R Programming Notes

Srikar Katta and Edoardo Airoldi

Contents

R Programming

Introduction

The rise of data science's popularity is in large part due to the easy use of computational techniques to simplify processes. What used to take five hours by hand can now be done in a matter of only a few seconds through the use of a computer. Throughout this book, we will use the statistical programming language `R` to complement the statistical concepts taught. While there are many languages that can be used for data science–including Python, Java, and Matlab–`R` is an easy-to-use programming language that is constantly growing, largely due to the fact that it is open-source (i.e., anyone can contribute to `R`'s tools and functionalities). `R` has an incredibly strong community and is often used in scientific research and in the industry. Because `R` is quite expansive and has many features, it is quite difficult to understand all of `R` in one book. However, the content we discuss provides the reader enough information to understand the core aspects of `R` for statistics.

Before we dive into `R`, there are a few steps to tackle. First, we must install `R` and the `R` editor `RStudio` on our local computers by following the instructions on the `RStudio website` or by making an account with `RStudio Cloud`, the cloud version of`RStudio`.

Within `RStudio`, there are four windows: the console tab, the workspace tab, the files tab, and the `R` scripts tab.

The console tab will display outputs; the workspace tab will display any variables or data created in any `R` scripts; the files tab will display the files in our current working directory, and the `R` scripts tab will open up files that we write `R` code in.

To open a new `R` script, press "File" –> "New File"–> "R Script". `R` scripts have the `.R` file extension and execute `R` commands. However, there are many times we want to add content to our scripts and do not want to have the code execute; rather than deleting the code, we "comment" the code out by putting a `#` in front of the line we do not want executed. If we are interested in commenting out multiple lines of code, we can use the `RStudio` shortcut, `command + shift + c` on Macs or `control + shift + c` on Linux and Windows. Comments are also incredibly useful for writing descriptions about a chunk of code's functions, allowing others (and ourselves) to quickly understand our goal when programming.

To ensure RStudio and R work, let us create our first R script that prints the statement "Hello, world!" Open a new R script and save it as \texttt{hello_world.R}. In order to get into the habit of proper coding practices, we want to add our name, date, and a short description of the script at the top as comments. For example, here is what I would write:

```
## Srikar Katta
## August 23, 2021
## My first R script
```

Now, write the statement `print('Hello, world!')`. While we will explain what character data types and `print` statements are later in this chapter, at a high level, this command simply tells the computer to print out the statement inside its parentheses. In addition to the actual command, let us write a comment that explains what is happening so that we can get into the practice of describing our code:

```
## prints the statement "Hello, world!"
print('Hello, world!')
```

```
## [1] "Hello, world!"
```

To actually execute the command, we will highlight the chunk of code we are interested in running and hit `Run` in the top right of the R script tab or press `command + enter` on Macs or `control + enter` on Windows/Linux. In the console tab, we should see the following:

```
> print('Hello, world!')
```

```
[1] "Hello, world!"
```

Variables and Data Types

While R will simplify your workflow, there is a large trade-off that must be made: as the programmer, you must be very precise and very exact in your instructions. For instance, when explaining how to make a sandwich to a friend, you might say, "Take two slices of bread, put your fixens on one slice, and put the other slice on top of the fixes," and your friend would understand. However, the computer requires incredibly exact instructions. It has no idea what bread, fixens, or slices are; it does not understand what it means to "put" something on something else. Specificity is key when programming.

One frustration may be the fact that redefining "bread" for each and every instruction could be very tedious and repetitive. To overcome this, R (as well as other programming languages) has variables.

Definition 0.1: Variable

Variables are spaces of memory reserved in the computer for storing specific values. The process of saving a value to a variable is known as **assignment**.

For instance, one variable may store the value 5 and another may store the value 6; these two variables can then be added up rather than writing "5 + 6." To assign a variable in R, the common convention is to write the name of the variable on the left, then "<-", and lastly the value that should be stored in the variable. For example, the following assigns the value 5 to the variable `x`:

```
x <- 5
```

Variables in R must follow certain naming conventions:

- The variable *must* start with a letter, but it can contain numbers, letters, underscore, and periods
- Keywords are not allowed to be defined as variable names. For example, 'for' is a keyword in R, so you are not allowed to save a value to the variable 'for'.

- Special characters like ‘¡ and ‘*‘ and white space like tabs and spaces are not allowed in the variable name

Because variables reserve an allocated amount of memory, the variable must know how much memory is required for a value, and depending on the data type, each value requires a different amount of data. While we will not discuss the exact specifics of memory allocation or even memory requirements for different data types, understanding the differences between data types is essential for a programmer.

Definition 0.2: Data type

A **data type** is a piece of information that informs the computer of properties that can be act upon it. For instance, integer data types can be added together but character data types cannot.

The key data types for data science are logical, numeric, integer, and character.

Definition 0.3: Logical data type

A **logical data type** has only two values: `TRUE` and `FALSE`. In `R`, `TRUE` and 1 are equivalent and `FALSE` and 0 are equivalent.

Definition 0.4: Numeric data type

A **numeric data type** is any real number. The following are examples: 1, -5, 10284.99678, -901374.112. There are also keywords that represent special numeric data, like `pi` represents the number 3.14..., and `Inf` represents ∞.

Definition 0.5: Character data type

A **character data type** is anything that resides between quotes. For example, even though 1 is a numeric, when it is stored within a pair of quotes, it becomes a character: "1". Character data can be denoted with either single (i.e., ') or double (i.e., ") quotes.

Each data type has special operations that can be performed on them. For instance, logical values have logical operators–and, or, and xor–denoted as `&`, `|`, and `xor` in `R` respectively. The following table describes the result of the logical operators on logical data types:

Logical Value 1	Logical Value 2	`&` outcome	`|` outcome	`xor` outcome
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

Additionally, we can apply the note operator–denoted as `!`–to logical data. This will turn a `TRUE` value to `FALSE` and vice versa.

