# Tutorial 6: Functions

Leah Brooks

February 24, 2020

Mid-way through the semester, we take a break from data visualization to teach you how to create your own functions in R.

Most of the commands you use in R are functions themselves, so you already know how to use a function (even if you didn't realize it). Today you will learn how to make your very own R function.

Make sure to write today's tutorial in an R script, rather than R Markdown. In addition, put

```
rm(list=ls())
```

at the beginning of your code. If strange things are happening in your code, consider running it from the beginning, and the above command will clear everything in R's memory. I tell you write a R script so that you can more easily keep track of what functions are defined where and apply to what.

As to when to use a function, in the immortal words of Hadley Wickham, "You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code)" (see this here).

The ability to create functions is one of the most powerful tools that statistical programming gives you. Good programming relies heavily on functions. Cut and paste is prone to errors. Furthermore, while cut and paste always seems easiest when you need to do things once, my experience with programming suggests that your first graph is never your last graph. When you need to make modifications, particularly ones that are the same across a set of graphs, you will be happy you chose to make a function.

This class gives you an initial introduction to functions. If you'd like to deeper, I recommend Hadley Wickman's Advanced R notes.

## A. What is a function?

In this section, we define what a function is and explain its parts.

The example below shows the bones of any basic function. The first part `function.name` is the name of the function. You can choose any name you'd like for the function.

```
function.name <- function(arg1, arg2){
  # stuff your function does
}
```

Beware, however, that if you use the name of an existing function in R, such as `plot`, when you call `plot`, you will get your new function, instead of R's usual 'plot'. Bottom line: stay away from existing names if at all possible. You can ask R whether it has any functions of a given name by typing

```
? sum
```

```
## starting httpd help server ... done
```

When you type the above, a help window pops up – this means there is an existing function named sum. So don't use `sum`! What about `dogs`?

```
? dogs
```

```
## No documentation for 'dogs' in specified packages and libraries:
## you could try '??dogs'
```

R tells us it can't find any functions called `dogs`, and you are good to go.

Every function also needs the word `function` – this is the same across all functions, and tells R that you are making a function. You cannot change this word.

The variables `arg1` and `arg2` are the inputs to the new function. They define what you can put into the function.

Once you've given R values for `arg1` and `arg2`, R undertakes the commands inside the curly brackets. These commands are known as the "function body."

# B. A first function

## B.1. A very first function

Let's begin with a very simple function that takes one value to the power of the other. As we construct it, this function has arguments `x` and `y`. Whatever value you give R for `x`, it will take it to the power `y`. We call this new function `summer`.

Before I define it, I check to see if a function by this name already exists.

```
? summer
```

```
## No documentation for 'summer' in specified packages and libraries:
## you could try '??summer'
```

No function exists by this name. This check is not required, but it is good practice, since you can create trouble by naming your function with a name that already exists as a R function.

Now let's define the function:

```
summer <- function(x,y){
  x^y
}
```

In the `summer` function, the arguments are `x` and `y`. The body is $x^y$.

Having defined the function, we'd now like to call it. The most clear way of calling a function is to associate each argument with its value, as in

```
summer(x=1,y=2)
```

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

This call should look familiar. You've been using functions all semester. Now you're writing one of your own.

The summer function returns a value of 1. Note that

$$1^2 = 1$$

, so all is good. as it should.

Alternatively, you can make the same call by writing

```r
summer(1,2)
```

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

This works, but is bad practice. Code like this is hard to decipher and debug.

Now try

```r
summer(x = 2, y = 1)
```

```
## [1] 2
```

Note that this does not yield the same outcome. Homework question 1 asks you why.

Finally, the call below does not work at all. We've specified nothing for `y`, and all arguments are mandatory unless a default value is specified (which we'll learn how to do in a bit).

```r
summer(x = 1,)
```

## B.2. A Slightly More Complicated Function

The example in B.1. was so simple that you might wonder why we bother with functions. Let us start working toward something slightly more complicated.

Suppose you'd like to know the marginal tax rate for a specific income. Maybe you'd like to automatically print a chart title that says what the marginal tax rate of the mean income is, for example (so you can update the picture without looking up the marginl tax rate each time).

The marginal tax rate is the tax you pay on your last dollar of earnings. In the US system, rates are progressive, so that higher incomes pay higher tax rates. In other words, your first \$$x$ of income is taxed at rate $a$. Income greater than $x$, but less that $y$ is taxed at rated $b$, where $b < a$. The rate associated with your "tax bracket" is your marginal tax rate.

A starting point for this kind of work is a function that delivers a marginal tax rate based on an input income. We do this below, with a thanks to Bankrate for helpful marginal tax rates (for single people; page also has married, if you're curious).

```r
single.marg.tax.rate <- function(income){
  mr <-
    ifelse(income < 9325,0.10,
      ifelse(income < 37950, 0.15,
        ifelse(income < 91500, 0.25,
          ifelse(income < 191650, 0.28,
            ifelse(income < 416700, 0.33,
              ifelse(income < 418400, 0.35, 0.396))))))
  print(paste0("marginal tax rate for income ",income, " is ", mr))
}
```

This function takes the argument `income` and finds the bracket into which that income fits. It then outputs the tax rate and income in a print statement.

Give it a try!

Here's my first attempt:

```r
single.marg.tax.rate(income = 10000)
```

```
## [1] "marginal tax rate for income 10000 is 0.15"
```

This seems to find the right marginal tax rate, according to the Bankrate page.

Does it work for other incomes?

```
single.marg.tax.rate(income = 50000)
```

```
## [1] "marginal tax rate for income 50000 is 0.25"
```

```
single.marg.tax.rate(income = 500000)
```

```
## [1] "marginal tax rate for income 5e+05 is 0.396"
```

### B.3. A function in a separate file

Sometimes you build a function that you would like to use in multiple programs. If you'd like to do this, you can put your R function in its own .R file.

For example, I put a variant of the `summer` function we created above in a separate new R file, and saved it as summer2_func.R (don't use dots in the file name, except for the .R extension). My file looks like

```
summer2 <- function(x,y,z){
  x^y + z
}
```

I can now call this function and get a result:

```
source("H:/pppa_data_viz/2018/tutorials/lecture12/summer2_func.R")
summer2(x = 5, y = 3, z = 1)
```

```
## [1] 126
```

## C. Other Function Argument Basics

In this section, we discuss more features of function arguments: how they work, how you name them, and setting defaults.

### C.1. More on function arguments

Suppose that we would like to run the function `summmer2`, but we don't want to add anything to $x^y$ (what the z variable does).

Let's try to run it two ways:

```
summer2(x = 5, y = 3, z = 0)
```

```
## [1] 125
```

This one runs, and properly gives us $5^3 + 0 = 45$.

Now let's try

```
summer2(x = 5, y = 3)
```

This one should give an error message. Why? R is trying to evaluation `x^y + z`, but can't find a value for `z` – so it breaks.

To make sure you understand why R is breaking, look at the following example:

```
summer3 <- function(x,y,z){
  x^y
}

summer3(x = 5, y = 3)
```

```
## [1] 125
```

This does not generate an error. The homework asks you why, even without a value for `z`.

