Data Science for Economists

Lecture 5: Data cleaning & wrangling: (1) Tidyverse

Grant McDermott, adapted by Kyle Coombs
Bates College | EC/DCS 368

Table of contents

1. Prologue
2. Tidyverse basics
3. Data wrangling with dplyr
    - filter
    - arrange
    - select
    - mutate
    - summarise
    - joins
4. Data tidying with tidyr
    - pivot_longer / pivot_wider
5. Summary

Prologue

Why so many packages?

- You are probably wondering why there are so many packages in R that do similar things.
- How come you need to know this many packages? Isn't this a bit much?
- Think back to our clean code principles.
    - One of the key practices of clean code is to abstract away complexity.
    - This is what packages do. They abstract away the complexity to make code easier to read, write, and debug.
    - They offer a consistent interface and set of help documentation.
    - Different packages prioritize different goals -- so you can choose the one that best fits your needs.
    - e.g. the `tidyverse` packages prioritize relational database management (called "tidy" data)
    - `data.table` prioritizes speed and memory efficiency in completing data operations, assumes you're doing the RDBM yourself

Why so many packages?

- You are probably wondering why there are so many packages in R that do similar things.

- How come you need to know this many packages? Isn't this a bit much?

- Think back to our clean code principles.

    - One of the key practices of clean code is to abstract away complexity.
    - This is what packages do. They abstract away the complexity to make code easier to read, write, and debug.
    - They offer a consistent interface and set of help documentation.
    - Different packages prioritize different goals -- so you can choose the one that best fits your needs.
    - e.g. the `tidyverse` packages prioritize relational database management (called "tidy" data)
    - `data.table` prioritizes speed and memory efficiency in completing data operations, assumes you're doing the RDBM yourself

- Of course, different packages have different ways of abstracting away complexity.

- So yes, it is a bit much, but it's also a good thing.

Checklist

R packages you'll need for this lecture

☑ **tidyverse**

- This is a meta-package that loads a suite of other packages, including **dplyr** and **tidyr**, which includes the `starwars` dataset that we'll use for practice.

☑ **nycflights13**

Checklist

R packages you'll need for this lecture

☑ **tidyverse**

- This is a meta-package that loads a suite of other packages, including **dplyr** and **tidyr**, which includes the `starwars` dataset that we'll use for practice.

☑ **nycflights13**

The following code chunk will install (if necessary) and load everything for you.

```
if (!require(pacman)) install.packages('pacman', repos = 'https://cran.rstudio.com')
pacman::p_load(tidyverse, nycflights13)
```

What is "tidy" data?

Resources:

- Vignettes (from the **tidyr** package)
- Original paper (Hadley Wickham, 2014 JSS)

What is "tidy" data?

Resources:

- Vignettes (from the **tidyr** package)
- Original paper (Hadley Wickham, 2014 JSS)

Key points:

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

What is "tidy" data?

Resources:

- Vignettes (from the **tidyr** package)
- Original paper (Hadley Wickham, 2014 JSS)

Key points:

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.

Relational Database Management with R

- Remember Relational Database Management from our work on Empirical Organization?
- Today, we'll learn how to implement it using packages in the `tidyverse`
- We'll cover:
    - Subsetting data
    - Variable creation, renaming, selection
    - Grouping and summarizing data
    - Joining and appending datasets

Tidyverse basics

Tidyverse vs. base R

There is often a direct correspondence between a **tidyverse** command and its **base R** equivalent.

These generally follow a `tidyverse::snake_case` vs `base::period.case` rule:

tidyverse	base
`?readr::read_csv`	`?utils::read.csv`
`?dplyr::if_else`	`?base::ifelse`
`?tibble::tibble`	`?base::data.frame`

Etcetera.

If you call up the above examples, you'll see that the tidyverse alternative:

- Offers enhancements or other useful options (and some restrictions too)
- Better documentation
- More consistent syntax

Tidyverse vs. base R

There is often a direct correspondence between a **tidyverse** command and its **base R** equivalent.

These generally follow a `tidyverse::snake_case` vs `base::period.case` rule:

tidyverse	base
`?readr::read_csv`	`?utils::read.csv`
`?dplyr::if_else`	`?base::ifelse`
`?tibble::tibble`	`?base::data.frame`

Etcetera.

If you call up the above examples, you'll see that the tidyverse alternative:

- Offers enhancements or other useful options (and some restrictions too)
- Better documentation
- More consistent syntax

Remember: There are (almost) always multiple ways to achieve a single goal in R.

Tidyverse packages

Let's load the tidyverse meta-package and check the output.

```r
library(tidyverse)
```

Tidyverse packages

Let's load the tidyverse meta-package and check the output.

```
library(tidyverse)
```

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.)

