Lab 5: Data Wrangling

Table of Contents

You can download the .Rmd file [here](lab-5.Rmd). You can download the .doc file [here](lab-5.docx).

# Purpose

The purpose of today’s lab is to introduce you to the tidyverse as a framework for working with data structures in R. We will mostly focus on data wrangling (particularly data transformation), including how to extract specific observations and variables, how to generate new variables and how to summarize data.

For further resources on these topics, check out [*R for Data Science*](https://r4ds.had.co.nz/) by Hadley Wickham and [this cheatsheet on data wrangling](https://github.com/rstudio/cheatsheets/blob/master/data-transformation.pdf) from RStudio.

To quickly navigate to the desired section, click one of the following links:

1. [Intro to the tidyverse](#tidy)
2. [Manipulating observations](#obs)
3. [Manipulating variables](#vars)
4. [Summarizing data](#summarize)
5. [Grouping Data](#group)

# Intro to the tidyverse

* The tidyverse, according to its creators, is [“an opionated collection of R packages designed for data science.”](https://www.tidyverse.org/) It’s a suite of packages designed with a consistent philosophy and aesthetic. This is nice because all of the packages are designed to work well together, providing a consistent framework to do many of the most common tasks in R, including, but not limited to…
  + data manipulation (dplyr) **= our focus today**
  + reshaping data (tidyr)
  + data visualization (ggplot2)
  + working with strings (stringr)
  + working with factors (forcats)

To load all the packages included in the tidyverse, use:

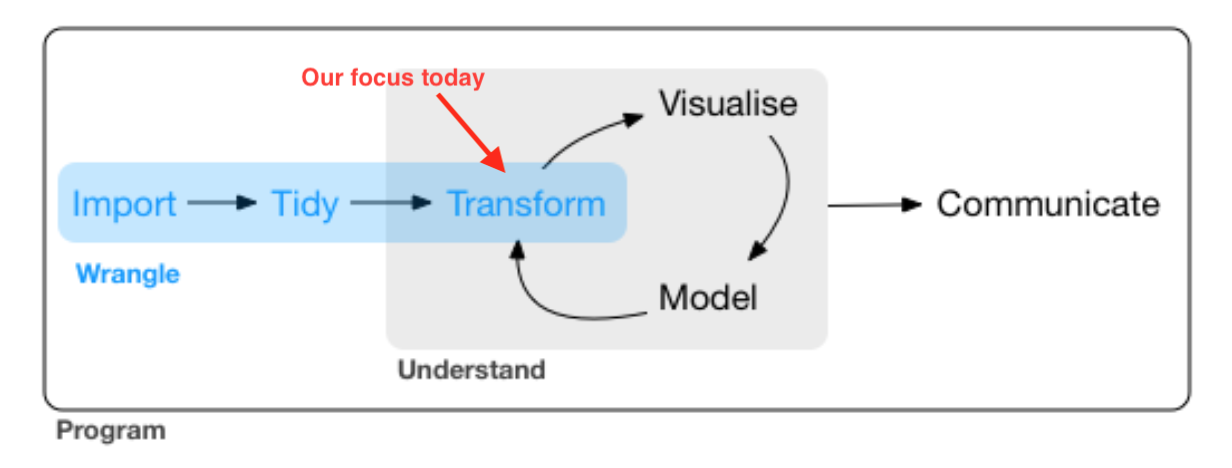
#install.packages("tidyverse")  
library(tidyverse)

* Three qualities of the tidyverse are worth mentioning at the outset:
  1. Packages are designed to be like *grammars* for their task, so we’ll be using functions that are named as *verbs* to discuss the tidyverse. The idea is that you can string these grammatical elements together to form more complex statements, just like with language.
  2. The first argument of (basically) every function we’ll review today is data (in the form of a data frame). This is very handy, especially when it comes to piping (discussed [below](#pipes)).
  3. Variable names are *usually* not quoted.

## What is data wrangling?

* Data wrangling, broadly speaking, means getting your data into a useful form for visualizing and modelling it. Hadley Wickham, who has developed a lot of the tidyverse, conceptualizes the main steps involved in data wrangling as follows:
  1. Importing your data (we covered this in [Week 1’s lab](https://uopsych.github.io/psy611/labs/lab-1.html#Importing_Data_into_R))
  2. Tidying your data (see brief overview below)
  3. Transforming your data (what we’ll cover today)

The figure below highlights the steps in data wrangling in relation to the broader scope of a typical data science workflow:



## What is tidy data?

* Data is considered “tidy” when:
  1. Each variable has its own column
  2. Each observation has its own row
  3. Each value has its own cell

The following figure from *R for Data Science* illustrates this visually.

