Introduction to R and RStudio

2025-05-21

Introduction

The purpose of this document is to provide an introduction to the basic structure and functions of R and RStudio for those who are unfamiliar. After working through the sections and code in this document you should be able to:

- Understand how RStudio is set up
- Perform basic calculations in R
- Store variables in R
- Understand what a working directory is, and how to set one
- Read in a dataset (from Excel or similar formats) into R
- Perform basic explorations of this dataset including: number of observations and variables, the basic structure of this dataset, how to view components of the dataset
- Verify the types of variables in your dataset and their structure

General Housekeeping

Before we start, let's make sure that we have a way to keep our files organized during this workshop. You should have a folder on your computer with materials for this class. You should save all materials related to this class in this folder and organize them with different folders for each lab and exam. To simplify things going forward, call this folder `seroanalytics_workshop`.

The Basic Setup of RStudio

When you open this document in RStudio, you will notice there are 4 main sub-sections in your window going clockwise starting in the upper left corner, they are:

1. The source pane: this includes the document you have opened and any other written code that you have saved.
2. The environment pane: this is a list of all variables, datasets and other objects that R has stored.
3. The plot, packages, and file pane: a window that allows you to see files and plots and to manage and install packages.
4. The console pane: this is where code is run.

Here are some important points about the console and and a saved document that contains code.

- The saved document you are working with is an Rmarkdown (Rmd) document. It allows you to write both code that you can run in R, and regular words, much like any other word processor.
- You can make alterations to your code in the document (located in box 1 from the list above) and save these changes, just like you can for a word document.

- If you type any code directly into the console, it will run, but it will not be saved and you cannot make changes to it.
- We recommend only writing and altering code in your document, and then running this code so that it is always saved.

Importantly, when you write and save code, you may need to come back to it at a later date. Therefore it is important to add comments to your code so you can orient yourself to your code when you return to it. Comments are little notes that tell you what a piece of code accomplishes, or why you wrote it, but they are not code: they do not "run" or produce output. We will provide examples in the following bits of R code.

R is a calculator

```
# this is a comment

# this code performs some basic calculations

3 + 3
```

```
## [1] 6
```

```
5 * 10
```

```
## [1] 50
```

```
exp(12)
```

```
## [1] 162754.8
```

```
log(3)
```

```
## [1] 1.098612
```

R can store variables

Here, we are assigning `x`, `y`, and `z` to have specific values with the use of an assignment arrow `<-`. If you run this code, and look in your environment tab, you will notice now R has remembered that `x`, `y`, and `z` have these specific values.

```
# this code stores numeric variables
x <- 3
x
```

```
## [1] 3
```

```
y <- x * 5
y
```

```
## [1] 15
```

```r
z <- log(y)
z
```

```
## [1] 2.70805
```

We can also assign variables to have values that are not numeric, for example characters or strings.

```r
introduction <- "My name is John"

introduction
```

```
## [1] "My name is John"
```

Every named variable in R is case specific, so if I try to type in `Introduction` with a capital letter `I`, R will return an error, I must use the name exactly as I have typed it.

As you begin to work with more variables in R, you will notice that naming variables is very important. You want to pick a name for variables that gives you information about what that variable contains. You also cannot use spaces or dashes (`-`) in your name. If you want to separate words in the name of a variable, you should use an underscore or a `.`. For example:

```r
# this code stores character variables

full_name <- "John Smith"

full_name
```

```
## [1] "John Smith"
```

The variable names of `full name` or `full-name` etc., are not valid.

If you start a variable name with a number or you do have spaces or a dash, you will need to enclose your variable names in this symbol “. For example:

```r
"full name" <- "John Smith"

"full-name" <- "John Smith"

"full-name"
```

```
## [1] "full-name"
```

We can also store vectors in R of numbers or characters:

```r
# this code creates vectors

vect_numeric <- c(1, 2, 3, 4, 5)
vect_numeric
```

```
## [1] 1 2 3 4 5
```

```
vect_character <- c("a", "b", "c", "d", "e")
vect_character
```

```
## [1] "a" "b" "c" "d" "e"
```

Importantly, each element of the vector is separated by a comma. So if we were to make a character vector in the following way:

```
vect_character1 <- c("a,b,c,d,e")
vect_character1
```

```
## [1] "a,b,c,d,e"
```

and compare the length of `vect_character`, and `vect_character1`, they would not be the same:

```
# this code measures the length of vectors

length(vect_character)
```

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

```
length(vect_character1)
```

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

We can also access a particular element of a vector in the following way:

```
# this code retrieves an element of a vector in position 3
vect_character[3]
```

```
## [1] "c"
```

This returns the letter `c`, because `c` is the third element of `vect_character`.

Working Directories and Reading in Data

Before we begin working with a dataset, we must establish a working directory. This is a file path that tells R where to locate and save files on your computer. This should be the folder that you created for this class on your computer.

We can start by seeing what the current working directory of R is:

```
# this code returns the current working directory
getwd()
```

If `getwd()` is not showing the correct file path, then one way to set the directory is to use the `setwd()` function as shown below. Here, we show an example of setting the working directory to the location where the *seroanalytics_workshop* folder is stored. You will need to replace the below *my_path* with the exact file path of your *seroanalytics_workshop* folder.

```
# replace the my_path location below with the location of
# your seroanalytics_workshop folder
my_path <- "/OneDrive-JohnsHopkins/seroanalytics_workshop"

setwd(my_path)
```

Another way to correctly set your working directory (if `getwd()` is not showing the directory we want), is to do the following:

1. Go to "Session" in the uppermost menu bar
2. Go to "Set Working Directory"
3. Then to "Choose Directory" and click this
4. A dialog box should appear, and you can manually navigate to the correct folder.
5. Once you have navigated to the correct folder, click open.

You must set your working directory every time you open an new RStudio session, and any time you read in data to R, you must ensure that your working directory is the folder that directly contains the file you wish to read in. Otherwise R will return an error.

Now, that we have the correct working directory, we can read in one dataset that we will work with for the introductory lab using the `read.csv()` function. These data are a set of meta data variables (age, sex) that link to the individual IDs from our training serosurvey dataset.

```
# this code reads in data from the set working directory
# meta_data <- read.csv('Training demographics df.csv')

# another way to read in this file is to use the my_path
# location of the working directory and paste it in front
# of the file name this is generally the best way to read
# in data in Rmd files
my_path <- "/OneDrive-JohnsHopkins/seroanalytics_workshop"

meta_data <- read.csv(paste(my_path, "Data/Training demographics df.csv",
    sep = "/"))
```

Exploring data

We will now begin exploring one data set, the `meta_data` data to discover:

1. How many rows and columns are in these data
2. Which variables are in these data
3. The types of these variables

Let's begin by viewing our full dataset with the `View` function.

```
# this code allows us to view our data

View(meta_data)
```

If we don't want to view the entire dataset (there are many rows), we can use the `head()` function which allows us to view the first six rows of the data.

```
head(meta_data)
```

```
##   Luminex_id Luminex_plate id age sex
## 1   Unknown1        Plate1  1   0   2
## 2   Unknown2        Plate1  2   0   2
## 3   Unknown3        Plate1  3   0   2
## 4   Unknown4        Plate1  4   0   2
## 5   Unknown5        Plate1  5   0   1
## 6   Unknown6        Plate1  6   0   2
```

Side note, if you forget what the function `head()` is used for or what the appropriate way to use it is, you can always remind yourself with the help function as follows. The details will appear in the help pane.

```
`?`(head())
```

The help function can be used to provide information on any function.

Now, we can observe the number of rows (with the function `nrow()`) and columns (with the function `ncol()`) in our data.

```
# this code returns the number of rows and columns in the
# data
nrow(meta_data)
```

```
## [1] 1000
```

```
ncol(meta_data)
```

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

```
# another way to look at this is to use the following
# function which outputs the number of rows first, then
# number of columns
dim(meta_data)
```

```
## [1] 1000    5
```

The general structure of our data can be observed with the function `str()`, including class of the object, variable names and what the variable type is (e.g., integer, character).

```
# this code returns the structure of the data
str(meta_data)
```

```
## 'data.frame':    1000 obs. of  5 variables:
##  $ Luminex_id   : chr  "Unknown1" "Unknown2" "Unknown3" "Unknown4" ...
##  $ Luminex_plate: chr  "Plate1" "Plate1" "Plate1" "Plate1" ...
##  $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ age          : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ sex          : int  2 2 2 2 1 2 2 2 1 2 ...
```

Another way to return the structure or class of the data object or specific variables is to use the `class()` function. In R, the `$` operator is used to extract a named column from a list or data frame by name—not by position or type.

```
# this code is another way to return the class or structure
# of the data
class(meta_data)
```

```
## [1] "data.frame"
```

```
class(meta_data$Luminex_id)
```

```
## [1] "character"
```

```
class(meta_data$age)
```

```
## [1] "integer"
```

We now know that our data is stored in R as a `data.frame`. This is a common way to store data and is compatible with many of the functions we will use later in this class.