Note that you can also put R objects into a function call. Let's let `natl.mn.income` be $53,719, which is the 2014 US mean income. We'll then use this object in the `summer` function call.

```
natl.mn.inc <- 53719
summer(x = natl.mn.inc,y = 1)
```

```
## [1] 53719
```

### C.2. Defaults

One way to avoid the error we have in section C.1. from calling `summer2(5,3)` would be to set a default value for z. Let us set the default value for `z` as 0.

```
summer4 <- function(x, y, z = 0){
  x^y + z
}

summer4(x = 5, y = 3)
```

```
## [1] 125
```

Now if you don't specify a value for `z`, R assumes that it is zero. If you do specify a value, that value replaces zero.

### C.3. Calling the right type of variable

You should also be careful that the type of input argument you give to the function matches how the function will use that argument.

The text below yields an error:

```
summer(x = "fred", y = "ted")
```

Explain why in homework question 3.

## D. What the function outputs

Sometimes you'd like a function to just calculate something and print the result to the screen. Other times, it's helpful to have a function return something to you that you can use in the rest of the code. For example, suppose we'd like to use the marginal tax rate that the function `single.marginal.tax.rate` creates.

Can I work with this new marginal tax rate in the rest of the code?

```
single.marg.tax.rate(income = 500000)
```

```
taxes.paid <- (500000 - 418401)*mr
```

This second command gives an eror. Why is this? Didn't we just create `mr` in this function? Why doesn't this object now exist?

This brings up a key element of functions. Everything that you create in the function is "local" to the function unless you specifically tell R you want to take it out of the function. To tell R to take something out of the function, you need to "return" the value. "Returning" the value means taking something that exists just in the function and making it exist in the rest of the code as well. We will learn how to do this in this section.

As an aside, when you write `mr` in plain R code, R will print the value of `mr`. When you write `mr` inside a function, R doesn't print the value of `mr` to the console. To see the value of `mr`, you need to write `print(mr)`. I also illustrate this point in the code below.

Running our original function again, we see that just running it in plain code delivers the marginal tax rate to the console.

```
single.marg.tax.rate(income = 500000)
```

```
## [1] "marginal tax rate for income 5e+05 is 0.396"
```

Making the function deliver something to a new object prints nothing, but gives no error.

```
out <- single.marg.tax.rate(income = 500000)
```

```
## [1] "marginal tax rate for income 5e+05 is 0.396"
```

Let's look at the new object:

```
out
```

```
## [1] "marginal tax rate for income 5e+05 is 0.396"
```

It's a text string! Which is the last thing the function did.

If we want the marginal tax rate as a number, we need to modify the function. Let's make a new function called `single.marg.tax.rate.v2`.

```
single.marg.tax.rate.v2 <- function(income){
  mr <-
    ifelse(income < 9325,0.10,
           ifelse(income < 37950, 0.15,
                  ifelse(income < 91500, 0.25,
                         ifelse(income < 191650, 0.28,
                                ifelse(income < 416700, 0.33,
                                       ifelse(income < 418400, 0.35, 0.396))))))
  print(paste0("marginal tax rate for income ",income, " is ", mr))
  mr
}
```

Let's now run the function, and also put its output into `out2`.

```
single.marg.tax.rate.v2(income = 500000)
```

```
## [1] "marginal tax rate for income 5e+05 is 0.396"
```

```
## [1] 0.396
```

```
out2 <- single.marg.tax.rate.v2(income = 500000)
```

```
## [1] "marginal tax rate for income 5e+05 is 0.396"
```

```
out2
```

```
## [1] 0.396
```

This output is now a number, since listing `mr` is the last thing the function did. We can now use `out2` in our main program. Below I calculate the taxes paid, for someone earning $500,000, on income above $418,401. (This person does not pay the marginal tax rate of 39.6 on all of their income – just income above $418,401. For income between $416,701 and $418,401, the person pays a rate of 35 percent. Below that – from about $200,000 to $400,000 they pay 33 percent.)

```
taxes.paid <- (500000 - 418401)*out2
taxes.paid
```

```
## [1] 32313.2
```

## E. Functions for graphs and maps

To illustrate the value of functions, let's automate some repetitive operations. A good practice when building a function is to write out one instance of what you'd like to do in plain code. Then work on the function that automates it. You can write the function directly, but this is best for when you are quite comfortable with functions.

Begin by loading the csv with crashes in DC, found here. Download the spreadsheet, read it into R, and find out what variables this dataframe has.

```
crashes <- read.csv("H:/pppa_data_viz/2020/tutorial_data/tutorial06/20200226_Crashes_in_DC.csv")
names(crashes)
```

```
##  [1] "ï..X"                       "Y"
##  [3] "OBJECTID"                   "CRIMEID"
##  [5] "CCN"                        "REPORTDATE"
##  [7] "ROUTEID"                    "MEASURE"
##  [9] "OFFSET"                     "STREETSEGID"
## [11] "ROADWAYSEGID"               "FROMDATE"
## [13] "TODATE"                     "MARID"
## [15] "ADDRESS"                    "LATITUDE"
## [17] "LONGITUDE"                  "XCOORD"
## [19] "YCOORD"                     "WARD"
## [21] "EVENTID"                    "MAR_ADDRESS"
## [23] "MAR_SCORE"                  "MAJORINJURIES_BICYCLIST"
## [25] "MINORINJURIES_BICYCLIST"    "UNKNOWNINJURIES_BICYCLIST"
## [27] "FATAL_BICYCLIST"            "MAJORINJURIES_DRIVER"
## [29] "MINORINJURIES_DRIVER"       "UNKNOWNINJURIES_DRIVER"
## [31] "FATAL_DRIVER"               "MAJORINJURIES_PEDESTRIAN"
## [33] "MINORINJURIES_PEDESTRIAN"   "UNKNOWNINJURIES_PEDESTRIAN"
## [35] "FATAL_PEDESTRIAN"           "TOTAL_VEHICLES"
## [37] "TOTAL_BICYCLES"             "TOTAL_PEDESTRIANS"
## [39] "PEDESTRIANSIMPAIRED"        "BICYCLISTSIMPAIRED"
## [41] "DRIVERSIMPAIRED"            "TOTAL_TAXIS"
## [43] "TOTAL_GOVERNMENT"           "SPEEDING_INVOLVED"
## [45] "NEARESTINTROUTEID"          "NEARESTINTSTREETNAME"
## [47] "OFFINTERSECTION"            "INTAPPROACHDIRECTION"
## [49] "LOCATIONERROR"              "LASTUPDATEDATE"
## [51] "MPDLATITUDE"                "MPDLONGITUDE"
## [53] "MPDGEOX"                    "MPDGEOY"
## [55] "BLOCKKEY"                    "SUBBLOCKKEY"
## [57] "FATALPASSENGER"              "MAJORINJURIESPASSENGER"
## [59] "MINORINJURIESPASSENGER"      "UNKNOWNINJURIESPASSENGER"
```

Your file may differ slightly from mine if it has been updated since February 26, 2020. My version says it was last modified December 10, 2019.