The tidyverse actually comes with a lot more packages than those loaded automatically.[1]

```
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] "tidyr"         "xml2"          "tidyverse"
```

We'll use most of these packages during the remainder of this course.

- **lubridate** for dates, **rvest** for webscraping, **broom** to `tidy()` R objects into tables
- However, packages still have to be loaded separately with `library()`

[1] It also includes a *lot* of dependencies upon installation. This is a matter of some controversy.

Tidyverse packages (cont.)

Today, however, I'm only really going to focus on two packages:

1. **dplyr**
2. **tidyr**

These are the workhorse packages for cleaning and wrangling data.

- Data cleaning and wrangling occupies an inordinate amount of time, no matter where you are in your research career.
- I cannot underscore this enough
- This course can add structure to the cleaning and wrangling, but it is still a time-consuming process.
- It can be a real bummer, so pick data projects that you are excited about.

dplyr

Key dplyr verbs

There are five key dplyr verbs that you need to learn.

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. `summarise`: Collapse multiple rows into a single summary value.[1]

[1] `summarize` with a "z" works too, but Hadley Wickham is from New Sealand.

Learn the verbs

Practice these commands together using the `starwars` data frame that comes pre-packaged with dplyr. **Stop** when you hit the last `summarise` slide (approx. 33).

```
starwars
```

```
## # 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 Luke Sk…     172    77 blond      fair       blue            19   male  mascu…
##  2 C-3PO        167    75 <NA>       gold       yellow         112   none  mascu…
##  3 R2-D2         96    32 <NA>       white, bl… red             33   none  mascu…
##  4 Darth V…     202   136 none       white      yellow          41.9 male  mascu…
##  5 Leia Or…     150    49 brown      light      brown           19   fema… femin…
##  6 Owen La…     178   120 brown, gr… light      blue            52   male  mascu…
##  7 Beru Wh…     165    75 brown      light      blue            47   fema… femin…
##  8 R5-D4         97    32 <NA>       white, red red             NA   none  mascu…
##  9 Biggs D…     183    84 black      light      brown           24   male  mascu…
## 10 Obi-Wan…     182    77 auburn, w… fair       blue-gray       57   male  mascu…
## # ℹ 77 more rows
## # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

1) dplyr::filter

Filter means "subset" the rows of a data frame based on some condition(s).

```
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  mascu…
## 2 Qui-Gon …    193    89 brown      fair       blue            92   male  mascu…
## 3 Dooku        193    80 white      fair       brown          102   male  mascu…
## 4 Bail Pre…    191    NA black      tan        brown           67   male  mascu…
## # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

We can chain multiple commands with the pipe `%>%` as we've seen[1].

[1] Pipes were invented by Doug McIlroy in 1964, are widely used in Unix shells (e.g. bash) and other programming languages (e.g. `F#`). They pass the preceding object as the first argument to the following function. In R, they allow you to chain together code in a way that reads from left to right.

The pipe

- The pipe `%>%` is important for making your code readable, and minimizing balanced-parentheses errors
- It takes whatever is on its left and makes it the first argument of the function on the right
- So whatever object you're working with you take, ship it along to the next function, process, then ship along again, then ship along again! Like a conveyer belt
- Notice that all **dplyr** functions take the data frame as the first argument, making it easy to chain them
- "Ships along" anything, including vectors or single numbers, not just data frames! Track what the object being shipped is in each step.

The pipe

- See how clean it can make the code!

```
mean(starwars[starwars$species == "Human" & starwars$height>=190,]$height, na.rm = TRUE)
```

```
## [1] 194.75
```

vs.

```
starwars %>% # Specify data
  filter(species == "Human", height >= 190) %>% # Specify filter
  pull(height) %>% # Specify the column you want
  mean(na.rm = TRUE) # Calculate the mean
```

```
## [1] 194.75
```

1) dplyr::filter *cont.*

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

```
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>      <chr>          <dbl> <chr> <chr>
## 1 Arvel Cr…     NA    NA brown      fair       brown             NA male  mascu…
## 2 Finn          NA    NA black      dark       dark              NA male  mascu…
## 3 Rey           NA    NA brown      light      hazel             NA fema… femin…
## 4 Poe Dame…     NA    NA brown      light      brown             NA male  mascu…
## 5 BB8           NA    NA none       none       black             NA none  mascu…
## 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))`. Try this yourself.

2) dplyr::arrange

