
## 2 Leia Organa       150     49 brown      light       brown        19 female feminin...
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

`pull()`: Extract a column from as a data frame as a vector or scalar.

```
R> starwars %>% filter(gender=="feminine") %>% pull(height)

## [1] 150 165 150 163 178 184 157 170 166 165 168 213 167 96 178 NA 165
```

# Other dplyr goodies *cont.*

`count()` and `distinct()`: Number and isolate unique observations.

```
R> starwars %>% count(species) %>% head(6)
```

```
## # A tibble: 6 × 2
##   species     n
##   <chr>     <int>
## 1 Aleena      1
## 2 Besalisk    1
## 3 Cerean      1
## 4 Chagrian    1
## 5 Clawdite    1
## 6 Droid       6
```

```
R> starwars %>% distinct(species) %>% pull() %>% sort() %>% magrittr::extract(1:5)
```

```
## [1] "Aleena"    "Besalisk"   "Cerean"    "Chagrian"   "Clawdite"
```

You could also use a combination of `mutate()`, `group_by()`, and `n()`, e.g. `starwars %>% group_by(species) %>% mutate(num = n())`.

# Other dplyr goodies *cont.*

`where()`: Select the variables for which a function returns true.

```
R> starwars %>% select(where(is.numeric)) %>% names()  
## [1] "height"      "mass"        "birth_year"
```

`across()`: Summarize or mutate multiple variables in the same way. More information [here](#).

```
R> starwars %>%  
+   mutate(across(where(is.numeric), scale)) %>%  
+   head(3)  
  
## # A tibble: 3 × 14  
##   name  height[,1] mass[,1] hair_color skin_color eye_color birth_year[,1] sex  
##   <chr>     <dbl>    <dbl> <chr>       <chr>       <chr>           <dbl> <chr>  
## 1 Luke...    -0.0678   -0.120 blond      fair       blue          -0.443 male  
## 2 C-3PO     -0.212    -0.132 <NA>       gold      yellow         0.158 none  
## 3 R2-D2     -2.25     -0.385 <NA>      white, bl... red          -0.353 none  
## # i 6 more variables: gender <chr>, homeworld <chr>, species <chr>,  
## #   films <list>, vehicles <list>, starships <list>
```

# Other dplyr goodies *cont.*

`case_when()`: Vectorize multiple `if_else()` (or base R `ifelse()`) statements.

```
R> starwars %>%  
+   mutate(  
+     height_cat = case_when(  
+       height < 160 ~ "tiny",  
+       height ≥ 160 & height < 190 ~ "medium",  
+       height ≥ 190 & height < 220 ~ "tall",  
+       height ≥ 220 ~ "giant"  
+     )  
+   ) %>%  
+   pull(height_cat) %>% table()  
  
## .  
## giant medium tall tiny  
##      5      45     18     13
```

There are also a whole class of **window functions** for getting leads and lags, ranking, creating cumulative aggregates, etc.  
See `vignette("window-functions")`.

`inner_join()`, `left_join()`, `right_join()`: Enough already, we'll talk about this in the session on databases!

# Data tidying with `tidyverse`

---

# Key `tidyverse` verbs

`tidyverse` is part of the core tidyverse. There are four key `tidyverse` verbs that you need to learn.

1. `pivot_longer()`: Pivot wide data into long format (i.e. "melt").<sup>1</sup>
2. `pivot_wider()`: Pivot long data into wide format (i.e. "cast").<sup>2</sup>
3. `separate()`: Separate (i.e. split) one column into multiple columns.
4. `unite()`: Unite (i.e. combine) multiple columns into one.



<sup>1</sup> Updated version of `tidyverse::gather()`.

<sup>2</sup> Updated version of `tidyverse::spread()`.

# On "longer" and "wider" datasets

Remember the **key philosophy for tidy data?**

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

One of the most common tasks for data scientists is to **reshape** data from one form to the other.

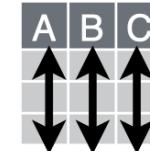
There are **multiple ways to store the same data in a dataset** (or across multiple tables; but more on that in the session on databases).

Here, we learn how to shift between

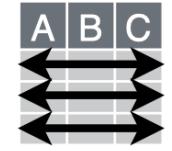
- "**wider**" formats, i.e. data being stored across more columns and
- "**longer**" formats, i.e. data being stored across more rows.