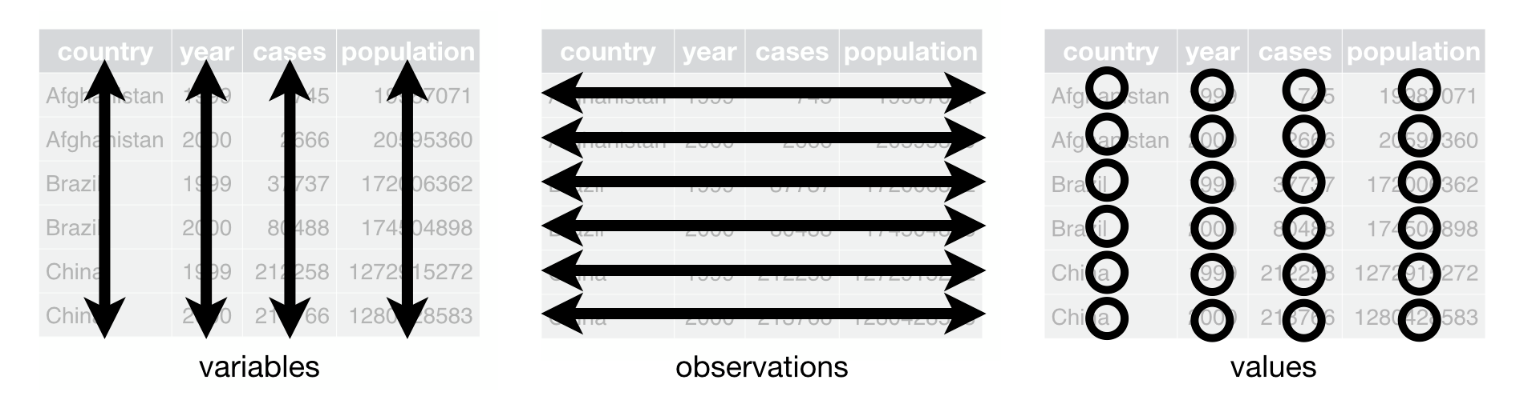


Illustration of tidy data

* If your data is not already in tidy format when you import it, you can use functions from the {tidyR} package, e.g. pivot\_longer() and pivot\_wider(), that allow you to “reshape” your data to get it into tidy format.
* However, this term we are mostly going to work with simpler datasets that are already tidy, so we are not going to focus on these functions today. These functions will become especially useful in the future when we work with repeated measures data that has multiple observations for each subject. If you are interested in learning more about reshaping your data with {tidyR}, check out [this chapter](https://r4ds.had.co.nz/tidy-data.html#introduction-6) from *R for Data Science*.

## Today’s focus: {dplyr}

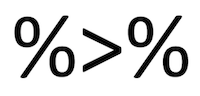
* Most of the functions we’ll go over today come from the {dplyr} package. Essentially, you can think of this package as a set of “pliers” that you can use to tweak data frames, hence its name (and hex sticker).



* {dplyr} is a “grammar” of data manipulation. As such, its functions are *verbs*:
  + mutate() adds new variables that are functions of existing variables
  + select() picks variables based on their names.
  + filter() picks cases based on their values.
  + summarize() reduces multiple values down to a single summary.
  + arrange() changes the ordering of the rows.
* Note that {dplyr} functions always take a data frame as the first argument and return a modified data frame back to you. The fact that you always get a data frame back is useful down the road when you are modelling and visualizing data.

## Pipes

* Pipes come from the {magrittr} package are available when you load the tidyverse. (Technically, the pipe is imported with {dplyr}.) Pipes are a way to write strings of functions more easily, creating *pipelines*. They are extremely powerful and useful. A pipe looks like this:



* You can enter a pipe with the shortcut CTRL+Shift+M for PC or CMD+Shift+M for Mac.

