Lab 2: Working with datasets

PSC 103B - Spring 2024

Installing and loading packages

There are many functions available in base R. By this I mean, if you open R right after installing it, there are a number of functions already available for you to use. All the functions we've used in last week's lab are base R functions. They're available as soon as you install R.

But we can add more functions (other than the ones available in base R), by installing and loading packages that contain different functions.

There's something important to understand about packages. There is a difference between installing and loading a package. First, you have to install the package. This downloads it onto your computer and puts it somewhere R can find it. You only have to install a package once (unless you uninstall and install R again).

We can use the `install.packages()` function to do this. The name of the package goes in the parenthesis and must be **inside quotation marks**!

```
install.packages("tidyverse", dependencies = TRUE)
```

`dependencies = TRUE` will tell your computer to also install any packages that package needs to work properly. This usually makes for a smoother installation process. While you are doing this, you may be prompted to answer a question with yes (y) or no (n). You should answer that by typing y in the console and pressing enter. It's generally safe to answer the questions with yes if you don't understand them.

After installing the package it is now on your computer. And you can easily use it **after** loading the package. Every time you open R you must load (But not install!) all the packages you want to use. You can use the library() function to do this. And you don't need quotation marks.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.3     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.4     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

Summary:

- A package must be installed before it can be loaded (use `install.packages()`).
- A package must be loaded before it can be used (use `library()`).

Tidyverse

In R, there are always multiple ways to do the same thing, so you might see code that looks different than what you're about to learn, but does the same thing. That's okay. Today we'll learn how to read data into R (for example, an excel file or other types of data you might have on your computer) and how to select and change the information on that dataset. We'll do all this using a set of packages called tidyverse. Tidyverse is nice because the functions are more intuitive, and we hope that will make it easier to learn. There are also plenty of online resources that can help you when you run into problems (which you inevitably will – this is just a normal part of coding!).

So if you haven't done that yet, go ahead and load tidyverse using the `library()` function.

Working Directory & Reading Data into R

There are different kinds of datafiles that you might want to read into R. Different data files often require different code to be read in successfully. Today we'll cover how to read in one kind of data file In the future, if you need to read in another kind and this code isn't working… Google it! An example google search… "How to read a .rda file in R".

You should have all downloaded a the datafile 'Lab2Data.csv' from Canvas. We'll go over how to load this dataset into R.

(If you are using RStudio Could, this process will be a little different, but not too complicated. You'll need to first upload the dataset you downloaded from canvas into RStudio Cloud using the upload button that's in the Files tab on the lower right hand corner of your screen.)

First we need to set the working directory. We need to tell R where to look for the file we want to read into R. So we set the working directory to the folder where the data file is. If you just downloaded the datafile it'll likely be in your Downloads or Desktop folder. Check where the file is before setting the working directory!

We can use the `setwd()` function. We can put the path to the folder inside the function. **IMPORTANT**: This path *MUST* be within quotations.

```
# Example
setwd("C:/Users/marwin/OneDrive/Documentos/Rprojects/phd/24-Spring/psc103b-sq24/lab2")
       # You will need to change this ^^^ path!
```

The `getwd()` function tells you what folder is currently the working directly. This is helpful to check that the working directory is set to the folder you want.

Another way to change the working directory in RStudio is by using the point & click options. Go to Session > Set Working Directory > Choose Directory. This will open up a new window. Navigate to the folder you want to be your working directory. When you get to that folder, click the option 'Open'. This also works for RStudio Cloud in a similar way. You can use `getwd()` to check that it worked.

The kind of file we have now is a CSV file, or a comma separated values file. This is just a simplified type of spreadsheet that has rows and columns and values in the cells of those rows and columns.

To read in a csv file, we can use the `read_csv()` function that comes in one of the packages in tidyverse. When you read in a datafile, you need to save it under some label. We'll save our data file as dat.

```
dat <- read_csv(file = 'Lab2Data.csv')
```

```
## Rows: 205 Columns: 10
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): fave_animal
## dbl (9): id, e, n, c, a, o, pa, na, ls
##
```

```
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
  # Name of data file needs to be within quotations
```

Note: never name your data "data". This is because `data()` is a function is R and if you name any of your variables or objects the same name as one of its functions, it won't know which one you're talking about.

Note 2: `read_csv()` helpfully tells you the data type it assigned each of the columns you just read in. col_double means it's numeric, while col_character means… you guessed it, it's character! Can you guess what it says when it identifies one of the columns as being type logical?