Tidy data in a nutshell

A table is tidy if:



&

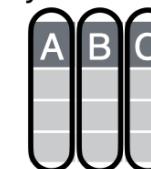


Each **variable** is in its own **column**

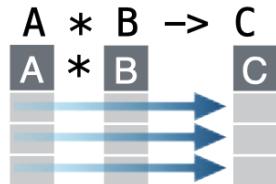
Each **observation, or case**, is in its own **row**

Benefits of tidy data

Tidy data:



Makes variables easy to access as vectors



Preserves cases during vectorized operations

# From wide to long to wide

## From wider to longer

- `pivot_longer()` pivots `cols` columns, moving column names into a `names_to` column, and column values into a `values_to` column.
- Recall a panel study design with multiple observations per unit.
- In the classical long format, each row represents one observation.
- Note how this is approaching the ideal of **tidy data**.

`pivot_longer()`

The diagram illustrates the `pivot_longer()` function's behavior. On the left, a wide-format table has three columns: `country`, `1999`, and `2000`. The `1999` column contains values "0.7K" and "37K". The `2000` column contains values "2K" and "80K". An arrow points to the right, leading to a long-format table where each country has two rows, one for each year. The `country` column lists A, B, and C. The `year` column lists 1999 and 2000. The `cases` column lists the values from the wide-format table: "0.7K", "37K", "2K", and "80K".

| country | 1999 | 2000 |
|---------|------|------|
| A       | 0.7K | 2K   |
| B       | 37K  | 80K  |
| C       | 212K | 213K |

→

| country | year | cases |
|---------|------|-------|
| A       | 1999 | 0.7K  |
| B       | 1999 | 37K   |
| C       | 1999 | 212K  |
| A       | 2000 | 2K    |
| B       | 2000 | 80K   |
| C       | 2000 | 213K  |

## From longer to wider

- `pivot_wider()` pivots a `names_from` and a `values_from` column into a rectangular field of cells.
- In a panel study design, this would allow you to have one variable per measurement (e.g., pre- and posttreatment outcome variable).
- While this is nice for the human eye, it is sometimes not what fits the tidyverse workflow. Also, wenn you have multiple repeated measurements (think: variables in a population survey), the number of columns is quickly inflated. Be ready to `pivot_longer()`.

`pivot_wider()`

The diagram illustrates the `pivot_wider()` function's behavior. On the left, a long-format table has four columns: `country`, `year`, `type`, and `count`. The `country` column lists A, A, A, A, B, B, B, B, C, C. The `year` column lists 1999, 1999, 2000, 2000, 1999, 1999, 2000, 2000, 1999, 1999. The `type` column lists "cases" and "pop" repeated for each country-year combination. The `count` column lists values like "0.7K", "19M", etc. An arrow points to the right, leading to a wide-format table where each country has two rows, one for each year. The `country` column lists A, B, C. The `year` column lists 1999 and 2000. The `cases` and `pop` columns list the values from the long-format table: "0.7K", "2K", "37K", "80K", "212K", "213K", "19M", "20M", "172M", "174M", "NA", "NA".

| country | year | type  | count |
|---------|------|-------|-------|
| A       | 1999 | cases | 0.7K  |
| A       | 1999 | pop   | 19M   |
| A       | 2000 | cases | 2K    |
| A       | 2000 | pop   | 20M   |
| B       | 1999 | cases | 37K   |
| B       | 1999 | pop   | 172M  |
| B       | 2000 | cases | 80K   |
| B       | 2000 | pop   | 174M  |
| C       | 1999 | cases | 212K  |
| C       | 1999 | pop   | 1T    |

→

| country | year | cases | pop  |
|---------|------|-------|------|
| A       | 1999 | 0.7K  | 19M  |
| A       | 2000 | 2K    | 20M  |
| B       | 1999 | 37K   | 172M |
| B       | 2000 | 80K   | 174M |
| C       | 1999 | 212K  | 1T   |
| C       | 2000 | NA    | NA   |