```
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  mascu…
##  2 IG-88        200 140   none       metal      red             15   none  mascu…
##  3 Luke Sk…     172  77   blond      fair       blue            19   male  mascu…
##  4 Leia Or…     150  49   brown      light      brown           19   fema… femin…
##  5 Wedge A…     170  77   brown      fair       hazel           21   male  mascu…
##  6 Plo Koon     188  80   none       orange     black           22   male  mascu…
##  7 Biggs D…     183  84   black      light      brown           24   male  mascu…
##  8 Han Solo     180  80   brown      fair       brown           29   male  mascu…
##  9 Lando C…     177  79   black      dark       brown           31   male  mascu…
## 10 Boba Fe…     183  78.2 black      fair       brown           31.5 male  mascu…
## # ℹ 77 more rows
## # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

2) dplyr::arrange

```
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  mascu…
##  2 IG-88        200 140   none       metal      red             15   none  mascu…
##  3 Luke Sk…     172  77   blond      fair       blue            19   male  mascu…
##  4 Leia Or…     150  49   brown      light      brown           19   fema… femin…
##  5 Wedge A…     170  77   brown      fair       hazel           21   male  mascu…
##  6 Plo Koon     188  80   none       orange     black           22   male  mascu…
##  7 Biggs D…     183  84   black      light      brown           24   male  mascu…
##  8 Han Solo     180  80   brown      fair       brown           29   male  mascu…
##  9 Lando C…     177  79   black      dark       brown           31   male  mascu…
## 10 Boba Fe…     183  78.2 black      fair       brown           31.5 male  mascu…
## # 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. Try this yourself by arranging according to the "name" column.

2) dplyr::arrange *cont.*

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

```
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  mascu…
##  2 Jabba D…    175  1358 <NA>       green-tan… orange           600 herm… mascu…
##  3 Chewbac…    228   112 brown      unknown    blue             200 male  mascu…
##  4 C-3PO       167    75 <NA>       gold       yellow           112 none  mascu…
##  5 Dooku       193    80 white      fair       brown            102 male  mascu…
##  6 Qui-Gon…    193    89 brown      fair       blue              92 male  mascu…
##  7 Ki-Adi-…    198    82 white      pale       yellow            92 male  mascu…
##  8 Finis V…    170    NA blond      fair       blue              91 male  mascu…
##  9 Palpati…    170    75 grey       pale       yellow            82 male  mascu…
## 10 Cliegg …    183    NA brown      fair       blue              82 male  mascu…
## # ℹ 77 more rows
## # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

3) dplyr::select

Select means subset the columns of a data frame based on their names.

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

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

```
## # A tibble: 6 × 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
```

3) dplyr::select *cont.*

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

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

```
## # A tibble: 6 × 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
```

3) dplyr::select *cont.*

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

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

```
## # A tibble: 6 × 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
```

If you just want to rename columns without subsetting them, you can use `rename`. Try this now by replacing `select( ... )` in the above code chunk with `rename( ... )`.

4) dplyr::mutate

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

```
starwars %>%
  select(name, birth_year) %>%
  mutate(dog_years = birth_year * 7) %>%
  mutate(comment = paste0(name, " is ", dog_years, " in dog years.")) %>%
  head()
```

```
## # 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.
```

4) dplyr::mutate *cont.*

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

```
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 labe
```

```
## # 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:

```
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
```

5) dplyr::summarise

Particularly useful in combination with the `group_by`[1] command.

```
starwars %>%
  group_by(species, gender) %>%
  summarise(mean_height = mean(height, na.rm = TRUE)) %>%
  head()
```

```
## # A tibble: 6 × 3
## # Groups:   species [6]
##   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
## 6 Droid    feminine           96
```

Note: **dplyr** 1.0.0 also notifies you about grouping variables every time you do operations on or with them. YMMV, but I switch them off with `options(dplyr.summarise.inform = FALSE)` in my `.Rprofile`.

5) dplyr::summarise *cont.*

Note that including "na.rm = TRUE" (or, its alias "na.rm = T") is usually a good idea with summarise functions. Otherwise, your output will be missing too.

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

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

```
## Much better
starwars %>%
  summarise(mean_height = mean(height, na.rm = TRUE))
```

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

5) dplyr::summarise *cont.*

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

```
starwars %>%
  group_by(species) %>%
  summarise(across(where(is.numeric), ~mean(.x, na.rm=T))) %>%
  head()
```

```
## # A tibble: 6 × 4
##   species  height  mass birth_year
##   <chr>     <dbl> <dbl>      <dbl>
## 1 Aleena      79    15        NaN
## 2 Besalisk   198   102        NaN
## 3 Cerean     198    82         92
## 4 Chagrian   196   NaN        NaN
## 5 Clawdite   168    55        NaN
## 6 Droid      131.   69.8       53.3
```

5) dplyr::summarise *cont.*

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