We also know there are 1000 observations (or rows) of 5 variables (or columns). These variables are:

1. Luminex_id - the individual sample identification number from the luminex plate
2. Luminex_plate - the plate identification number from the luminex plate
3. id - the individual identification number, which is unique for each person, or each row.
4. age - age in years at the time of survey of the individual
5. sex - sex of the individual (female=1, male=2)

We also know the type of each of the five variables. Variables 1-2 are `chr`, which means they are character variables, and variables 3-5 are `int` or integers.

Suppose we would like know which values all the variables in the dataset can take on, and we use the `summary()` function. We can also see what values specific variables can take on using the `summary()` function, including for the variable `Luminex_id`.

```
# this code summarizes all variables in the data frame
# simultaneously
summary(meta_data)
```

```
##   Luminex_id        Luminex_plate            id              age
##  Length:1000        Length:1000        Min.   :   1.0   Min.   :  0.00
##  Class :character   Class :character   1st Qu.: 250.8   1st Qu.: 16.00
##  Mode  :character   Mode  :character   Median : 500.5   Median : 34.50
##                                        Mean   : 500.5   Mean   : 38.75
##                                        3rd Qu.: 750.2   3rd Qu.: 58.25
##                                        Max.   :1000.0   Max.   :100.00
##       sex
##  Min.   :1.000
##  1st Qu.:1.000
##  Median :1.000
##  Mean   :1.495
##  3rd Qu.:2.000
##  Max.   :2.000
```

```
# this code summarizes the variable Luminex_id
summary(meta_data$Luminex_plate)
```

```
##    Length     Class      Mode
##      1000 character character
```

We only know there are 1000 observations that are characters; this is not very informative. We can switch the type of variable from character to factor, which is a good way to deal with categorical variables in R.

```
# this code re-assigns the Luminex plate variable as a
# factor

meta_data$Luminex_plate <- as.factor(meta_data$Luminex_plate)
```

Now, if we try to summarize the variable `Luminex_plate`, we can get information about the values of this variable:

```
summary(meta_data$Luminex_plate)
```

```
##  Plate1 Plate10 Plate11 Plate12 Plate13 Plate14 Plate15 Plate16  Plate2  Plate3
##      64      64      64      64      64      64      64      40      64      64
##  Plate4  Plate5  Plate6  Plate7  Plate8  Plate9
##      64      64      64      64      64      64
```

Now, we can see there are 16 different plates run, all but 1 plate has 64 individual samples run. Only 1 plate (Plate 16) has 40 samples run on it.

We can also make integer or numeric variables into character variables. Let's try this for the sex variable and relabel the variable 1=female and 2=male. We can do this using an ifelse statement that will make values that equal 1 state 'female' and values that equal 2 state 'male'. Labeling or changing the variable class can help analyses downstream.

```
# this code uses an ifelse statement to re-assign the sex
# variable and create a new character sex variable
meta_data$sex1 <- ifelse(meta_data$sex == 1, "Female", ifelse(meta_data$sex ==
    2, "Male", NA))

# now check how the class of the sex variable changed
str(meta_data)
```

```
## 'data.frame':    1000 obs. of  6 variables:
##  $ Luminex_id   : chr  "Unknown1" "Unknown2" "Unknown3" "Unknown4" ...
##  $ Luminex_plate: Factor w/ 16 levels "Plate1","Plate10",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ age          : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ sex          : int  2 2 2 2 1 2 2 2 1 2 ...
##  $ sex1         : chr  "Male" "Male" "Male" "Male" ...
```

More data exploration: mean, median, quantiles, counts and proportions, standard deviation

Base R has built-in functions to calculate key statistics measures. For continuous variables, like age, we can calculate mean, median, quantiles, and standard deviation.

```
mean(meta_data$age)
```

```
## [1] 38.749
```

If any missing values exist in the age variable, then you will get a NA output. To get a summary statistics, we need to either filter out the missing variables or specify that we want to obtain a summary measure *excluding* the missing values. This is accomplished by adding `na.rm=TRUE` in the parenthesis after specifying the variable:

```
mean(meta_data$age, na.rm = TRUE)
```

```
## [1] 38.749
```

We use the same syntax to calculate median, standard deviation, maximum, and minimum of the variable:

```
median(meta_data$age, na.rm = TRUE)  # median of the variable
```

```
## [1] 34.5
```

```
sd(meta_data$age, na.rm = TRUE)  # standard deviation
```

```
## [1] 26.73063
```

```
max(meta_data$age, na.rm = TRUE)  # maximum
```

```
## [1] 100
```

```
min(meta_data$age, na.rm = TRUE)  # minimum
```

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

The `quantile` function allows us to calculate any quantile of interest by specifying the quantile in the `probs` option. We can calculate a single quantile at a time, or multiple. As for the measures above, we will need to explicitly specify that we are excluding missing values (using `na.rm=FALSE`), or the output will return an error.

Let's practice returning 50th percentile (median), 2.5th percentile, and 97.5th percentile:

```
quantile(meta_data$age, probs = c(0.5), na.rm = TRUE)  # return a single quantile -
```

```
##  50%
## 34.5
```

```r
# 50th percentile, or median
quantile(meta_data$age, probs = c(0.025), na.rm = TRUE)  # return a single
```

```
## 2.5%
##    1
```

```r
# quantile - 2.5th percentile
quantile(meta_data$age, probs = c(0.025, 0.5, 0.975), na.rm = TRUE)  # return three
```

```
##  2.5%   50% 97.5%
##   1.0  34.5  92.0
```

```r
# quantiles - 2.5th, 50th (median), and 97.5th
```

For categorical variables, we can tabulate the number of responses by category. Let's look at the variable for Luminex plate number (`Luminex_plate`). We can use function `table()` as follows:

```r
table(meta_data$Luminex_plate)
```

```
##
##  Plate1 Plate10 Plate11 Plate12 Plate13 Plate14 Plate15 Plate16  Plate2  Plate3
##      64      64      64      64      64      64      64      40      64      64
##  Plate4  Plate5  Plate6  Plate7  Plate8  Plate9
##      64      64      64      64      64      64
```

We can also us the table function to calculate the proportion of data that has a particular value by category. Let's apply this to the character `sex1` variable.

```r
table(meta_data$sex1)
```

```
##
## Female   Male
##    505    495
```

```r
prop.table(table(meta_data$sex1))
```

```
##
## Female   Male
##  0.505  0.495
```

Note that by default, this does not show how many missing values there are. It's generally a good practice to either force R to output number of missing values *if there are any such values* (using `useNA="ifany"` option), or to *always* output the number of missing values (even if there are no such values), using `useNA="always"` option.

```r
table(meta_data$sex1, useNA = "ifany")  # display number of missing / NA values
```

```
##
## Female   Male
##    505    495
```

```
# IF THERE ARE ANY IN THE DATA

table(meta_data$sex1, useNA = "always")  # display number of missing / NA values
```

```
##
## Female   Male   <NA>
##    505    495      0
```

```
# at all times (including if there are no missing values)
```

There is also another way to calculate the proportion of individuals that are female or male. One option is to count the number of females, and divide that by number of females and males in the dataset. Note the use of `==` sign, and the use of `|` (which means OR) when counting number of instances when sex is *either* female or male.

```
sum(meta_data$sex1 == "Female")/sum(meta_data$sex1 == "Female" |
    meta_data$sex1 == "Male")
```

```
## [1] 0.505
```

The proportion of households with females is thus 0.505.