# 1) tidyverse::pivot\_longer()

```
R> stocks = data.frame( ## Could use "tibble" instead of "data.frame" if you prefer
+   time = as.Date('2009-01-01') + 0:1,
+   X = rnorm(2, 0, 1),
+   Y = rnorm(2, 0, 2),
+   Z = rnorm(2, 0, 4)
+ )
R> stocks
```

|      | time       | X           | Y        | Z        |
|------|------------|-------------|----------|----------|
| ## 1 | 2009-01-01 | -0.01532541 | 3.105318 | 1.417412 |
| ## 2 | 2009-01-02 | -1.06660899 | 2.866468 | 0.918123 |

```
R> tidy_stocks ← stocks %>% pivot_longer(-time, names_to = "stock", values_to = "price")
R> tidy_stocks
```

|      | time       | stock | price   |
|------|------------|-------|---------|
| ## 1 | 2009-01-01 | X     | -0.0153 |
| ## 2 | 2009-01-01 | Y     | 3.11    |
| ## 3 | 2009-01-01 | Z     | 1.42    |
| ## 4 | 2009-01-02 | X     | -1.07   |
| ## 5 | 2009-01-02 | Y     | 2.87    |

## 2) `tidyverse::pivot_wider()`

```
R> tidy_stocks %>% pivot_wider(names_from = stock, values_from = price)
```

```
R> tidy_stocks %>% pivot_wider(names_from = time, values_from = price)
```

*Note:* The second example – which has combined different pivoting arguments – has effectively transposed the data.

### 3) `tidyr::separate()`

Sometimes, cell values provide information that should be stored in separate columns. `separate()` offers one way of doing this. (Side note: Once you learn regular expressions, you will have an even more powerful tool for this task.)

```
R> economists = data.frame(name = c("Adam.Smith", "Paul.Samuelson", "Milton.Friedman"))  
R> economists
```

```
##          name  
## 1      Adam.Smith  
## 2  Paul.Samuelson  
## 3 Milton.Friedman
```

`separate()` in action:

```
R> economists %>% separate(name, c("first_name", "last_name"))  
  
##   first_name last_name  
## 1      Adam     Smith  
## 2      Paul Samuelson  
## 3    Milton  Friedman
```

You can also specify the separation character with `separate( ... , sep=".")`. The way `sep` works also depends on column type (character vs. numeric). Check out the [function reference](#).

### 3) `tidy::separate()` cont.

A related function is `separate_rows()`, for splitting up cells that contain multiple fields or observations (a frustratingly common occurrence with survey data).

```
R> jobs = data.frame(  
+   name = c("Jack", "Jill"),  
+   occupation = c("Homemaker", "Philosopher, Philanthropist, Troublemaker")  
+ )  
R> jobs
```

```
##      name          occupation  
## 1  Jack           Homemaker  
## 2 Jill Philosopher, Philanthropist, Troublemaker
```

`separate_rows()` in action:

```
R> jobs %>% separate_rows(occupation)
```

```
## # A tibble: 4 × 2  
##   name    occupation  
##   <chr>   <chr>  
## 1 Jack    Homemaker  
## 2 Jill    Philosopher
```

## 4) `tidyverse::unite()`

`separate()` has a complementary function, `unite()`. Unsurprisingly, it unites values from multiple columns into one.

```
R> gdp = data.frame(  
+   yr = rep(2016, times = 3),  
+   mnth = rep(1, times = 3),  
+   dy = 1:3,  
+   gdp = rnorm(3, mean = 100, sd = 2)  
+ )  
R> gdp  
  
##      yr mnth dy      gdp  
## 1 2016     1  1 96.62137  
## 2 2016     1  2 98.92069  
## 3 2016     1  3 99.44646  
  
R> ## Combine "yr", "mnth", and "dy" into one "date" column  
R> gdp %>% unite(date, c("yr", "mnth", "dy"), sep = "-")  
  
##      date      gdp  
## 1 2016-1-1 96.62137  
## 2 2016-1-2 98.92069  
## 3 2016-1-3 99.44646
```

## 4) `tidyr::unite()` cont.

Note that `unite()` will automatically create a character variable. You can see this better if we convert it to a tibble.

```
R> gdp_u = gdp %>%  
+   unite(date,  
+         c("yr", "mnth", "dy"),  
+         sep = "-") %>%  
+   as_tibble()  
R> gdp_u  
  
## # A tibble: 3 × 2  
##   date      gdp  
##   <chr>    <dbl>  
## 1 2016-1-1  96.6  
## 2 2016-1-2  98.9  
## 3 2016-1-3  99.4
```