All numeric data can be added, subtracted, multiplied, divided (unless the denominator is 0), exponentiated. The operations are shown below:

```
x <- 5.5
y <- 11

add <- x + y ## use + for addition operator
sub <- x - y ## use - for subtraction operator
mul <- x * y ## use * for multiplication operator
div <- y / x ## use / for division operator
xpo <- x ^ y ## use ^ for exponentiation operator
```

In addition to operators existing for specific data types, there are some operations that exist for numerics and characters both. These generally deal with data being compared. While it may seem odd that `y` and `z` are both characters and the less than and greater than operators are being used, recall earlier that we said all data are stored as sequences of 0s and 1s in `R`. Since 0s and 1s are numeric data, they can be compared logically. While applying the greater than or less than operators on characters may not be as common as comparing numeric data in data science, it is still helpful to know that these operators exist. Essentially, the rule for characters is that special characters (e.g., *) are "smaller" than lower case letters, which are "smaller" than upper case letters. These comparisons will return logical data types (i.e., `TRUE` if the result is true and `FALSE` if the result is false).

```
x <- 5
y <- 'a'
z <- 'b'

less <- x < y ## use < for less than operator
grtr <- x > y ## use <= for less than or equal to operator
lseq <- x <= y ## use > for greater than operator
greq <- x >= y ## use >= for greater than or equal to operator
equl <- x == y ## use == for equal to operator
nteq <- x != y ## use != for the not equal to operator
```

While these "scalar" data types are useful, there are many instances in data science when it is useful to couple many different values together. For instance, suppose we have a sequence of 10,000 product prices off of an online marketplace. It is likely that we will apply the same operations to all 10,000 data, so it makes sense to have a data type to save all the values together. In `R`, multiple values can be saved in a vector.

Definition 0.6: Vector data type

A **vector** is a data type in which multiple values can be stored together. These are denoted by `c([insert data here])` with commas separating values.

It is important to note that vectors do not have to consist of the same data types. Suppose we are trying to capture information about products in an online marketplace. While product prices will be numeric, the product's name or brand or supplier will most likely be character data types; using vectors, it is possible to store these data together. It is also possible to combine vectors together by creating a new vector where the two elements are the old vector. Additionally, we may be interested in accessing a specific value of the vector, which we can do by indexing the vector; this is accomplished by calling the variable storing the list, placing square brackets after the variable name, and placing the position of the element of interest inside the square brackets. It is important to note that counting starts from one in `R`. So the first element in the vector will be called using the index 1.

```
vector1 <- c('apple', 1, 'red')
vector2 <- c('banana', 0.5, 'yellow')

c(vector1, vector2) ## displays vector1 and vector2 together
```

```
## [1] "apple"  "1"      "red"    "banana" "0.5"    "yellow"
```

```
vector2[2] ## accesses the second element of vector2: 0.5
```

```
## [1] "0.5"
```

One limitation of vectors however is that they can only store one dimensional data. In the previous example, while we may have been able to intuit that `'apple'` represents the product, it may be better to explicitly store what each value represents, which is possible with a list. We can access data within a list by either providing the index of element of interest or by writing the name of the element of interest as a character if the name is provided. We can also access named elements by writing '[insert name of list variable]$[insert

name of data in the vector]'.

Definition 0.7: List data type

A **list** is a data type in which multiple values can be stored together and the values can be named. Lists can also store other lists or vectors inside of them. These are denoted by `list([insert data here])` with commas separating values. To name the values of a list, the name of the data is followed by `=` and then the data to be stored: `list([insert name] = [insert data], [insert name] = [insert data], ...)`.

```
list1 <- list(fruit = 'apple', price = 1, color = 'red')
list2 <- list(fruit = 'banana', price = 0.5, color = 'yellow')

list1 ## displays list 1
```

```
## $fruit
## [1] "apple"
##
## $price
## [1] 1
##
## $color
## [1] "red"
```

```
list(list1, list2) ## displays list1 and list2 together
```

```
## [[1]]
## [[1]]$fruit
## [1] "apple"
##
## [[1]]$price
## [1] 1
##
## [[1]]$color
## [1] "red"
##
##
## [[2]]
## [[2]]$fruit
## [1] "banana"
##
## [[2]]$price
## [1] 0.5
##
## [[2]]$color
## [1] "yellow"
```

```
list1['fruit'] ## accesses the 'fruit' element of list1
```

```
## $fruit
## [1] "apple"
```

```
list2[2] ## accesses the second element of list2: 0.5
```

```
## $price
## [1] 0.5
```

```
list2$color
```

```
## [1] "yellow"
```

There are also many cases in which a data scientist may be interested in representing multidimensional data more succinctly. For example, `list1` and `list2` both have the same named data, so it may be better to store the data as a matrix instead, a two-dimensional data type. Generally, matrix columns represent the same value of interest (e.g., price) while rows represent data for each observation (e.g., data on each product). Data can be accessed in matrixes by specifying the row of the element of interest followed by the column of the element of interest within square brackets, similar to a vector. If the matrix has names, the data can be accessed using the names of the row and column the element is located in. It is also possible for a matrix to access more than one element through a single call by providing a vector of row indexes and column indexes that are both aligned to the elements of interest.

Definition 0.8: Matrix data type

A **matrix** is a data type that has both rows and columns, with entries of all the same data type. Unlike the other data types, matrices take in more information; specifically, they require a vector of data, the number of rows, the number of columns, and the manner in which the vector should be reshaped. They can be formed by calling `matrix(data = [insert vector], nrow = [insert integer representing number of rows], ncol = [inser integer representing number of columns], byrow = [insert logical representing whether the matrix should be formed rowise or not], dimnames = [insert list of length two gicing the row and column names respectively or NULL if not necessary])`.

```
vector1 <- c('apple', 1, 'red')
vector2 <- c('banana', 0.5, 'yellow')
## make matrix row-wise
m1 <- matrix(data = c(vector1, vector2),
       nrow = 2,
       ncol = 3,
       byrow = TRUE,
       dimnames = NULL)
## make matrix column-wise
m2 <- matrix(data = c(vector1, vector2),
       nrow = 3,
       ncol = 2,
       byrow = FALSE,
       dimnames = NULL)
## make matrix row-wise with names
m3 <- matrix(data = c(vector1, vector2),
       nrow = 2,
       ncol = 3,
       byrow = TRUE,
       dimnames = list(c('product1', 'product2'),
                       c('product', 'price', 'color')))
m1
```

```
##      [,1]     [,2]  [,3]
## [1,] "apple"  "1"   "red"
## [2,] "banana" "0.5" "yellow"
```

```
m2
```

```
##      [,1]    [,2]
## [1,] "apple" "banana"
```

```
## [2,] "1"     "0.5"
## [3,] "red"   "yellow"
```

```
m3
```

```
##          product  price color
## product1 "apple"  "1"   "red"
## product2 "banana" "0.5" "yellow"
```

```
## access data in m1
m1[1, 3] ## row 1, column 3 is 'yellow'
```

```
## [1] "red"
```

```
m3['product2', 'product'] ## 'banana'
```

```
## [1] "banana"
```

```
m2[c(1, 2), c(1, 2)]
```

```
##      [,1]    [,2]
## [1,] "apple" "banana"
## [2,] "1"     "0.5"
```