Let's look at our data:

```
head(dat)
```

```
## # A tibble: 6 x 10
##      id     e     n     c     a     o    pa    na    ls fave_animal
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
## 1     1  2.05  4.75  3.2   5     4.65    17    16   1.4 cat
## 2     2  1.3   5     2.75  3.2   1.8     17    25   1.4 dog
## 3     3  2.95  4.35  3.8   3.9   3.15    19    12   3   other
## 4     4  1.5   4.85  3.35  2.85  4.6     10    25   1   cat
## 5     5  1.85  4.1   3.2   4.2   3.25    17    18   1   dog
## 6     6  1.4   3.9   4.45  4.45  2.85    24    32   1   other
```

```
# prints out 5 first rows, to see whole dataset use View(dat)
# or click the object in the Environment pane on the right.
```

- Columns labels key
  - e = extraversion
  - n = neuroticism
  - c = conscientiousness
  - a = agreeableness
  - o = openness
  - pa = positive affect
  - na = negative affect
  - ls = life satisfaction

We can look up some features of the dataset we just read in:

```
class(dat) # What kind of data structure is dat?
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
dim(dat) # What are the dimensions of dat?
```

```
## [1] 205  10
```

```
str(dat)
```

```
## spc_tbl_ [205 x 10] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ id         : num [1:205] 1 2 3 4 5 6 7 8 9 10 ...
##  $ e          : num [1:205] 2.05 1.3 2.95 1.5 1.85 1.4 2.4 2.1 2.2 2.3 ...
##  $ n          : num [1:205] 4.75 5 4.35 4.85 4.1 3.9 3.8 3.2 3.05 3.55 ...
##  $ c          : num [1:205] 3.2 2.75 3.8 3.35 3.2 4.45 2.3 3.35 4.1 1.85 ...
##  $ a          : num [1:205] 5 3.2 3.9 2.85 4.2 4.45 2.65 4.4 3.45 3.9 ...
##  $ o          : num [1:205] 4.65 1.8 3.15 4.6 3.25 2.85 2.15 3.7 2.9 4.4 ...
##  $ pa         : num [1:205] 17 17 19 10 17 24 14 12 26 25 ...
##  $ na         : num [1:205] 16 25 12 25 18 32 30 16 12 12 ...
```

```
##  $ ls         : num [1:205] 1.4 1.4 3 1 1 1 2 1 6 2.4 ...
##  $ fave_animal: chr [1:205] "cat" "dog" "other" "cat" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   id = col_double(),
##   ..   e = col_double(),
##   ..   n = col_double(),
##   ..   c = col_double(),
##   ..   a = col_double(),
##   ..   o = col_double(),
##   ..   pa = col_double(),
##   ..   na = col_double(),
##   ..   ls = col_double(),
##   ..   fave_animal = col_character()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

```
# This is a new function.
# By looking at the output what do you think the str() function tells you?
```

```
summary(dat)
```

```
##        id            e               n               c               a
##  Min.   :  1   Min.   :1.150   Min.   :1.000   Min.   :1.650   Min.   :1.700
##  1st Qu.: 52   1st Qu.:2.900   1st Qu.:2.100   1st Qu.:3.100   1st Qu.:3.350
##  Median :103   Median :3.300   Median :2.650   Median :3.500   Median :3.800
##  Mean   :103   Mean   :3.266   Mean   :2.662   Mean   :3.513   Mean   :3.741
##  3rd Qu.:154   3rd Qu.:3.750   3rd Qu.:3.150   3rd Qu.:3.900   3rd Qu.:4.150
##  Max.   :205   Max.   :4.800   Max.   :5.000   Max.   :4.900   Max.   :5.000
##                NA's   :5                                       NA's   :8
##        o               pa             na              ls
##  Min.   :1.800   Min.   :10.0   Min.   :10.00   Min.   :1.000
##  1st Qu.:3.300   1st Qu.:24.0   1st Qu.:10.00   1st Qu.:3.200
##  Median :3.700   Median :30.0   Median :12.00   Median :4.800
##  Mean   :3.731   Mean   :30.6   Mean   :15.05   Mean   :4.439
##  3rd Qu.:4.150   3rd Qu.:37.0   3rd Qu.:18.00   3rd Qu.:5.800
##  Max.   :5.000   Max.   :50.0   Max.   :46.00   Max.   :7.000
##                                                 NA's   :2
##  fave_animal
##  Length:205
##  Class :character
##  Mode  :character
##
##
##
##
```

```
# The summary() function will give you the breakdown of a bunch of stats
# for each column.
```

How many missing values are there in the extraversion variable?