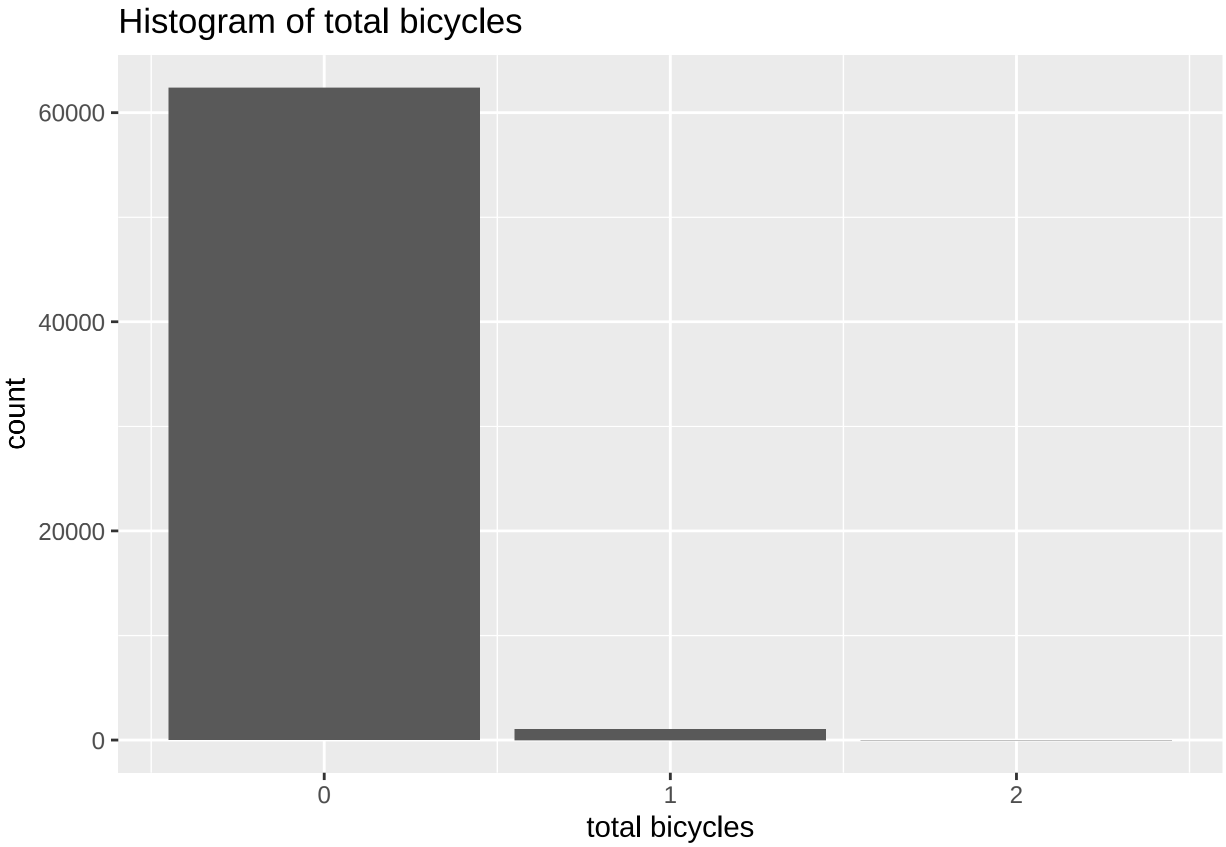
Suppose we first want to get a sense of how many crashes with bicycles there are. To do this, I can look at the distribution of this variable with `summary()` and make a histogram with `ggplot`. Let's do both.

```
library(ggplot2)
summary(crashes$TOTAL_BICYCLES)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.00000 0.00000 0.01722 0.00000 2.00000
```

```
ggplot() +
    geom_histogram(data = crashes,
                   mapping = aes(TOTAL_BICYCLES),
                   stat="count") +
    labs(title = "Histogram of total bicycles",
         x = "total bicycles")
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



But this is a new dataset, and we'd like to explore a bunch of variables. How can we do this efficiently for a variety of variables? Let's write a function that reports on the distribution of the variable and then makes a histogram.

The argument of the function below is `var`, which I intend to be the variable name. Unfortunately, we can't refer to the input as `crashes$var`, since R will go look for a variable in `crashes` named `var` and get mad when it can't find it. To get around this, I do a few things

- When I call the function, I put the variable name in quotes. This makes this text that R can evaluate, rather than an object it will try to look for
- Inside the function, I make the character string "`crashes$[variable name]`" and assign it to `vname`
- To use either `var` or `vname`, I need to tell R to evaluate these strings. I can do this by saying to R `eval(parse(text=vname))` or `eval(parse(text=var))`. This tells R to parse a text string (`parse(text=)`); not entirely sure why this part is needed, but it's definitely needed) and then evaluate it (`eval()`). The evaluating means read the text string as an R object.

If you want to get a sense of why the last part is important, try running the function below without the `eval()` and `parse()` bits, and note the error messages.

For the current function, I create it as below.

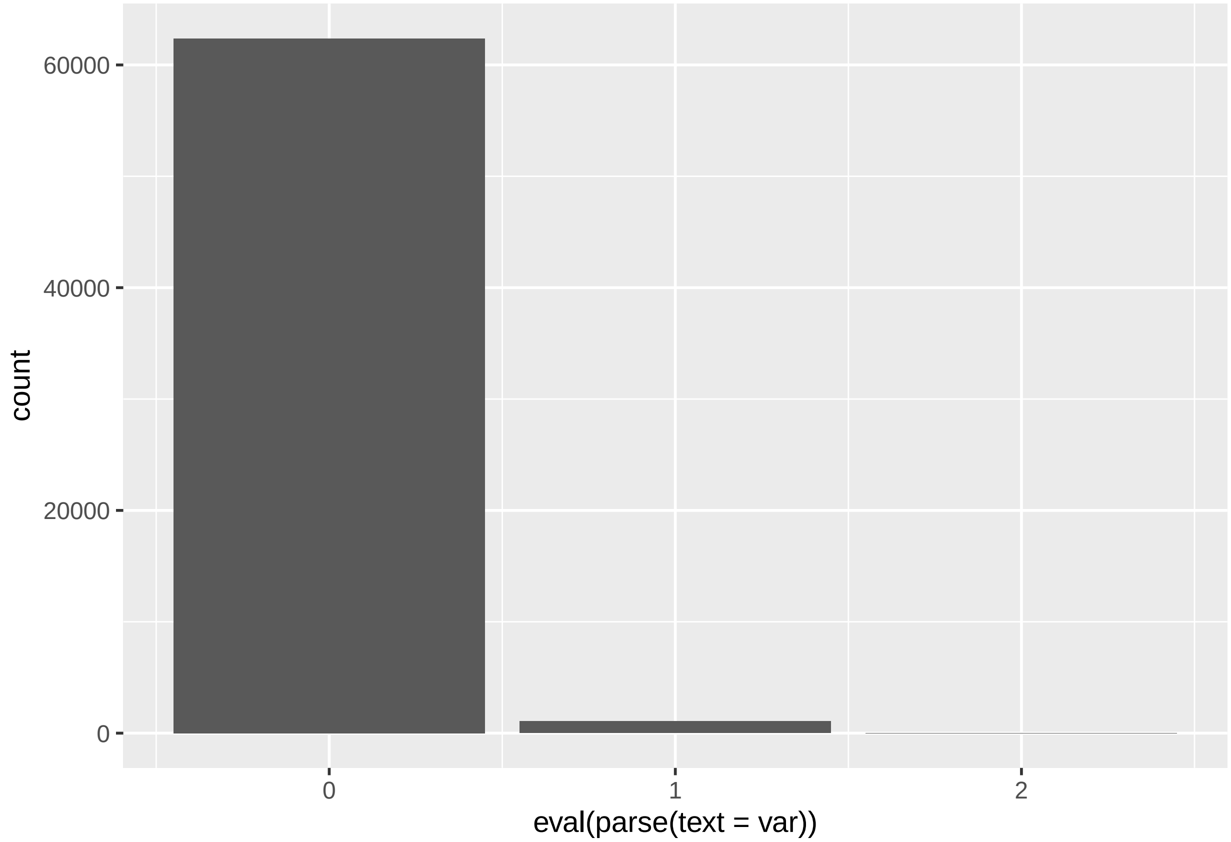
```
checkit <- function(var){
  vname <- paste0("crashes$",var)
  print(summary(eval(parse(text=vname))))
  ggplot() +
    geom_histogram(data = crashes,
                   mapping = aes(eval(parse(text=var))),
                   stat="count")
}
```

I call it by giving it a variable name in quotes:

```
checkit(var = "TOTAL_BICYCLES")
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.00000 0.00000 0.01722 0.00000 2.00000
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



This is good, but it fails to illuminate the power of functions. What we did above we could do just as well in a loop – so why bother with a function? A function is great if you'd like to change more than one element at a time – which is all a loop will allow you to do.

In our next iteration, let's add a second element to change – title text to make the graph more reasonable.

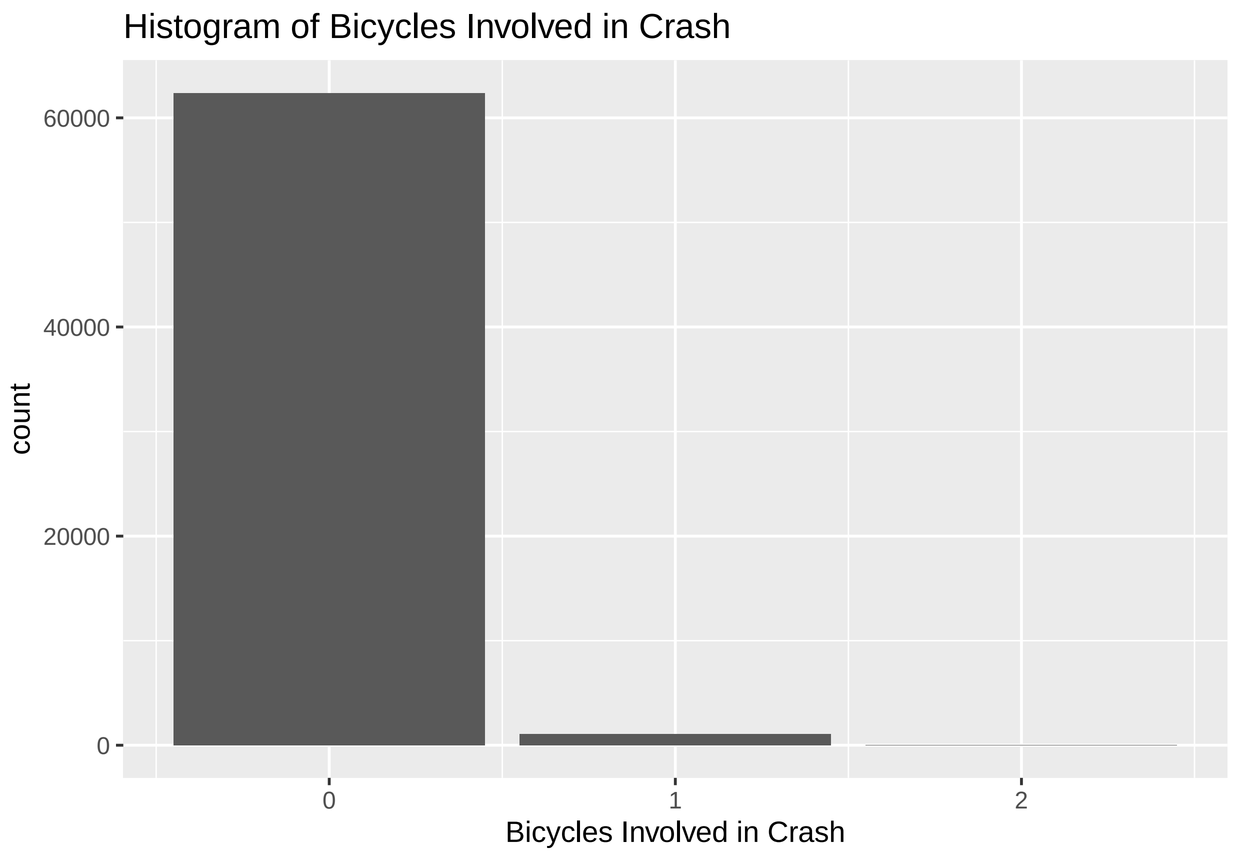
```
checkit <- function(var,namer1){
  vname <- paste0("crashes$",var)
  print(summary(eval(parse(text=vname))))
  ggplot() +
    geom_histogram(data = crashes,
                   mapping = aes(eval(parse(text=var))),
                   stat="count") +
    labs(title = paste0("Histogram of ",namer1),
         x = namer1)
}
```

