

Big Data in Economics

Lecture 4: Data cleaning and wrangling: (1) Tidyverse

Grant McDermott

University of Oregon | EC 510

Table of contents

1. Prologue
2. Tidyverse basics
3. Data wrangling with dplyr
4. Data tidying with tidyr
5. Summary

Prologue

What is "tidy" data?

Resources:

- [Vignette](#) (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.

Checklist

☑ You should already have installed the **tidyverse** in the first lecture.

- However, I also want you to install the development version of **dplyr**:

```
remotes::install_github('tidyverse/dplyr')
```

- If the above command doesn't work for you, make sure that you've installed the **remotes** package first. Similarly, if you get a question about upgrading other packages, select "All" (normally option 1).
- I'll get back to this stuff later on in the lecture.

☑ You will also need the **nycflights13** package.

- Install it now:

```
install.packages('nycflights13', repos =  
'https://cran.rstudio.com')
```

Tidyverse basics

Tidyverse vs. base R

Much digital ink has been spilled over the "tidyverse vs. base R" debate.

I won't delve into this debate here, because I think the answer is **obvious**: We should teach the tidyverse first (or, at least, early).

- The documentation and community support are outstanding.
- Having a consistent philosophy and syntax makes it much easier to learn.
- The tidyverse provides a convenient "front-end" to some key big-data tools that we'll use later in the course.
- For data cleaning, wrangling and plotting... the tidyverse is really a no-brainer.¹

But this certainly shouldn't put you off learning base R alternatives.

- Base R is extremely flexible and powerful, esp. when combined with other libraries.
- 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.

¹ I'm also a huge fan of **data.table**. This package will be the subject of our next lecture...

Tidyverse vs. base R (cont.)

One point of convenience is that 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. E.g. Compare:

tidyverse	base
<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>

Etcetera.

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.

- Remember: There are always many ways to achieve a single goal in R.

Tidyverse packages

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

```
library(tidyverse)
```

```
## — Attaching packages
```

```
## ✓ ggplot2 3.3.0          ✓ purrr 0.3.3  
## ✓ tibble 3.0.0           ✓ dplyr 0.8.99.9002  
## ✓ tidyr 1.0.2            ✓ stringr 1.4.0  
## ✓ readr 1.3.1           ✓ forcats 0.5.0
```

```
## — Conflicts
```

```
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag()     masks stats::lag()
```

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.)

The tidyverse actually comes with a lot more packages than those that are just loaded automatically.¹

```
tidyverse_packages()
```

```
## [1] "broom"      "cli"        "crayon"     "dbplyr"     "dplyr"
## [6] "forcats"    "ggplot2"    "haven"      "hms"        "httr"
## [11] "jsonlite"   "lubridate"  "magrittr"   "modelr"     "pillar"
## [16] "purrr"      "readr"      "readxl"     "reprex"     "rlang"
## [21] "rstudioapi" "rvest"      "stringr"    "tibble"     "tidyr"
## [26] "xml2"       "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 webscraping.

- However, bear in mind that these packages will have to be loaded separately.

.

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

Tidyverse packages (cont.)

I hope to cover most of the tidyverse packages over the length of this course.

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. They are thus the ones that you will likely make the most use of (alongside **ggplot2**, which we already met back in Lecture 1).

- Data cleaning and wrangling occupies an inordinate amount of time, no matter where you are in your research career.

An aside on pipes: %>%

We already learned about pipes in our lecture on the bash shell. In R, the pipe operator is denoted `%>%` and is automatically loaded with the tidyverse.

I want to reiterate how cool pipes are, and how using them can dramatically improve the experience of reading and writing code. Compare:

```
## These next two lines of code do exactly the same thing.  
mpg %>% filter(manufacturer="audi") %>% group_by(model) %>% summarise(hwy_mean =  
  summarise(group_by(filter(mpg, manufacturer="audi"), model), hwy_mean = mean(hwy))
```

The first line reads from left to right, exactly how I thought of the operations in my head.

- Take this object (mpg), do this (filter), then do this (group by), etc.

The second line totally inverts this logical order (the final operation comes first!)

- Who wants to read things inside out?

An aside on pipes: %>% (cont.)

The piped version of the code is even more readable if we write it over several lines. Here it is again and, this time, I'll run it for good measure so you can see the output:

```
mpg %>%  
  filter(manufacturer=="audi") %>%  
  group_by(model) %>%  
  summarise(hwy_mean = mean(hwy))
```

