

# Lecture 3: Data Cleaning & Wrangling with Tidyverse

James Sears\*

AFRE 891 SS 24

Michigan State University

\*Parts of these slides are adapted from “[Data Science for Economists](#)” by Grant McDermott.

# Table of Contents

1. Prologue
2. Tidyverse Overview
3. Pipes
4. Data Wrangling with dplyr
  - filter, arrange, select, mutate, summarise, joins
5. Data Tidying with tidyr
  - pivot\_longer / pivot\_wider
  - separate
  - unite
6. Summary

# Prologue

# What is "tidy" data?

## Resources:

- **Vignette** (from the **tidyr** package)
- **Original paper** (Hadley Wickham, 2014 JSS)
- **Online Book: R 4 Data Science**

## 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

R packages you'll need today

☒ **tidyverse**

☒ **nycflights13**

Let's hold off on loading/installing these right now

# Tidyverse Overview

# Tidyverse vs. base R

One thing to note before we dive into **tidyverse**: 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>

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.

# Tidyverse vs. base R

One thing to note before we dive into **tidyverse**: 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>

**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.

```
library(tidyverse)
```

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

The tidyverse actually comes with a **lot more packages** that aren't loaded automatically.

```
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 several of these additional packages during the remainder of this course: **haven** for loading Stata files, **lubridate** for working with dates, **rvest** package for webscraping.

# Tidyverse Packages

This week we're going to focus on two packages:

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

These are the workhorse packages for cleaning and wrangling data that you will likely make the most use of (alongside **ggplot2**, which we'll cover more in depth later on)

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

# Pipes



# Pipes

The tidyverse pipe operator `%>%` is one of its coolest features.

- It lets you send (i.e. "pipe") intermediate output to another command.
- In other words, it allows us to chain together a sequence of simple operations and thereby implement a more complex operation while preserving legibility

Let's demonstrate with an example.

# Pipes

Suppose you're a big German car fan and want to see average fuel efficiency of their models for 1999-2008. The simple operations involve

1. Load the dataset (`mpg` "loaded" by tidyverse)
2. Filter the data to Audi and Volkswagen (`filter()`)
3. Group the data by model (`group_by()`)
4. Summarise average highway mileage (`summarise()`)

Without pipes, we would need to

1. Assign/reassign intermediate objects to memory after each step (repetitive), or
2. Nest a lot of functions (hard to read)

# Without Pipes: 1. Assign/Reassign

```
cars ← mpg  
german_cars ← filter(mpg, manufacturer %in% c("audi", "volkswagen"))  
german_cars_grp ← group_by(german_cars, manufacturer, model)  
summarise(german_cars_grp, hwy_mean = mean(hwy))
```

This is stack-to to read and leaves us with a bunch of intermediate objects that we'll need to deal with.

```
rm(cars, german_cars, german_cars_grp)
```

# Without Pipes: 2. Nest

The nested approach is harder to read and **totally inverts the logical order**

```
summarise(group_by(filter(mpg, manufacturer %in% c("audi", "volkswagen")))
```

- The final operation comes first!
- Who wants to read things inside out??



# With Pipes

The below line does exactly the same thing through the power of pipes:

```
mpg %>% filter(manufacturer %in% c("audi", "volkswagen")) %>% group_by(mar
```

With pipes the 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.

# Pipes: Improved Readability

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 so we can see the output:

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

```
## # A tibble: 7 × 3  
## # Groups:   manufacturer [2]  
##   manufacturer model      hwy_mean  
##   <chr>         <chr>      <dbl>  
## 1 audi         a4          28.3  
## 2 audi         a4 quattro  25.8  
## 3 audi         a6 quattro  24  
## 4 volkswagen   gti         27.4  
## 5 volkswagen   jetta        29.1  
## 6 volkswagen   new beetle  32.8  
## 7 volkswagen   passat       27.6
```

# Pipes: Improved Readability


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

At each line, the **upstream object/output** (i.e. the `mpg` df) is being passed into the **downstream function** (i.e. `filter()`) as the first argument.



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 (hence the **not-a-pipe image**) earlier, which can do some other cool things if you're inclined to explore.

# The base R pipe: |>

The magrittr pipe has proven so successful and popular, that as of R 4.1.0 the R core team **added a "native" pipe**, denoted .<sup>1</sup> For example:

```
mtcars |> subset(cyl=4) |> head()
```

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

dplyr

# Aside: Updating Packages

- Please make sure that you are running at least **dplyr** 1.0.0 before continuing.
- 1.0.0 has been around for a while now (currently on 1.1.4), but if you have an old old version of dplyr, these functions won't be available
- As well, it's a good idea to frequently update all your packages!

```
packageVersion('dplyr')
```

```
## [1] '1.1.1'
```

```
# install.packages('dplyr') ## install updated version if < 1.0.0
```

Note: You can turn off **dplyr's** notifications about grouping variables with

```
options(dplyr.summarise.inform = FALSE) ## Add to .Rprofile to make perma
```

# 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 or arrange) columns by their names:
4. `mutate`: Create new columns or modify existing columns.
5. `summarise`: Collapse multiple rows into a single summary value.<sup>2</sup>

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

<sup>2</sup> `summarize` with a "z" works too, R doesn't discriminate.

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



# 1) dplyr::filter

Regular expressions work well too.

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

```
## # A tibble: 3 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex    ge  
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <c  
## 1 Luke Sky...   172    77 blond      fair       blue        19   male  ma  
## 2 Anakin S...   188    84 blond      fair       blue       41.9  male  ma  
## 3 Shmi Sky...   163    NA black      fair       brown       72   fema... fe  
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,  
## #   vehicles <list>, starships <list>
```

# 1) dplyr::filter

Or **stringr** functions

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

```
## # A tibble: 3 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex    ge  
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <c  
## 1 Luke Sky...   172    77 blond      fair       blue        19   male  ma  
## 2 Anakin S...   188    84 blond      fair       blue       41.9  male  ma  
## 3 Shmi Sky...   163    NA black      fair       brown       72   fema... fe  
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,  
## #   vehicles <list>, starships <list>
```

# Identifying Missing Data

A very common `filter` use case is identifying (or removing) observation with missing values

```
starwars %>%
  filter(is.na(height))

## # A tibble: 6 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   ge
##   <chr>    <int> <dbl> <chr>      <chr>    <chr>      <dbl> <chr> <c
## 1 Arvel Cr...    NA     NA brown      fair      brown            NA male  ma
## 2 Finn          NA     NA black      dark      dark            NA male  ma
## 3 Rey           NA     NA brown      light     hazel            NA fema... fe
## 4 Poe Dame...    NA     NA brown      light     brown            NA male  ma
## 5 BB8           NA     NA none       none      black            NA none  ma
## 6 Captain ...    NA     NA unknown   unknown   unknown          NA <NA> <M
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

# Removing Missing Data

To remove missing observations, simply use negation:

```
filter(is.na(height))
```

Or use the convenient `drop_na()` verb:

```
dim(starwars)
```

```
## [1] 87 14
```

```
starwars %>%  
  drop_na(height) %>%  
  dim()
```

```
## [1] 81 14
```

## 2) dplyr::arrange

`arrange()` sorts rows based on values of a variable/variables

- **numeric:** ascending order
- **character:** alphabetically (try this on `name` variable)

```
starwars %>%  
  arrange(birth_year) %>%  
  head()
```

```
## # A tibble: 6 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex    ge  
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>  
## 1 Wicket S...    88    20 brown      brown      brown         8 male  ma  
## 2 IG-88         200   140 none       metal      red          15 none  ma  
## 3 Luke Sky...   172    77 blond      fair       blue          19 male  ma  
## 4 Leia Org...   150    49 brown      light      brown         19 fema... fe  
## 5 Wedge An...   170    77 brown      fair       hazel         21 male  ma  
## 6 Plo Koon     188    80 none       orange     black         22 male  ma  
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>, 29 / 85  
## #   vehicles <list>, starships <list>
```