Re-running with this new function, we find

```
checkit(var = "TOTAL_BICYCLES", namer1 = "Bicycles Involved in Crash")
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.00000 0.00000 0.01722 0.00000 2.00000
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



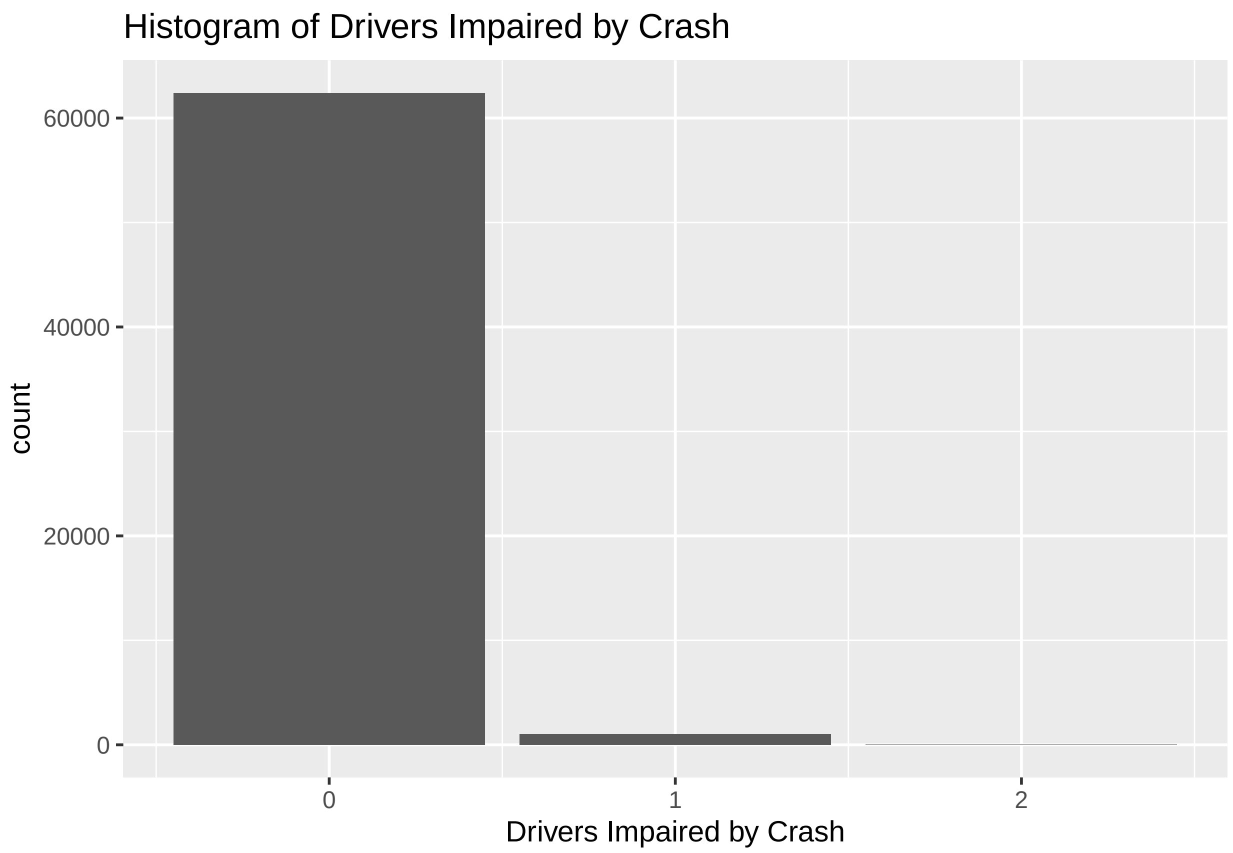
And now it is perfectly easy to do a bunch more variables to get a sense of what these data look like:

```
checkit(var = "DRIVERSIMPAIRED", namer1 = "Drivers Impaired by Crash")
checkit(var = "SPEEDING_INVOLVED", namer1 = "Whether Speeding Was Involved")
checkit(var = "TOTAL_TAXIS", namer1 = "Number of Taxis Involved")
checkit(var = "WARD", namer1 = "Ward")
checkit(var = "TOTAL_GOVERNMENT", namer1 = "don't know")
```

These commands produce a lot of output:

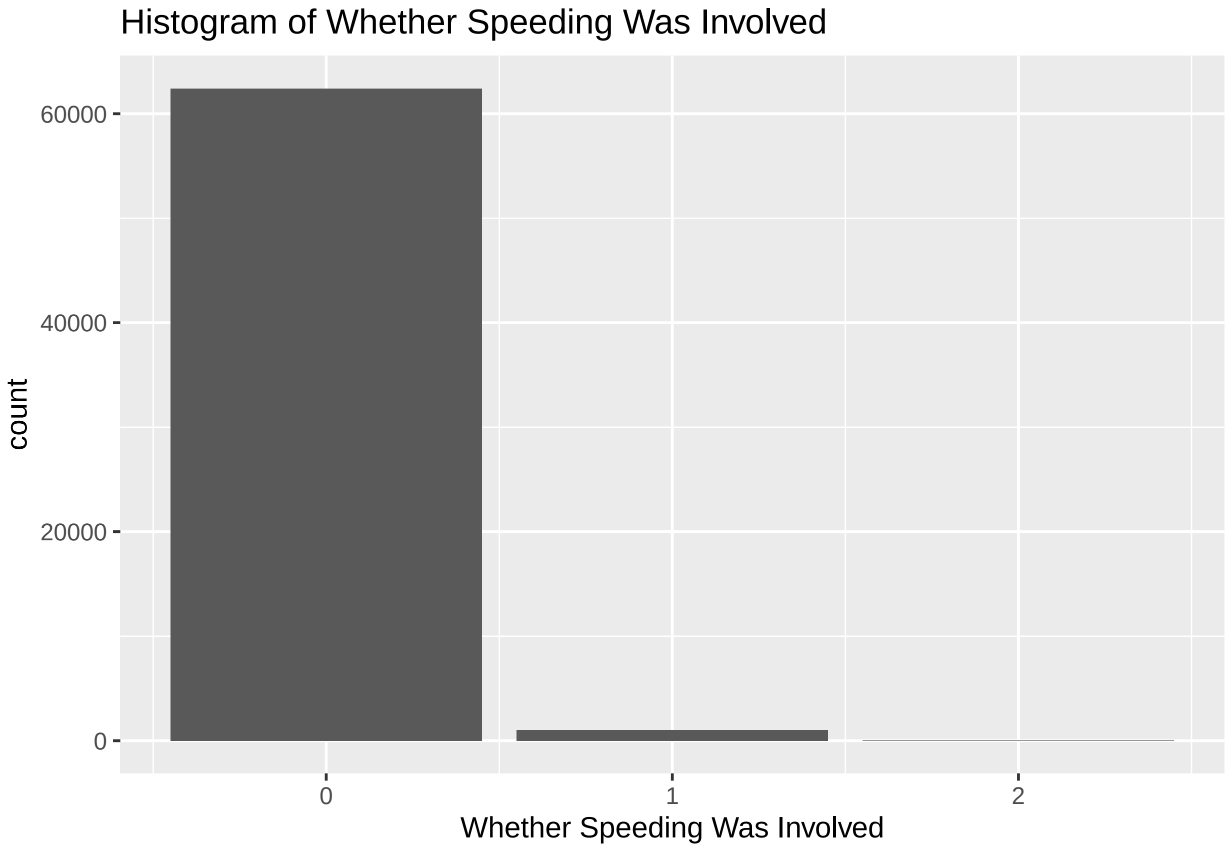
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.00000 0.00000 0.01662 0.00000 2.00000
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.00000 0.00000 0.01645 0.00000 2.00000
```