If a function or operation can be applied to all values in a vector, list, or matrix, then an operation can be applied by treating the variable just like another variable. For example, if we have a vector `x` of numeric data, then we could add 1 to all the values in `x` by calling `x + 1`:

```
vector1 <- c(1.5, 2, 2.5)
vector1 + 1
```

```
## [1] 2.5 3.0 3.5
```

However, if all the values are not of the same data type a function cannot be applied to that data, then there will be an error:

A dataframe data type is very similar to a matrix in the sense that it is two dimensional and has rows and columns. However, dataframes are a generalized form of a matrix and allows for different data types: each column is a vector of values of the same data types but the rows can differ. For example, notice that even though 1 in `R` is a numeric, it is a character in the matrix `m1`, as evidenced by the quotes around its entry. However, in the `dataframe`, 1 is a numeric.

\begin{defintion}{Dataframe data type} A **dataframe** is a generalized matrix whose columns may be of different data types. \end{definition_term}

`data.frame` constructions take the following form:

```
[insert variable name] <- data.frame(
  [insert column name] = [insert vector of data],
  [insert column name] = [insert vector of data],
  ...
  [insert column name] = [insert vector of data]
)
```

Because the columns are vectors of the same data type, we must first create vectors representing data we want in our columns; keep in mind that all vectors must be the same length or else `R` will throw an error.

```
vector1 <- c('apple', 'banana')
vector2 <- c(1, 0.5)
vector3 <- c('red', 'yellow')

df1 <- data.frame(
```

```
  product = vector1,
  price = vector2,
  color = vector3
)

print(df1)
```

```
##   product price  color
## 1   apple   1.0    red
## 2  banana   0.5 yellow
```

Conditional Logic

In addition to variables and data types, conditional statements are essential for data science. Conditionals often follow the form of "if...then." For instance, "if a person is from Philadelphia, then they are likely fans of the Eagles," would be a conditional statement because it follows an "if...then" style statement.

Definition 0.9: Conditional

A **conditional** is a statement that follows the form of "if [insert condition], then [insert conclusion]". The condition must be a logical type: `TRUE` or `FALSE`.

Suppose we are creating a simulation of the real estate market, which vary significantly by geographic location: real estate in Manhattan is different than real estate in Boston, which is very different than real estate in Wichita, Kansas. These conditional statements can be very helpful in these situations. The "if" portion of the conditional will be denoted by `if([insert logical statement])` in `R`. Within the `if` statement must go a logical statement (i.e., a statement that returns `TRUE` or `FALSE`). The "if" portion is then followed by the conclusion, which is contained within brackets. To create multi-level conditionals, we use the `else` operator.

Note that it is possible for for two conditional to be true; in the real estate market, a house will be located in Boston and the northeast. However, if both conditions are used in the same if-else logic, then only the first true condition will be considered. In other words, as soon as the first true condition occurs, the computer stops evaluating all other else statements.

```
location <- 'Boston'

## simple conditional statement
if(location == 'Boston') {
  mean_house_price <- 800000
}

mean_house_price
```

```
## [1] 8e+05
```

```
## if-else conditional statement
if(location != 'Boston') {
  mean_house_price <- 300000
} else {
  mean_house_price <- 800000
}
mean_house_price
```

```
## [1] 8e+05
```

```r
## multi-level if-else conditional statement
if(location == 'Boston') {
  mean_house_price <- 800000
} else if(location == 'Manhattan') {
  mean_house_price <- 1500000
} else if(location == 'Northeast') {
  mean_house_price <- 500000
} else {
  mean_house_price <- 250000
}
mean_house_price
```

```
## [1] 8e+05
```

Functions

Returning to the example in the introduction where we discussed "making a sandwich," variables allow you to define what "bread" and "fixens" are; however, the computer still may not know what it means to "put" fixens on bread. Functions are actions in computer programming that allow us to overcome the problems of repeating ourselves; rather than continuously telling the computer what "put" means, we can define a function to perform the action for us.

Definition 0.10: Functions in programming

Functions are blocks of codes that perform actions in programming that take in inputs and provide zero or one outputs.

Functions follow a particular construction: functions take in inputs, called arguments, that can be data structures that the function acts upon (e.g., take the mean of a vector) or it may be a way to alter the method in which a function acts (e.g., return a different mean for "Northeast" and "Boston" inputs). However, functions can also have zero inputs and always perform the same action (i.e., execute the same block of code) by calling the function. It is important to note that arguments often require a specific data structure, or else the code will throw an error or not execute properly. Ensuring the proper data type is being input can save us from headaches down the road.

Additionally, functions in programming return either one or zero data structures that can then be saved to other variables. However, if we would like to return two values–the mean and the variance, for example–we could save the results in a named list and access the values by returning the list.

The following are a few examples of commonly used functions in `R`:

```r
## print function
##   input: character
##   returns: nothing
##   action: displays character to screen
print('Hello, world')
```

```
## [1] "Hello, world"
```

```r
## mean function
##   input: vector of numeric data
##   output: numeric
##   action: calculates the mean of the numeric data
mean(c(1, 2, 3))
```

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

```
mean('Hello, world') ## using the wrong input data type can lead to errors
```

```
## Warning in mean.default("Hello, world"): argument is not numeric or logical:
## returning NA
```

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

If using RStudio, we can find information on functions by typing ?[insert function name], which will then prompt the "Help" tab to display information on the function.