```
starwars %>%
  group_by(species) %>%
  summarise(across(where(is.numeric), ~mean(.x, na.rm=T))) %>%
  head()
```

```
## # A tibble: 6 × 4
##   species  height  mass birth_year
##   <chr>     <dbl> <dbl>      <dbl>
## 1 Aleena      79    15         NaN
## 2 Besalisk   198   102         NaN
## 3 Cerean     198    82          92
## 4 Chagrian   196   NaN         NaN
## 5 Clawdite   168    55         NaN
## 6 Droid      131.   69.8        53.3
```

Try to intuit what `.x` does above!

Other dplyr goodies

`group_by` and `ungroup`: For (un)grouping.

- Particularly useful with the `summarise` and `mutate` commands, as we've already seen.

Other dplyr goodies

`group_by` and `ungroup`: For (un)grouping.

- Particularly useful with the `summarise` and `mutate` commands, as we've already seen.

`slice`: Subset rows by position rather than filtering by values.

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

Other dplyr goodies

`group_by` and `ungroup`: For (un)grouping.

- Particularly useful with the `summarise` and `mutate` commands, as we've already seen.

`slice`: Subset rows by position rather than filtering by values.

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

`pull`: Extract a column as a vector or scalar.

- `starwars %>% filter(gender=="female") %>% pull(height)` returns `height` as a vector

Other dplyr goodies

`group_by` and `ungroup`: For (un)grouping.

- Particularly useful with the `summarise` and `mutate` commands, as we've already seen.

`slice`: Subset rows by position rather than filtering by values.

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

`pull`: Extract a column as a vector or scalar.

- `starwars %>% filter(gender=="female") %>% pull(height)` returns `height` as a vector

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

- `starwars %>% count(species)`, or `starwars %>% distinct(species)`
- Or use `mutate`, `group_by`, and `n()`, e.g. `starwars %>% group_by(species) %>% mutate(num = n())`.

There are also window functions for leads and lags, ranks, cumulative aggregation, etc.

- See `vignette("window-functions")`.

Quick quiz

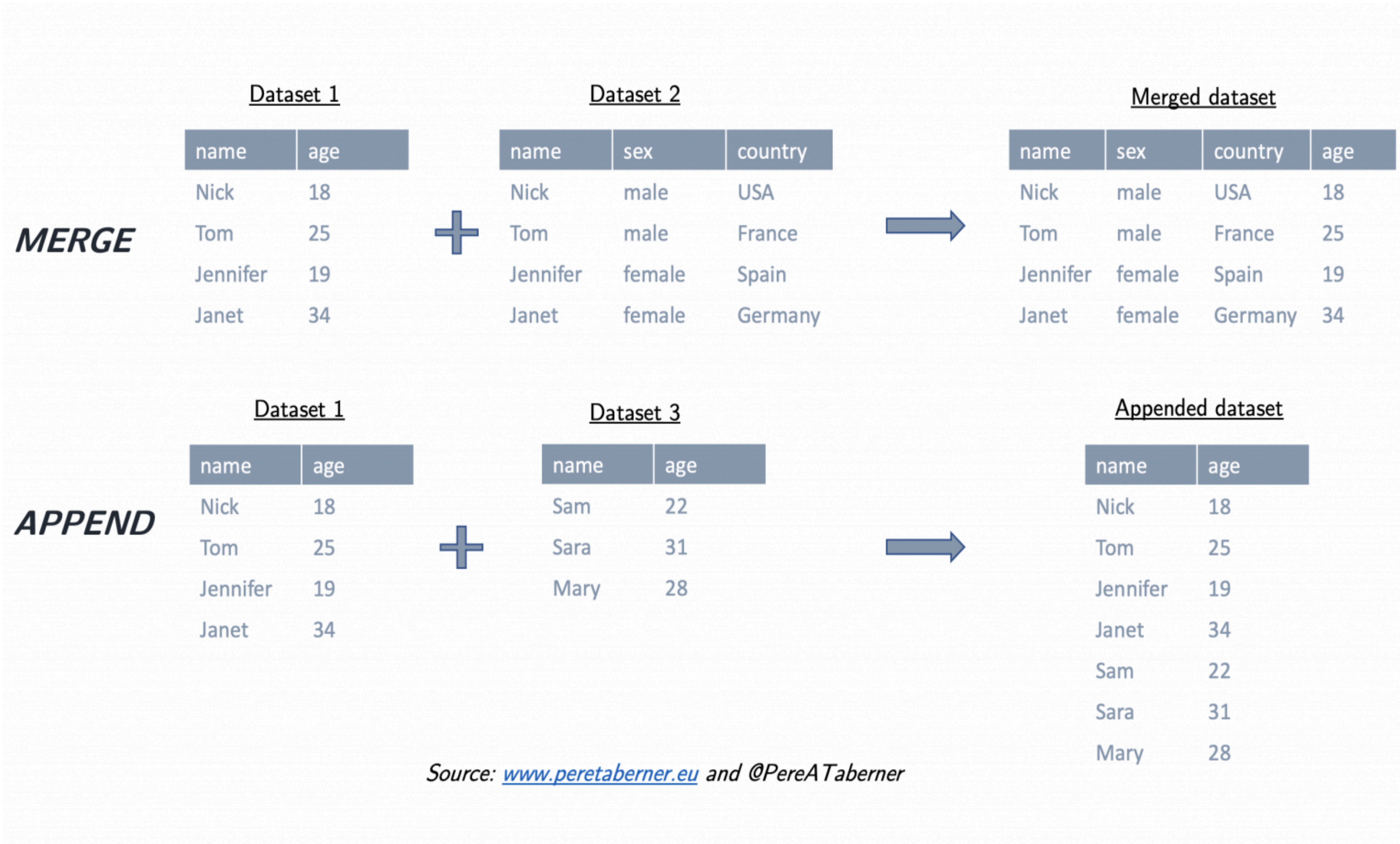
Write me code that will tells me the average birth year of characters by homeworld of the human characters in the `starwars` dataset.

Combining data frames

The final set of dplyr "goodies" are the family of **append** and **join** operations. However, these are important enough that I want to go over some concepts in a bit more depth...

- We will encounter and practice these many more times as the course progresses.
- Imagine you have two data frames, `df1` and `df2`, that you want to combine.
  - You can **append** or **bind**: stack the datasets on top of each other and match up the columns using `bind_rows()`
  - You can **merge** or **join**: match the rows based on a common identifier using `left_join()`, `inner_join()`, etc.
- The appropriate choice depends on the task you are trying to accomplish
  - Are you trying to add new observations or new variables?

Visualize the difference



MERGE

Dataset 1

name	age
Nick	18
Tom	25
Jennifer	19
Janet	34

Dataset 2

name	sex	country
Nick	male	USA
Tom	male	France
Jennifer	female	Spain
Janet	female	Germany

Merged dataset

name	sex	country	age
Nick	male	USA	18
Tom	male	France	25
Jennifer	female	Spain	19
Janet	female	Germany	34

APPEND

Dataset 1

name	age
Nick	18
Tom	25
Jennifer	19
Janet	34

Dataset 3

name	age
Sam	22
Sara	31
Mary	28

Appended dataset

name	age
Nick	18
Tom	25
Jennifer	19
Janet	34
Sam	22
Sara	31
Mary	28

Source: www.peretaberner.eu and @PereATaberner

Taken from Pere A. Taberner.

Appending

- One way to append in the `tidyverse` is with `bind_rows()`
  - Base R has `rbind()`, which requires column names to match
  - `data.table` has `rbindlist()`, which requires column names to match unless you specify `fill`