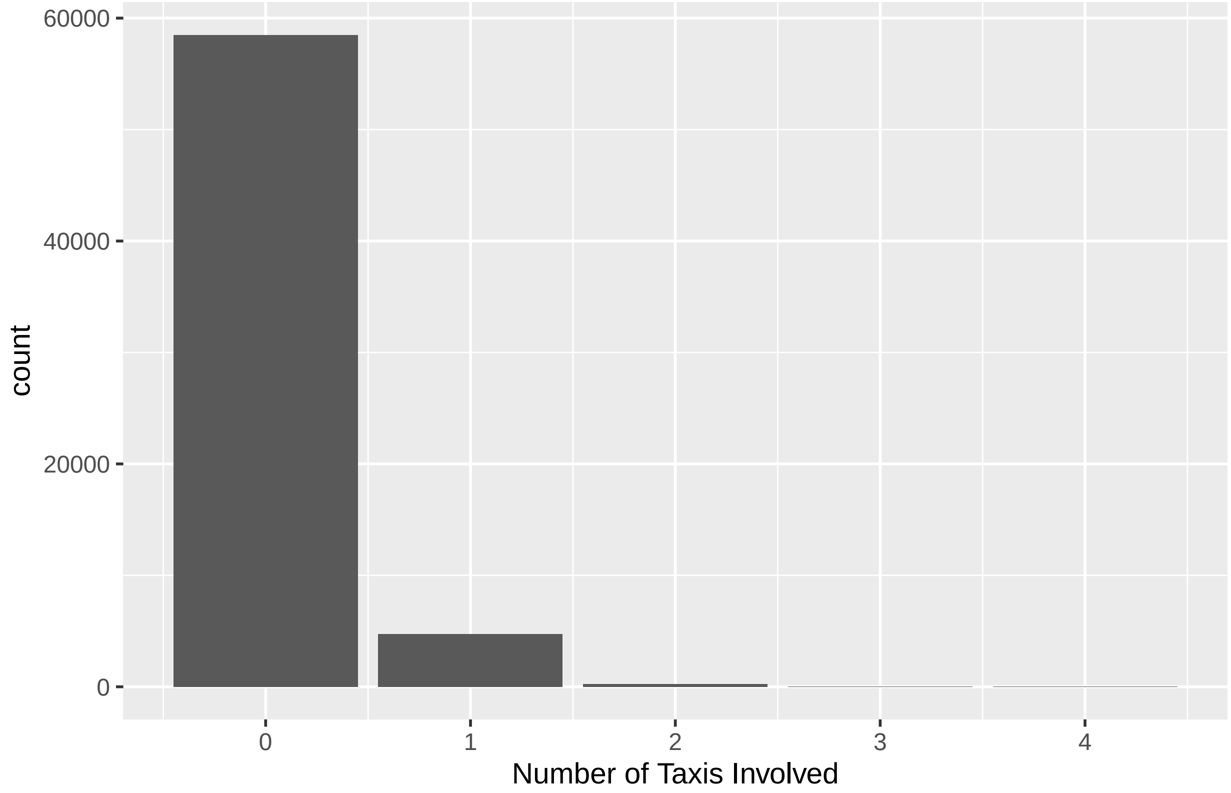
```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.00000 0.00000 0.08209 0.00000 4.00000
```

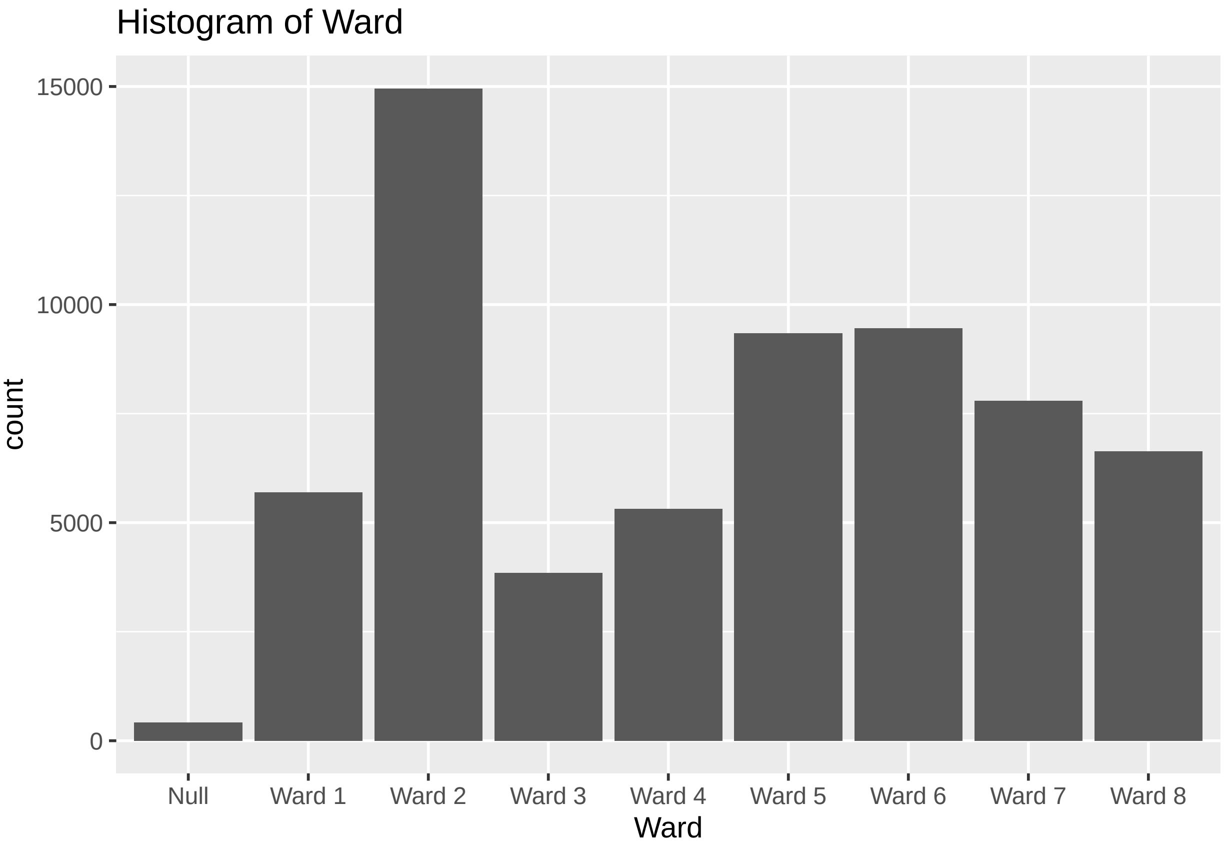
```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```

Histogram of Number of Taxis Involved



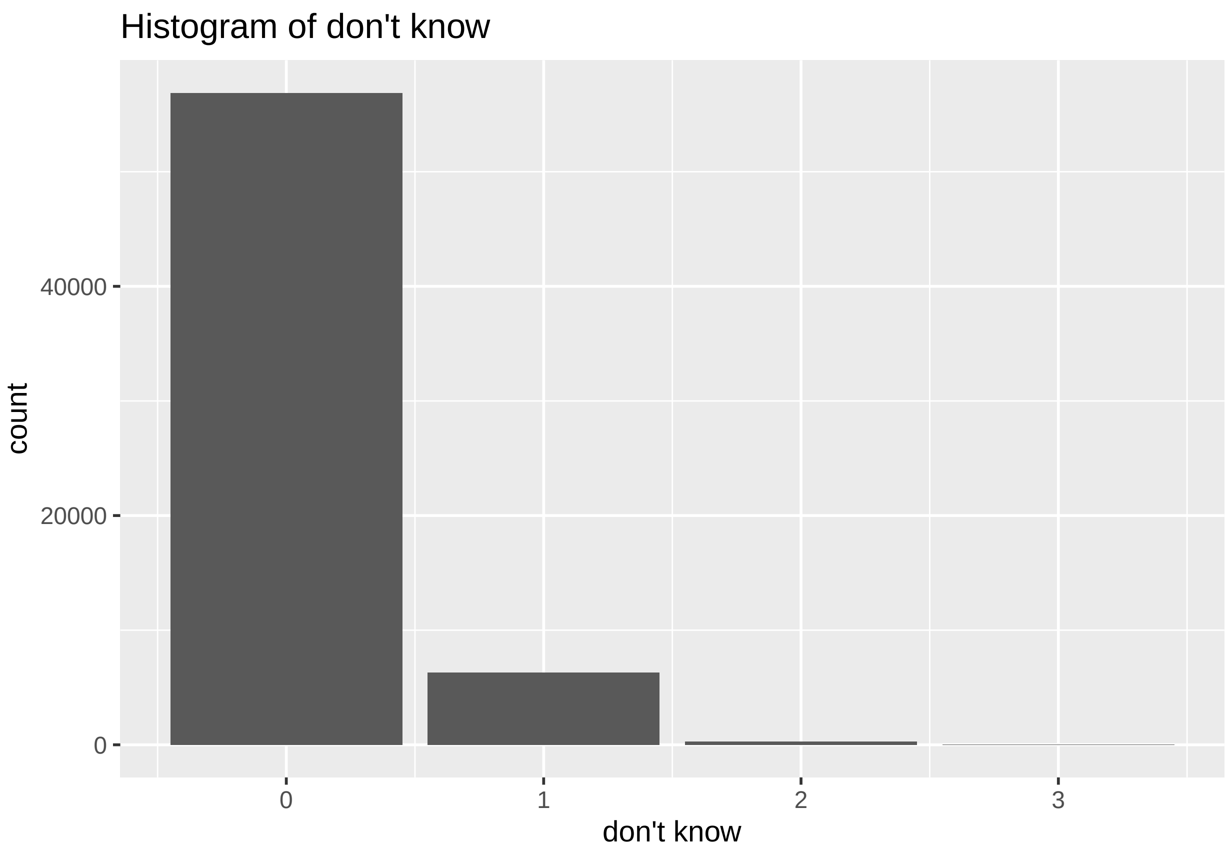
```
##   Null Ward 1 Ward 2 Ward 3 Ward 4 Ward 5 Ward 6 Ward 7 Ward 8
##    415   5697  14957   3843   5320   9343   9460   7794   6640
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.0000  0.0000  0.1085  0.0000  3.0000
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



