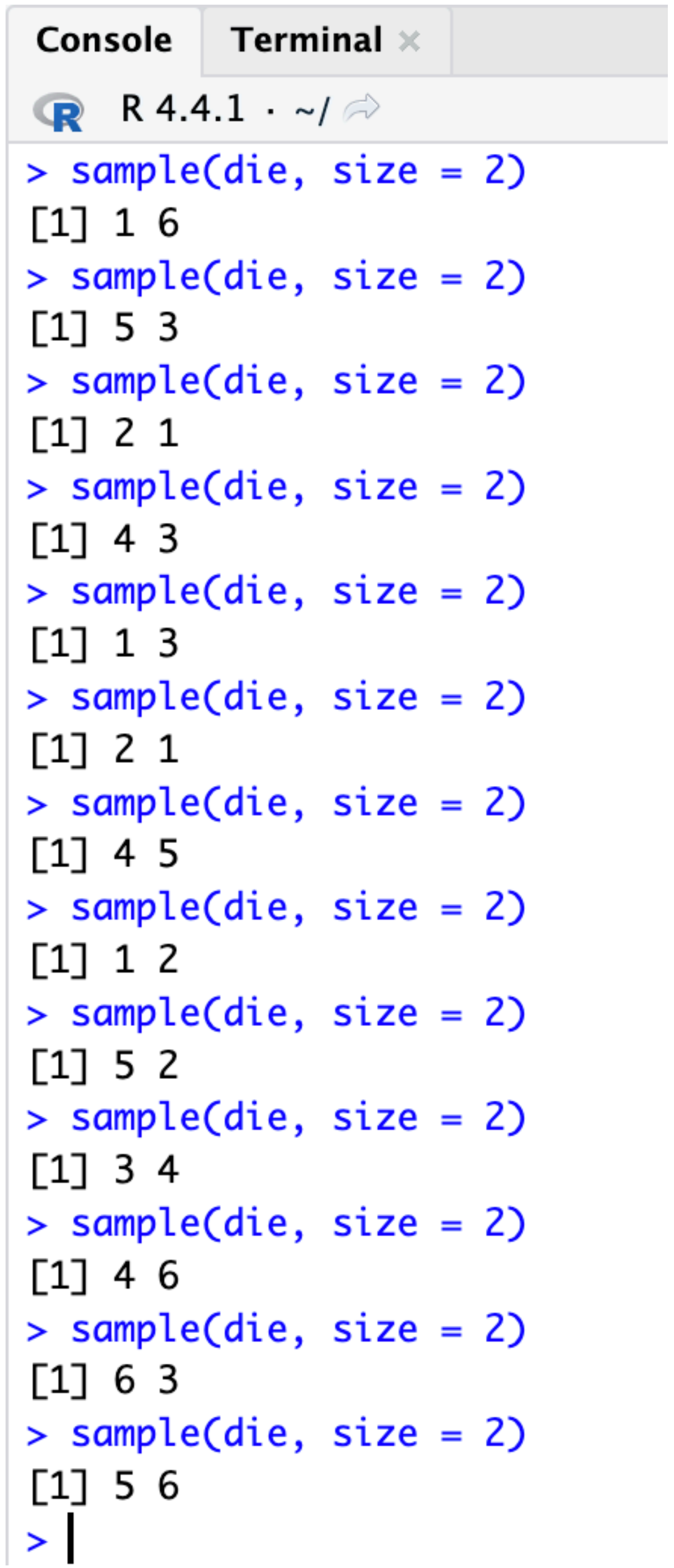
Basics

Sample with replacement

- With `sample` if you set `size = 2`, you can *almost* simulate a pair of dice.
- “Almost” because if you use it many times, you’ll notice that the second die never has the same value as the first die.
- By default, `sample` builds a sample without replacement.