If you want to convert it to something else (e.g. date or numeric) then you will need to modify it using `mutate()`. See below for an example, using the `lubridate` package's super helpful date conversion functions.

```
R> library(lubridate)  
R> gdp_u %>% mutate(date = ymd(date))  
  
## # A tibble: 3 × 2  
##   date      gdp  
##   <date>    <dbl>  
## 1 2016-01-01  96.6  
## 2 2016-01-02  98.9  
## 3 2016-01-03  99.4
```

# Other tidyverse goodies

`crossing()`: Get the full combination of a group of variables.<sup>1</sup>

```
R> crossing(side=c("left", "right"), height=c("top", "bottom"))

## # A tibble: 4 × 2
##   side  height
##   <chr> <chr>
## 1 left   bottom
## 2 left   top
## 3 right  bottom
## 4 right  top
```

`drop_na(data, ...)`: Drop rows containing NAs in `...` columns.

`fill(data, ..., direction = c("down", "up"))`: Fill in NAs in `...` columns with most recent non-NA values.

<sup>1</sup> See `?expand()` and `?complete()` for more specialized functions that allow you to fill in (implicit) missing data or variable combinations in existing data frames. Base R alternative: `expand.grid()`.

# Coding style

---

# Coding style: the basics

## Why adhering to a particular style of coding?

- It reduces the number of arbitrary decisions you have to consciously make during coding. We make an arbitrary decision (convention) once, not always ad hoc.
- It provides consistency.
- It makes code easier to write.
- It makes code easier to read, especially in the long term (i.e. two days after you've closed a script).

## What are questions of style?

- Questions of style are a matter of opinion.
- We will mostly follow Hadley Wickham's opinion as expressed in the "[tidyverse style guide](#)".
- We'll consider how to
  - name,
  - comment,
  - structure, and
  - write.

# Naming things

**Surprisingly many things can go wrong with naming...**

"There are only two hard things in Computer Science:  
cache invalidation and naming things." - *Phil Karlton*

Credit [karlton.org](http://karlton.org)



Credit [Mashable](#)

# Naming files

- Code file names should be meaningful and end in `.R`.
- Avoid using special characters in file names. Stick with numbers, letters, dashes (`-`), and underscores (`_`).
- Some examples:

```
# Good
fit_models.R
utility_functions.R
```

```
# Bad
fit models.R
foo.r
stuff.r
```

- If files should be run in a particular order, prefix them with numbers:

```
00_download.R
01_explore.R
...
09_model.R
10_visualize.R
```

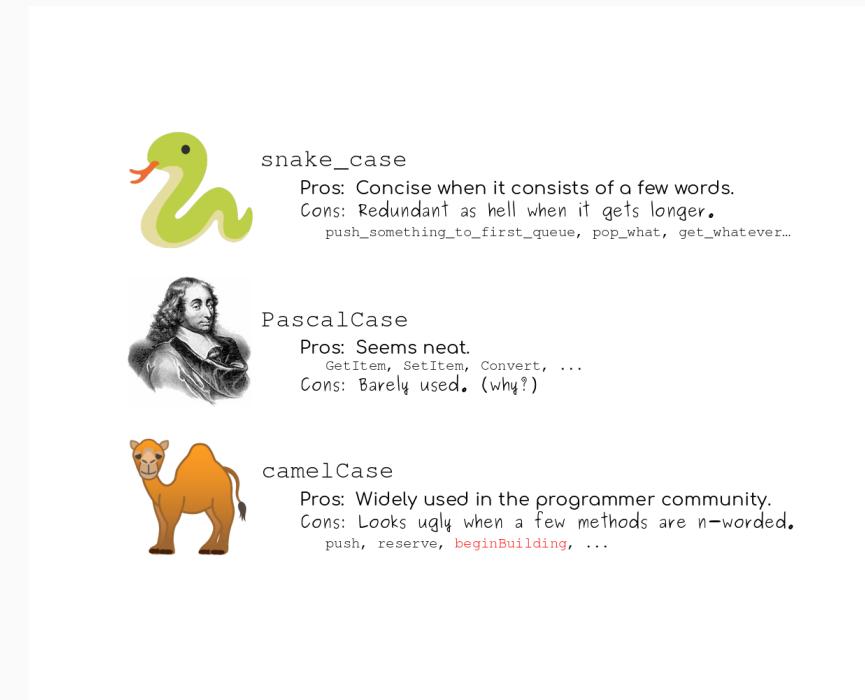
# Naming objects and variables

- There are various conventions of how to write phrases without spaces or punctuation. Some of these have been adapted in programming, such as `camelCase`, `PascalCase`, or `snake_case`.
- The `tidyverse` way: Object and variable names should use only lowercase letters, numbers, and underscores.
- Examples:

```
# Good
day_one # snake_case
day_1 # snake_case
```

```
# Less good
dayOne # camelCase
DayOne # PascalCase
day.one # dot.case
```

```
# Dysfunctional
day-one # kebab-case
```