Finally, I create a few more variables – I was interested if there was a time pattern in crashes, so I extract the date portion of the `FROMDATE` variable using `substr` (as we've done previously) and create a R date variable using `as.Date()`.

I am optimistic this is the right variable and that it's formatted properly for this because I've taken a quick look at the two major date variables in the original dataset that come as text:

```
crashes[1:10,c("FROMDATE","TODATE")]
```

```
##                    FROMDATE TODATE
## 1  2012-04-17T04:00:00.000Z     NA
## 2  2012-04-11T04:00:00.000Z     NA
## 3  2014-01-13T05:00:00.000Z     NA
## 4  2012-04-09T04:00:00.000Z     NA
## 5  2012-04-19T04:00:00.000Z     NA
## 6  2012-04-12T04:00:00.000Z     NA
## 7  2012-04-19T04:00:00.000Z     NA
## 8  2012-04-11T04:00:00.000Z     NA
## 9  2012-04-13T04:00:00.000Z     NA
## 10 2012-04-17T04:00:00.000Z     NA
```

A date variable is a special kind of variable that allows us to find the month or the day of the week of that date. Month you could imagine finding on your own, but the day of the week is tricky.

Here's how I create these new variables (remember that we already did a very similar task in Tutorial 3).

```
crashes$dayof <- as.Date(substr(crashes$FROMDATE,1,10))
crashes$day.of.week <- weekdays(crashes$dayof)
table(crashes$day.of.week)
```

```
##
##    Friday    Monday  Saturday    Sunday  Thursday   Tuesday Wednesday
##     10537      8204      8879      7106      9685      9482      9574
```

```
crashes$month <- months(crashes$dayof)
table(crashes$month)
```

```
##
##     April    August  December  February   January      July      June     March
##      7131      3497      2687      2374      2562      5269      6773      6038
##       May  November   October September
##      7853      6691      8230      4362
```

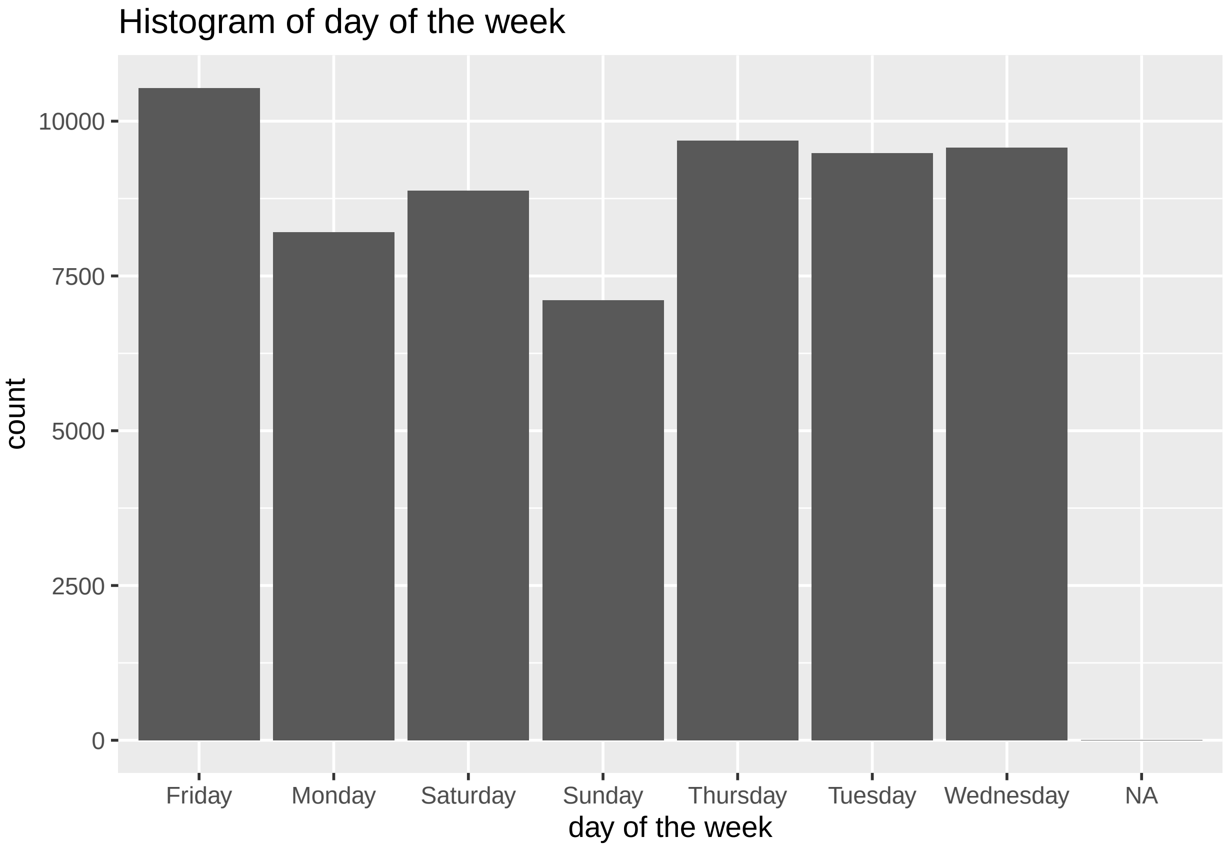
It's hard to interpret whether these numbers are ok from a table. Luckily, we have a function that can make a picture for us. We'll apply our function to these new variables.

```
checkit(var = "day.of.week", namer1 = "day of the week")
checkit(var = "month", namer1 = "month")
```