#practice entering a pipe with the shortcut here

* A pipe passes an object on the left-hand side as the first argument (or . argument) of whatever function is on the right-hand side.
  + x %>% f(y) is the same as f(x, y)
  + y %>% f(x, ., z) is the same as f(x, y, z )

Example: I want to calculate the mean of the mpg variable from the mtcars data set and round our answer to 2 decimal places. I can accomplish this by nesting:

round(mean(mtcars$mpg, na.rm = TRUE), 2)

Or, we could use pipes. Grammatically, you can think of a pipe as “then.” I have a variable, the mile per gallon of cars, THEN I want to take the mean of that variable, and THEN I want to round that answer to two decimal places.

mtcars$mpg %>% # select the `mpg` variable from the `mtcars` dataset  
 mean(na.rm = TRUE) %>% # calculate the mean  
 round(2) # round to 2 decimal places

Now, rewrite the following code using pipes.

round(sqrt(sum(mtcars$cyl)), 1)  
  
mtcars$cyl %>%   
 sum() %>%   
 sqrt() %>%   
 round(1)

mtcars$cyl %>%   
 sum() %>%   
 sqrt() %>%   
 round(1)

### Why use pipes?

1. Cleaner code
   * This is nice, because it helps make your code more readable by other humans (including your future self).

. Cleaner environment \* When you use pipes, you have basically no reason to save objects from intermediary steps in your data wrangling / analysis workflow, because you can just pass output from function to function without saving it. \* Finding objects you’re looking for is easier.

. Efficiency in writing code \* Naming objects is hard; piping means coming up with fewer names.

. More error-proof \* Because naming is hard, you might accidentally re-use a name and make an error.

## Example dataset

* Because you are already familiar with the World Happiness dataset, we will use this as a running example today (we’ll use the same version from Homework 1). You can import the data with the following code:

world\_happiness <- rio::import("https://raw.githubusercontent.com/uopsych/psy611/master/labs/resources/lab5/data/world\_happiness.csv")

### Clean names

* If we look at the names of the variables in world\_happiness, we’ll notice that all of the variable names are capitalized.

names(world\_happiness)

## [1] "Country" "Happiness" "GDP" "Support" "Life"   
## [6] "Freedom" "Generosity" "Corruption" "World"

* Personally, I find it annoying to have to remember to capitalize the first letter whenever I reference a variable name. The clean\_names() function from the {janitor} package will (by default) convert all variable names to snake\_case (but there are several other options…see [here](https://cran.r-project.org/web/packages/janitor/vignettes/janitor.html#clean-data.frame-names-with-clean_names) for more info).

#install.packages("janitor") # if not already installed  
library(janitor)

# clean variable names and re-save the data  
world\_happiness <- world\_happiness %>%   
 clean\_names()

Now all of our variable names are lower case.

names(world\_happiness)

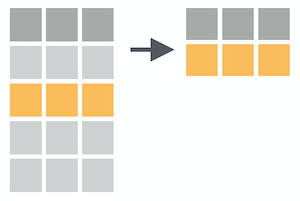
## [1] "country" "happiness" "gdp" "support" "life"   
## [6] "freedom" "generosity" "corruption" "world"

* **Note**: Remember to save your new data frame to an object of the same name as your old data frame if you want to overwrite the old one.

# Manipulating observations

## Extract rows with filter()

* The filter() function is used to subset observations based on their values. The result of filtering is a data frame with the same number of columns as before but fewer rows, as illustrated below…