```
sum(is.na(dat$e))
```

```
## [1] 5
```

If you look at the last answer we got, it's a little confusing to read, right? You have to start with the innermost

thing, and think about what that is (a vector of numbers), and then imagine what the next function that's around that will do to it (test to see if it's NA or not and return a vector of TRUE/FALSES) and then what the other thing around it does (sums all the TRUE values). With just three steps, it's not too terrible, but you can imagine it can get a lot worse if you need to use a lot of functions, one on top of the other. It's just not natural for most people to read from the inside out. Thankfully, there's a better way!

Pipe

This is called a pipe: `%>%` . You can insert one by holding Command+Shift+M (Mac) or Ctrl+Shift+M (Windows). So you don't have to type all the symbols. But what is it for? Basically, for sending the result of one operation into another function. Taking the example above, we could do the following:

```
sum(is.na(dat$e))
```

```
## [1] 5
```

or we could use a pipe, and do the following:

```
dat$e %>% is.na() %>% sum()
```

```
## [1] 5
```

The result of whatever comes before the pipe gets inserted into the function that comes after it. This makes it easier to read. You can also put each thing on a different line, like this:

```
dat$e %>%
  is.na() %>%
  sum()
```

```
## [1] 5
```

This might seem silly with short, simple functions like these, but as functions get more complex, this will make writing and reading code a lot easier and more organized.

Subsetting Datasets

Last week we talked about subsetting vectors.

```
ex.vec <- c(1:15)
```

Do you remember how to subset the 10th element in ex.vec?

```
ex.vec[10]
```

```
## [1] 10
```

And do you remember how to subset when we have a two-dimensional data structure? We can still use `[]` , but now we have to tell R what rows *and* what columns we want to subset `dat[r#, c#]`.

If we only need one row (e.g., extraversion, the 2nd row), that's easy:

```
dat[ , 2] # just leave the row number empty. Remember the comma though!
```

```
## # A tibble: 205 x 1
##        e
##    <dbl>
##  1  2.05
##  2  1.3
##  3  2.95
##  4  1.5
```

```
##  5  1.85
##  6  1.4
##  7  2.4
##  8  2.1
##  9  2.2
## 10  2.3
## # i 195 more rows
```

```
dat[ ,"e"] # we can also do this
```

```
## # A tibble: 205 x 1
##        e
##    <dbl>
##  1  2.05
##  2  1.3
##  3  2.95
##  4  1.5
##  5  1.85
##  6  1.4
##  7  2.4
##  8  2.1
##  9  2.2
## 10  2.3
## # i 195 more rows
```

```
dat$e #or this
```

```
##   [1] 2.05 1.30 2.95 1.50 1.85 1.40 2.40 2.10 2.20 2.30 2.50   NA 2.65 2.00 3.00
##  [16] 3.00 3.45 3.15 3.45 3.20 3.25 2.90 3.00 3.30 4.10 3.00 4.55 4.10 3.60 2.80
##  [31] 3.05 3.15 2.95 3.55 3.15 3.40 4.15 3.05 3.20 2.65 3.30 3.70 1.85 2.95 1.15
##  [46] 2.45 2.55 2.10 3.55 3.10 2.20 2.55 3.60 3.60 2.90 3.70 2.85 2.75 3.20 3.40
##  [61] 2.35 2.90 1.25 3.35 2.70 4.20 2.70 2.35 3.55 3.70 3.65 3.15 2.95   NA 2.95
##  [76] 2.70 3.60 3.75 3.40 2.30 3.35 2.75 2.35 2.40   NA 2.55 2.45 3.80 3.25 3.75
##  [91] 2.20 4.00 4.00 3.00 3.60 3.80 2.65 3.85 3.30 3.25 3.25 2.20 3.80 3.35 3.25
## [106] 2.95 3.70 3.55 3.95 3.25 3.30 3.40 3.60 3.30 3.50 3.70 3.50 3.15 3.85 3.60
## [121] 3.75 2.65 3.05 3.60 3.75 3.35 3.90 3.45 3.40 3.00 4.00 3.50 3.05 3.45 3.60
## [136] 3.85 3.35 2.90 3.75 3.85 3.60 2.90 2.80 2.70 3.25 3.65 3.90 1.30 3.70 3.85
## [151] 2.95 4.30 4.30 2.70 2.70 3.10 3.70   NA 3.15 3.65 3.20 3.15 3.05 1.70 2.75
## [166] 3.70 3.45 4.40 4.00 3.90 4.10 4.60 3.45 4.35 3.20 2.80 3.70 4.35 3.55 3.15
## [181] 4.05 3.70 4.25 3.15 3.20 4.30 4.05 3.95 3.15 4.30 3.90   NA 4.45 3.85 3.80
## [196] 4.75 3.90 4.15 4.45 4.00 2.60 4.05 4.80 4.30 3.40
```

But how do we do this if we want more columns? Try to use what we learned last week to pull out all the Big Five columns (e, a, c, n, and o) and save them to a new data frame, big5.