In this particular instance, because there are no missing values, we can also use the length of variable as our denominator:

```
sum(meta_data$sex1 == "Female")/length(meta_data$sex1)
```

```
## [1] 0.505
```

We can also look at counts of values of variable across two variables. This is a common type of table in descriptive epidemiology (2x2 table). Let's see what the breakdown of sex is by Luminex plate number. We can do this using the `table()` function. The first variable that we specify will go in the rows of the table. The second variable is the column variable.

```
table(meta_data$Luminex_plate, meta_data$sex1)  # rows = Luminex plate; columns = sex
```

```
##
##           Female Male
##   Plate1      23   41
##   Plate10     33   31
##   Plate11     41   23
##   Plate12     38   26
##   Plate13     39   25
##   Plate14     31   33
##   Plate15     25   39
##   Plate16     13   27
##   Plate2      27   37
##   Plate3      35   29
##   Plate4      30   34
##   Plate5      29   35
##   Plate6      32   32
##   Plate7      34   30
##   Plate8      38   26
##   Plate9      37   27
```

Merge datasets

Sometimes merging datasets is useful so that we can connect the meta data like age and sex to the outcome data like serostatus or median fluorescence intensity (MFI - as output from the Luminex machine). First, we'll load the data.frame containing the serology outcome data. Then we'll join the new serology data.frame with the meta_data data.frame we loaded earlier using the individual identification number (id) to join the data.

```
# this code reads in data
sero_data <- read.csv(paste(my_path, "Data/Training serostatus df.csv",
    sep = "/"))

# let's make the serostatus variables characters
sero_data$CHIKV_serostatus <- as.character(sero_data$CHIKV_serostatus)
sero_data$DENV_serostatus <- as.character(sero_data$DENV_serostatus)

# this code joins the sero_data with the meta_data by the
# individual identification number
merged_data <- merge(meta_data, sero_data, by = "id")

# this is a left join: keep all meta_data rows, add
# sero_data where id matches
merged_data <- merge(meta_data, sero_data, by = "id", all.x = TRUE)

head(merged_data)
```

```
##   id Luminex_id Luminex_plate age sex   sex1 DENV_serostatus CHIKV_serostatus
## 1  1   Unknown1        Plate1   0   2   Male               0                0
## 2  2   Unknown2        Plate1   0   2   Male               0                0
## 3  3   Unknown3        Plate1   0   2   Male               0                0
## 4  4   Unknown4        Plate1   0   2   Male               0                0
## 5  5   Unknown5        Plate1   0   1 Female               0                0
## 6  6   Unknown6        Plate1   0   2   Male               0                0
##   DENV_MFI CHIKV_MFI
## 1      310       155
## 2      486       387
## 3      713       349
## 4      215       206
## 5      396       146
## 6      441       253
```

"Wide" versus "long" data sets

In a **wide** data set, each variable has its own column (e.g., one column per time point), common in spreadsheet-style data or for visualization (e.g., plots), and is easier to read but harder to manipulate in some R functions. In a **long** data set, each observation is a row and observations are sometimes stacked, variables like 'time' and 'value' are separate columns. This approach is preferred for some functions (e.g., ggplot2) and is easier for grouped operations and reshaping.

Our current merged_data data.frame is a wide data set. To transform this data.frame into a long data.frame so that we can create a column for antigen (DENV, CHIKV) and outcome type (serostatus, MFI).

```r
# Reshape serostatus to long format
df_long <- reshape(merged_data, varying = list(c("DENV_serostatus",
    "CHIKV_serostatus"), c("DENV_MFI", "CHIKV_MFI")), v.names = c("serostatus",
    "MFI"), timevar = "antigen", times = c("DENV", "CHIKV"),
    idvar = "id", direction = "long")

head(df_long)
```

```
##        id Luminex_id Luminex_plate age sex   sex1 antigen serostatus MFI
## 1.DENV  1   Unknown1        Plate1   0   2   Male    DENV          0 310
## 2.DENV  2   Unknown2        Plate1   0   2   Male    DENV          0 486
## 3.DENV  3   Unknown3        Plate1   0   2   Male    DENV          0 713
## 4.DENV  4   Unknown4        Plate1   0   2   Male    DENV          0 215
## 5.DENV  5   Unknown5        Plate1   0   1 Female    DENV          0 396
## 6.DENV  6   Unknown6        Plate1   0   2   Male    DENV          0 441
```

Visualizing data

Now let's cover what a package is. A package in R is a collection of functions, data, and documentation bundled together to extend R's capabilities. It adds tools for specific tasks like data analysis, visualization, or modeling. You install a package once (e.g., `install.packages("ggplot2")`) and load it with `library(ggplot2)` to use its functions.

First, let's install the following packages. You only need to install packages once (with the `install.packages()` function), but they need to be loaded each time you start R to use them.

```r
# install packages that will be needed; the
# install.packages() function can be used
install.packages("ggplot2")
install.packages("dplyr")
install.packages("tidyr")
install.packages("epitools")
install.packages("MASS")
install.packages("pROC")
install.packages("mclust")
install.packages("devtools")

# install the flexfit package to use the devtools package
# and install the flexfit package accurately you need to
# have R version 4.0 or higher this package is not on CRAN
# so we instead install it from github (using the
# install_github function). This function is in the
# devtools package, so we first load devtools (using the
# library function)
library(devtools)
install_github("EPPIcenter/flexfit")
```

Next, we can load the other packages we've installed using the library function

```r
# the library function loads each package
library(ggplot2)
library(dplyr)
```

```r
library(tidyr)
library(epitools)
library(MASS)
library(pROC)
library(mclust)
```

Here we will learn how to make plots using the `ggplot2` package. While all plots can be made using base R (not using the ggplot2 package), we will make different types of common plots using the `ggplot2` package. Here, you'll learn how to make and interpret the following types of plots, including: scatterplot, boxplot, bar graph, and histogram.

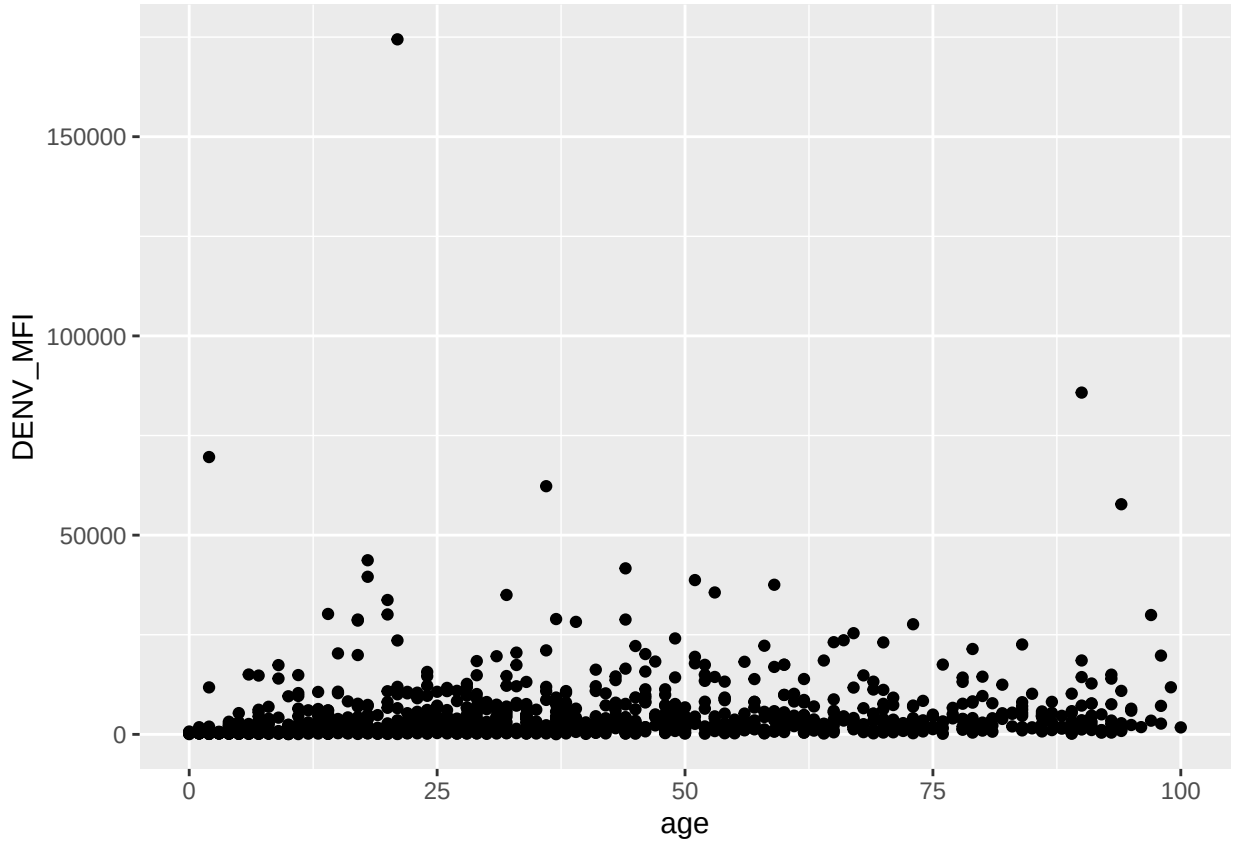
According to the developers of `ggplot2`: *It's hard to succinctly describe how `ggplot2` works because it embodies a deep philosophy of visualisation. However, in most cases you start with `ggplot()`, supply a dataset and aesthetic mapping (with `aes()`). You then add on layers (like `geom_point()` or `geom_histogram()`), scales (like `scale_colour_brewer()`), faceting specifications (like `facet_wrap()`) and coordinate systems (like `coord_flip()`).* Reference: https://ggplot2.tidyverse.org/.