## 2) dplyr::arrange

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	ge
	<chr>	<int>	<dbl>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>
## 1	Yoda	66	17	white	green	brown	896	male	ma
## 2	Jabba D...	175	1358	<NA>	green-tan...	orange	600	herm...	ma
## 3	Chewbac...	228	112	brown	unknown	blue	200	male	ma
## 4	C-3PO	167	75	<NA>	gold	yellow	112	none	ma
## 5	Dooku	193	80	white	fair	brown	102	male	ma
## 6	Qui-Gon...	193	89	brown	fair	blue	92	male	ma
## 7	Ki-Adi-...	198	82	white	pale	yellow	92	male	ma
## 8	Finis V...	170	NA	blond	fair	blue	91	male	ma
## 9	Palpati...	170	75	grey	pale	yellow	82	male	ma
## 10	Cliegg ...	183	NA	brown	fair	blue	82	male	ma

```
## # i 77 more rows
```

```
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>, 30 / 85
```

## 2) dplyr::arrange

We can also nested sort by including multiple variables

```
starwars %>%  
  arrange(desc(birth_year))
```

```
## # A tibble: 87 × 14
```

	name	height	mass	hair_color	skin_color	eye_color	birth_year	sex	ge
	<chr>	<int>	<dbl>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>
## 1	Yoda	66	17	white	green	brown	896	male	ma
## 2	Jabba D...	175	1358	<NA>	green-tan...	orange	600	herm...	ma
## 3	Chewbac...	228	112	brown	unknown	blue	200	male	ma
## 4	C-3PO	167	75	<NA>	gold	yellow	112	none	ma
## 5	Dooku	193	80	white	fair	brown	102	male	ma
## 6	Qui-Gon...	193	89	brown	fair	blue	92	male	ma
## 7	Ki-Adi-...	198	82	white	pale	yellow	92	male	ma
## 8	Finis V...	170	NA	blond	fair	blue	91	male	ma
## 9	Palpati...	170	75	grey	pale	yellow	82	male	ma
## 10	Cliegg ...	183	NA	brown	fair	blue	82	male	ma

```
## # i 77 more rows
```

```
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>, 31 / 85
```

### 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 "-".

- Variables will appear in the order you specify the arguments

```
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
```



### 3) dplyr::select

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

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

```
## # A tibble: 6 × 3  
##   alias      crib      gender  
##   <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

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

```
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			

### 3) dplyr::select

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 × 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>
```

### 3) dplyr::select

**dplyr** has an entire group of **selection helpers** that can be used in many functions:

<code>start_with("D")</code>	names starting with "D"
<code>ends_with("_hh")</code>	names ending with "_hh"
<code>contains("d")</code>	names containing "d"
<code>matches("^[a-d]")</code>	names matching regular expression "^[a-d]"
<code>num_range(x, 1:10)</code>	names following patten <code>x1</code> , <code>x2</code> , ..., <code>x10</code>
<code>all_of(vars)</code> / <code>any_of(vars)</code>	matches names stored in character vector <code>vars</code>
<code>last_col()</code>	further right column
<code>where(is.numeric)</code>	all variables where <code>is.numeric()</code> returns <code>TRUE</code>

# Aside: dplyr::relocate

Note that the function `relocate()` uses the same syntax as `select()` to move groups of columns at once.

Add variables separated by commas to move them **to the front**

```
starwars %>%  
  relocate(  
    ends_with("_color"), homeworld  
  ) %>%  
  head()
```