Functions can also have default arguments. For instance, in the mean function, it actually takes in multiple arguments: a vector of numerics x, a numeric scalar trim that represents "the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint," and a logical na.rm that "[indicates] whether NA values should be stripped before the computation proceeds." However, we did not need to specify the trim and na.rm arguments when previously calling mean because they have defaults trim = 0 and na.rm = FALSE. The following displays the differences in calling the mean function when changing defaults:

```
vector1 <- c(1, 2, 2, 4)
print(mean(vector1)) ## mean with trim = 0 default
```

```
## [1] 2.25
```

```
print(mean(vector1, trim = 0.25)) ## mean when removing top and bottom 25% of values
```

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

We can also create our own functions by calling

```
[insert name] <- function([insert arguments], [insert default argument] = [insert default input]) {
  [insert code]

  return([insert data structure to return])
}
```

As we discussed earlier, functions do not need to take in any arguments or return any values. So, we can define a function with neither of these as such:

```
[insert name] <- function() {
  [insert code]
}
```

Additionally, function names follow the same rules/conventions as variable names:

- The variable *must* start with a letter, but it can contain numbers, letters, underscore, and periods
- Keywords are not allowed to be defined as variable names. For example, 'for' is a keyword in R, so you are not allowed to save a value to the variable 'for'.
- Special characters like '¡ and '*' and white space like tabs and spaces are not allowed in the variable name

It is very common to see functions within functions as well. Suppose we will be printing out three lines: the first statement being a person's name, the second being a person's age, and the third being the number of classes a person has taken. For the letter grades, we will ask how many classes did they receive with that letter grade. We can do so by defining the function \texttt{print_person}:

```
print_person <- function(name, age, n_classes) {
  print(paste('Name: ', name))
  print(paste('Age: ', age))
  print(paste('Number of classes: ', n_classes))
```

```
}
print_person(
  name = 'Srikar',
  age = '21',
  n_classes = 25
)
```

```
## [1] "Name:  Srikar"
## [1] "Age:  21"
## [1] "Number of classes:  25"
```

Now, we can extend this to also print out someone's GPA. However, in order to do this, we need someone's letter grades, which we will obtain by creating a new function called \texttt{print_person_plus_gpa} that also has parameters \texttt{a_grade, b_grade, c_grade, d_grade, f_grade} that each take in a numeric that details the number of classes a person has earned that letter grade in. First, we need to convert the \texttt{a_grade, b_grade, c_grade, d_grade, f_grade} numerics into GPA points: each A is worth 4 points, each B is worth 3, each C is worth 2, each D is worth 1, and Fs are worth 0. We then divide the GPA points by the number of classes taken

```
calc_gpa <- function(a_grade, b_grade, c_grade, d_grade, f_grade) {
  gpa_points <-
    (a_grade * 4) + (b_grade * 3) + (c_grade * 2) + (d_grade * 1) + (f_grade * 0)
  n_classes <- sum(c(a_grade, b_grade, c_grade, d_grade, f_grade))
  gpa <- gpa_points/n_classes
  return(gpa)
}
```

Then, we can put the \texttt{print_person} and \texttt{print_gpa} statements together.

```
print_person_plus <- function(name, age, a_grade, b_grade, c_grade, d_grade, f_grade) {

  ## calculate GPA
  gpa <- calc_gpa(a_grade = a_grade,
                  b_grade = b_grade,
                  c_grade = c_grade,
                  d_grade = d_grade,
                  f_grade = f_grade)

  ## calculate number of classes taken
  n_classes <- sum(c(a_grade, b_grade, c_grade, d_grade, f_grade))

  ## print information
  print_person(name = name,
               age = age,
               n_classes = n_classes)
  print(paste('GPA: ', gpa))
}

print_person_plus(name = 'Srikar',
                  age = 21,
                  a_grade = 25,
                  b_grade = 0,
                  c_grade = 0,
                  d_grade = 0,
                  f_grade = 0
```

```
                  )
```

```
## [1] "Name:  Srikar"
## [1] "Age:  21"
## [1] "Number of classes:  25"
## [1] "GPA:  4"
```

Loops

Functions are very useful for modularizing code and removing redundancies. In a similar vein, loops allow us to remove repetition in our code by executing the same block of code multiple times.

Definition 0.11: Loops in programming

A **loop** is a control sequence that runs a block of code multiple times.

To illustrate its use case, suppose we were asked to find the GPA of 25 students. While we have a matrix with each row representing the following data for an individual student: \texttt{name, age, a_grade, b_grade, c_grade, d_grade, f_grade}, where each column name represents the same value as its corresponding parameters in the \texttt{print_person_plus} function.

```
##        names ages a_grades b_grades c_grades d_grades f_grades
## 1       Liam   27        9       12       20        2        7
## 2     Olivia   19        3       17       20        4        6
## 3      Kanye   18       10       12       11        0       17
## 4       Noah   25       11       15       20        3        1
## 5      Rohan   31       14       14       15        4        3
## 6       Emma   18       11       12       11        5       11
## 7     Oliver   20       10       16       18        7        0
## 8        Ava   23       11       14        9        5       11
## 9     Elijah   23       10       11       14        5       10
## 10 Charlotte   22        5       15       14        2       14
## 11   William   20       12        6       14        6       12
## 12    Sophia   27        9       17       25        4        0
## 13     James   26        8        8       17        3       14
## 14       Jun   20        9       10       14        4       13
## 15    Amelia   15       14       11       15        6        4
## 16  Benjamin   21       10       15       21        5        0
## 17  Isabella   15        9       20       15        4        2
## 18     Lucas   19       11       18       17        6        0
## 19       Mia   26        7       11       15        5       12
## 20    Tashin   28        9       13       17        2        9
## 21     Henry   15       13       12       12        3       10
## 22    Evelyn   25        7       13       12        9        9
## 23    Miriam   17       13       15       12        5        5
## 24 Alexander   18       11       16       23        9        0
## 25    Harper   14        5       14       25        7        0
```

While the \texttt{print_person_plus} function allows us to avoid retyping all of the code multiple times, we still have to write the \texttt{print_person_plus} function each and every time. For example,

```
## info for person 1
print_person_plus(name = grades_table$names[1],
                  age = grades_table$ages[1],
                  a_grade = grades_table$a_grades[1],
```

```
                  b_grade = grades_table$b_grades[1],
                  c_grade = grades_table$c_grades[1],
                  d_grade = grades_table$d_grades[1],
                  f_grade = grades_table$f_grades[1]
                  )
```

```
## [1] "Name:  Liam"
## [1] "Age:  27"
## [1] "Number of classes:  50"
## [1] "GPA:  2.28"
```

```
## info for person 2
print_person_plus(name = grades_table$names[2],
                  age = grades_table$ages[2],
                  a_grade = grades_table$a_grades[2],
                  b_grade = grades_table$b_grades[2],
                  c_grade = grades_table$c_grades[2],
                  d_grade = grades_table$d_grades[2],
                  f_grade = grades_table$f_grades[2]
                  )
```

```
## [1] "Name:  Olivia"
## [1] "Age:  19"
## [1] "Number of classes:  50"
## [1] "GPA:  2.14"
```

Instead we can use a for loop, which we usually use to iterate over a sequence of some sort; the loop stops executing after it has iterated through all the elements in the sequence.