Below, we provide the code to produce different plot types using `ggplot2`. This is by no means a comprehensive summary! There are **many** online resources for learning how to make `ggplot2` figures, including the reference above.

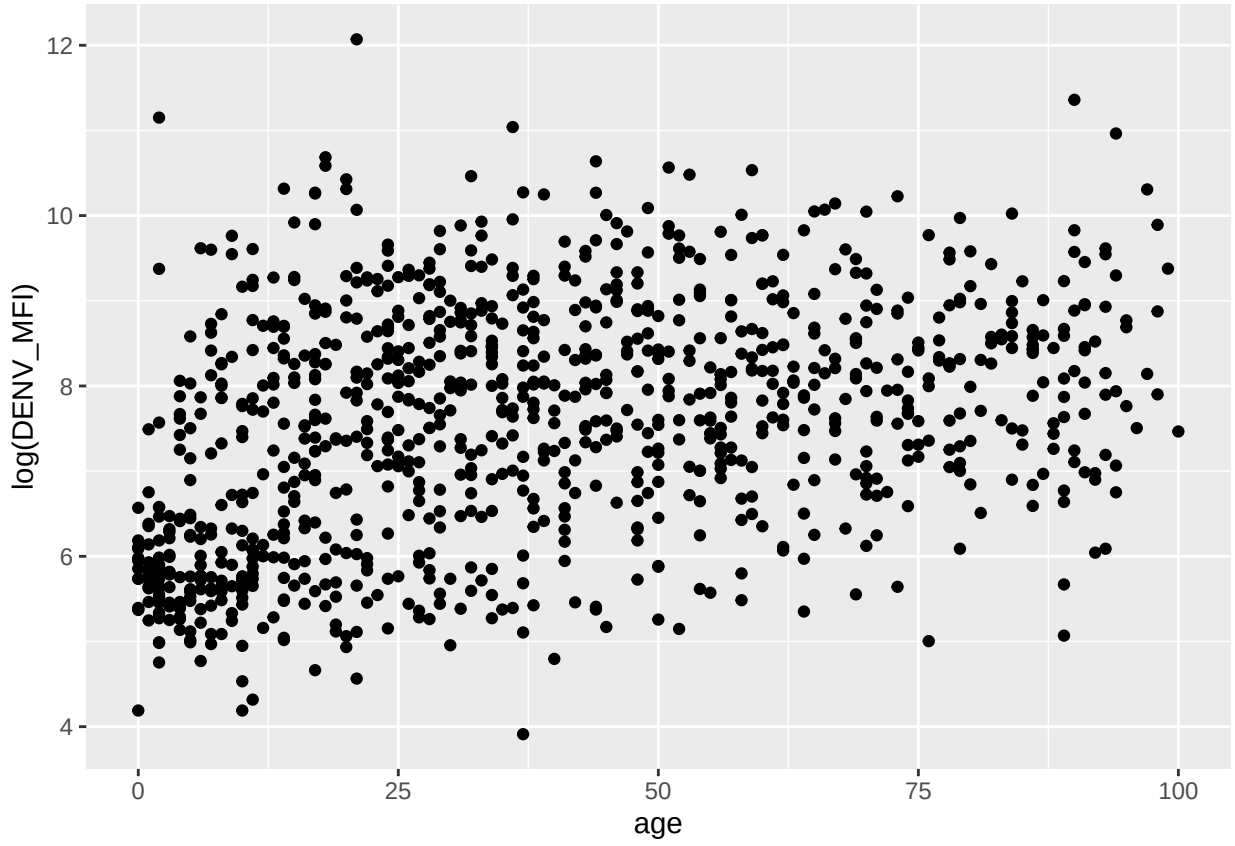
1. Scatterplot

Scatterplots are typically used to look at two numeric variables.

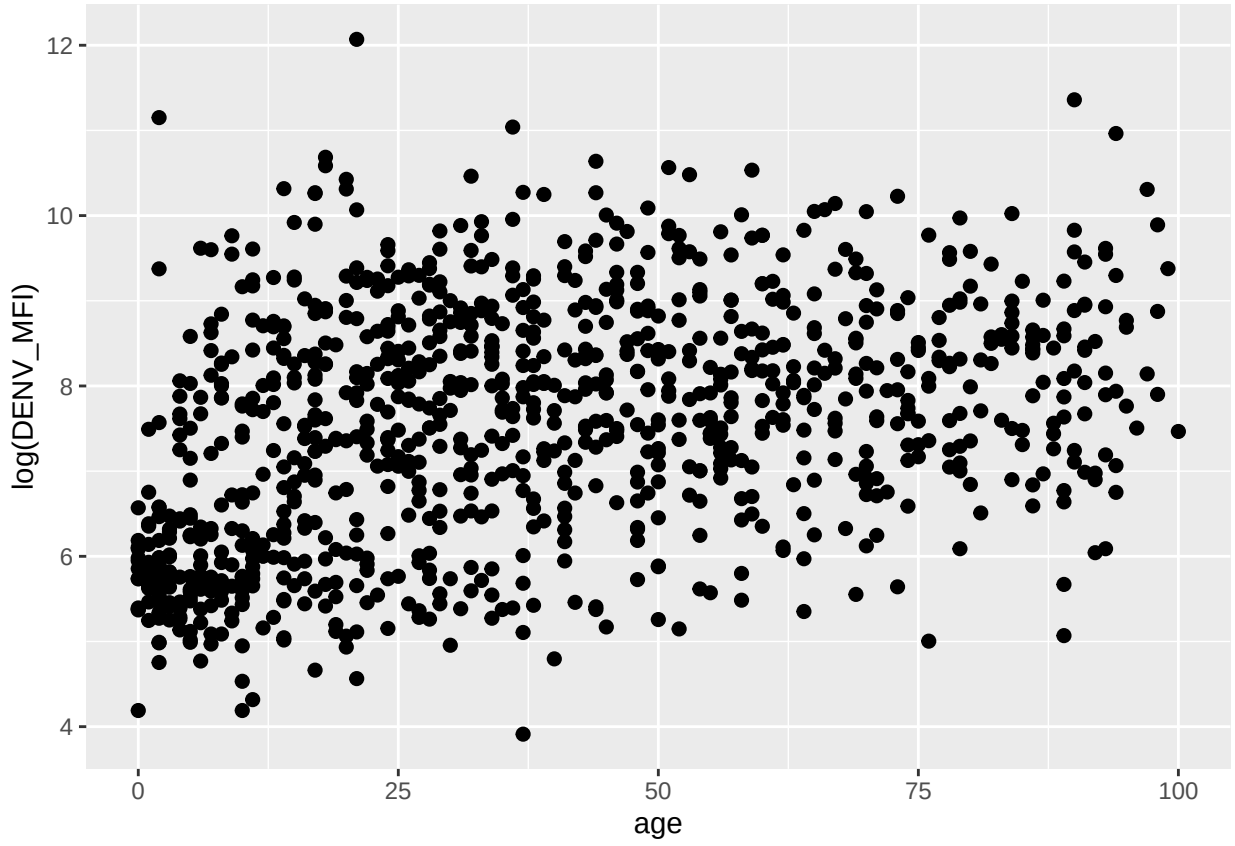
```r
# plot age vs. DENV MFI
ggplot(data = merged_data, aes(x = age, y = DENV_MFI)) + geom_point()
```