```r
df1 <- data.frame(x = 1:3, y = 4:6)
df2 <- data.frame(x = 1:4, y = 10:13, z=letters[1:4])

## Append df2 to df1
bind_rows(df1, df2)
```

```
##   x  y    z
## 1 1  4 <NA>
## 2 2  5 <NA>
## 3 3  6 <NA>
## 4 1 10    a
## 5 2 11    b
## 6 3 12    c
## 7 4 13    d
```

Joins

One of the mainstays of the dplyr package is merging data with the family join operations.

- `inner_join(df1, df2)`
- `left_join(df1, df2)`
- `right_join(df1, df2)`
- `full_join(df1, df2)`
- `semi_join(df1, df2)`
- `anti_join(df1, df2)`

Joins are how you get **Relational Database Managment** (RDBM) to work in R.

(See visual depictions of the different join operations here.)

Joins (cont.)

Datasets to merge:

name	country
Nick	USA
Tom	France
Sara	France

name	age
Nick	18
Tom	25
Jennifer	19

Outputs:

inner_join()

name	country	age
Nick	USA	18
Tom	France	25

full_join()

name	country	age
Nick	USA	18
Tom	France	25
Sara	France	
Jennifer		19

left_join()

name	country	age
Nick	USA	18
Tom	France	25
Sara	France	

right_join()

name	country	age
Nick	USA	18
Tom	France	25
Jennifer		19

semi_join()

name	country
Nick	USA
Tom	France

anti_join()

name	country
Sara	France

Source: www.peretaberner.eu and @PereATaberner

Relational Database Management with R

- Remember relational database management?
- Each dataframe has a unique identifier (a "key") that links it to other dataframes.
- All the dataframes have the keys in common, so you can match them up
- Let's get a less abstract example using flights

nycflights13 data

The `flights` data frame contains information flights that departed from NYC in 2013.

- All flight information is stored in the `flights` data frame.
- Information about the planes (like year built) in the `planes` data frame.