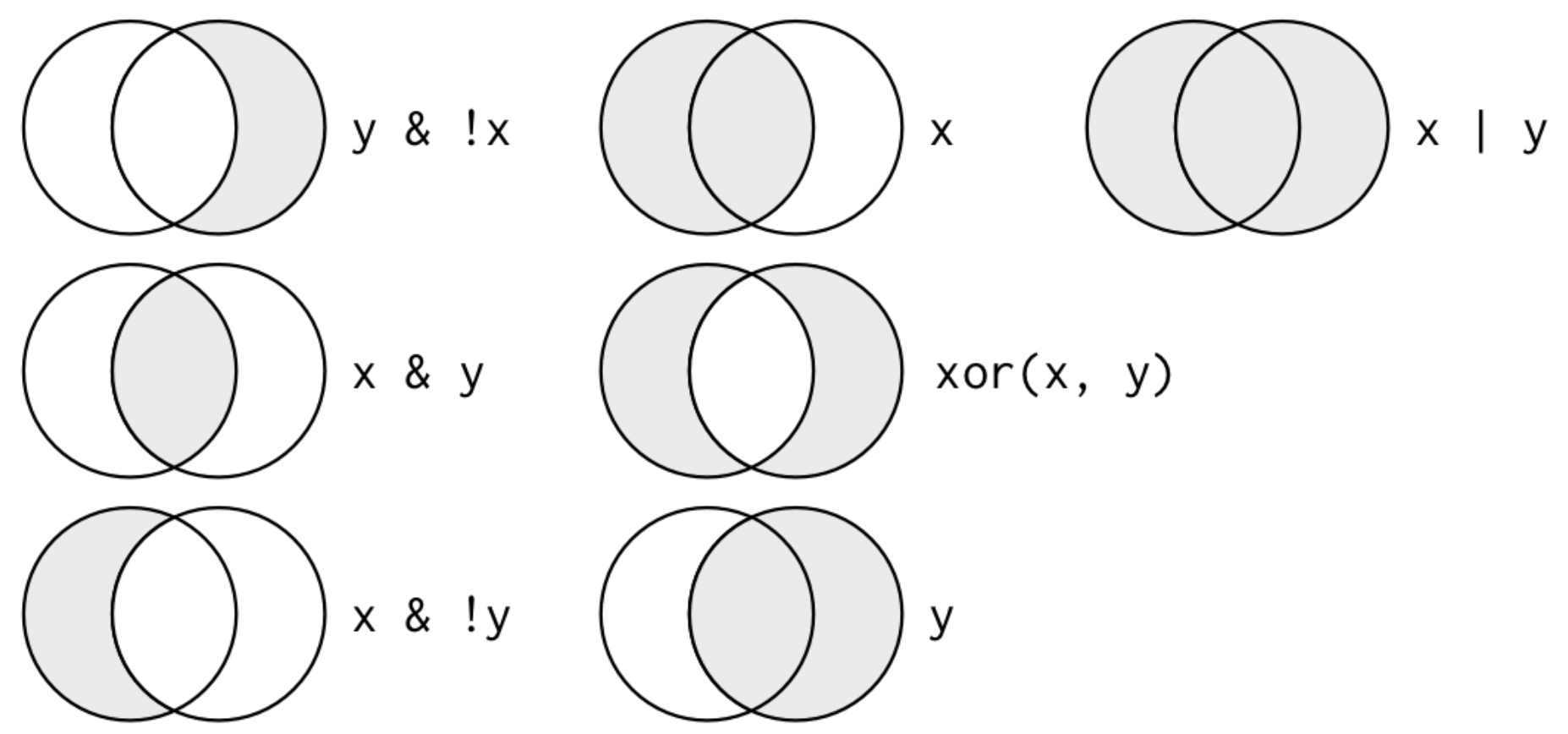
* The first argument is data and subsequent arguments are logical expressions that tell you which observations to retain in the data frame.

For example, we can filter rows to retain data only for the United States.

world\_happiness %>%   
 filter(country == "United States")

## Logical operators

* The == we just used is an example of a comparison operator that tests for equality. The other comparison operators available are :
  + > (greater than)
  + >= (greater than or equal to)
  + < (less than)
  + <= (less than or equal to)
  + != (not equal to)
* You can combine multiple arguments to filter() with Boolean operators. The figure below from [*R for Data Science*](https://r4ds.had.co.nz/transform.html#logical-operators) shows the complete set of Boolean operators.



* For example, let’s select observations for the United States, Mexico and Canada.

world\_happiness %>%   
 filter(country == "United States" | country == "Mexico" | country == "Canada")

* Since it is somewhat cumbersome to write country three times, we can use a special short-hand here with the %in% operator. Generally speaking, specifying x %in% y will select every row where x is one of the values in y.

So we could have written our filter statement like this:

world\_happiness %>%   
 filter(country %in% c("United States", "Mexico", "Canada"))  
  
country\_names = c("United States", "Mexico", "Canada")  
  
world\_happiness %>%   
 filter(country %in% country\_names)

### You try

* Filter for observations that are greater than the mean of happiness

# your code here

world\_happiness %>%   
 filter(happiness > mean(happiness, na.rm= TRUE))

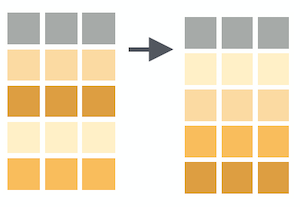
* Filter for observations that are greater than the mean of happiness but less than the mean of gdp

# your code here

world\_happiness %>%   
 filter(happiness > mean(happiness, na.rm= TRUE),   
 gdp < mean(gdp, na.rm = TRUE))

## Sort rows with arrange()

* The arrange() function keeps the same number of rows but changes the *order* of the rows in your data frame, as illustrated below…



* The first argument is data and subsequent arguments are name(s) of columns to order the rows by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

For example, let’s re-order observations by happiness. Note that rows are sorted in ascending order by default.

world\_happiness %>%   
 arrange(happiness) %>% # sorts in ascending order by default  
 head()  
  
world\_happiness %>%   
 arrange(desc(happiness)) %>%   
 head()# sort in descending order

# Manipulating variables

## Extract columns with select()

* The select() function subsets columns in your data frame. This is particularly useful when you have a data set with a huge number of variables and you want to narrow down to the variables that are relevant for your analysis.