```
big5 <- dat[,c("e", "a", "c", "n", "o")]
```

But that's a little clunky. There's a better way to do that using `select()`, one of the tidyverse functions. `select()` does exactly what it sounds like: it selects the columns you tell it to. All you need to do is tell it where to find the columns, and then which columns you want, like this:

```
dat %>%
  select(c(e, a, c, n, o))
```

```
## # A tibble: 205 x 5
##        e     a     c     n     o
##    <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  2.05  5     3.2   4.75  4.65
```

```
##  2  1.3   3.2   2.75  5     1.8
##  3  2.95  3.9   3.8   4.35  3.15
##  4  1.5   2.85  3.35  4.85  4.6
##  5  1.85  4.2   3.2   4.1   3.25
##  6  1.4   4.45  4.45  3.9   2.85
##  7  2.4   2.65  2.3   3.8   2.15
##  8  2.1   4.4   3.35  3.2   3.7
##  9  2.2   3.45  4.1   3.05  2.9
## 10  2.3   3.9   1.85  3.55  4.4
## # i 195 more rows
```

We can save that to a new object:

```
big5 <- dat %>%
  select(c(e, a, c, n, o))
```

It knows to expect the first argument–that means the thing that comes first after the parenthesis–to be a dataframe, and everything that comes after to be column names in that dataframe. Please note that, within the select function (and many other tidyverse functions) you don't need to put the column names in quotes, which makes it simpler and faster to use.

Describing the Data

A useful way to count and display frequencies in R is the `table()` function. For example, we can use the `table()` function to count values in a column.

```
table(dat$fave_animal)
```

```
##
##   cat   dog other
##    72    74    55
```

Can we use it with a continuous variable?

```
table(dat$e)
```

```
##
## 1.15 1.25  1.3  1.4  1.5  1.7 1.85    2 2.05  2.1  2.2  2.3 2.35  2.4 2.45  2.5
##    1    1    2    1    1    1    2    1    1    2    4    2    3    2    2    1
## 2.55  2.6 2.65  2.7 2.75  2.8 2.85  2.9 2.95    3 3.05  3.1 3.15  3.2 3.25  3.3
##    3    1    4    6    3    3    1    5    7    6    5    2   10    6    7    5
## 3.35  3.4 3.45  3.5 3.55  3.6 3.65  3.7 3.75  3.8 3.85  3.9 3.95    4 4.05  4.1
##    5    6    6    3    5   10    3   10    5    4    6    5    2    5    3    3
## 4.15  4.2 4.25  4.3 4.35  4.4 4.45 4.55  4.6 4.75  4.8
##    2    1    1    5    2    1    2    1    1    1    1
```

We can also describe our data by calculating some descriptive statistics.

Descriptive statistics functions

- `mean()`
- `var()`
- `sd()`
- `median()`

For example if I want to calculate the mean of extraversion. I can use the `mean()` function but I have to tell the function what column in the data set I want to use

```r
mean(big5$e) # What do you need to add to this line of code to calculate the mean of extraversion?
```

```
## [1] NA
```

Did you get an NA? Why do you think that is?

Clue: are there any missing values in the variable extraversion?

```r
mean(big5$e, na.rm = TRUE)
```

```
## [1] 3.266
```

What about if I want to calculate the variance of extraversion?

```r
var(big5$e, na.rm = TRUE)
```

```
## [1] 0.4936372
```

And the standard deviation?

```r
sd(big5$e, na.rm = TRUE)
```

```
## [1] 0.7025932
```

Sometimes you'll want to compute your descriptive statistics by group. Specifically, you need to do this if your research question focuses on group differences (e.g., between experimental conditions or gender)

You can do this using the `group_by()` and `summarize()` functions. For example, let's say that we want to know the means & standard deviations for agreeableness for cat people vs. dog people in our dataset.

First, we have a small problem. Can you spot it by looking at this table?