Definition 0.12: For loop in programming

A **for loop** is a control sequence that iterates through a vector of elements (generally a sequence) and stops once ever element in the vector has been iterated through. The syntax is usually `for([insert variable name] in [insert vector]) { [insert code] }`

In this case, we will create a sequence from 1 to 25 so that we can go through each and every observation in the dataframe, using `i` as our indexing variable.

```
for(i in 1:25) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
```

```
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
```

Next, we will replace the inner code chunk with the function of interest:

```
for(i in 1:25) {
  print_person_plus(name = grades_table$names[i],
                    age = grades_table$ages[i],
                    a_grade = grades_table$a_grades[i],
                    b_grade = grades_table$b_grades[i],
                    c_grade = grades_table$c_grades[i],
                    d_grade = grades_table$d_grades[i],
                    f_grade = grades_table$f_grades[i]
                    )
}
```

```
## [1] "Name:  Liam"
## [1] "Age:  27"
## [1] "Number of classes:  50"
## [1] "GPA:  2.28"
## [1] "Name:  Olivia"
## [1] "Age:  19"
## [1] "Number of classes:  50"
## [1] "GPA:  2.14"
## [1] "Name:  Kanye"
## [1] "Age:  18"
## [1] "Number of classes:  50"
## [1] "GPA:  1.96"
## [1] "Name:  Noah"
## [1] "Age:  25"
## [1] "Number of classes:  50"
## [1] "GPA:  2.64"
## [1] "Name:  Rohan"
## [1] "Age:  31"
## [1] "Number of classes:  50"
## [1] "GPA:  2.64"
## [1] "Name:  Emma"
## [1] "Age:  18"
## [1] "Number of classes:  50"
## [1] "GPA:  2.14"
## [1] "Name:  Oliver"
## [1] "Age:  20"
## [1] "Number of classes:  51"
## [1] "GPA:  2.56862745098039"
## [1] "Name:  Ava"
## [1] "Age:  23"
```

```
## [1] "Number of classes:  50"
## [1] "GPA:  2.18"
## [1] "Name:  Elijah"
## [1] "Age:  23"
## [1] "Number of classes:  50"
## [1] "GPA:  2.12"
## [1] "Name:  Charlotte"
## [1] "Age:  22"
## [1] "Number of classes:  50"
## [1] "GPA:  1.9"
## [1] "Name:  William"
## [1] "Age:  20"
## [1] "Number of classes:  50"
## [1] "GPA:  2"
## [1] "Name:  Sophia"
## [1] "Age:  27"
## [1] "Number of classes:  55"
## [1] "GPA:  2.56363636363636"
## [1] "Name:  James"
## [1] "Age:  26"
## [1] "Number of classes:  50"
## [1] "GPA:  1.86"
## [1] "Name:  Jun"
## [1] "Age:  20"
## [1] "Number of classes:  50"
## [1] "GPA:  1.96"
## [1] "Name:  Amelia"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.5"
## [1] "Name:  Benjamin"
## [1] "Age:  21"
## [1] "Number of classes:  51"
## [1] "GPA:  2.58823529411765"
## [1] "Name:  Isabella"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.6"
## [1] "Name:  Lucas"
## [1] "Age:  19"
## [1] "Number of classes:  52"
## [1] "GPA:  2.65384615384615"
## [1] "Name:  Mia"
## [1] "Age:  26"
## [1] "Number of classes:  50"
## [1] "GPA:  1.92"
## [1] "Name:  Tashin"
## [1] "Age:  28"
## [1] "Number of classes:  50"
## [1] "GPA:  2.22"
## [1] "Name:  Henry"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.3"
```

```
## [1] "Name:  Evelyn"
## [1] "Age:  25"
## [1] "Number of classes:  50"
## [1] "GPA:  2"
## [1] "Name:  Miriam"
## [1] "Age:  17"
## [1] "Number of classes:  50"
## [1] "GPA:  2.52"
## [1] "Name:  Alexander"
## [1] "Age:  18"
## [1] "Number of classes:  59"
## [1] "GPA:  2.49152542372881"
## [1] "Name:  Harper"
## [1] "Age:  14"
## [1] "Number of classes:  51"
## [1] "GPA:  2.33333333333333"
```

Suppose that instead of indexing until 25, we indexed until 26. Then, we would get the following output:

```
for(i in 1:26) {
  print_person_plus(name = grades_table$names[i],
                    age = grades_table$ages[i],
                    a_grade = grades_table$a_grades[i],
                    b_grade = grades_table$b_grades[i],
                    c_grade = grades_table$c_grades[i],
                    d_grade = grades_table$d_grades[i],
                    f_grade = grades_table$f_grades[i]
                    )
}
```

```
## [1] "Name:  Liam"
## [1] "Age:  27"
## [1] "Number of classes:  50"
## [1] "GPA:  2.28"
## [1] "Name:  Olivia"
## [1] "Age:  19"
## [1] "Number of classes:  50"
## [1] "GPA:  2.14"
## [1] "Name:  Kanye"
## [1] "Age:  18"
## [1] "Number of classes:  50"
## [1] "GPA:  1.96"
## [1] "Name:  Noah"
## [1] "Age:  25"
## [1] "Number of classes:  50"
## [1] "GPA:  2.64"
## [1] "Name:  Rohan"
## [1] "Age:  31"
## [1] "Number of classes:  50"
## [1] "GPA:  2.64"
## [1] "Name:  Emma"
## [1] "Age:  18"
## [1] "Number of classes:  50"
## [1] "GPA:  2.14"
## [1] "Name:  Oliver"
```

```
## [1] "Age:  20"
## [1] "Number of classes:  51"
## [1] "GPA:  2.56862745098039"
## [1] "Name:  Ava"
## [1] "Age:  23"
## [1] "Number of classes:  50"
## [1] "GPA:  2.18"
## [1] "Name:  Elijah"
## [1] "Age:  23"
## [1] "Number of classes:  50"
## [1] "GPA:  2.12"
## [1] "Name:  Charlotte"
## [1] "Age:  22"
## [1] "Number of classes:  50"
## [1] "GPA:  1.9"
## [1] "Name:  William"
## [1] "Age:  20"
## [1] "Number of classes:  50"
## [1] "GPA:  2"
## [1] "Name:  Sophia"
## [1] "Age:  27"
## [1] "Number of classes:  55"
## [1] "GPA:  2.56363636363636"
## [1] "Name:  James"
## [1] "Age:  26"
## [1] "Number of classes:  50"
## [1] "GPA:  1.86"
## [1] "Name:  Jun"
## [1] "Age:  20"
## [1] "Number of classes:  50"
## [1] "GPA:  1.96"
## [1] "Name:  Amelia"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.5"
## [1] "Name:  Benjamin"
## [1] "Age:  21"
## [1] "Number of classes:  51"
## [1] "GPA:  2.58823529411765"
## [1] "Name:  Isabella"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.6"
## [1] "Name:  Lucas"
## [1] "Age:  19"
## [1] "Number of classes:  52"
## [1] "GPA:  2.65384615384615"
## [1] "Name:  Mia"
## [1] "Age:  26"
## [1] "Number of classes:  50"
## [1] "GPA:  1.92"
## [1] "Name:  Tashin"
## [1] "Age:  28"
## [1] "Number of classes:  50"
```

```
## [1] "GPA:  2.22"
## [1] "Name:  Henry"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.3"
## [1] "Name:  Evelyn"
## [1] "Age:  25"
## [1] "Number of classes:  50"
## [1] "GPA:  2"
## [1] "Name:  Miriam"
## [1] "Age:  17"
## [1] "Number of classes:  50"
## [1] "GPA:  2.52"
## [1] "Name:  Alexander"
## [1] "Age:  18"
## [1] "Number of classes:  59"
## [1] "GPA:  2.49152542372881"
## [1] "Name:  Harper"
## [1] "Age:  14"
## [1] "Number of classes:  51"
## [1] "GPA:  2.33333333333333"
## [1] "Name:  NA"
## [1] "Age:  NA"
## [1] "Number of classes:  NA"
## [1] "GPA:  NA"
```