```
## # A tibble: 6 × 6
##   flight tailnum  year month   day dep_time
##    <int> <chr>   <int> <int> <int>    <int>
## 1   1545 N14228   2013     1     1      517
## 2   1714 N24211   2013     1     1      533
## 3   1141 N619AA   2013     1     1      542
## 4    725 N804JB   2013     1     1      544
## 5    461 N668DN   2013     1     1      554
## 6   1696 N39463   2013     1     1      554
```

```
## # A tibble: 6 × 4
##   tailnum  year manufacturer     model
##   <chr>   <int> <chr>            <chr>
## 1 N10156   2004 EMBRAER          EMB-145XR
## 2 N102UW   1998 AIRBUS INDUSTRIE A320-214
## 3 N103US   1999 AIRBUS INDUSTRIE A320-214
## 4 N104UW   1999 AIRBUS INDUSTRIE A320-214
## 5 N10575   2002 EMBRAER          EMB-145LR
## 6 N105UW   1999 AIRBUS INDUSTRIE A320-214
```

Joins (cont.)

Let's perform a left join on the flights and planes datasets.

- *Note*: I'm going subset columns after the join, but only to keep text on the slide.

Joins (cont.)

Let's perform a left join on the flights and planes datasets.

- *Note*: I'm going subset columns after the join, but only to keep text on the slide.

```
left_join(flights, planes) %>%
  select(year, month, day, dep_time, arr_time, carrier, flight, tailnum, type, model)
```

```
## Joining with by = join_by(year, tailnum)
```

```
## # A tibble: 336,776 × 10
##     year month   day dep_time arr_time carrier flight tailnum type  model
##    <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr> <chr>
##  1  2013     1     1      517      830 UA        1545 N14228  <NA>  <NA>
##  2  2013     1     1      533      850 UA        1714 N24211  <NA>  <NA>
##  3  2013     1     1      542      923 AA        1141 N619AA  <NA>  <NA>
##  4  2013     1     1      544     1004 B6         725 N804JB  <NA>  <NA>
##  5  2013     1     1      554      812 DL         461 N668DN  <NA>  <NA>
##  6  2013     1     1      554      740 UA        1696 N39463  <NA>  <NA>
##  7  2013     1     1      555      913 B6         507 N516JB  <NA>  <NA>
##  8  2013     1     1      557      709 EV        5708 N829AS  <NA>  <NA>
##  9  2013     1     1      557      838 B6          79 N593JB  <NA>  <NA>
## 10  2013     1     1      558      753 AA         301 N3ALAA  <NA>  <NA>
## # ℹ 336,766 more rows
```

Joins (cont.)

(*continued from previous slide*)

Note that dplyr made a reasonable guess about which columns to join on (i.e. columns that share the same name). It also told us its choices:

```
## Joining, by = c("year", "tailnum")
```

However, there's a problem here: the variable "year" does not have a consistent meaning across our joining datasets!

- In one it refers to the *year of flight*, in the other it refers to *year of construction*.

Joins (cont.)

(*continued from previous slide*)

Note that dplyr made a reasonable guess about which columns to join on (i.e. columns that share the same name). It also told us its choices:

```
## Joining, by = c("year", "tailnum")
```

However, there's a problem here: the variable "year" does not have a consistent meaning across our joining datasets!

- In one it refers to the *year of flight*, in the other it refers to *year of construction*.

Luckily, there's an easy way to avoid this problem.

- See if you can figure it out before turning to the next slide.
- Get help with `?dplyr::join`

Joins (cont.)

(*continued from previous slide*)

You just need to be more explicit in your join call by using the `by =` argument.

- You can also rename any ambiguous columns to avoid confusion.

```
left_join(
  flights,
  planes %>% rename(year_built = year), ## Not necessary w/ below line, but helpful
  by = "tailnum" ## Be specific about the joining column
  ) %>%
  select(year, month, day, dep_time, arr_time, carrier, flight, tailnum, year_built, type, m
  head(3) ## Just to save vertical space on the slide
```

```
## # A tibble: 3 × 11
##    year month   day dep_time arr_time carrier flight tailnum year_built type
##   <int> <int> <int>    <int>    <int> <chr>    <int> <chr>        <int> <chr>
## 1  2013     1     1      517      830 UA        1545 N14228        1999 Fixed w…
## 2  2013     1     1      533      850 UA        1714 N24211        1998 Fixed w…
## 3  2013     1     1      542      923 AA        1141 N619AA        1990 Fixed w…
## # ℹ 1 more variable: model <chr>
```

Joins (cont.)

(*continued from previous slide*)

Last thing I'll mention for now; note what happens if we again specify the join column... but don't rename the ambiguous "year" column in at least one of the given data frames.