And the output is

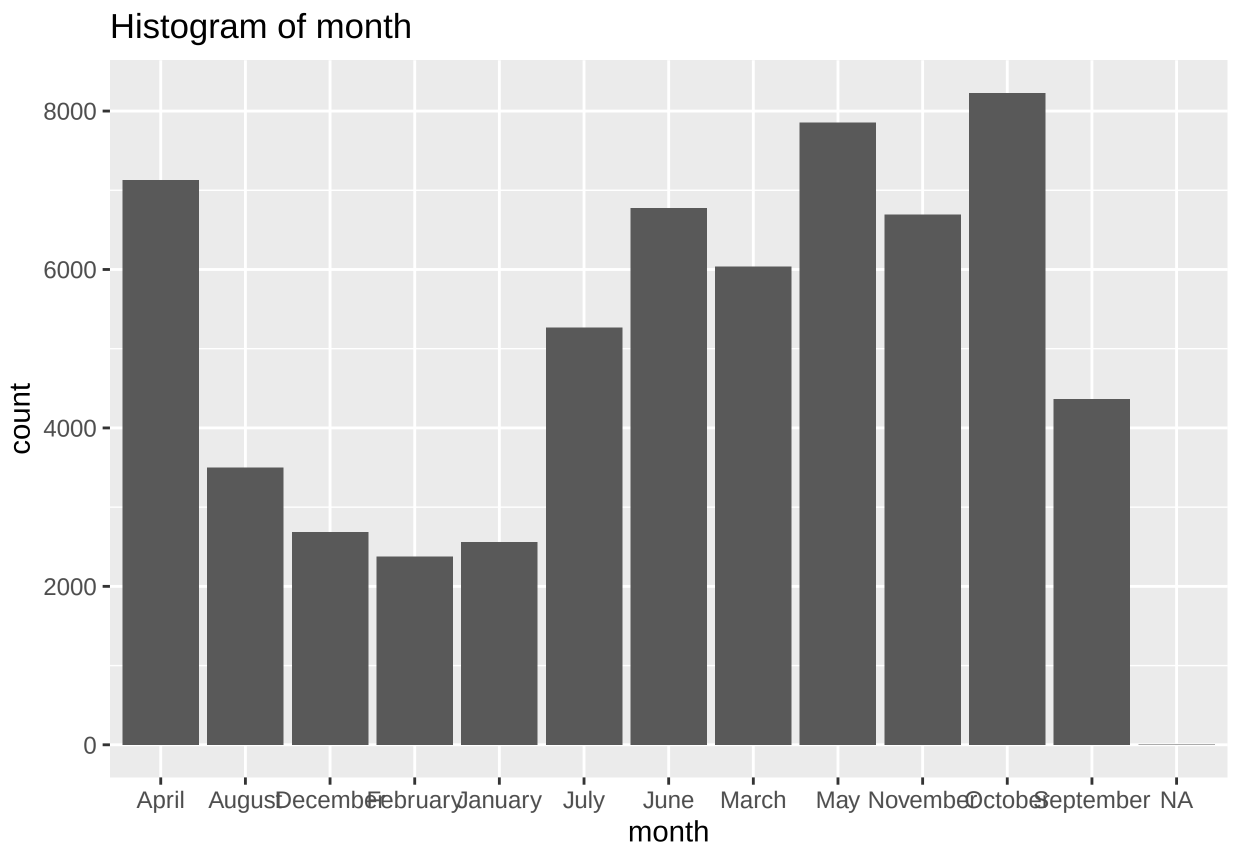
```
##    Length     Class      Mode
##     63469 character character
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



```
##    Length     Class      Mode
##     63469 character character
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



Things look ok! (Last year, the data had an alarming spike in January; DC seems to have cleaned this up.)

## F. Functions and ggplot

After all this build-up, I now need to tell you that not all parts of `ggplot()` work as one might hope in a function. After I explain this, I show you a non-terrible work-around.

### F.1. Why ggplot is different

It seems that this kind of function should work, where we tell R some input variables:

```
graphit <- function(var,namer1){
  ggplot() +
    geom_histogram(data = crashes,
         mapping = aes(x= var),
         stat="count") +
    labs(title = paste0("Histogram of ",namer1),
         x = namer1)
}
```

But when you test it, it does not:

```
graphit(var = WARD,
        namer1 = "Ward")
```

This is because of the non-standard way in which all `tidyverse` packages evaluate R code. For more on that, see this document.

### F.2. Fixes

However, there is a happy fix that allows you to "plug in" variables in text into the function. Instead of `aes()`, use `aes_string()`.

Here is a very general example of the type of function you can write. Note that the input variables *must be in quotes.*

```
testplot <-
  function(datum = meansdf,
           xvar = "condition",
           yvar = "means",
           fillvar = "condition"){
    p <- ggplot(data = datum,
                mapping = aes_string(x = xvar, y= yvar, fill = fillvar)) +
         geom_bar(position="dodge", stat="identity")
}
```

(This example is taken from Stack Overflow.)

## G. Homework

1. In section B.1., why do

```
summer(x = 1, y = 2)
summer(x = 2, y = 1)
```

not yield the same output? Write in math what each one does.

2. In section C.1., why does the call `summer3(x = 5, y = 3)` return a value when `summer2(x = 5, y = 3)` does not?
3. In C.2., why does `summer(x = "fred", y = "ted")` yield an error?
4. Fix the function in part E to remove the graph background.
5. Make a function that automates a graphics operation of interest to you, using a dataset not from this tutorial.