

Lab 02: Data and Reproducibility

Maghfira Ramadhani

! Due date

This lab is due on **Monday, September 22 at 11:59pm**. To be considered on time, the following must be done by the due date:

- Final .pdf file submitted on Gradescope

Introduction

The main goal is to learn data processing using tidyverse and introduce you to version control using Github.

Learning goals

By the end of the lab, you will learn:

1. Tidyverse basics
2. Data wrangling with dplyr
3. Data tidying with tidyr

The tidyverse

A whole “universe” of functions within R

- The most powerful, intuitive, and popular approach to data cleaning, wrangling, and visualization in R

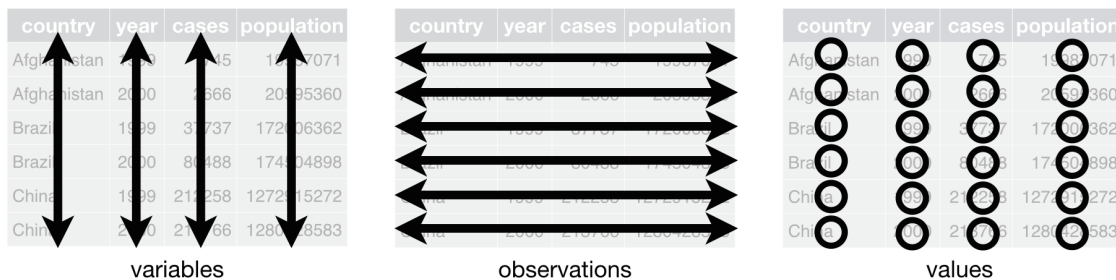
Advantages:

- Consistent philosophy and syntax
- “Verb” based approach makes it more familiar to users of Stata/SAS/SPSS
- Serves as the front-end for many other big data and ML tools

Tidying Data

The two most important properties of tidy data are:

1. Each column is a unique variable.
2. Each row is a single observation.



[Image is from “[R for Data Science](#)” by Hadley Wickham & Garrett Grolemund, used under [CC BY-NC-ND 3.0](#)]

Tidy data is easier to work with, because you have a consistent way of referring to variables and observations. It then becomes easy to manipulate, visualize, and model.

Wide vs. Long Formats

Both of these data sets display information on heart rate observed in individuals across 3 different time periods:

```

      name time1 time2 time3
1  Wilbur    67    56    70
2 Petunia    80    90    67
3 Gregory    64    50   101

```

```

      name time heartrate
1  Wilbur     1         67
2 Petunia     1         80
3 Gregory     1         64

```

4	Wilbur	2	56
5	Petunia	2	90
6	Gregory	2	50
7	Wilbur	3	70
8	Petunia	3	67
9	Gregory	3	10

Which dataframe is in *tidy* format?

Wide data:

- Row = patient. Columns = repeated observations over time.
- Often easier to take in at a glance (as in a spreadsheet).

Long data:

- Row = one observation. Columns = ID variables + observed variable.
- Usually easier to clean, merge with other data, and avoid errors.

Tidy data is more likely to be **long**.

- Most R packages have been written assuming your data is in long format.

“Tidy datasets are all alike but every messy dataset is messy in its own way.”

– Hadley Wickham

Tidyverse packages

We need to install and load a couple of packages. Run these preliminaries:

```
# load and install package if necessary
if (!require("pacman")) install.packages("pacman")
pacman::p_load(
  tidyverse,
  nycflights13
)
```

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](#).

The tidyverse actually comes with a lot more packages than those that are just 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"
```

All of these are super useful

- **lubridate** helps us work with dates
- **rvest** is for webscraping

This lab will focus on two that are automatically loaded: **dplyr** and **tidyr**.

Pipes: `|>` or `%>%`

Pipes take the **output** of one function and feed it into the **first argument** of the next (which you then skip).

`dataframe |> filter(condition)` is equivalent to `filter(dataframe, condition)`.

Note: `|>` on these slides is generated by the two characters `| >`, without the space.

Older version of the pipe: `%>%` * From the **magrittr** package loaded with the tidyverse *
Works identically to `|>` in most situations.

Keyboard shortcut: Ctl/Cmd + Shift + M

- Have to turn on a setting in RStudio options to make `|>` the default

Pipes 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) |>
  summarize(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
```

```
summarize(group_by(filter(mpg, manufacturer=="audi"), model), 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
```

The first line reads from left to right, exactly how you think about the operations.