Credit [cassert24/Reddit](#)

# Naming objects and variables *cont.*

**Table 1:**

File folder structure for organizing model and analysis files used in the proposed DARTH coding framework.

| Folder name | Folder function  |
|-------------|--|
| data-raw    | This is where raw data is stored alongside “R” scripts that read in raw data, process these data, and calls <code>use_this::use_data(&lt;processed data&gt;)</code> to save .rda formatted data files in the “data” folder. These data could include a “.csv” file with input parameters derived from the published literature, as well as internal R data files (with .RData, rds, or .rda extensions) containing primary data from which model input values will be estimated through statistical models embedded into the analysis.                     |
| data        | This is where input data is stored to be used in the different components of the CEA. These data could be generated from raw data stored in the “data-raw” folder. Essentially, this folder stores the cleaned or processed versions of raw data that has been gathered from elsewhere   |
| R           | This is where “R” files that define functions to be used as part of the analysis are stored. These are functions that are specific to the analysis. The model will be one such function; however, other functions will likely be used, such as computing the fit of the model output to the specific calibration targets of the analysis. This folder also stores “R” scripts that document the datasets in the “data” folder.   |
| analysis    | This is where interactive scripts of the analysis would be stored. These scripts control the overall flow of the analysis. This is also where many operations that ultimately become functions will be developed and debugged.   |
| output      | This is where output files of the analysis should be stored. These files may be internal R data files (“.RData”, “.rds”, “.rda”) or external data files (such as “.csv”). Examples of files stored here would be the output of the model calibration component or the PSA dataset generated in the uncertainty analysis component. These data files can then be loaded by other components without having to first rerun previous components (e.g. the calibrated model values can be loaded for a base case analysis without re-running the calibration). |
| figs        | For analyses that will include figures, we generally create a separate figures folder. Though these could be stored in the output folder, it can be helpful to have a separate folder so that the images of the figure files can be easily previewed. This is particularly important for analyses that generate a large number of figures.   |
| tables      | This folder includes tables to be included in a publication or report, such as the table of intervention costs and effects and ICERs.  |
| report      | A report folder could be used to store R Markdown files to describe in detail the model-based CEA by using all the functions and data of the framework, run analyses and display figures. The R Markdown files can be compiled into .html, .doc or .pdf files to generate a report of the CEA. This report could be the document submitted to HTA agencies accompanying the R code of the model-based CEA.   |
| vignettes   | A vignettes folder could be used to describe the usage of the functions and data of each of some or all components of the framework through accompanying R Markdown files as documentation. The R Markdown file can use all the functions, outputs, and figures to integrate the R code into the Markdown text.  |
| tests       | A tests folder includes “R” scripts that runs all the unit tests of the functions in the framework. A good practice is to have one file of tests for each complicated function or for each of the components of the framework.   |

Credit Alarid et al. 2019

**Table 2:**

File and variable naming conventions in the proposed DARTH coding framework.

| Object type | Naming recommendation  | Examples  |
|-------------|--|---|
| Files       | dir/<component number>_<description>.<ext>   | <ul style="list-style-type: none"> <li>• analysis/01_model_inputs.R</li> <li>• R/02_simulation_model_functions.R</li> </ul>   |
| Functions   | <action!>_<description>  | <ul style="list-style-type: none"> <li>• generate_init_params()</li> <li>• generate_psa_params()</li> </ul>   |
| Variables   | <x>_<y>_<var_name><br>where<br>x = data type prefix<br>y = variable type prefix<br>var_name = brief descriptor | <ul style="list-style-type: none"> <li>• n_samp</li> <li>• hr_S1D</li> <li>• v_r_mort_by_age</li> <li>• a_M</li> <li>• l_params_all</li> <li>• df_out_ce</li> </ul> |

**Table 3:**

Recommended prefixes in variable names that encode data and variable type.