* The first argument is data, followed by the name(s) of the column(s) you want to subset. Note that you can use variable positions rather than their names, but this is usually not as useful. Let’s go through some simple examples of common uses of select().
* Select one variable

world\_happiness %>%   
 select(country)

* Select multiple variables

world\_happiness %>%   
 select(country, freedom, corruption)

* Select a range of variables

world\_happiness %>%   
 select(country:support)  
names(world\_happiness)

* Rearrange the order of variables
  + Note: everything() is a helper function that gives us all the remaining variables in the data frame (see more on [helper functions](#helper) below)

world\_happiness %>%   
 select(country, world, everything())

* De-select variables with a minus sign (-)

world\_happiness %>%   
 select(-happiness) %>%   
 names()

* De-select range of variables

world\_happiness %>%   
 select(-(gdp:world)) %>%   
 names()

### You try

* Produce a data frame of the variables country, gdp, and happiness for countries whose gdp is greater than average.

# your code here

world\_happiness %>%   
 janitor::clean\_names() %>%   
 select(country, gdp, happiness) %>%   
 filter(gdp>mean(gdp, na.rm = T)) %>%   
 head()  
  
#mean(as.numeric(world\_happiness$GDP), na.rm = TRUE)

* Produce a data frame that has only countries coded as 1 that have greater than average levels of freedom. Arrange the rows by freedom scores in descending order, and only display the country and freedom variables (in that order). How many observations are you left with?

# your code here

world\_happiness %>%   
 filter(world == 1 & freedom > mean(freedom, na.rm = TRUE)) %>%  
 arrange(desc(freedom)) %>%   
 select(country, freedom)

### Helper functions for select()

There are some “helper” functions that you can use along with select() that can sometimes be more efficient than selecting your variables explicitly by name.

|  |  |
| --- | --- |
| function | what it does |
| starts\_with() | selects columns starting with a string |
| ends\_with() | selects columns that end with a string |
| contains() | selects columns that contain a string |
| matches() | selects columns that match a regular expression |
| num\_ranges() | selects columns that match a numerical range |
| one\_of() | selects columns whose names match entries in a character vector |
| everything() | selects all columns |
| last\_col() | selects last column; can include an offset. |

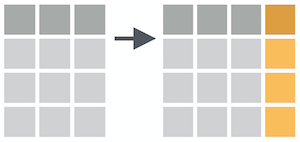
Quick example:

world\_happiness %>%   
 select(starts\_with("c"))