```r
table(dat$fave_animal)
```

```
##
##   cat   dog other
##    72    74    55
```

We're only interested in cat and dog people, but there are a bunch of people who said their favorite animal is something else, or "other". So first we need to separate the observations for cat and dog people from those other people's who have other favorite animals. One way to do that is to `filter()` our dataset.

The `filter()` function works very similarly to the `select()` function we just learned about, but instead of selecting columns to keep, it selects rows to keep. This is a similar function to the filter function in spreadsheets, if you are familiar with those.

Because this is a close cousin to `select()`, it also expects the dataframe you are trying to filter as the first argument. Then, you need to tell it which rows to keep. But how do you do that if rows don't have names like columns? You use a condition. For example, in this case, we want to keep only the people who are cat people or the people who are dog people. The other kind of people in our dataset are "other" people, and we are not interested in them. So we want the rows (people) who are NOT "other". The way to tell R this is to use the following statement: `fave_animal != "other"`

The symbol `!=` means "not equal", so we are telling the filter function to give us all rows where fave_animal is NOT equal to "other", which are the rows where fave_animal is equal to either cat or dog. There are other symbols you can use although some only work on numeric variables:

- `==` means "equal to"
- `>` means "greater than"
- `<` means "smaller than"
- `>=` means "greater or equal to"
- `<=` means "smaller or equal to"

Here is the complete line of code:

```
filter(dat, fave_animal != "other")
```

```
## # A tibble: 146 x 10
##       id     e     n     c     a     o    pa    na    ls fave_animal
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
##  1     1  2.05  4.75  3.2   5     4.65    17    16   1.4 cat
##  2     2  1.3   5     2.75  3.2   1.8     17    25   1.4 dog
##  3     4  1.5   4.85  3.35  2.85  4.6     10    25   1   cat
##  4     5  1.85  4.1   3.2   4.2   3.25    17    18   1   dog
##  5     7  2.4   3.8   2.3   2.65  2.15    14    30   2   cat
##  6     8  2.1   3.2   3.35  4.4   3.7     12    16   1   dog
##  7    10  2.3   3.55  1.85  3.9   4.4     25    12   2.4 cat
##  8    11  2.5   3.85  2.35  3.05  3.55    29    22   2.2 cat
##  9    12 NA     3.6   4.15  3.1   2.95    21    10   5.2 dog
## 10    16  3     3.15  2.95  2.7   2.9     38    39   5   cat
## # i 136 more rows
```

We should save this new, smaller dataset

```
cats_dogs <- filter(dat, fave_animal != "other")
```

Now you try it! Can you select only the rows with cat people?

```
cats_only <- filter(dat, fave_animal == "cat")
```

What about the rows with people who have extraversion higher than 3?

```
high_e <- filter(dat, e > 3)
```

Great, now that we have a smaller dataset with only cat and dog people, we can calculate the means and SDs for agreeableness for each group. First, we start by grouping the dataset we have with `group_by()`. Like `select()` and `filter()`, it expects the dataset as the first argument. The second argument should be the variable we want to use to make the groups, `fave_animal`. Finally, we want to use `summarize()` to get the mean of the variable `a`, and we want that mean to be called `M`. This is how we describe that.

You can think of `group_by()` as sort of separating your dataset into smaller datasets based on the different levels of the variable you give it; so if I group by `fave_animal`, what's going on behind the scenes is R and tidyverse are splitting this into one dataset of cat people, and then one dataset of dog people.

Then, `summarize()` will compute whatever descriptive you tell it in each of those datasets.

```
group_by(cats_dogs, fave_animal) %>%
  summarize(M = mean(a))
```

```
## # A tibble: 2 x 2
##   fave_animal     M
##   <chr>       <dbl>
## 1 cat            NA
## 2 dog            NA
```

When we run this, you'll see that it tells us the mean for each group is NA. Where have we seen this problem before? Can you fix it?

```
group_by(cats_dogs, fave_animal) %>%
  summarize(M = mean(a, na.rm = TRUE))
```

```
## # A tibble: 2 x 2
##   fave_animal     M
##   <chr>       <dbl>
```

```
## 1 cat          3.71
## 2 dog          3.75
```

How would I do the same thing except for the standard deviation?

```
group_by(cats_dogs, fave_animal) %>%
  summarize(M = sd(a, na.rm = TRUE))
```

```
## # A tibble: 2 x 2
##   fave_animal     M
##   <chr>       <dbl>
## 1 cat         0.646
## 2 dog         0.644
```

You can also group by more than one grouping variable. I'll make one from one of our continuous variables: extraversion (We haven't learned how to do this yet, so don't worry if the code below looks confusing, but you might be able to understand it based on the other things we already do know how to do. If you're curious, I encourage you to google :)).

Let's separate our rows into "high" extraversion (>3) and "low" extraversion (<=3)

```
cats_dogs <- mutate(
  cats_dogs, # mutate is used to create a new variable in the dataframe cats_dogs
  e_level = if_else(
    e > 3, "high", "low"
    )
  ) # we're creating e_level, and saying that it's "high"
    # if e is bigger than 3, "low" otherwise
```

Now, let's summarize the means & standard deviations for agreeableness by favorite animal and extraversion level.