```
## # A tibble: 6 × 14
```

	hair_color	skin_color	eye_color	homeworld	name	height	mass	birth_year	species
	<chr>	<chr>	<chr>	<chr>	<chr>	<int>	<dbl>	<dbl>	<chr>
## 1	blond	fair	blue	Tatooine	Luke...	172	77	19	Human
## 2	<NA>	gold	yellow	Tatooine	C-3PO	167	75	112	Protocol Droid
## 3	<NA>	white, bl...	red	Naboo	R2-D2	96	32	33	Astromech Droid
## 4	none	white	yellow	Tatooine	Dart...	202	136	41.9	Human
## 5	brown	light	brown	Alderaan	Leia...	150	49	19	Human
## 6	brown, grey	light	blue	Tatooine	Owen...	178	120	52	Human

# Aside: dplyr::relocate

Can also use arguments `.after` / `.before` to place the column(s) in specific locations

```
starwars %>%  
  relocate(  
    species,  
    .before = height  
  ) %>%  
  head()
```

```
## # A tibble: 6 × 14  
##   name      species height  mass hair_color skin_color eye_color birth_year s  
##   <chr>    <chr>    <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr>  
## 1 Luke Sk... Human      172    77 blond      fair        blue        19    m  
## 2 C-3PO     Droid      167    75 <NA>      gold        yellow      112    r  
## 3 R2-D2     Droid       96    32 <NA>      white, bl... red         33    r  
## 4 Darth V... Human      202   136 none       white       yellow      41.9 m  
## 5 Leia Or... Human      150    49 brown     light      brown       19    f  
## 6 Owen La... Human      178   120 brown, gr... light      blue        52    m  
## # i 5 more variables: gender <chr>, homeworld <chr>, films <list>,  
38 / 85
```

## 4) dplyr::mutate

You can use `mutate()` to **create new columns** from scratch, or (more commonly) **transform existing columns**.

```
starwars %>%  
  select(name, birth_year) %>%  
  mutate(dog_years = birth_year * 7) %>%  
  head()
```

```
## # A tibble: 6 × 3  
##   name          birth_year dog_years  
##   <chr>         <dbl>     <dbl>  
## 1 Luke Skywalker      19        133  
## 2 C-3PO              112        784  
## 3 R2-D2               33        231  
## 4 Darth Vader        41.9       293.  
## 5 Leia Organa        19        133  
## 6 Owen Lars          52        364
```

## 4) dplyr::mutate

**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.")  
  ) %>% 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       293.3 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.
```



## 4) dplyr::mutate

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
```

```
## # 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

Lastly, combining `mutate` with the recent `across` feature<sup>3</sup> allows you to easily work on a **subset of variables**:

```
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
```

<sup>3</sup> This workflow (i.e. combining `mutate` and `across`) supersedes the old "scoped" variants of `mutate` that you might have used previously

## 5) dplyr::summarise

`summarise()` lets us manually specify summary statistics. It's particularly useful in combination with the `group_by` command.

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

```
## # A tibble: 42 × 3  
## # Groups:   species [38]  
##   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  
## 7 Droid     masculine    140  
## 8 Dug       masculine    112  
## 9 Ewok      masculine     88
```

## 5) dplyr::summarise

Note that including `na.rm = TRUE` (or `na.rm = T`) is usually a good idea, otherwise, missing values will result in `NA`

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

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

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

## 5) dplyr::summarise

Note that including `na.rm = TRUE` (or `na.rm = T`) is usually a good idea, otherwise, missing values will result in `NA`

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

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

## 5) dplyr::summarise

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, na.rm=T)) %>%  
  head(5)
```

```
## # A tibble: 5 × 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
```

# Other dplyr Goodies:

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

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

```
starwars %>%  
  group_by(species) %>%  
  mutate(species_mass = mean(mass, na.rm = T),  
         species_mass_diff = mass - species_mass) %>%  
  select(name, starts_with("species")) %>%  
  ungroup() %>% head()
```

```
## # A tibble: 6 × 4  
##   name          species species_mass species_mass_diff  
##   <chr>         <chr>         <dbl>         <dbl>  
## 1 Luke Skywalker Human           82.8          -5.78  
## 2 C-3PO         Droid           69.8           5.25  
## 3 R2-D2         Droid           69.8          -37.8  
## 4 Darth Vader   Human           82.8           53.2  
## 5 Leia Organa   Human           82.8          -33.8
```

# Other dplyr Goodies: slice

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

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