```
left_join(
  flights,
  planes, ## Not renaming "year" to "year_built" this time
  by = "tailnum"
  ) %>%
  select(contains("year"), month, day, dep_time, arr_time, carrier, flight, tailnum, type, 
  head(3)
```

```
## # A tibble: 3 × 11
##   year.x year.y month   day dep_time arr_time carrier flight tailnum type  model
##    <int>  <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr> <chr>
## 1   2013   1999     1     1      517      830 UA        1545 N14228  Fixe… 737-…
## 2   2013   1998     1     1      533      850 UA        1714 N24211  Fixe… 737-…
## 3   2013   1990     1     1      542      923 AA        1141 N619AA  Fixe… 757-…
```

Joins (cont.)

(*continued from previous slide*)

Last thing I'll mention for now; note what happens if we again specify the join column... but don't rename the ambiguous "year" column in at least one of the given data frames.

```
left_join(
  flights,
  planes, ## Not renaming "year" to "year_built" this time
  by = "tailnum"
  ) %>%
  select(contains("year"), month, day, dep_time, arr_time, carrier, flight, tailnum, type, 
  head(3)
```

```
## # A tibble: 3 × 11
##   year.x year.y month   day dep_time arr_time carrier flight tailnum type  model
##    <int>  <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr> <chr>
## 1   2013   1999     1     1      517      830 UA        1545 N14228  Fixe… 737-…
## 2   2013   1998     1     1      533      850 UA        1714 N24211  Fixe… 737-…
## 3   2013   1990     1     1      542      923 AA        1141 N619AA  Fixe… 757-…
```

Make sure you know what "year.x" and "year.y" are. Again, it pays to be specific.

tidyr

Key tidyr verbs

1. `pivot_longer`: Pivot wide data into long format.

2. `pivot_wider`: Pivot long data into wide format.

3. `separate`, `unite`, `fill`, `expand`, `nest`, `unnest`: Various other data tidying operations.

    - There are many utilities in the `tidyr` package that help you clean and wrangle data.
    - But they are best learned through experience

Key tidyr verbs

1. `pivot_longer`: Pivot wide data into long format.

2. `pivot_wider`: Pivot long data into wide format.

3. `separate`, `unite`, `fill`, `expand`, `nest`, `unnest`: Various other data tidying operations.

    - There are many utilities in the `tidyr` package that help you clean and wrangle data.
    - But they are best learned through experience

Let's practice these verbs together in class.

- Side question: Which of `pivot_longer` vs `pivot_wider` produces "tidy" data?

1) tidyr::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))
stocks
```

```
##         time          X          Y         Z
## 1 2009-01-01  0.4139186 -0.3254475  2.087752
## 2 2009-01-02 -1.2610702 -3.8178951 -3.455760
```

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

```
## # A tibble: 6 × 3
##   time       stock  price
##   <date>     <chr>  <dbl>
## 1 2009-01-01 X      0.414
## 2 2009-01-01 Y     -0.325
## 3 2009-01-01 Z      2.09
## 4 2009-01-02 X     -1.26
## 5 2009-01-02 Y     -3.82
## 6 2009-01-02 Z     -3.46
```

2) tidyr::pivot_wider

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

```
## # A tibble: 2 × 4
##   time            X     Y     Z
##   <date>      <dbl> <dbl> <dbl>
## 1 2009-01-01 0.0231 -2.08 -2.22
## 2 2009-01-02 1.25   -3.45  6.01
```

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

```
## # A tibble: 3 × 3
##   stock `2009-01-01` `2009-01-02`
##   <chr>        <dbl>        <dbl>
## 1 X           0.0231         1.25
## 2 Y          -2.08          -3.45
## 3 Z          -2.22           6.01
```

2) tidyr::pivot_wider

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

```
## # A tibble: 2 × 4
##   time            X     Y     Z
##   <date>      <dbl> <dbl> <dbl>
## 1 2009-01-01 0.0231 -2.08 -2.22
## 2 2009-01-02 1.25   -3.45  6.01
```

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

```
## # A tibble: 3 × 3
##   stock `2009-01-01` `2009-01-02`
##   <chr>        <dbl>        <dbl>
## 1 X           0.0231         1.25
## 2 Y          -2.08          -3.45
## 3 Z          -2.22           6.01
```

Note that the second example — which has combined different pivoting arguments — has effectively transposed the data.