```
group_by(cats_dogs, fave_animal, e_level) %>%
  summarize(M = mean(a, na.rm = TRUE))
```

```
## `summarise()` has grouped output by 'fave_animal'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 6 x 3
## # Groups:   fave_animal [2]
##   fave_animal e_level     M
##   <chr>       <chr>   <dbl>
## 1 cat         high     3.80
## 2 cat         low      3.49
## 3 cat         <NA>     3.95
## 4 dog         high     3.82
## 5 dog         low      3.65
## 6 dog         <NA>     3.33
```

You can see that all combinations of favorite animal and extraversion level that exist in the data are represented here, including cat and dog people who don't have a value for extraversion (and therefore don't have a value for extraversion level).

Now try to do it for SD:

```
group_by(cats_dogs, fave_animal, e_level) %>%
  summarize(M = sd(a, na.rm = TRUE))
```

```
## `summarise()` has grouped output by 'fave_animal'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 6 x 3
## # Groups:   fave_animal [2]
##   fave_animal e_level      M
##   <chr>       <chr>    <dbl>
## 1 cat         high     0.592
## 2 cat         low      0.736
## 3 cat         <NA>    NA
## 4 dog         high     0.597
## 5 dog         low      0.689
## 6 dog         <NA>     1.07
```

You can also summarize variables with more than one function at a time:

```
group_by(cats_dogs, fave_animal) %>%
  summarize(M = mean(a, na.rm = TRUE), SD = sd(a, na.rm = TRUE))
```

```
## # A tibble: 2 x 3
##   fave_animal     M    SD
##   <chr>       <dbl> <dbl>
## 1 cat          3.71 0.646
## 2 dog          3.75 0.644
```

Remember, don't name your mean variable mean or your sd variable sd so they don't have the same name as the functions mean() and sd().

Confidence Intervals

We can create an interval around our mean. If we compute a 95% CI this only tells us that roughly 95% of intervals will contain the true population mean.

Steps for calculating the confidence interval:

1. Calculate the sample mean
2. Calculate the standard error

- $SE = \frac{SD}{\sqrt{n}}$
- Let's assume sd = 1

3. Calculate the z-score multiplier for your desired CI

- When calculating a 95% CI for a normal distribution, this multiplier (z_{95}) always equals 1.96, but we'll learn to extract that value from a function in R so we can calculate other CIs

4. Construct CI around sample mean

- $CI = \bar{x} \pm z_{95} \times SE$

Let's calculate the confidence interval for extraversion for all the observations (dat)

Step 1

```
e.mean <- mean(dat$e, na.rm = TRUE)
```

Step 2

How could we find out the sample size?

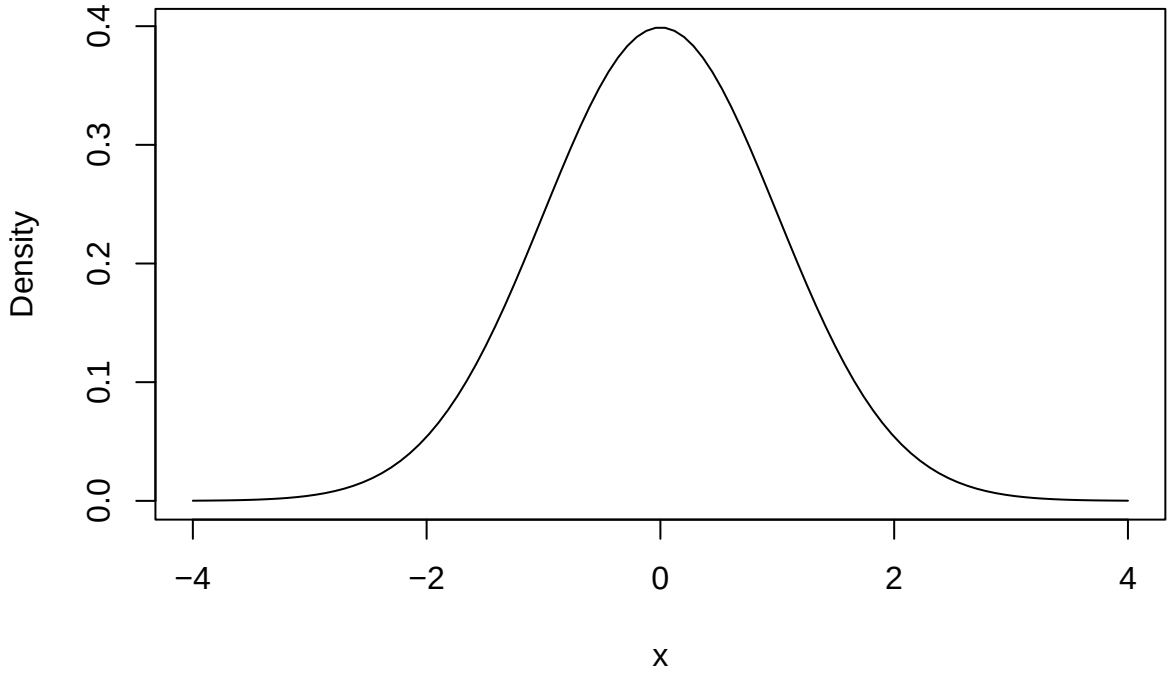
```
n <- nrow(dat)
se <- 1/sqrt(n)
```

Step 3

Let's find that multiplier! This part is a little confusing, so bear with me. So 1.96 is the value in the normal distribution that leaves 2.5% of the values to one side, and 2.5% to the other. There are several functions in R that can tell us the values we want for specific distributions and the functions for the normal distribution are called `_norm()` where `_` can be `q`, `r`, `d`, and `p`. We'll only use `qnorm()` for now.

`qnorm()` takes one argument and gives you one value back. It needs you to tell it what cumulative proportion of the normal curve you want (area under the curve) and in return, it will give you the z value (the value on the x axis) back. For a visualization, run the three lines below. You should see a plot on the lower right side of the screen.