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

Best practice is to put each function on its own line and indent. Look how much more readable this is:

```
mpg |>
  filter(manufacturer == "audi") |>
  group_by(model) |>
  summarize(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
```

Vertical space costs nothing and makes for much more readable/writable code than cramming things horizontally.

All together, this multi-line line of code is called a **pipeline**.

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. **summarize**: Collapse multiple rows into a single summary value.

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

Exercise 1: dplyr::filter

Subset Observations (Rows)



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 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>
```

Regular expressions work well too.

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

```
# A tibble: 3 x 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 Sky~    172    77 blond      fair        blue         19  male  mascu~
2 Anakin S~    188    84 blond      fair        blue        41.9  male  mascu~
3 Shmi Sky~    163    NA black      fair        brown         72  fema~ femin~
# i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>
```

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

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

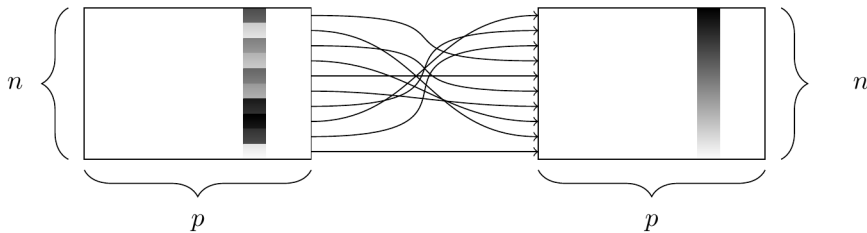
```
# A tibble: 6 x 14
  name      height  mass hair_color skin_color eye_color birth_year sex  gender
  <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
1 Arvel Cr~    NA    NA brown      fair        brown         NA  male  mascu~
2 Finn        NA    NA black      dark        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 none       none        unknown       NA  fema~ femin~
# 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.

```
clean_starwars<-starwars |>
  filter(is.na(height))
clean_starwars
```

```
# A tibble: 6 x 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 none      none        unknown       NA fema~ femin~
# i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>
```

Exercise 2: dplyr::arrange



`arrange` sorts your data frame by a particular column (numerically, or alphabetically)

```
starwars |>
  arrange(birth_year)
```

```
# A tibble: 87 x 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>
```

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

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

```
# A tibble: 87 x 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>
```

Exercise 3: dplyr::select

Subset Variables (Columns)



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
# i 77 more rows
```

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

```
starwars |>
  select(alias=name, planet=homeworld)
```

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

If you just want to rename columns without subsetting them, you can use `rename`. Try this!

```
starwars_rename<-starwars |>
  rename(alias=name)
names(starwars_rename)
```

```
[1] "alias"      "height"     "mass"       "hair_color" "skin_color"
[6] "eye_color"  "birth_year" "sex"        "gender"     "homeworld"
[11] "species"    "films"      "vehicles"   "starships"
```

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
# i 77 more rows
```

Some other selection helpers: `starts_with()`, `ends_with()`, `all_of(c("name1", "name2"))`, `matches()`.

Exercise 4: dplyr::mutate

Make New Variables



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-3P0	112	784	C-3P0 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.
7	Beru Whitesun Lars	47	329	Beru Whitesun Lars is 329 in dog year~
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.

i 77 more rows

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 year~
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.
# i 77 more rows
```

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 = if_else(height > 180, "Tall", "Short"))
```

```
# 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
```

Lastly, combining `mutate` with `across` allows you to easily perform the same operation on a subset of variables.

```
starwars |>
  select(name:eye_color) |>
  mutate(across(where(is.character), toupper))
```

```
# A tibble: 87 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
6 OWEN LARS	178	120 BROWN, GREY	LIGHT	BLUE
7 BERU WHITESUN LARS	165	75 BROWN	LIGHT	BLUE
8 R5-D4	97	32 <NA>	WHITE, RED	RED
9 BIGGS DARKLIGHTER	183	84 BLACK	LIGHT	BROWN
10 OBI-WAN KENOBI	182	77 AUBURN, WHITE	FAIR	BLUE-GRAY

i 77 more rows

Exercise 5: dplyr::summarize

Summarise Data



Particularly useful in combination with the `group_by` command.