2) tidyr::pivot_longer with prefix

Let's pivot the pre-loaded billboard data: showing weekly rankings of top 100 in the year 2000

```
head(billboard)
```

```
## # A tibble: 6 × 79
##   artist       track date.entered   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8
##   <chr>        <chr> <date>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 2 Pac        Baby… 2000-02-26      87    82    72    77    87    94    99    NA
## 2 2Ge+her      The … 2000-09-02      91    87    92    NA    NA    NA    NA    NA
## 3 3 Doors Do… Kryp… 2000-04-08      81    70    68    67    66    57    54    53
## 4 3 Doors Do… Loser 2000-10-21      76    76    72    69    67    65    55    59
## 5 504 Boyz     Wobb… 2000-04-15      57    34    25    17    17    31    36    49
## 6 98^0         Give… 2000-08-19      51    39    34    26    26    19     2     2
## # ℹ 68 more variables: wk9 <dbl>, wk10 <dbl>, wk11 <dbl>, wk12 <dbl>,
## #   wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>,
## #   wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>,
## #   wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>, wk30 <dbl>,
## #   wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>, wk36 <dbl>,
## #   wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>, wk41 <dbl>, wk42 <dbl>,
## #   wk43 <dbl>, wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, wk48 <dbl>, …
```

2) tidyr::pivot_longer with prefix *cont.*

Wait, why is there 'wk' in the 'week' column?

```
billboard %>%
  pivot_longer(cols=starts_with('wk'), names_to="week",
    values_to="rank") %>%
  head()
```

```
## # A tibble: 6 × 5
##   artist track                   date.entered week   rank
##   <chr>  <chr>                   <date>       <chr> <dbl>
## 1 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk1      87
## 2 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk2      82
## 3 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk3      72
## 4 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk4      77
## 5 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk5      87
## 6 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk6      94
```

2) tidyr::pivot_longer with prefix *cont.*

That fixed it.

```
billboard %>%
  pivot_longer(cols=starts_with('wk'), names_to="week",
    values_to="rank",names_prefix='wk') %>%
  mutate(week=as.numeric(week)) %>% # Make week a numeric variable
  head()
```

```
## # A tibble: 6 × 5
##   artist track                   date.entered  week  rank
##   <chr>  <chr>                   <date>       <dbl> <dbl>
## 1 2 Pac  Baby Don't Cry (Keep... 2000-02-26       1    87
## 2 2 Pac  Baby Don't Cry (Keep... 2000-02-26       2    82
## 3 2 Pac  Baby Don't Cry (Keep... 2000-02-26       3    72
## 4 2 Pac  Baby Don't Cry (Keep... 2000-02-26       4    77
## 5 2 Pac  Baby Don't Cry (Keep... 2000-02-26       5    87
## 6 2 Pac  Baby Don't Cry (Keep... 2000-02-26       6    94
```

Aside: Remembering the pivot_* syntax

There's a long-running joke about no-one being able to remember Stata's "reshape" command. (Exhibit A.)

It's easy to see this happening with the `pivot_*` functions too. Remember the documentation is your friend!

```
?pivot_longer
```

And GitHub CoPilot, ChatGPT and other AI tools are also your friends if you use precise language about what you want the AI tool to do and you try their suggestions carefully.[1]

[1] Back in my day we had to scour StackOverflow for hours to find the right answer. And we liked it!

Other tidyr goodies

- `separate`: Split a single column into multiple columns.
  - `separate(df, col, into = c("A", "B"), sep = "-")` will split `col` into columns `A` and `B` at the `-` separator.
- `unite`: Combine multiple columns into a single column.
  - `unite(df, col, A, B, sep = "-")` combines columns `A` and `B` into column `col` with `-` as the separator.
- `fill`: Fill in missing values with the last non-missing value.
  - `fill(df, starts_with("X"))` will fill in all columns that start with "X".
- `drop_na`: Drop rows with missing values.
- `expand`: Create a complete set of combinations from a set of factors.
- `nest` and `unnest`: Combine columns into lists within a single cell or split a column of lists into separate rows.
  - Try with the `starwars` data frame: `unnest(starwars, films,names_sep='')`

Summary

Key verbs

dplyr

1. `filter`
2. `arrange`
3. `select`
4. `mutate`
5. `summarise`

tidyr

1. `pivot_longer`
2. `pivot_wider`

Key verbs

dplyr

1. `filter`
2. `arrange`
3. `select`
4. `mutate`
5. `summarise`

tidyr

1. `pivot_longer`
2. `pivot_wider`

Other useful items include: pipes (`%>%`), grouping (`group_by`), joining functions (`left_join`, `inner_join`, etc.).

Next lecture: Scraping data!