Notice that the code does not break; instead, it prints `NA` values, which would hurt our analyses. To overcome this issue, we will instead replace the "to" index with an automatic calculation of the number of rows in the dataframe, using the `nrow` function:

```
for(i in 1:nrow(grades_table)) {
  print_person_plus(name = grades_table$names[i],
                    age = grades_table$ages[i],
                    a_grade = grades_table$a_grades[i],
                    b_grade = grades_table$b_grades[i],
                    c_grade = grades_table$c_grades[i],
                    d_grade = grades_table$d_grades[i],
                    f_grade = grades_table$f_grades[i]
                    )
}
```

```
## [1] "Name:  Liam"
## [1] "Age:  27"
## [1] "Number of classes:  50"
## [1] "GPA:  2.28"
## [1] "Name:  Olivia"
## [1] "Age:  19"
## [1] "Number of classes:  50"
## [1] "GPA:  2.14"
## [1] "Name:  Kanye"
## [1] "Age:  18"
## [1] "Number of classes:  50"
## [1] "GPA:  1.96"
## [1] "Name:  Noah"
## [1] "Age:  25"
```

```
## [1] "Number of classes:  50"
## [1] "GPA:  2.64"
## [1] "Name:  Rohan"
## [1] "Age:  31"
## [1] "Number of classes:  50"
## [1] "GPA:  2.64"
## [1] "Name:  Emma"
## [1] "Age:  18"
## [1] "Number of classes:  50"
## [1] "GPA:  2.14"
## [1] "Name:  Oliver"
## [1] "Age:  20"
## [1] "Number of classes:  51"
## [1] "GPA:  2.56862745098039"
## [1] "Name:  Ava"
## [1] "Age:  23"
## [1] "Number of classes:  50"
## [1] "GPA:  2.18"
## [1] "Name:  Elijah"
## [1] "Age:  23"
## [1] "Number of classes:  50"
## [1] "GPA:  2.12"
## [1] "Name:  Charlotte"
## [1] "Age:  22"
## [1] "Number of classes:  50"
## [1] "GPA:  1.9"
## [1] "Name:  William"
## [1] "Age:  20"
## [1] "Number of classes:  50"
## [1] "GPA:  2"
## [1] "Name:  Sophia"
## [1] "Age:  27"
## [1] "Number of classes:  55"
## [1] "GPA:  2.56363636363636"
## [1] "Name:  James"
## [1] "Age:  26"
## [1] "Number of classes:  50"
## [1] "GPA:  1.86"
## [1] "Name:  Jun"
## [1] "Age:  20"
## [1] "Number of classes:  50"
## [1] "GPA:  1.96"
## [1] "Name:  Amelia"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.5"
## [1] "Name:  Benjamin"
## [1] "Age:  21"
## [1] "Number of classes:  51"
## [1] "GPA:  2.58823529411765"
## [1] "Name:  Isabella"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.6"
```

```
## [1] "Name:  Lucas"
## [1] "Age:  19"
## [1] "Number of classes:  52"
## [1] "GPA:  2.65384615384615"
## [1] "Name:  Mia"
## [1] "Age:  26"
## [1] "Number of classes:  50"
## [1] "GPA:  1.92"
## [1] "Name:  Tashin"
## [1] "Age:  28"
## [1] "Number of classes:  50"
## [1] "GPA:  2.22"
## [1] "Name:  Henry"
## [1] "Age:  15"
## [1] "Number of classes:  50"
## [1] "GPA:  2.3"
## [1] "Name:  Evelyn"
## [1] "Age:  25"
## [1] "Number of classes:  50"
## [1] "GPA:  2"
## [1] "Name:  Miriam"
## [1] "Age:  17"
## [1] "Number of classes:  50"
## [1] "GPA:  2.52"
## [1] "Name:  Alexander"
## [1] "Age:  18"
## [1] "Number of classes:  59"
## [1] "GPA:  2.49152542372881"
## [1] "Name:  Harper"
## [1] "Age:  14"
## [1] "Number of classes:  51"
## [1] "GPA:  2.33333333333333"
```

Our code can now also account for a new 26th student that may join the dataset.

Visualizations with ggplot2

R has become one of the most popular languages for data science because of its strong community of independent developers, which means that many people contribute to developing R's functionality. R's extensions come from packages.

Definition 0.13: Packages in R

Packages are collections of code, tests, data, objects, and documentation that are assembled in a specific format to allow for easy installation.