```
x <- seq(-4, 4, length=100)
Density <- dnorm(x)
plot(x, Density,  type="l", lty=1)
```



Looking at this plot, if you started coloring the area under the curve from left to right, what value do you think would be on the x-axis when you hit 50%? You can test if you're right by asking `qnorm`:

```
qnorm(.5)
```

```
## [1] 0
```

Did you get the value you expected?

But we don't want 50%, we want a value that will tell us what the x-axis value is on both sides when we color the middle 95% of the area under the curve. Look at the curve again. If we did that, how much would be left on each side?

If you said 5%, that's a good guess, but it's wrong. The curve is symmetrical, so if you filled in 95% in the middle, you'd have 2.5% left on one side, and 2.5% left on the other! What we want to know, then, is what

value do we give `qnorm` to get it to tell us what the x-axis value is when 2.5% is left.

Imagine you're coloring under the curve from the left side all the way to right. How much do you need to fill in to leave 2.5%?

```
qnorm(0.975)
```

```
## [1] 1.959964
```

Because the curve is symmetrical, you could also get the negative value by entering 0.025 (2.5%) into `qnorm()`:

```
qnorm(.025)
```

```
## [1] -1.959964
```

Check your understanding: what value would you give `qnorm()` if you wanted to find the multiplier to construct a 90% CI? What about a 99% CI?

```
# 90% CI
qnorm(0.95)
```

```
## [1] 1.644854
```

```
# 99% CI
qnorm(.995)
```

```
## [1] 2.575829
```

Let's save that `z_95` value so we can use it on the next step:

```
z_95 <- qnorm(.975)
```

Step 4

```
lowerlim <- e.mean - (z_95 * se) #this is the lower limit of your CI
upperlim <- e.mean + (z_95 * se) #this is the upper limit of your CI

lowerlim
```

```
## [1] 3.12911
```

```
upperlim
```

```
## [1] 3.40289
```

Z scores

We also talked about z scores in the lectures this past week. They're standardized values, where the mean of the variable is 0 and the standard deviation is 1

We can calculate z scores in R (or by hand!)

Formula: $z = \frac{(x - \bar{x})}{\sigma_x}$

- x is a specific value, for example one person's extraversion score
- $\bar{x}$ is the sample mean of extraversion
- σ_x is the sample standard deviation of extraversion

Let's calculate the z-score for one extraversion score:

```r
e.score <- dat[5 , "e"]

e.sd <- sd(dat$e, na.rm = TRUE)
```

We already saved the mean for extraversion so we can use that here. Let's calculate the score:

```r
z.e <- (e.score - e.mean) / e.sd
```

How can we interpret this z-score?

We can interpret this z-score as: This individual has an extraversion score that is about 2 standard deviations below the mean.

The nice thing about R is that we don't need to calculate each z score one by one. We can just get R to calculate all of them at once for us. This is called a "vectorized operation" because the operations happen on a whole vector all at once!

Let's repeat the steps above for all extraversion scores at once:

```r
e.scores <- dat$e
# this is the entire vector for all the extraversion scores, check it out!
```

We already have the SD and Mean for e, so all we need to do now is:

```r
zs.e <- (e.scores - e.mean) / e.sd
```

Look at zs.e, what does it look like? What is its mean and SD?