## country corruption  
## 1 Albania 0.88479304  
## 2 Argentina 0.85090619  
## 3 Armenia 0.90146220  
## 4 Australia 0.35655439  
## 5 Austria 0.55747962  
## 6 Azerbaijan 0.61555255  
## 7 Bahrain NA  
## 8 Bangladesh 0.72060090  
## 9 Belarus 0.66867816  
## 10 Belgium 0.46878463  
## 11 Benin 0.85009819  
## 12 Bhutan 0.63395578  
## 13 Bolivia 0.86237395  
## 14 Bosnia and Herzegovina 0.95985365  
## 15 Botswana 0.86029297  
## 16 Brazil 0.77133906  
## 17 Burkina Faso 0.69272399  
## 18 Cambodia 0.82513022  
## 19 Cameroon 0.86804903  
## 20 Canada 0.42715225  
## 21 Chad 0.88863939  
## 22 Chile 0.81151134  
## 23 China NA  
## 24 Colombia 0.84289932  
## 25 Congo (Brazzaville) 0.84135950  
## 26 Congo (Kinshasa) 0.86637801  
## 27 Costa Rica 0.76141941  
## 28 Croatia 0.84854555  
## 29 Cyprus 0.89279515  
## 30 Czech Republic 0.88646746  
## 31 Denmark 0.19101639  
## 32 Dominican Republic 0.75528818  
## 33 Ecuador 0.66582751  
## 34 Egypt 0.68449807  
## 35 El Salvador 0.80454427  
## 36 Estonia 0.56873447  
## 37 Ethiopia 0.56702733  
## 38 Finland 0.22336966  
## 39 France 0.64060205  
## 40 Gabon 0.86677748  
## 41 Georgia 0.50241679  
## 42 Germany 0.41216829  
## 43 Ghana 0.94543612  
## 44 Greece 0.82395965  
## 45 Guatemala 0.82165492  
## 46 Guinea 0.76215202  
## 47 Haiti 0.77740395  
## 48 Honduras 0.84808272  
## 49 Hungary 0.90753031  
## 50 India 0.77643496  
## 51 Indonesia 0.94596726  
## 52 Iran NA  
## 53 Iraq 0.76216716  
## 54 Ireland 0.40875691  
## 55 Israel 0.78942990  
## 56 Italy 0.91275305  
## 57 Ivory Coast 0.74424964  
## 58 Japan 0.65444309  
## 59 Jordan NA  
## 60 Kazakhstan 0.71384430  
## 61 Kenya 0.85254985  
## 62 Kosovo 0.85064709  
## 63 Kuwait NA  
## 64 Kyrgyzstan 0.85772502  
## 65 Latvia 0.80840039  
## 66 Lebanon 0.88895327  
## 67 Liberia 0.90267265  
## 68 Libya NA  
## 69 Lithuania 0.92417407  
## 70 Luxembourg 0.37539047  
## 71 Macedonia 0.82417899  
## 72 Madagascar 0.86095339  
## 73 Malawi 0.83482540  
## 74 Malaysia 0.83789223  
## 75 Mali 0.80004674  
## 76 Malta 0.66388631  
## 77 Mauritania 0.71535844  
## 78 Mexico 0.70797193  
## 79 Moldova 0.94311881  
## 80 Mongolia 0.90021819  
## 81 Montenegro 0.78123259  
## 82 Morocco 0.86777443  
## 83 Myanmar 0.63330519  
## 84 Nepal 0.82350838  
## 85 Netherlands 0.41182211  
## 86 New Zealand 0.18588871  
## 87 Nicaragua 0.72799838  
## 88 Niger 0.70254970  
## 89 Nigeria 0.92610925  
## 90 North Cyprus 0.65918028  
## 91 Norway 0.29881436  
## 92 Pakistan 0.71664119  
## 93 Palestinian Territories 0.77430135  
## 94 Panama 0.80994290  
## 95 Paraguay 0.86288828  
## 96 Peru 0.88373041  
## 97 Philippines 0.75519156  
## 98 Poland 0.81009632  
## 99 Portugal 0.94105077  
## 100 Qatar NA  
## 101 Romania 0.96165097  
## 102 Russia 0.91341829  
## 103 Rwanda 0.09460447  
## 104 Saudi Arabia NA  
## 105 Senegal 0.76549017  
## 106 Serbia 0.85935801  
## 107 Sierra Leone 0.82482803  
## 108 Singapore 0.09894388  
## 109 Slovakia 0.92754513  
## 110 Slovenia 0.89219791  
## 111 Somalia 0.41023576  
## 112 South Africa 0.85269475  
## 113 South Korea 0.84072161  
## 114 Spain 0.82166493  
## 115 Sri Lanka 0.85947096  
## 116 Sweden 0.23196414  
## 117 Switzerland 0.20953351  
## 118 Syria 0.68523693  
## 119 Taiwan 0.85719484  
## 120 Tajikistan 0.74168962  
## 121 Tanzania 0.90642261  
## 122 Thailand 0.91365111  
## 123 Togo 0.73326176  
## 124 Tunisia 0.81482500  
## 125 Turkey 0.80607623  
## 126 Turkmenistan NA  
## 127 Ukraine 0.95247275  
## 128 United Arab Emirates NA  
## 129 United Kingdom 0.45613372  
## 130 United States 0.69754261  
## 131 Uruguay 0.67347568  
## 132 Uzbekistan 0.47091693  
## 133 Venezuela 0.81309682  
## 134 Vietnam NA  
## 135 Yemen 0.82909757  
## 136 Zimbabwe 0.81045735

## Make new variables with mutate()

* The mutate() function is most commonly used to add new columns to your data frame that are functions of existing columns.



* mutate() requires data as its first argument, followed by a set of expressions defining new columns. Let’s take a couple examples…
* Create new variables
  + **Note**: New variables are automatically added at the end of the data frame (scroll to the right to see them)

world\_happiness %>%   
 mutate(corruption\_z = scale(corruption), # z-score `corruption` variable  
 life\_int = round(life, 0)) %>% # round `life` variable to a whole number  
 head()

* Change existing variables

When we imported our data, the world variable was automatically categorized as an integer.

class(world\_happiness$world)

However, this variable refers to discrete categories, and we want to change it to be a factor. We can do this using mutate().