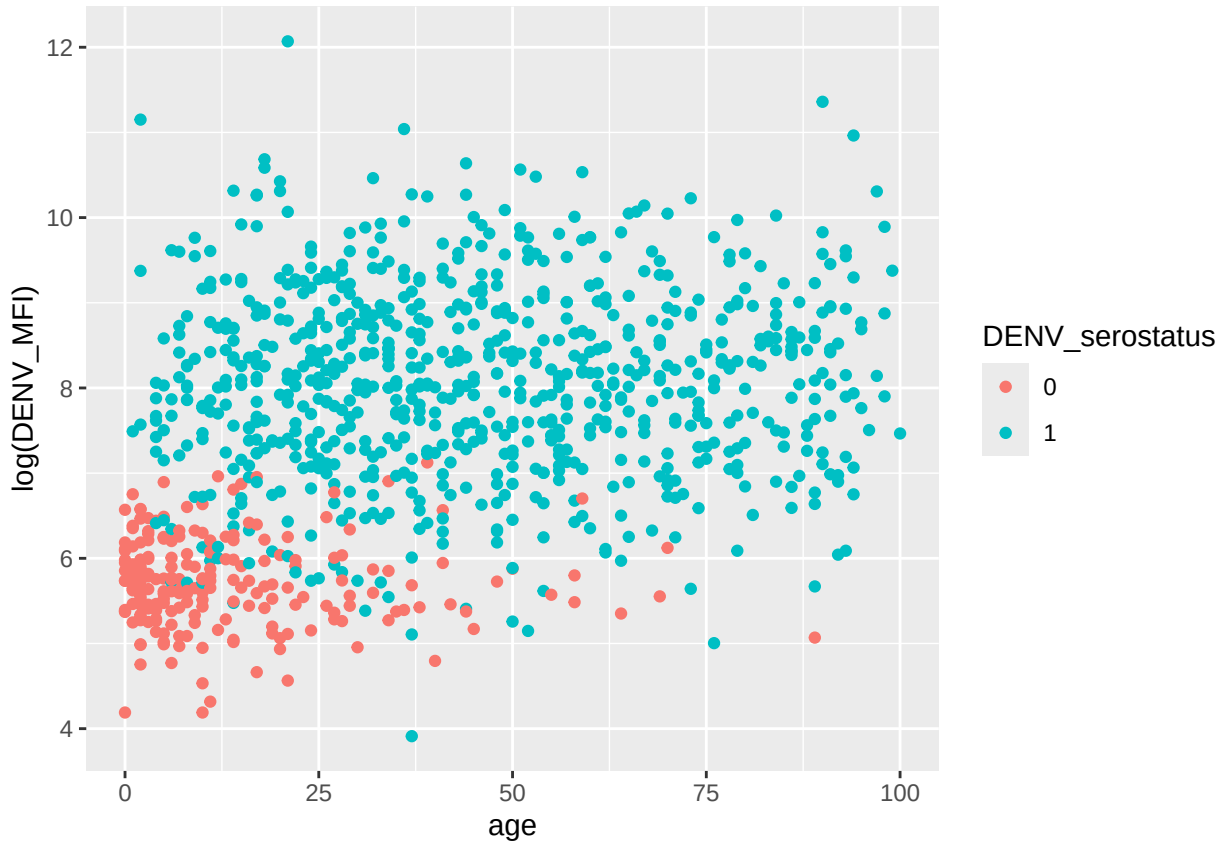

```
ggplot(data = merged_data, aes(x = age, y = log(DENV_MFI))) +
    geom_point()
```



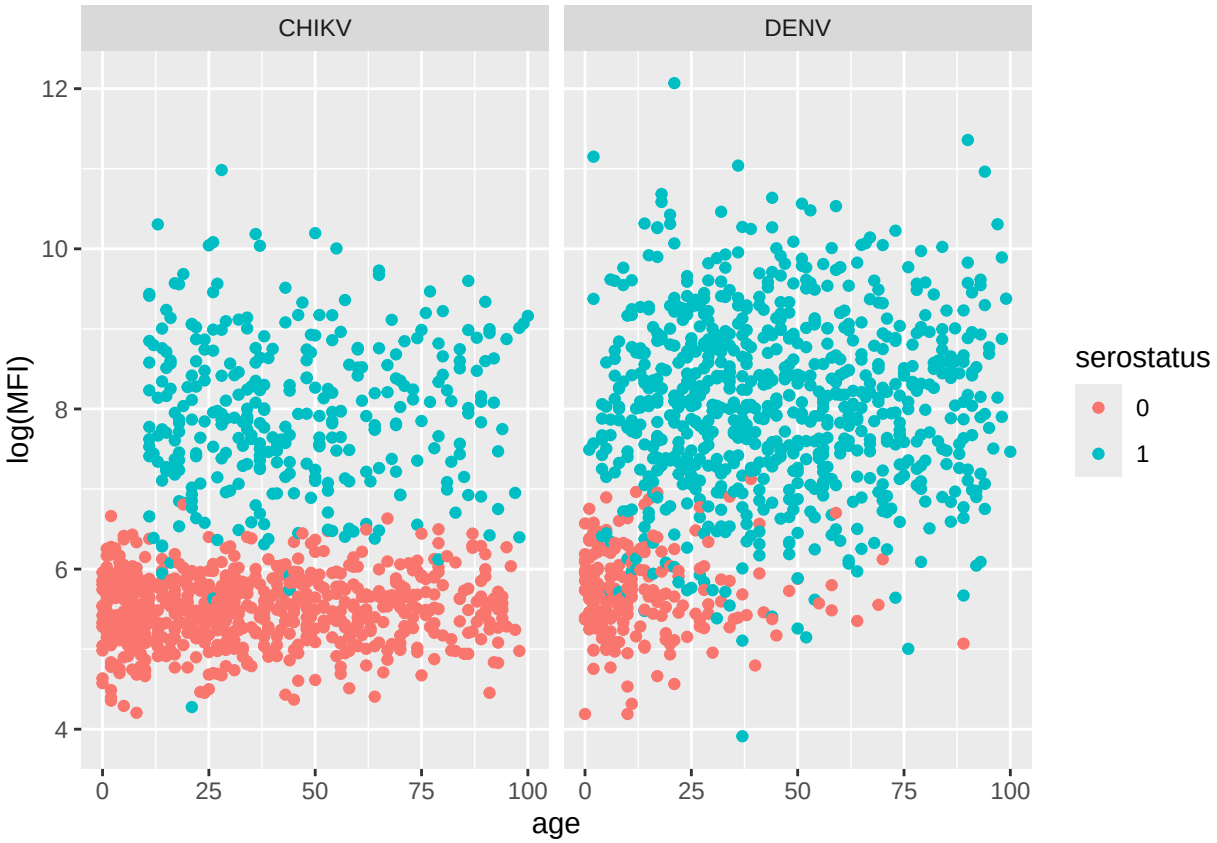

```
ggplot(data = merged_data, aes(x = age, y = log(DENV_MFI))) +
    geom_point(size = 2)
```




```
ggplot(data = merged_data, aes(x = age, y = log(DENV_MFI))) +
    geom_point(aes(colour = DENV_serostatus))
```