To install a package that is listed on R's public repository, CRAN, we simply have to type

```
## package name must be within quotes
install.packages('[insert package name]')
```

Once the package is installed, we can load it in by typing

```
## package name can have quotes or not -- it does not matter here
library([insert package name])
```

One of the most popular packages in R is the ggplot2 package, which creates great visualizations built on the idea of the "Grammar of Graphics". First we will install ggplot2 and then load it into our R environment.

```
## package name must be within quotes
install.packages('ggplot2')
```

```
library(ggplot2)
```

While the ggplot2 syntax may be difficult when first starting, it will become second nature in the long run. Unlike other plotting libraries we may run into in R, ggplot2 takes in dataframes and then layers other features on top.

Let us first load in data to use from the ggplot2 library:

```
## load data
data('diamonds', package = 'ggplot2')
```

In the "Environment" tab of RStudio, we should now see an object called diamonds, the dataframe we will use. To explore what the columns represent, we can use ?diamonds because it is an object in the ggplot2 package and will have documentation for it.

```
## show the first five rows of the data
print(head(diamonds))
```

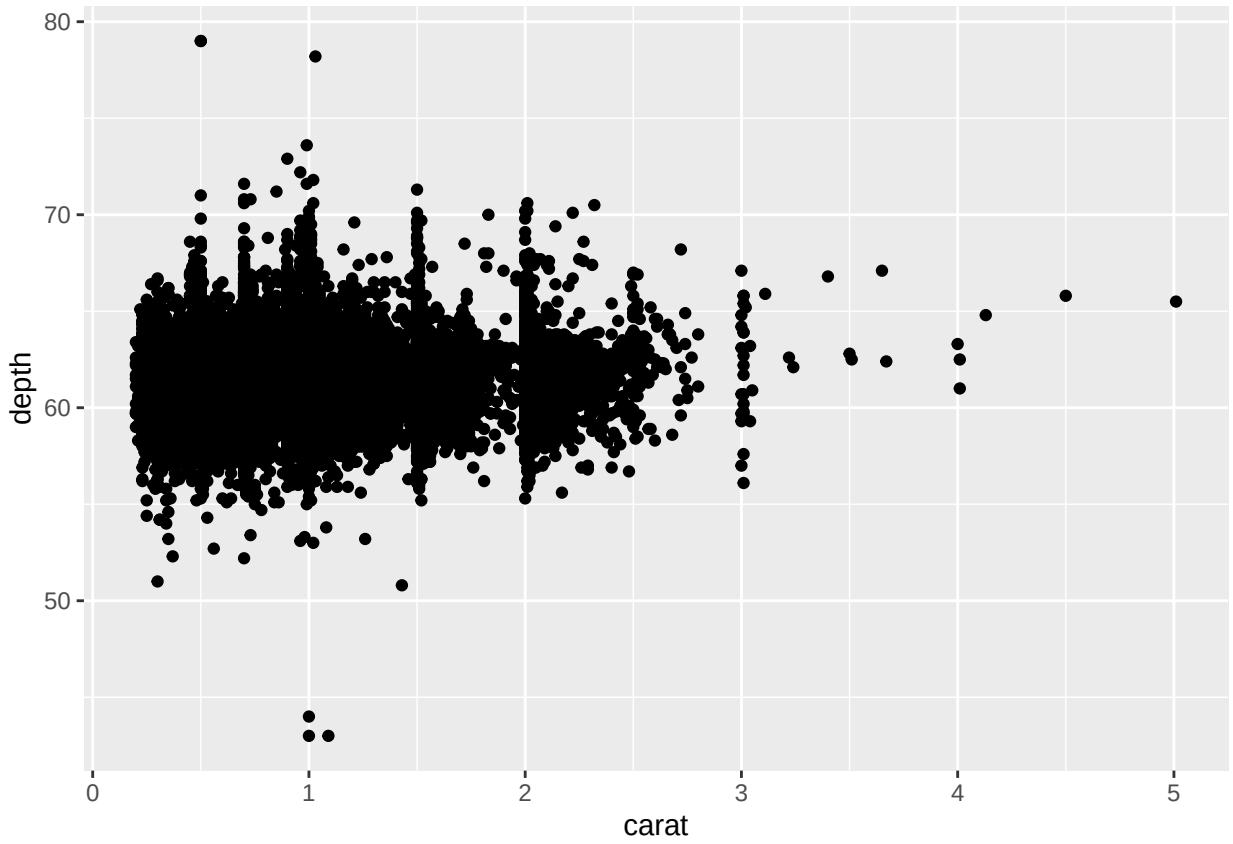
```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
## 3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