```
## # A tibble: 2 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex    ge  
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <c  
## 1 Luke Sky...   172    77 blond      fair       blue        19 male  ma  
## 2 Leia Org...   150    49 brown      light     brown        19 fema... fe  
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,  
## #   vehicles <list>, starships <list>
```



# Other dplyr Goodies: pull

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

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

```
## integer(0)
```

# Other dplyr Goodies: count and distinct

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

```
starwars %>% count(species)
```

```
## # A tibble: 38 × 2
##   species      n
##   <chr>    <int>
## 1 Aleena        1
## 2 Besalisk       1
## 3 Cerean         1
## 4 Chagrian       1
## 5 Clawdite       1
## 6 Droid          6
## 7 Dug            1
## 8 Ewok            1
## 9 Geonosian      1
## 10 Gungan        3
## # i 28 more rows
```

```
starwars %>% distinct(species)
```

# Other dplyr Goodies: window functions

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.

# 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)`

(You can find some helpful visual depictions of the different join operations **here**.)

# Joins

For our join examples, we'll use some data sets that come bundled with the **nycflights13** package.

- Load it now and then inspect these data frames<sup>4</sup> in your own console.

<sup>4</sup> These datasets are technically stored as tibbles, which are an **opinionated, modern version of data frames**. For our uses we can treat them essentially interchangeably, or forcibly go between types with `as.data.frame()/as.tibble()`

# Joins: Example Datasets

The `flights` dataset contains information on all flights that departed NYC in 2013:

```
head(flights)
```

```
## # A tibble: 6 × 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
## 1  2013     1     1     517             515           2     830             81
## 2  2013     1     1     533             529           4     850             83
## 3  2013     1     1     542             540           2     923             85
## 4  2013     1     1     544             545          -1    1004            102
## 5  2013     1     1     554             600          -6     812             83
## 6  2013     1     1     554             558          -4     740             72
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

# Joins: Example Datasets

The `planes` dataset contains metadata for all plane tailnumbers within the FAA aircraft registry

```
head(planes)
```

```
## # A tibble: 6 × 9
##   tailnum  year type      manufacturer model engines seats speed en
##   <chr>    <int> <chr>      <chr>          <chr>   <int> <int> <int> <chr>
## 1 N10156   2004 Fixed wing multi ... EMBRAER      EMB-...     2    55    NA Tu
## 2 N102UW   1998 Fixed wing multi ... AIRBUS INDU... A320...     2   182    NA Tu
## 3 N103US   1999 Fixed wing multi ... AIRBUS INDU... A320...     2   182    NA Tu
## 4 N104UW   1999 Fixed wing multi ... AIRBUS INDU... A320...     2   182    NA Tu
## 5 N10575   2002 Fixed wing multi ... EMBRAER      EMB-...     2    55    NA Tu
## 6 N105UW   1999 Fixed wing multi ... AIRBUS INDU... A320...     2   182    NA Tu
```

# Joins

Let's perform a **left join** to bring variables from the planes dataset into the flights dataset.

- `left_join(df1, df2)` keeps all rows of `df1`, adds variables from `df2`
- **Note:** I'm subsetting columns, but only for the sake of slide legibility

```
left_join(flights, planes) %>%  
  select(year:dep_time, carrier, flight, tailnum, type, model, engine)
```

```
## # A tibble: 336,776 × 10
```

```
##   year month   day dep_time carrier flight tailnum type  model engine  
##   <int> <int> <int>   <int> <chr>   <int> <chr>   <chr> <chr> <chr>  
## 1  2013     1     1     517 UA      1545 N14228 <NA> <NA> <NA>  
## 2  2013     1     1     533 UA      1714 N24211 <NA> <NA> <NA>  
## 3  2013     1     1     542 AA      1141 N619AA <NA> <NA> <NA>  
## 4  2013     1     1     544 B6        725 N804JB <NA> <NA> <NA>  
## 5  2013     1     1     554 DL        461 N668DN <NA> <NA> <NA>  
## 6  2013     1     1     554 UA      1696 N39463 <NA> <NA> <NA>  
## 7  2013     1     1     555 B6        507 N516JB <NA> <NA> <NA>  
## 8  2013     1     1     557 EV      5708 N820AS <NA> <NA> <NA>
```



# Joins

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`.