| Prefix         | Data type  | Prefix | Variable type |
|----------------|------------|--------|---------------|
| <> (no prefix) | scalar     | n      | number        |
| v              | vector     | p      | probability   |
| m              | matrix     | r      | rate          |
| a              | array      | u      | utility       |
| df             | data frame | c      | cost          |
| dtb            | data table | hr     | hazard ratio  |
| l              | list       | rr     | relative risk |

# Naming functions

- In addition to following the general advice for object names, strive to use verbs for function names:

```
# Good
add_row()
permute()
```

```
# Bad
row_adder()
permutation()
```

- Also, try avoiding function names that already exist, in particular those that come with a loaded package.
- This often implies a trade-off between shortness and uniqueness. In any case, you would try to avoid situations that force you disambiguate functions with the same name (as in `dplyr::select`; see "[R packages](#)").
- Check out this [Wikipedia page](#)) or this [Stackoverflow post](#) for more background on naming conventions in programming!
- For more good advice on how to name stuff, see [this legendary presentation](#) by Jenny Bryan.

# Commenting on things

## Why commenting at all?

- It's often tempting to set up a project assuming that you will be the only person working on it, e.g. as homework. But that's almost never true.
- You have project partners, co-authors, principals.
- Even if not, there's someone else who you always have to keep happy: Future-you.
- Comment often to make Future-you happy about Past-you by documenting what Present-You is doing/thinking/planning to do.

Past-you



Present-you



Future-you



# Commenting on things *cont.*

## General advice

- Each line of a comment should begin with the comment symbol and a single space: #
- Use comments to record important findings and analysis decisions.
- If you need comments to explain what your code is doing, consider rewriting your code to be clearer.
- But: comments can work well as "sub-headlines".
- If you discover that you have more comments than code, consider switching to R Markdown.
- (Longer) comments generally work better if they get their own line.

```
R> # define job status  
R> dat$at_work ← dat$job %in% c(2, 3)  
R> dat$at_work ← dat$job %in% c(2, 3) # define job
```

## Giving structure

- Use commented lines together with dashes to break up your file into easily readable chunks.
- RStudio automatically detects these chunks and turns them into sections in the script outline.

```
R> # Input/output -----  
R>  
R> # input  
R> c("data/survey2021.csv")  
R>  
R> # output  
R> c("survey_2021_cleaned.RData",  
+     "resp_ids.csv")  
R>  
R> # Load data -----  
R>  
R> # Plot data -----
```

# Other stuff

- Use **spaces** generously, but not too generously.  
Always put a space after a comma, never before, just like in regular English.
- Use `←`, not `=`, for **assignment**.
- For **logical operators**, prefer `TRUE` and `FALSE` over `T` and `F`.
- To facilitate readability, **keep your lines short**. Strive to limit your code to about 80 characters per line.
- If a **function call is too long** to fit on a single line, use one line each for the function name, each argument, and the closing bracket.
- Use **pipes**. When you use them, they should always have a space before it, and should usually be followed by a new line.

## Spacing

```
R> # Good  
R> mean(x, na.rm = TRUE)  
R> height ← (feet * 12) + inches  
R>  
R> # Bad  
R> mean(x,na.rm=TRUE)  
R> mean ( x, na.rm = TRUE )  
R> height←feet*12+inches
```

## Piping

```
R> babynames %>%  
+   filter(name %>% equals("Kim")) %>%  
+   group_by(year, sex) %>%  
+   summarize(total = sum(n)) %>%  
+   qplot(year, total, color = sex, data = .,  
+         geom = "line") %>%  
+   add(ggtitle('People named "Kim"')) %>%  
+   print
```

# Summary

---

# FAQ

## Q: How much time should I invest to learn the tidyverse?

A: A week clearly is not enough. You will automatically practice more over the course of the semester. Coding is also self-learning, though. Look out for other tidyverse packages that sound interesting, and practice them!

## Q: Should I still learn base R?

A: You are going to, automatically. All I've done is to nudge you to a certain preference. But base R is not evil. It's just a bit less accessible.

## Q: Does the tidyverse also work for Big Data

A: Sure! However, when dealing with large datasets, you might want to consider the `data.table` package as an alternative to `dplyr`. Or just use `dtplyr`, a `data.table` backend for `dplyr` that allows you to write `dplyr` code that is automatically translated to the equivalent, but usually much faster, `data.table` code.

## Q: What from the tidyverse should I learn next?

```
R> sample(tidyverse_packages(), 1)
```