```
starwars |>
  group_by(species) |>
  summarize(mean_height = mean(height))
```

```
# A tibble: 38 x 2
  species    mean_height
  <chr>         <dbl>
1 Aleena         79
```

```

2 Besalisk      198
3 Cerean        198
4 Chagrian      196
5 Clawdite      168
6 Droid         NA
7 Dug           112
8 Ewok          88
9 Geonosian     183
10 Gungan       209.
# i 28 more rows

```

Notice that some of these summarized values are missing. If we want to ignore missing values, use `na.rm = T`:

```

## Much better
starwars |>
  group_by(species) |>
  summarize(mean_height = mean(height, na.rm = T))

```

```

# A tibble: 38 x 2
  species    mean_height
  <chr>         <dbl>
1 Aleena         79
2 Besalisk      198
3 Cerean        198
4 Chagrian      196
5 Clawdite      168
6 Droid        131.
7 Dug           112
8 Ewok          88
9 Geonosian     183
10 Gungan       209.
# i 28 more rows

```

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

```

starwars |>
  group_by(species) |>
  summarize(across(where(is.numeric), mean))

```

```
# A tibble: 38 x 4
  species    height  mass birth_year
  <chr>      <dbl> <dbl>      <dbl>
1 Aleena      79     15         NA
2 Besalisk   198    102         NA
3 Cerean     198     82         92
4 Chagrian   196     NA         NA
5 Clawdite   168     55         NA
6 Droid       NA     NA         NA
7 Dug       112     40         NA
8 Ewok       88     20          8
9 Geonosian  183     80         NA
10 Gungan   209.     NA         NA
# i 28 more rows
```

The same `across`-based workflow that we saw with `mutate` a few slides back also works with `summarize`. Though to add arguments, we have to use an **anonymous function**:

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

```
# A tibble: 38 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     NaN         NaN
5 Clawdite   168     55         NaN
6 Droid     131.    69.8     53.3
7 Dug       112     40         NaN
8 Ewok       88     20          8
9 Geonosian  183     80         NaN
10 Gungan   209.    74         52
# i 28 more rows
```


Other dplyr goodies

`ungroup`: For ungrouping after using `group_by`. - Use after doing your grouped `summarize` or `mutate` operation, or everything else you do will be super slow.

`slice`: Subset rows by position rather than filtering by values. - E.g. `starwars |> slice(1:10)`

`pull`: Extract a column from a data frame as a vector or scalar. - E.g. `starwars |> filter(sex=="female") |> pull(height)`

`distinct` and `count`: List unique values, with or without their number of appearances. - E.g. `starwars |> distinct(species)`, or `starwars |> count(species)` - `count` is equivalent to `group_by` and `summarize` with `n()`:

```
starwars |> group_by(species) |> summarize(n = n())
```

```
# A tibble: 38 x 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
```

Challenge 1

List the most common eye colors among female Star Wars characters in descending order of frequency.

As usual, there are multiple solutions.

```
starwars |>
  filter(sex == "female") |>
  count(eye_color) |>
  arrange(desc(n))
```

```
# A tibble: 6 x 2
  eye_color      n
  <chr>        <int>
1 blue          6
2 brown         4
3 black         2
4 hazel         2
5 unknown       1
6 yellow        1
```

```
starwars |>
  filter(sex == "female") |>
  group_by(eye_color) |>
  summarize(n = n()) |>
  arrange(desc(n))
```

```
# A tibble: 6 x 2
  eye_color      n
  <chr>        <int>
1 blue          6
2 brown         4
3 black         2
4 hazel         2
5 unknown       1
6 yellow        1
```

Explain what each line in the codes do:

Answer:

1. Line one takes the output of the starwars function and feeds it into the first argument of this code. Basically, line one starts the code.
2. Line two (“filter”) organizes the rows based on their values, so it organizes the rows to only include females.
3. Line three (“count”) lists the eye colors with their number of appearances, so it tells the amount of females with each eye color.

4. Line four (“arrange”) sorts the data table into columns and orders the rows based on their values.
5. Line five does the same thing as line one, feeding the output of the `starwars` function and feeding it directly into the next function.
6. Line six (“group_by”) organizes the data into groups by eye color.
7. Line seven (“summarize”) collapses the multiple rows into a single row containing a summary of the data.
8. Line eight (“arrange”) does the same thing as line four, which is arranging the rows into columns based on their values.

Storing results in memory

So far we haven’t been saving the dataframes that result from our code in memory. Usually, we will want to use them for the next task. Create a new object each time you write a pipeline.