# Joins: by

**Solution:** state explicitly which variables to join on by using the `by` argument.

- You can also rename any ambiguous columns to avoid confusion

```
left_join(
  flights,
  planes %>% rename(year_built = year),
  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 × 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
## 2  2013     1     1     533     850 UA      1714 N24211     1998 Fixed
## 3  2013     1     1     542     923 AA      1141 N619AA     1990 Fixed
## # i 1 more variable: model <chr>
```

# Joins: Name Conflicts

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)
  head(3)
```

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

Make sure you know what "year.x" and "year.y" are!

# Joining on Multiple Columns

Often we need to join on **multiple variables** (i.e. unit and time for panels).

Two main ways to use `by` when merging on multiple columns:

1. Rename matching columns before merging to have the same names
2. Specify columns with different names to match on with

```
by = c("yvar1" = "xvar1", "yvar2" = "xvar2", ... )
```

# Joining on Multiple Columns

To see these, let's get info from the `weather` dataset:

```
weather_sub <- select(weather, year, month, day, hour, temp, humid, start:
```

This dataset contains info on the temperature, humidity, and wind conditions at each hour of the day in NYC during 2013 - useful information for understanding reasons for flight delays!

# Joining on Multiple Columns

Suppose we want to have an approximation of the weather conditions before each flight. Since weather is only to the nearest hour, let's round flight departure to the closest hour<sup>5</sup>:

```
flights <- mutate(flights,  
  dep_hr = case_when(  
    nchar(dep_time) == 3 ~ substr(dep_time,1,1) %>% as.numeric()  
    nchar(dep_time) == 4 ~ substr(dep_time,1,2) %>% as.numeric()  
    TRUE ~ as.numeric(NA)  
  )  
)
```

<sup>5</sup> I'm doing this to get to a starting point of different variable names - in reality we could just jump straight to merging on the same names here.

# Joins: by (Renaming First)

Here we want to join on time (year, month, and departure hour).

We could begin by renaming `hour` in the weather dataset to match the flights data:

```
left_join(  
  flights,  
  weather_sub %>% rename(dep_hr = hour), ## Rename to match  
  by = c("year", "month", "day", "dep_hr") ## Specify join columns  
) %>%  
  select(year, month, day, dep_hr, flight, temp, humid) %>%  
  head(3) ## Just to save vertical space on the slide
```

```
## # A tibble: 3 × 7  
##   year month   day dep_hr flight  temp humid  
##   <int> <int> <int>   <dbl>   <int> <dbl> <dbl>  
## 1  2013     1     1       5    1545  39.0  64.4  
## 2  2013     1     1       5    1545  39.0  61.6  
## 3  2013     1     1       5    1545  39.9  54.8
```

# Joins: by (Merging on Different Names)

Alternatively, we could perform the same join without renaming (R will keep the X data's variable name for any naming differences)

```
left_join(
  flights,
  weather_sub,
  by = c("year" = "year", "month" = "month", "day" = "day", "dep_hr" = "hr") %>%
  select(year, month, day, dep_hr, flight, temp, humid) %>%
  head(3) ## Just to save vertical space on the slide
```