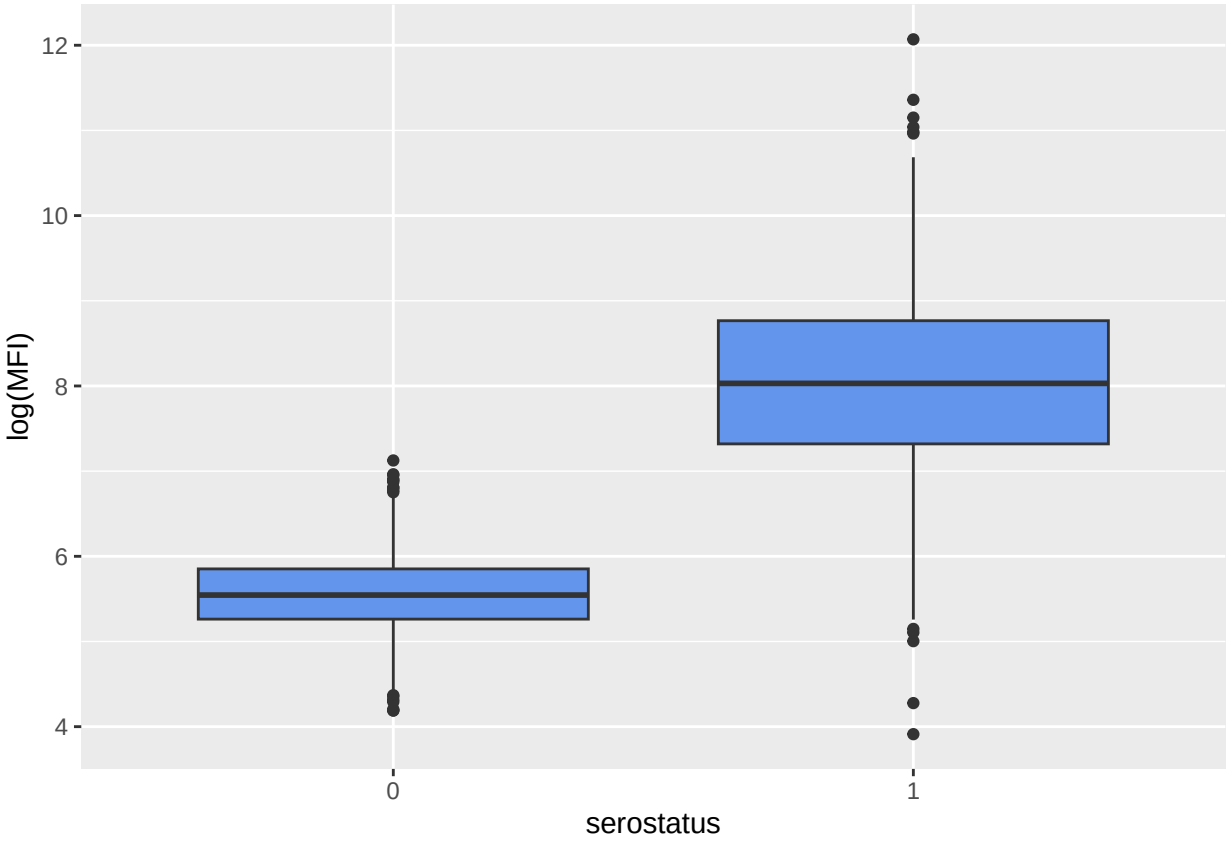

```
# we can also use the long data frame to plot the same
# scatterplot by antigen side by side
ggplot(data = df_long, aes(x = age, y = log(MFI))) + geom_point(aes(colour = serostatus)) +
    facet_wrap(. ~ antigen)
```



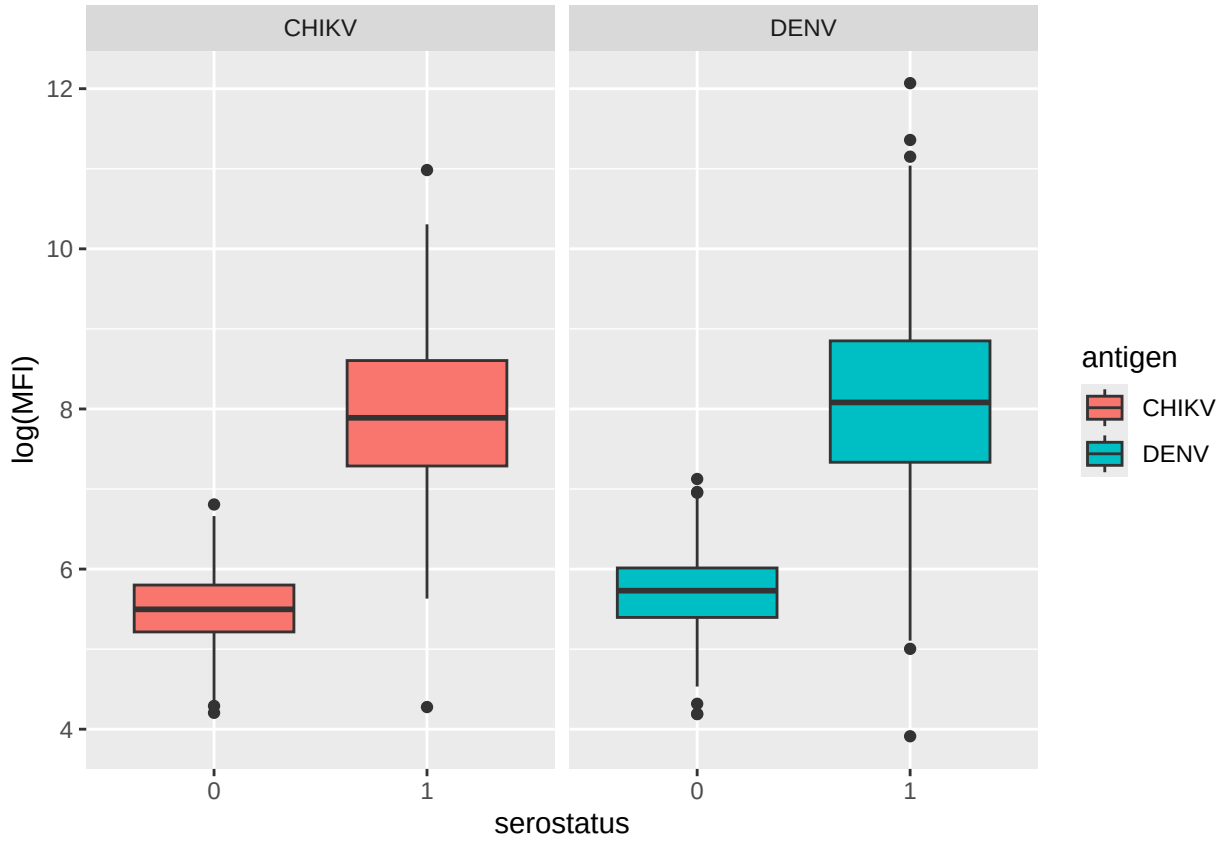
2. Boxplot

Boxplots are typically used to look at the distribution of a group, or of several groups.

```
ggplot(data = df_long, aes(x = serostatus, y = log(MFI))) + geom_boxplot(fill = "cornflowerblue")
```



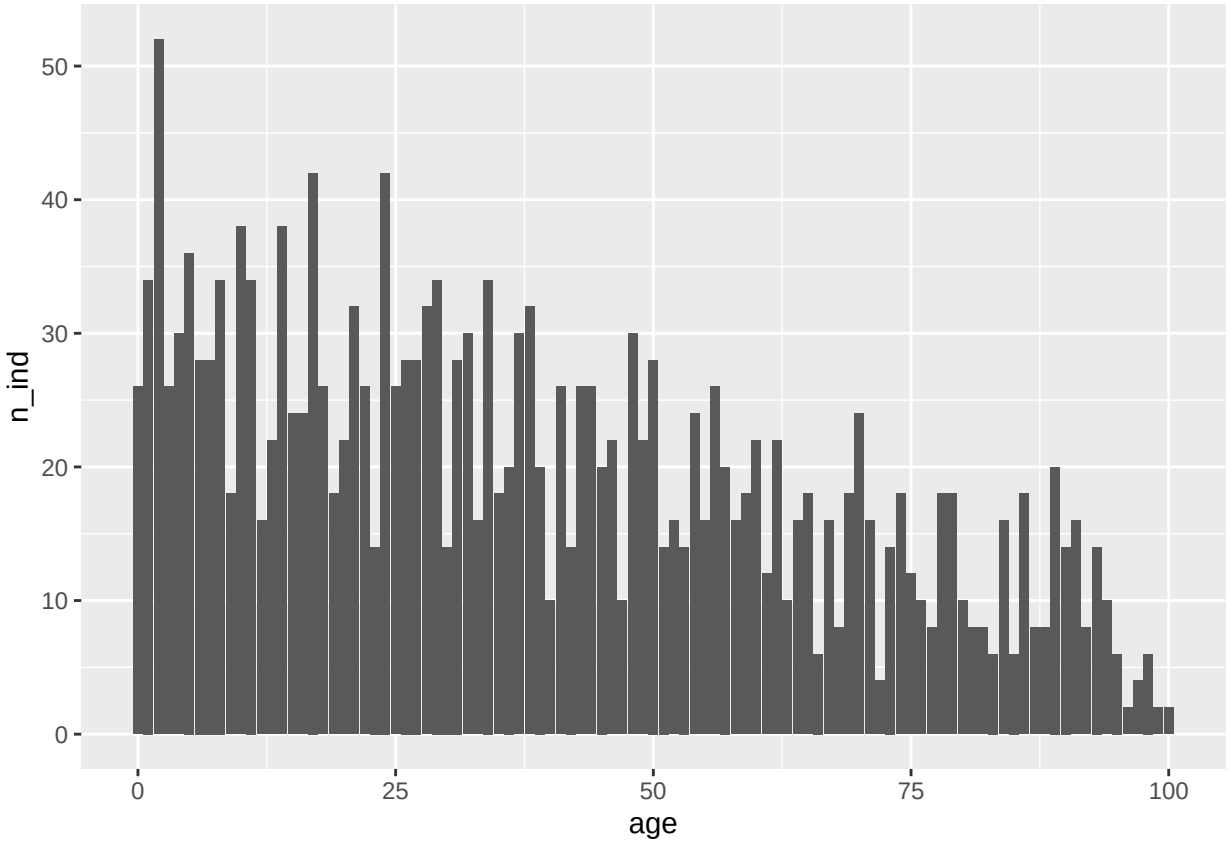

```
ggplot(data = df_long, aes(x = serostatus, y = log(MFI))) + geom_boxplot(aes(fill = antigen)) +
    facet_wrap(. ~ antigen)
```