```r
zs.e
```

```
##   [1] -1.73073127 -2.79820533 -0.44976240 -2.51354558 -2.01539102 -2.65587546
##   [7] -1.23257671 -1.65956634 -1.51723646 -1.37490659 -1.09024684          NA
##  [13] -0.87675203 -1.80189621 -0.37859747 -0.37859747  0.26188697 -0.16510265
##  [19]  0.26188697 -0.09393772 -0.02277278 -0.52092734 -0.37859747  0.04839216
##  [25]  1.18703115 -0.37859747  1.82751559  1.18703115  0.47538178 -0.66325721
##  [31] -0.30743253 -0.16510265 -0.44976240  0.40421684 -0.16510265  0.19072203
##  [37]  1.25819609 -0.30743253 -0.09393772 -0.87675203  0.04839216  0.61771165
##  [43] -2.01539102 -0.44976240 -3.01170014 -1.16141177 -1.01908190 -1.65956634
##  [49]  0.40421684 -0.23626759 -1.51723646 -1.01908190  0.47538178  0.47538178
##  [55] -0.52092734  0.61771165 -0.59209228 -0.73442215 -0.09393772  0.19072203
##  [61] -1.30374165 -0.52092734 -2.86937027  0.11955709 -0.80558709  1.32936103
##  [67] -0.80558709 -1.30374165  0.40421684  0.61771165  0.54654672 -0.16510265
##  [73] -0.44976240          NA -0.44976240 -0.80558709  0.47538178  0.68887659
##  [79]  0.19072203 -1.37490659  0.11955709 -0.73442215 -1.30374165 -1.23257671
##  [85]          NA -1.01908190 -1.16141177  0.76004153 -0.02277278  0.68887659
##  [91] -1.51723646  1.04470128  1.04470128 -0.37859747  0.47538178  0.76004153
##  [97] -0.87675203  0.83120647  0.04839216 -0.02277278 -0.02277278 -1.51723646
## [103]  0.76004153  0.11955709 -0.02277278 -0.44976240  0.61771165  0.40421684
## [109]  0.97353634 -0.02277278  0.04839216  0.19072203  0.47538178  0.04839216
## [115]  0.33305191  0.61771165  0.33305191 -0.16510265  0.83120647  0.47538178
## [121]  0.68887659 -0.87675203 -0.30743253  0.47538178  0.68887659  0.11955709
## [127]  0.90237140  0.26188697  0.19072203 -0.37859747  1.04470128  0.33305191
## [133] -0.30743253  0.26188697  0.47538178  0.83120647  0.11955709 -0.52092734
## [139]  0.68887659  0.83120647  0.47538178 -0.52092734 -0.66325721 -0.80558709
## [145] -0.02277278  0.54654672  0.90237140 -2.79820533  0.61771165  0.83120647
## [151] -0.44976240  1.47169090  1.47169090 -0.80558709 -0.80558709 -0.23626759
## [157]  0.61771165          NA -0.16510265  0.54654672 -0.09393772 -0.16510265
## [163] -0.30743253 -2.22888583 -0.73442215  0.61771165  0.26188697  1.61402078
## [169]  1.04470128  0.90237140  1.18703115  1.89868052  0.26188697  1.54285584
```

```
## [175] -0.09393772 -0.66325721  0.61771165  1.54285584  0.40421684 -0.16510265
## [181]  1.11586622  0.61771165  1.40052596 -0.16510265 -0.09393772  1.47169090
## [187]  1.11586622  0.97353634 -0.16510265  1.47169090  0.90237140          NA
## [193]  1.68518571  0.83120647  0.76004153  2.11217534  0.90237140  1.25819609
## [199]  1.68518571  1.04470128 -0.94791696  1.11586622  2.18334027  1.47169090
## [205]  0.19072203
```

```
mean(zs.e, na.rm = TRUE)
```

```
## [1] -2.382223e-17
```

```
sd(zs.e, na.rm = TRUE)
```

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

Coding Challenge

You're now going to do the things that we've covered in this lab but without code to follow along to. Work with each other, but don't copy others' answers!

1. Pick a variable continuous from our dat dataset. In other words a variable other than `fave_animal` and `e_level`.
2. Explore the variable by looking at it's mean, variance, sd and whether there are any missing values. Also calculate these statistics based on sub-groups.
3. Calculate a confidence interval around the mean of that variable.
4. Convert some (or all!) of the scores into z-scores.