```
women = starwars |> filter(sex == "female")
brown_eyed_women = women |> filter(eye_color == "brown")
```

Resist the temptation to use the same object name. This is called **clobbering** since it overwrites the previous version. It ruins your ability to easily go back to previous steps.

```
# DON'T do this
starwars = starwars |> filter(sex == "female")
```

By keeping multiple copies of very similar dataframes, will you waste your computer’s memory? Usually, no – R is smart and stores only the changes between objects.

Key tidy verbs

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

Which of `pivot_longer` vs `pivot_wider` produces “tidy” data?

Exercise 6: 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, 10, 1),
  Y = rnorm(2, 10, 2),
  Z = rnorm(2, 10, 5)
)
stocks
```

	time	X	Y	Z
1	2009-01-01	10.459809	11.46898	8.490222
2	2009-01-02	9.461827	13.87764	1.900462

```
tidy_stocks = stocks |>
  pivot_longer(cols=X:Z, names_to="stock", values_to="price")
tidy_stocks
```

```
# A tibble: 6 x 3
  time      stock price
  <date>    <chr> <dbl>
1 2009-01-01 X      10.5
2 2009-01-01 Y      11.5
3 2009-01-01 Z       8.49
4 2009-01-02 X       9.46
5 2009-01-02 Y      13.9
6 2009-01-02 Z       1.90
```

Exercise 7: tidyr::pivot_wider

Now we can use pivot_wider to go back to the original dataframe:

```
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 10.5  11.5  8.49
2 2009-01-02  9.46  13.9  1.90
```

Or, we can put it into a new (“transposed”) format, in which the observations are stocks and the columns are dates:

```
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          10.5         9.46
2 Y          11.5        13.9
3 Z           8.49         1.90
```

Exercise 8: tidyr::separate

`separate` helps when you have more than one value in a single column:

```
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 the `sep` argument:

```
economists |> separate(name, c("first_name", "last_name"), sep = "_")
```

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

Exercise 9: tidyr::separate

Related is `separate_rows`, for splitting cells with multiple values into multiple rows:

```
jobs = data.frame(  
  name = c("Joe", "Jill"),  
  occupation = c("President", "First Lady, Professor, Grandmother")  
)  
jobs
```

	name	occupation
1	Joe	President
2	Jill	First Lady, Professor, Grandmother

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

```
# A tibble: 5 x 2  
  name occupation  
  <chr> <chr>  
1 Joe   President  
2 Jill  First  
3 Jill  Lady  
4 Jill  Professor  
5 Jill  Grandmother
```

Related is `separate_rows`, for splitting cells with multiple values into multiple rows:

```
jobs = data.frame(  
  name = c("Joe", "Jill"),  
  occupation = c("President", "First Lady, Professor, Grandmother")  
)  
jobs
```

	name	occupation
1	Joe	President
2	Jill	First Lady, Professor, Grandmother

```
# Now split out Jill's various occupations into different rows  
jobs |> separate_rows(occupation, sep = ", ")
```

```
# A tibble: 4 x 2
  name occupation
  <chr> <chr>
1 Joe   President
2 Jill  First Lady
3 Jill  Professor
4 Jill  Grandmother
```

Exercise 10: 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	102.66463
2	2016	1	2	99.25005
3	2016	1	3	99.03425
4	2016	1	4	102.81068

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

	date	gdp
1	2016-1-1	102.66463
2	2016-1-2	99.25005
3	2016-1-3	99.03425
4	2016-1-4	102.81068

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

Note that `unite` will automatically create a character variable.

If you want to convert it to something else (e.g. date or numeric) then you will need to modify it using `mutate`. This example uses the [lubridate](#) package's super helpful date conversion functions.

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

	date	gdp
1	2016-01-01	102.66463
2	2016-01-02	99.25005
3	2016-01-03	99.03425
4	2016-01-04	102.81068

Challenge 2

Using `nycflights13`, create a table of average arrival delay (in minutes) by day (in rows) and carrier (in columns).

Hint: Recall that you can tabulate summary statistics using `group_by` and `summarize`:

```
flights |>
  group_by(carrier) |>
  summarize(avg_late = mean(arr_delay, na.rm=T))
```

```
# A tibble: 16 x 2
  carrier avg_late
  <chr>      <dbl>
1 9E         7.38
2 AA         0.364
3 AS        -9.93
4 B6         9.46
5 DL         1.64
6 EV        15.8
7 F9        21.9
8 FL        20.1
9 HA        -6.92
10 MQ        10.8
11 OO        11.9
12 UA         3.56
13 US         2.13
14 VX         1.76
15 WN         9.65
16 YV        15.6
```


Solution:

```
delay_long = flights |>
  group_by(carrier, day) |>
  summarize(avg_late = mean(arr_delay, na.rm=T))
delay_wide = delay_long |>
  pivot_wider(names_from=carrier, values_from=avg_late)
head(delay_wide, 4)
```

```
# A tibble: 4 x 17
  day `9E`   AA   AS   B6   DL   EV   F9   FL   HA   MQ   OO
<int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1     1  7.21 -1.23  -5.96 11.9    0.866 21.3  21.7  22.7 -15.4 12.9  NA
2     2  7.35 -0.905 -13.7  9.90    3.05 18.0   7.71  20.9 -16.1  9.04 NaN
3     3  5.80 -3.09 -20.8  5.26   -0.204 15.3  18.0  19.6 -16.8 13.7  0.5
4     4 -2.11 -5.80 -22.3 -0.0939 -6.24   3.87 14.5   4.38 -15.2  3.82 -12
# i 5 more variables: UA <dbl>, US <dbl>, VX <dbl>, WN <dbl>, YV <dbl>
```

Explain what each line in the codes do:

Answer:

1. Line one is a pipe, meaning it takes the output of the flights function and feeds it into the first argument of the next function.
2. Line two (“group_by”) organizes the data into separate groups by carrier type. Each group contains rows that share similar values.
3. Line three (“summarize”) collapses the multiple rows into a single row containing a summary of the data. Basically, it collapses the dataset into one row per group.
4. Line four is also a pipe, meaning it takes the output of the function and feeds it into the argument of the next function. So, it takes the data on flights and feeds it into the next function. “delay_long” indicates that the next function will be stored under the “delay_long” variable.
5. Line five (“group_by”) organizes the data into groups based on carrier and day. Because it has “|>”, it also includes a pipe, which will take the output of the function and feed it into the next function.
6. Line six (“summarize”) will collapse multiple rows into a single row containing a summary of the data. So, it will take the mean of the “arr_delay” column and create a new column with this mean.
7. Line seven is another pipe, so it will take the delay_long function and feed it into the argument of the next function. The result of the pipe will be renamed as “delay_wide.”

8. Line eight will use the values in the carrier column as column names and fill them with the values from the “avg_late” columns. In other words, it will change the dataset from long format to wide format.
9. This will display only the first four rows of the “delay_wide” function.

Create GitHub Account

Go to github.com and sign up to create an account. Report your GitHub username via this form:

[GitHub SignUp Report Form](#)

Submission

You will submit the PDF documents in to Gradescope as part of your final submission.

Warning

Remember – you must turn in a PDF file to the Gradescope page before the submission deadline for full credit.

Instructions to combine PDFs:

- Preview (Mac): support.apple.com/guide/preview/combine-pdfs-prvw43696/mac
- Adobe (Mac or PC): helpx.adobe.com/acrobat/using/merging-files-single-pdf.html

To submit your assignment:

- Access Gradescope
- Click on the assignment, and you’ll be prompted to submit it.
- Mark the pages associated with each exercise. All of the pages of your lab should be associated with at least one question (i.e., should be “checked”).
- Select the first page of your .PDF submission to be associated with the “*Workflow & formatting*” section.

Grading

Component	Points
Replicating Ex 1-10	85
Challenge	5
Creating Github Account	5
Workflow & formatting	5

The “Workflow & formatting” grade is to assess the reproducible workflow and document format.