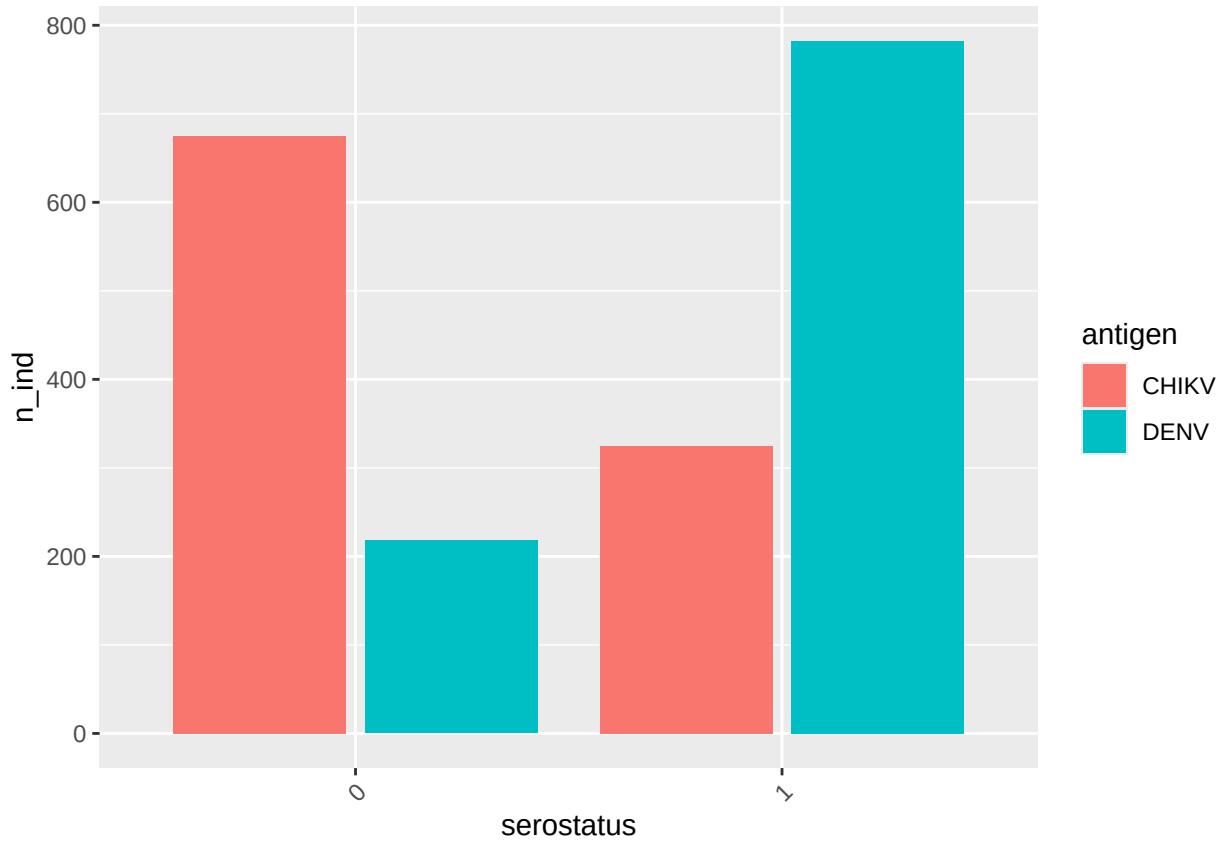
3. Bar graph

Bar graphs are typically used to look at the relationship between a numeric and a categorical variable.

```
# 1 categorical variable aggregate count of individuals by
# age
aggregate_data <- aggregate(id ~ age, data = df_long, FUN = length)
# rename the count column
names(aggregate_data)[2] <- "n_ind"
# plot
ggplot(data = aggregate_data, aes(x = age, y = n_ind)) + geom_col()
```



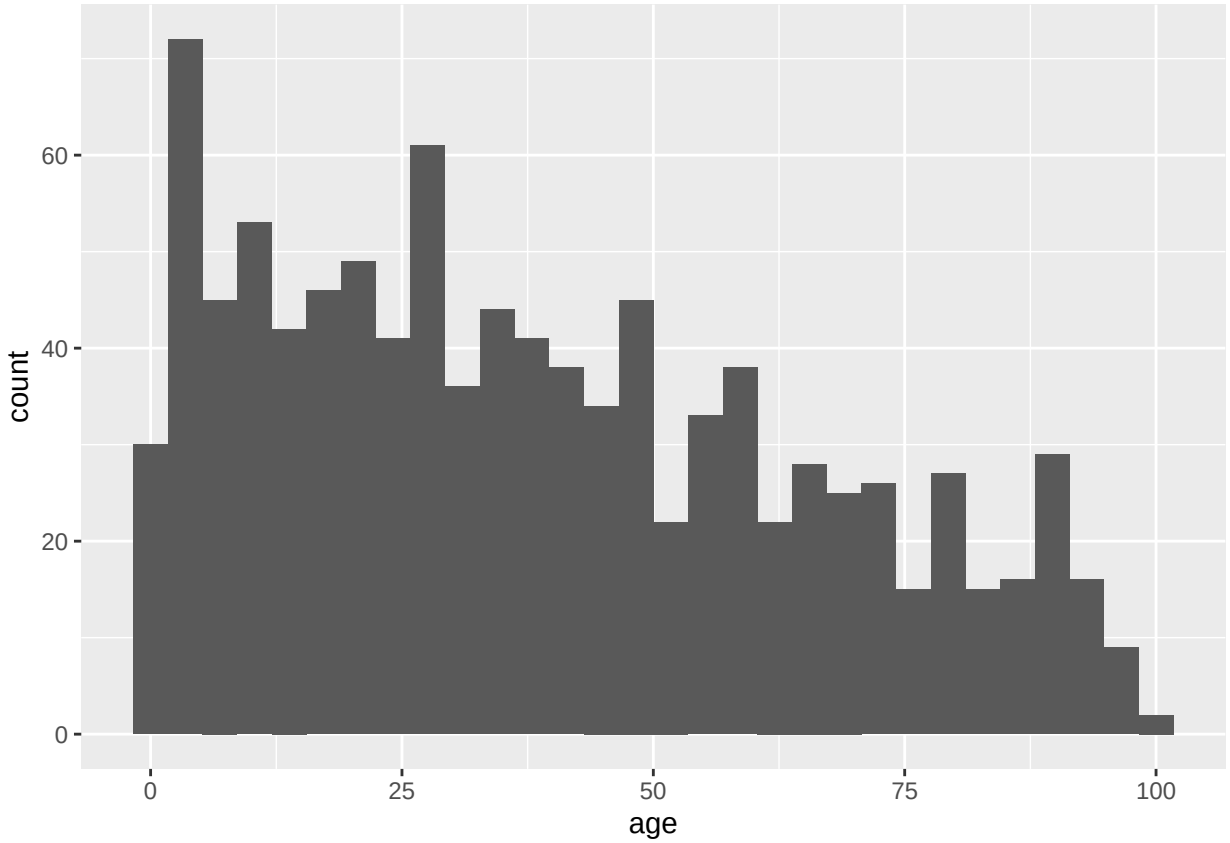

```r
# 2 categorical variables aggregate count of individuals by
# serostatus and antigen
aggregate_data <- aggregate(id ~ serostatus + antigen, data = df_long,
    FUN = length)
# rename the count column
names(aggregate_data)[3] <- "n_ind"
# plot
ggplot(data = aggregate_data, aes(x = serostatus, y = n_ind,
    fill = antigen)) + geom_col(position = "dodge2") + theme(axis.text.x = element_text(angle = 45,
    hjust = 1))
```