```
## # A tibble: 3 × 7
##   year month   day dep_hr flight  temp humid
##   <int> <int> <int>   <dbl>   <int> <dbl> <dbl>
## 1  2013     1     1       5    1545  39.0  64.4
## 2  2013     1     1       5    1545  39.0  61.6
## 3  2013     1     1       5    1545  39.9  54.8
```



# Mutating Joins

**left joins** are probably the most common join we'll do, but we can perform a wide range of **mutating joins** with other join functions:

Join Function	Description
<code>left_join(df1, df2)</code>	Add variables from <code>df2</code> into <code>df1</code> (keep all rows of <code>df1</code> )
<code>right_join(df1, df2)</code>	Add variables from <code>df1</code> into <code>df2</code> (keep all rows of <code>df2</code> )
<code>full_join(df1, df2)</code>	Combine <code>df1</code> and <code>df2</code> (keep all rows of <code>df1</code> and <code>df2</code> )
<code>inner_join(df1, df2)</code>	Keep only observations from <code>df1</code> with matches in <code>df2</code>
<code>semi_join(df1, df2)</code>	Combine <code>df1</code> and <code>df2</code> (keep all rows of <code>df1</code> and <code>df2</code> )

# Filtering Joins

We can also perform **filtering joins** to restrict samples based on matches/non-matches across datasets:

Join Function	Description
<code>semi_join(df1, df2)</code>	return all rows of <code>df1</code> with a match in <code>df2</code>
<code>anti_join(df1, df2)</code>	return all rows of <code>df1</code> <b>without</b> a match in <code>df2</code>

tidyr

# Key tidyr verbs

1. `pivot_longer`: Pivot wide data into long format (i.e. "melt").<sup>6</sup>
2. `pivot_wider`: Pivot long data into wide format (i.e. "cast").<sup>7</sup>
3. `separate_wider_delim/separate_longer_delim`: Separate (i.e. split) one column into multiple columns/multiple rows.
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?

<sup>6</sup> Updated version of `tidyr::gather`.

<sup>7</sup> Updated version of `tidyr::spread`.

# 1) tidyr::pivot\_longer

Use `pivot_longer()` to go **from wide to long**<sup>8</sup>:

```
stocks <- data.frame( ## Could use "tibble" instead of "data.frame" if you
  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  0.2827313 -2.5029815  3.204252
## 2 2009-01-02 -0.3161512 -0.8794891 -4.675430
```

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

```
## # A tibble: 6 × 3
##   time      stock price
##   <date>   <chr>   <dbl>
```

<sup>8</sup>Note that both pivot functions have a lot of handy options for modifying names.

```
## 1 2009-01-01 X      0.283
```

# 1) tidyr::pivot\_longer

We could also manually specify the columns to pivot (useful when we want to pivot on just a subset of columns)

```
stocks %>% pivot_longer(cols = c(X, Y, Z), names_to="stock", values_to="p:
```

```
## # A tibble: 6 × 3
##   time      stock price
##   <date>    <chr> <dbl>
## 1 2009-01-01 X      0.283
## 2 2009-01-01 Y     -2.50
## 3 2009-01-01 Z      3.20
## 4 2009-01-02 X     -0.316
## 5 2009-01-02 Y     -0.879
## 6 2009-01-02 Z     -4.68
```

# 1) tidyr::pivot\_longer

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 ← pivot_longer(stocks, -time, names_to="stock", values_to="|
```

## 2) tidyr::pivot\_wider

Use `pivot_wider()` to go **from long to wide**:

```
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  0.283 -2.50  3.20
## 2 2009-01-02 -0.316 -0.879 -4.68
```

```
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          0.283      -0.316
## 2 Y         -2.50      -0.879
## 3 Z          3.20      -4.68
```

Note the second ex. 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) Separate

tidyr has several `separate_direction_method` functions that make it easy to separate cells in a column into multiple columns/rows, where

- `direction` informs whether the data spread
  - `wide` (`_wider_`) or
  - expand each cell into multiple rows (`_longer_`)
- `method` instructs the way to split a cell:
  - `delim` to split on a delimiter (i.e. "." or "/")
  - `position` to split at fixed widths
  - `regex` to split with a regular expression (i.e. `a(? ≤ d)` )

### 3) Separate

Let's try splitting some economists' names.