```
## # A tibble: 3 x 2  
##   model      hwy_mean  
##   <chr>      <dbl>  
## 1 a4         28.3  
## 2 a4 quattro  25.8  
## 3 a6 quattro  24
```

Remember: Using vertical space costs nothing and makes for much more readable/writeable code than cramming things horizontally.

PS — The pipe is originally from the **magrittr** package ([geddit?](#)), which can do some other cool things if you're inclined to explore.

dplyr

Aside: Impending dplyr 1.0.0 release

The creators of dplyr are planning to release [version 1.0.0](#) of the package in a few weeks.

- This is a big deal, since it marks a stable code base (i.e. functions won't be changing much going forward) and also introduces a bunch of new features.
- If you haven't done so already, I want you to install the development version of dplyr so that you can access these new features:

```
# install.packages(remotes)  
remotes::install_github('tidyverse/dplyr')
```

P.S. See the original dplyr 1.0.0 announcement [here](#). The tidyverse blog is also running a series of posts on the new dplyr features (e.g. [here](#)).

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

Let's practice these commands together using the `starwars` data frame that comes pre-packaged with dplyr.

¹ `summarize` with a "z" works too. R doesn't discriminate against uncivilised nations of the world.

1) dplyr::filter

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

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

```
## # A tibble: 4 x 13  
##   name    height  mass hair_color skin_color eye_color birth_year gender homeworld  
##   <chr>   <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr>  <chr>  
## 1 Dart...   202   136 none       white      yellow      41.9 male   Tatooine  
## 2 Qui-...   193    89 brown      fair       blue        92  male   <NA>  
## 3 Dooku     193    80 white      fair       brown       102  male   Serenno  
## 4 Bail...   191    NA black      tan        brown       67  male   Alderaan  
## # ... with 4 more variables: species <chr>, films <list>, vehicles <list>,  
## #   starships <list>
```

1) dplyr::filter cont.

Regular expressions work well too.

```
starwars %>%  
  filter(grepl("Skywalker", name))
```

```
## # A tibble: 3 x 13  
##   name    height  mass hair_color skin_color eye_color birth_year gender homeworld  
##   <chr>   <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr>  <chr>  
## 1 Luke...   172    77 blond      fair        blue         19  male   Tatooine  
## 2 Anak...   188    84 blond      fair        blue        41.9  male   Tatooine  
## 3 Shmi...   163    NA black      fair        brown        72  female Tatooine  
## # ... with 4 more variables: species <chr>, films <list>, vehicles <list>,  
## #   starships <list>
```

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 x 13  
##   name    height    mass hair_color skin_color eye_color birth_year gender homeworld  
##   <chr>   <int>   <dbl> <chr>      <chr>      <chr>      <dbl> <chr>   <chr>  
## 1 Arve...     NA     NA brown      fair        brown          NA male   <NA>  
## 2 Finn        NA     NA black      dark        dark          NA male   <NA>  
## 3 Rey         NA     NA brown      light       hazel          NA female <NA>  
## 4 Poe ...     NA     NA brown      light       brown          NA male   <NA>  
## 5 BB8         NA     NA none       none        black          NA none   <NA>  
## 6 Capt...     NA     NA unknown   unknown    unknown          NA female <NA>  
## # ... with 4 more variables: 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 x 13  
##   name      height  mass hair_color skin_color eye_color birth_year gender  
##   <chr>    <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr>  
## 1 Wick...      88   20  brown      brown      brown         8  male  
## 2 IG-88      200  140  none       metal      red          15  none  
## 3 Luke...     172   77  blond      fair       blue         19  male  
## 4 Leia...     150   49  brown      light      brown         19  female  
## 5 Wedg...     170   77  brown      fair       hazel         21  male  
## 6 Plo ...     188   80  none       orange     black         22  male  
## 7 Bigg...     183   84  black      light      brown         24  male  
## 8 Han ...     180   80  brown      fair       brown         29  male  
## 9 Land...     177   79  black      dark       brown         31  male  
## 10 Boba...    183  78.2  black      fair       brown        31.5  male  
## # ... with 77 more rows, and 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 x 13  
##   name      height  mass hair_color skin_color eye_color birth_year gender  
##   <chr>    <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr>  
## 1 Yoda        66    17 white      green      brown        896 male  
## 2 Jabb...    175   1358 <NA>      green-tan... orange        600 herma...  
## 3 Chew...    228    112 brown      unknown    blue         200 male  
## 4 C-3PO     167     75 <NA>      gold       yellow        112 <NA>  
## 5 Dooku     193     80 white      fair       brown         102 male  
## 6 Qui-...    193     89 brown      fair       blue          92 male  
## 7 Ki-A...    198     82 white      pale       yellow         92 male  
## 8 Fini...    170     NA blond      fair       blue          91 male  
## 9 Palp...    170     75 grey      pale       yellow         82 male  
## 10 Clie...   183     NA brown      fair       blue          82 male  
## # ... with 77 more rows, and 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 "-".

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

```
## # A tibble: 87 x 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  
## # ... with 77 more rows
```

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)
```

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

3) dplyr::select cont.