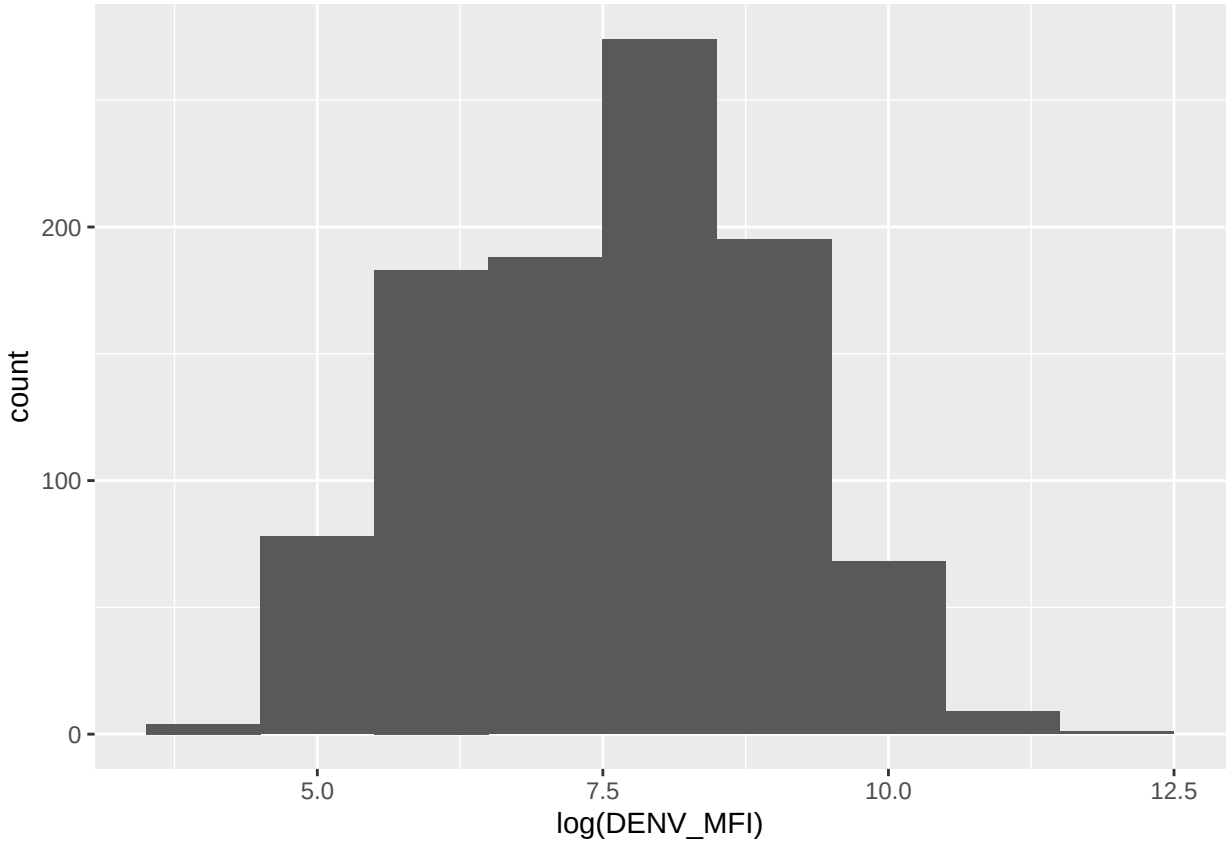
4. Histogram

A histogram is typically used to look at the distribution of one numeric variable.

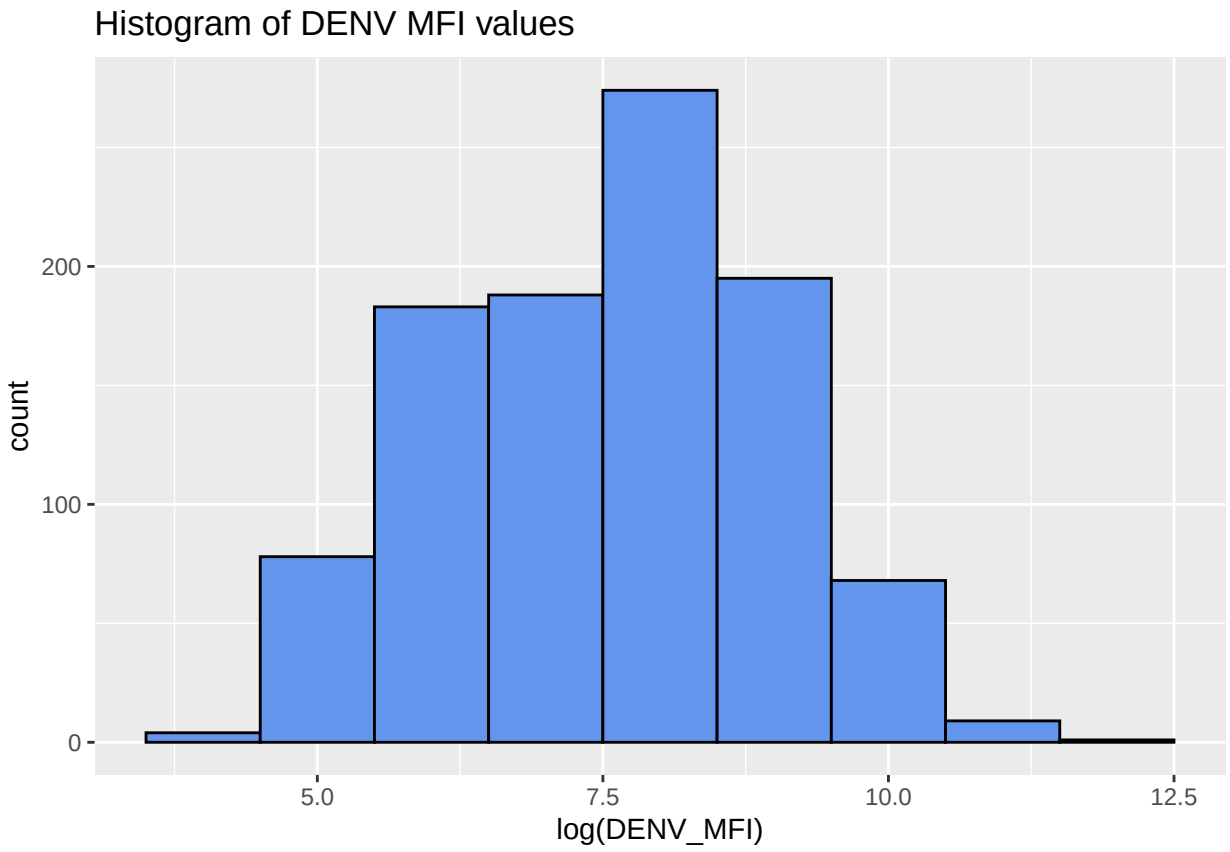
```
ggplot(merged_data, aes(x = age)) + geom_histogram()
```




```
ggplot(merged_data, aes(x = log(DENV_MFI))) + geom_histogram(binwidth = 1)
```




```
ggplot(merged_data, aes(x = log(DENV_MFI))) + geom_histogram(binwidth = 1,
    fill = "cornflowerblue", colour = "black") + ggtitle("Histogram of DENV MFI values")
```



Using the source function

The `source()` function in R runs a script file as if you typed its contents into the console. It's used to load and execute R code from an external file. This is often used to systematically load a set of necessary packages or functions needed for your Rmd script or project.

The below example R script (`my_script.R`) defines the `square()` function: `square <- function(x) x^2`. By executing the `my_script.R` file using the `source()` function, we are able to load and use the `square()` function. Source files are generally run at the beginning of every project to load all the necessary packages and functions needed.

```
# try to use the square function before executing the
# my_script.R file square(4)

# execute the my_script.R file using the source() function
source(paste(my_path, "Source/my_script.R", sep = "/"))

# try to use the square function now
square(4)
```

```
## [1] 16
```