# Note that I am re-saving the dataframe here to preserve this change  
world\_happiness <- world\_happiness %>%   
 mutate(world = as.factor(world))

Now check the type again…

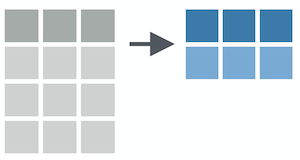
class(world\_happiness$world)

# Summarizing data

* The next dplyr verb we’ll cover is summarize(), which is used to summarize across rows of a dataset. Like all tidyverse functions, summarize() requires data as its first argument, and then you enter your summary functions separated by commas. Summary functions take vectors as inputs and return single values as outputs:



* The resulting dataset will have just the summary variables you created and will lose everything else. In other words, you are going from your raw data frame to a smaller summary data frame that only contains the summary variables you specify within summarize(), as illustrated below…



Let’s use summarize() to get the mean of gdp across all observations in the dataset.

world\_happiness %>%   
 summarize(mean\_gdp = mean(gdp, na.rm = TRUE))

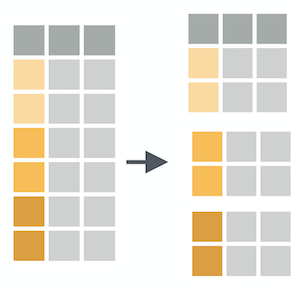
* Of course, we typically want to calculate more than just a mean. We can add other summary variables, separating them by commas.

world\_happiness %>%   
 summarize(mean\_gdp = mean(gdp, na.rm = TRUE), # mean  
 sd\_gdp = sd(gdp, na.rm = TRUE), # standard deviation  
 n = n()) # number of observations

* For a list of other common summary functions, check out the [cheat sheet](https://github.com/rstudio/cheatsheets/blob/master/data-transformation.pdf).

# Grouping data

* The group\_by() function creates groups based on one or more variables in the data. This affects all kinds of things that you then do with the data, such as mutating and/or summarizing. group\_by() requires data as its first argument, and the you name the variable(s) to group by.



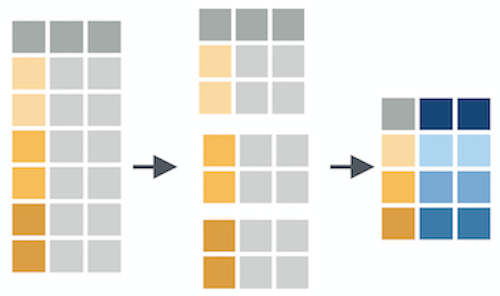
world\_happiness %>%   
 group\_by(world)

## # A tibble: 136 × 9  
## # Groups: world [4]  
## country happiness gdp support life freedom generosity corruption world  
## <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <int>  
## 1 Albania 4.61 9.25 0.639 68.4 0.704 -0.0823 0.885 2  
## 2 Argentina 6.70 NA 0.926 67.3 0.881 NA 0.851 4  
## 3 Armenia 4.35 8.97 0.723 65.3 0.551 -0.187 0.901 2  
## 4 Australia 7.31 10.7 0.952 72.6 0.922 0.316 0.357 1  
## 5 Austria 7.08 10.7 0.928 70.8 0.900 0.0891 0.557 1  
## 6 Azerbaijan 5.15 9.73 0.786 62.0 0.764 -0.223 0.616 2  
## 7 Bahrain 6.01 NA 0.853 65.8 0.850 NA NA 4  
## 8 Bangladesh 4.63 8.05 0.601 61.7 0.815 -0.0598 0.721 3  
## 9 Belarus 5.72 9.73 0.924 65.3 0.623 -0.101 0.669 2  
## 10 Belgium 6.90 10.6 0.885 71.3 0.869 0.0525 0.469 1  
## # … with 126 more rows

At first glance, it doesn’t appear that anything has happened. However, under the hood it has indeed grouped the data frame by the world variable. Copy and paste this code into the console–what do you notice?

## Combining group\_by() and summarize()

* group\_by() and summarize() can be combined to get group-level statistics. This is a great way to make tables of descriptive stats in R or to create aggregated datasets for some purposes.