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

```
starwars %>%  
  select(name, contains("color"))
```

```
## # A tibble: 87 x 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  
## # ... with 77 more rows
```


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.

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



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

Note: The new `relocate` function coming in dplyr 1.0.0 is bringing a lot more functionality to ordering of columns. See [here](#).

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."))
```

```
## # A tibble: 87 x 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        293.3 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.  
## 7 Beru Whitesun lars  47         329 Beru Whitesun lars is 329 in dog yea...  
## 8 R5-D4              NA          NA R5-D4 is NA in dog years.  
## 9 Biggs Darklighter  24         168 Biggs Darklighter is 168 in dog year...  
## 10 Obi-Wan Kenobi     57         399 Obi-Wan Kenobi is 399 in dog years.  
## # ... with 77 more rows
```

4) dplyr::mutate cont

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

```
starwars %>%  
  select(name, birth_year) %>%  
  mutate(  
    dog_years = birth_year * 7, ## Separate with a comma  
    comment = paste0(name, " is ", dog_years, " in dog years.")  
  )
```

```
## # A tibble: 87 x 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     293.3 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.  
## 7 Beru Whitesun lars  47       329 Beru Whitesun lars is 329 in dog yea...  
## 8 R5-D4              NA        NA R5-D4 is NA in dog years.  
## 9 Biggs Darklighter  24       168 Biggs Darklighter is 168 in dog year...  
## 10 Obi-Wan Kenobi    57       399 Obi-Wan Kenobi is 399 in dog years.  
## # ... with 77 more rows
```

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 ch
```

```
## # A tibble: 2 x 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 new `across` feature in dplyr 1.0.0 (or the development that you should've installed) allows you to easily work on a subset of variables, e.g.:

```
starwars %>%  
  select(name:eye_color) %>%  
  mutate(across(is.character, toupper)) %>%  
  head(5)
```

```
## # A tibble: 5 x 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: This workflow (i.e. combining `mutate` and `across`) supersedes the old "scoped" variants of `mutate` that you might have used before. More details [here](#) and [here](#).

5) dplyr::summarise

Particularly useful in combination with the `group_by` command.

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

```
## # A tibble: 43 x 3  
## # Groups:   species [38]  
##   species    gender mean_height  
##   <chr>      <chr>      <dbl>  
## 1 Aleena     male          79  
## 2 Besalisk  male         198  
## 3 Cerean     male         198  
## 4 Chagrian  male         196  
## 5 Clawdite  female        168  
## 6 Droid      none         200  
## 7 Droid      <NA>         120  
## 8 Dug        male         112  
## 9 Ewok       male          88  
## 10 Geonosian male         183  
## # ... with 33 more rows
```

5) dplyr::summarise *cont.*

Note that including "na.rm = T" is usually a good idea with summarise functions. Otherwise, any missing value will propagate to the summarised value too.

```
## Probably not what we want
```

```
starwars %>%  
  summarise(mean_height = mean(height))
```

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

```
## Much better
```

```
starwars %>%  
  summarise(mean_height = mean(height, na.rm = T))
```

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

4) 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(is.numeric, mean, na.rm=T)) %>%  
  head(5)
```

```
## # A tibble: 5 x 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     NA      NaN  
## 5 Clawdite   168     55      NaN
```

Note: Again, this functionality supersedes the old "scoped" variants of `summarise` and is only available with the development version of dplyr. Details [here](#) and [here](#).

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.

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

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

- E.g. `starwars %>% filter(gender="female") %>% pull(height)`

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

- E.g. `starwars %>% count(species)`, or `starwars %>% distinct(species)`
- 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.)

There are also a whole class of **window functions** for getting leads and lags, ranking, creating cumulative aggregates, etc.

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

The final set of dplyr "goodies" are the family of 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.

Joining operations

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)`

For the simple examples that I'm going to show here, we'll need some data sets that come bundled with the **nycflights13** package.

- Load it now and then inspect these data frames in your own console.

```
library(nycflights13)
flights
planes
```

Joining operations (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, mod

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

## # A tibble: 336,776 x 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>
## # ... with 336,766 more rows
```

Joining operations (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 an obvious 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.
- Try `?dplyr::join`.