```
## show details about the diamonds dataset in the Help tab
?diamonds
```

The first layer of the ggplot2 visualization specifies where the data comes from, which in this case is the diamonds dataset. The next layers decide what data we will display and how to display it. For instance, to make a bar plot, we use \texttt{geom_bar}; to make a scatter plot, we use \texttt{geom_point}. Within the layer function, we use the aes function to specify which vectors to map onto each axis.

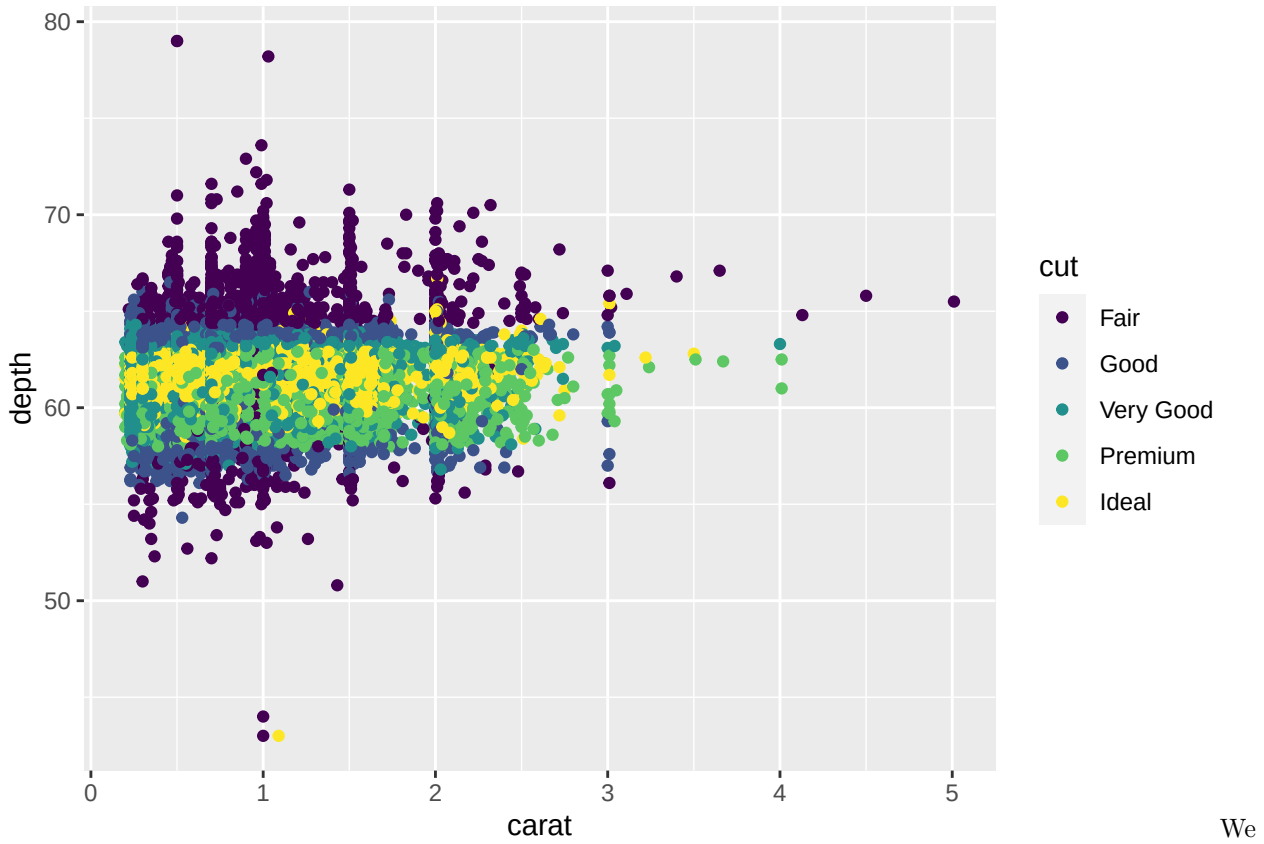
While plotting may seem complicated, perhaps an example will make more sense. Let us create a scatter plot with the number of carats on the x-axis and the depth on the y-axis:

```
ggplot(diamonds) +
  geom_point(aes(x = carat, y = depth))
```



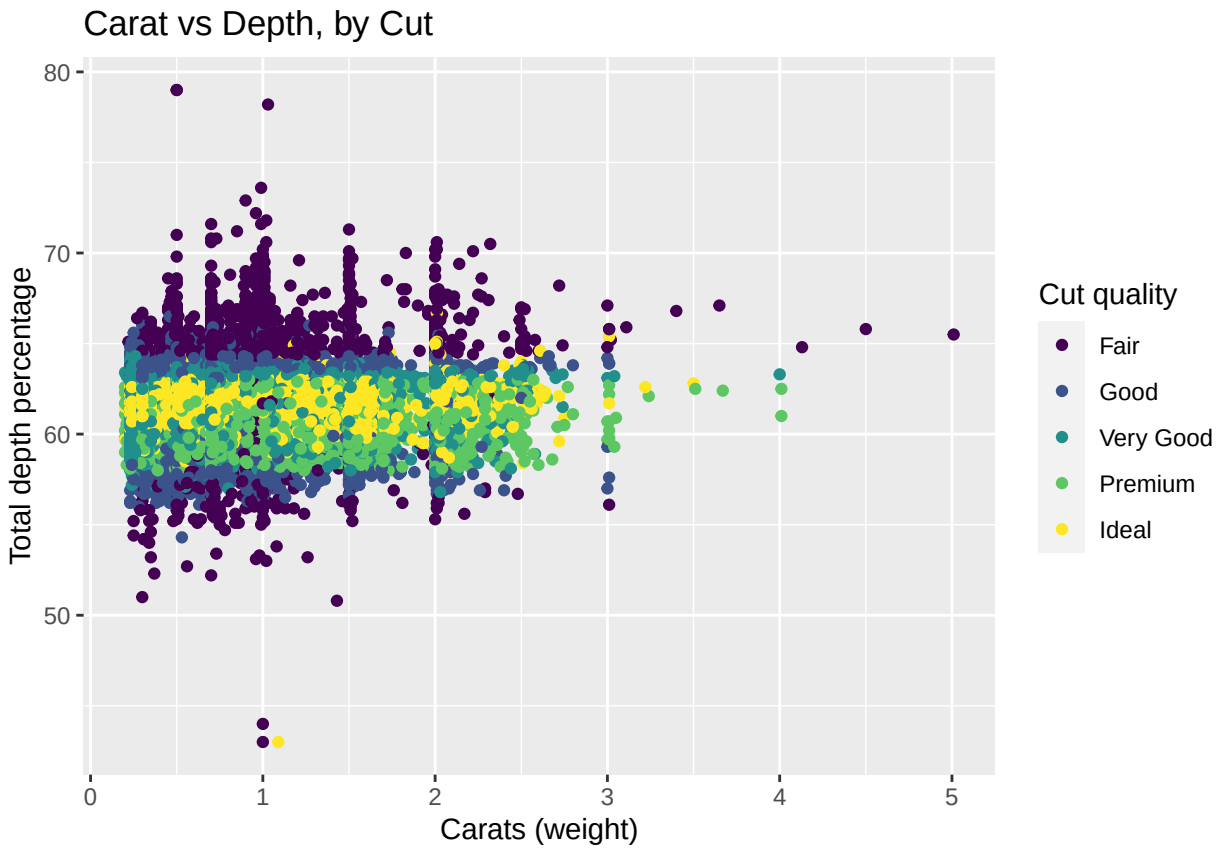
Notice that for each carat and depth observation, there is a cut associated with those values; we can take advantage of this mapping and further customize our visualization by coloring our points to display the cut of each observations:

```
ggplot(diamonds) +
  geom_point(aes(x = carat, y = depth, color = cut))
```



We can also add titles using the `labs` layer. Each aesthetic in the `aes` function can be relabeled in the `labs` function, as can the `title`, and the `subtitle`.

```
ggplot(diamonds) +
  geom_point(aes(x = carat, y = depth, color = cut)) +
  labs(title = 'Carat vs Depth, by Cut',
       x = 'Carats (weight)',
       y = 'Total depth percentage',
       color = 'Cut quality')
```



While `ggplot2` has many features, these basics should guide us through the rest of the content in this textbook.