* To use these together, you just run group\_by() followed by summarize() in a pipeline.

world\_happiness %>%   
 group\_by(world) %>% # group by the world variable  
 summarize(mean\_gdp = mean(gdp, na.rm = TRUE), # mean  
 sd\_gdp = sd(gdp, na.rm = TRUE), # standard deviation  
 n = n()) # number of observations

# Minihacks

For the minihacks today, we will be working with the [diamonds](https://ggplot2.tidyverse.org/reference/diamonds.html) dataset, which is built into R. This dataset contains the prices and various other attributes of about 54,000 different diamonds. Take a peek at the dataset with the following functions:

head(diamonds) # first few rows  
str(diamonds) # structure of the data frame

Here are what the variables refer to:

|  |  |
| --- | --- |
| variable | meaning |
| price | price in US dollars |
| carat | weight of the diamond |
| cut | quality of the cut (Fair, Good, Very Good, Premium, Ideal) |
| color | diamond colour, from D (best) to J (worst) |
| clarity | a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best)) |
| x | length in mm |
| y | width in mm |
| z | depth in mm |
| depth | total depth percentage |
| table | width of top of diamond relative to widest point |

## Minihack 1: Make code more legible

1. Take a look at the following chunk of code. If you’re like me, this will make your head hurt. See if you can understand what this code is trying to do.

arrange(select(filter(diamonds, carat > 3 & carat < 4, cut == "Premium", color == "G" | color == "H" | color == "I" | color == "J"), carat, color, price), color, desc(price))

1. Re-write this code using pipes (%>%) so it is easier to read. Make sure you get the same result that you get when you run the above code. For an extra challenge, try making the filtering step a little more concise.

# your code here

diamonds %>%   
 filter(carat > 3 & carat < 4,   
 cut == "Premium",   
 color %in% c("G", "H","I", "J")) %>%   
 select(carat, color, price) %>%   
 arrange(color, desc(price))

## Minihack 2: Translating into {dplyr} verbs

Answer the questions below using functions from {dplyr}. Think about the order in which you will need to do different operations.

1. On average, which cut of diamond is the most expensive?

# your code here

diamonds %>%  
 group\_by(cut) %>%  
 summarize(mean\_price = mean(price)) %>%   
 arrange(desc(mean\_price))

1. Which is more expensive on average, a diamond that is Fair and IF (worst cut, best clarity), or a diamond that is Ideal and I1(best cut, worst clarity)?

# your code here

# two steps  
diamonds %>%   
 filter(cut == "Fair",   
 clarity == "IF") %>%   
 summarize(mean\_price = mean(price, na.rm = TRUE))  
   
diamonds %>%   
 filter(cut == "Ideal",  
 clarity == "I1") %>%   
 summarize(mean\_price = mean(price, na.rm = TRUE))  
  
# one step  
diamonds %>%   
 group\_by(cut, clarity) %>%   
 summarize(mean\_price = mean(price, na.rm = TRUE)) %>%   
 filter(cut == "Fair" & clarity == "IF" |  
 cut == "Ideal" & clarity == "I1")

## Minihack 3: Summarizing data

1. Calculate the summary statistics listed below for the carat variable for each color of diamond. Give your summary variables the names indicated in parentheses.

* mean (mean)
* standard deviation (sd)
* number of observations (n)
* standard error of the mean (sem)
* 95% confidence interval (lower and upper bounds) around the mean (calculated using a *t* distribution). (ci\_lower and ci\_upper)
  + Hint: Refer to [this slide from class](https://uopsych.github.io/psy611/lectures/9-sampling.html#40) for an example

\*\*\*In your final summary output only include color, mean, ci\_lower and ci\_upper

# your code here

# group by color, CI based on t distribution  
diamonds %>%   
 group\_by(color) %>%   
 summarize(mean = mean(carat, na.rm = TRUE),  
 sd = sd(carat, na.rm = TRUE),  
 n = n(),  
 sem = sd/sqrt(n),  
 ci\_lower = mean - sem \* qt(p = .975, df = n-1),  
 ci\_upper = mean + sem \* qt(p = .975, df = n-1)) %>%   
 select(-n, -sd, -sem)

1. Repeat the same process above, but this time calculate summary statistics for the carat variable for each combination of color and cut. How many observations do you have in your summary data frame this time?

# your code here

diamonds %>%   
 group\_by(color, cut) %>%   
 summarize(mean = mean(carat, na.rm = TRUE),  
 sd = sd(carat, na.rm = TRUE),  
 n = n(),  
 sem = sd/sqrt(n),  
 ci\_lower = mean - sem \* qt(p = .975, df = n-1),  
 ci\_upper = mean + sem \* qt(p = .975, df = n-1)) %>%   
 select(-n, -sd, -sem)