Joining operations (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)
head(3) ## Just to save vertical space on the slide

## # A tibble: 3 x 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 Fixe...
## 2  2013     1     1     533     850 UA      1714 N24211     1998 Fixe...
## 3  2013     1     1     542     923 AA      1141 N619AA     1990 Fixe...
## # ... with 1 more variable: model <chr>
```

Joining operations (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)
head(3)
```

```
## # A tibble: 3 x 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 (i.e. "melt").¹
2. `pivot_wider`: Pivot long data into wide format (i.e. "cast").²
3. `separate`: Separate (i.e. split) one column into multiple columns.
4. `unite`: Unite (i.e. combine) multiple columns into one.

Let's practice these verbs together in class.

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

¹ Updated version of `tidyr::gather`.

² Updated version of `tidyr::spread`.

1) tidyr::pivot_longer

```
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  0.002747299  0.5968095 -0.6687848
## 2 2009-01-02 -2.588878073  4.5475938 -1.6870363
```

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

```
## # A tibble: 6 x 3
##   time      stock    price
##   <date>   <chr>   <dbl>
## 1 2009-01-01 X       0.00275
## 2 2009-01-01 Y       0.597
## 3 2009-01-01 Z      -0.669
## 4 2009-01-02 X      -2.59
## 5 2009-01-02 Y       4.55
## 6 2009-01-02 Z      -1.69
```

1) tidyr::pivot_longer cont.

Let's quickly save the "tidy" (i.e. long) stocks data frame for use on the next slide.

```
## Write out the argument names this time: i.e. "names_to=" and "values_to="  
tidy_stocks ←  
  stocks %>%  
  pivot_longer(-time, names_to="stock", values_to="price")
```

2) tidyr::pivot_wider

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

```
## # A tibble: 2 x 4
##   time          X      Y      Z
##   <date>      <dbl> <dbl> <dbl>
## 1 2009-01-01  0.00275 0.597 -0.669
## 2 2009-01-02 -2.59    4.55  -1.69
```

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

```
## # A tibble: 3 x 3
##   stock 2009-01-01 2009-01-02
##   <chr>      <dbl>      <dbl>
## 1 X          0.00275      -2.59
## 2 Y          0.597        4.55
## 3 Z         -0.669      -1.69
```

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

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. However, I find that I never forget the commands as long as I remember the argument order is "*names*" then "*values*".

3) tidyr::separate

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

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

```
economists %>% separate(name, c("first_name", "last_name"))
```

```
## first_name last_name  
## 1      Adam      Smith  
## 2      Paul Samuelson  
## 3     Milton   Friedman
```

This command is pretty smart. But to avoid ambiguity, you can also specify the separation character with `separate(... , sep=".")`.

3) tidyr::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).

```
jobs <- data.frame(
  name = c("Jack", "Jill"),
  occupation = c("Homemaker", "Philosopher, Philanthropist, Troublemaker")
)
jobs
```

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

```
## Now split out Jill's various occupations into different rows
jobs %>% separate_rows(occupation)
```

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

4) tidyr::unite

```
gdp <- data.frame(  
  yr = rep(2016, times = 4),  
  mnth = rep(1, times = 4),  
  dy = 1:4,  
  gdp = rnorm(4, mean = 100, sd = 2)  
)  
gdp
```

```
##      yr mnth dy      gdp  
## 1 2016     1  1 100.97242  
## 2 2016     1  2  99.80214  
## 3 2016     1  3  99.76375  
## 4 2016     1  4 100.68899
```

```
## Combine "yr", "mnth", and "dy" into one "date" column  
gdp %>% unite(date, c("yr", "mnth", "dy"), sep = "-")
```

```
##      date      gdp  
## 1 2016-1-1 100.97242  
## 2 2016-1-2  99.80214  
## 3 2016-1-3  99.76375  
## 4 2016-1-4 100.68899
```


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.

```
gdp_u <- gdp %>% unite(date, c("yr", "mnth", "dy"), sep = "-") %>% as_tibble()
gdp_u
```

```
## # A tibble: 4 x 2
##   date      gdp
##   <chr>    <dbl>
## 1 2016-1-1 101.
## 2 2016-1-2  99.8
## 3 2016-1-3  99.8
## 4 2016-1-4 101.
```

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

4) tidyr::unite cont.

(continued from previous slide)

```
library(lubridate)
gdp_u %>% mutate(date = ymd(date))
```

```
## # A tibble: 4 x 2
##   date          gdp
##   <date>        <dbl>
## 1 2016-01-01  101.
## 2 2016-01-02   99.8
## 3 2016-01-03   99.8
## 4 2016-01-04  101.
```

Other tidyr goodies

Use `crossing` to get the full combination of a group of variables.¹

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

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

See `?expand` and `?complete` for more specialised functions that allow you to fill in (implicit) missing data or variable combinations in existing data frames.

- You'll encounter this during your next assignment.

¹ Base R alternative: `expand.grid`.

Summary

Key verbs

dplyr

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

tidyr

1. `pivot_longer`
2. `pivot_wider`
3. `separate`
4. `unite`

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

Next lecture: Data cleaning and
wrangling: (2) data.table