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

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

### 3) tidyr::separate\_wider\_delim

To split names into two columns by splitting at the period, we can use

`separate_wider_delim`:

```
economists %>% separate_wider_delim(name, # column(s) to separate
                                   delim = ".", # delimiter to split on
                                   names = c("first_name", "last_name")) # name of new vari.
```

```
## # A tibble: 3 × 2
##   first_name last_name
##   <chr>      <chr>
## 1 Adam      Smith
## 2 Paul      Samuelson
## 3 Milton    Friedman
```

### 3) tidyr::separate\_wider\_regex

If you know regular expressions, you can use `separate_wider_regex` to accomplish the same task:

```
economists %>%  
  separate_wider_regex(name,  
    patterns = c(first_name = "[:alpha:]+", ".", last_name = "[:alp  
  
## # A tibble: 3 × 2  
##   first_name last_name  
##   <chr>      <chr>  
## 1 Adam      Smith  
## 2 Paul      Samuelson  
## 3 Milton    Friedman
```

### 3) tidyr::separate\_longer\_delim

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

Let's see its use with some occupation 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
```

### 3) tidyr::separate\_longer\_delim

We can expand the data to have one row for each name and occupation combination:

```
## Now split out Jill's various occupations into different rows  
jobs %>% separate_longer_delim(occupation, delim = ", ")
```

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

## 4) tidyr::unite

`unite()` allows us to **collapse multiple columns into a single column**

Suppose we have daily small business revenues:

```
rev <- data.frame(
  year = rep(2016, times = 4),
  month = rep(1, times = 4),
  day = 1:4,
  revenue = rnorm(4, mean = 100, sd = 10)
)
rev
```

```
##   year month day  revenue
## 1 2016     1   1 100.90731
## 2 2016     1   2  88.95349
## 3 2016     1   3  96.81105
## 4 2016     1   4 105.00220
```



## 4) tidyr::unite

We can use `unite` to combine the three date components into a single character column<sup>9</sup>:

```
## Combine "yr", "mnth", and "dy" into one "date" column
rev_u <- rev %>% unite(col = date, # name of new column
                      c("month", "day", "year", ), # columns to unite
                      sep = "-") # separator to use
```

```
rev_u
```

```
##      date    revenue
## 1 1-1-2016 100.90731
## 2 1-2-2016  88.95349
## 3 1-3-2016  96.81105
## 4 1-4-2016 105.00220
```

<sup>9</sup> Set the argument `remove = T` to keep the original input columns

## 4) tidyr::unite

If we want to convert the new character column to another type (e.g. date or numeric) then you will need to modify it using `mutate`.

For example, we can use the **lubridate** package's super helpful date conversion functions to convert our new variable to a date:

```
pacman::p_load(lubridate)
rev_u <- mutate(rev_u,
                date = mdy(date))
class(rev_u$date)
```

```
## [1] "Date"
```

# Other tidyr goodies

Use `crossing` to get the full combination of a group of variables.<sup>10</sup>

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

```
## # A tibble: 9 × 2  
##   side    height  
##   <chr>  <chr>  
## 1 center bottom  
## 2 center middle  
## 3 center top  
## 4 left   bottom  
## 5 left   middle  
## 6 left   top  
## 7 right  bottom  
## 8 right  middle  
## 9 right  top
```

<sup>10</sup> Base R alternative: `expand.grid`.

# Other tidyr goodies

Use `crossing` to get the full combination of a group of variables.<sup>11</sup>

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

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

<sup>11</sup> Base R alternative: `expand.grid`.

# Table of Contents

1. Prologue
2. Tidyverse Overview
3. Pipes
4. Data Wrangling with dplyr
  - filter, arrange, select, mutate, summarise, joins
5. Data Tidying with tidyr
  - pivot\_longer / pivot\_wider
  - separate
  - unite