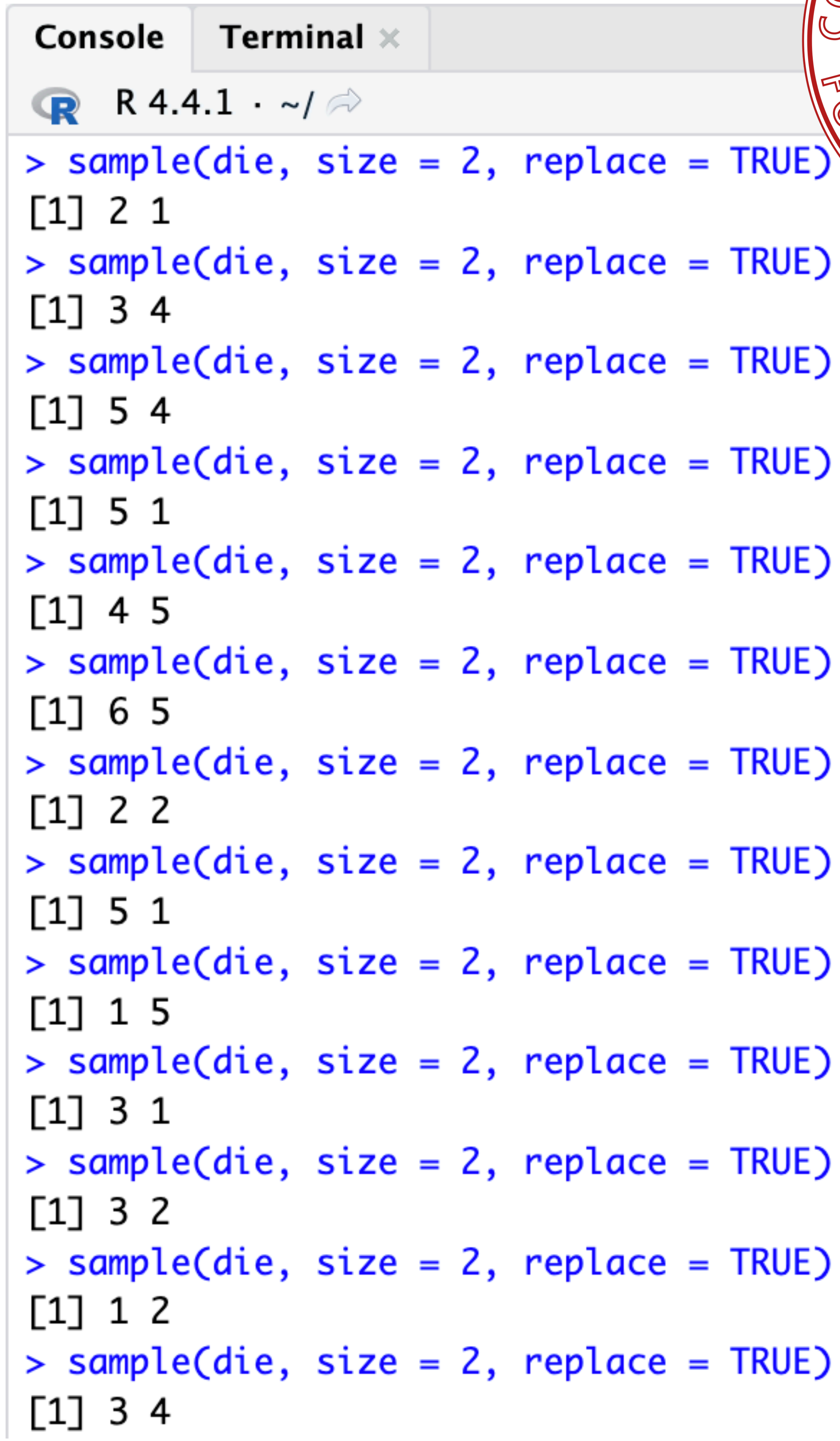


```
> sample(die, size = 2)
[1] 1 6
> sample(die, size = 2)
[1] 5 3
> sample(die, size = 2)
[1] 2 1
> sample(die, size = 2)
[1] 4 3
> sample(die, size = 2)
[1] 1 3
> sample(die, size = 2)
[1] 2 1
> sample(die, size = 2)
[1] 4 5
> sample(die, size = 2)
[1] 1 2
> sample(die, size = 2)
[1] 5 2
> sample(die, size = 2)
[1] 3 4
> sample(die, size = 2)
[1] 4 6
> sample(die, size = 2)
[1] 6 3
> sample(die, size = 2)
[1] 5 6
> |
```

Basics

Sample with replacement

- In the real world, when you roll a pair of dice, each die is independent of the other.
- If the first die comes up six, it does not prevent the second die from coming up six.
- You can recreate this behaviour in `sample` by adding the argument `replace = TRUE`
- The argument `replace = TRUE` causes `sample` to sample with replacement.

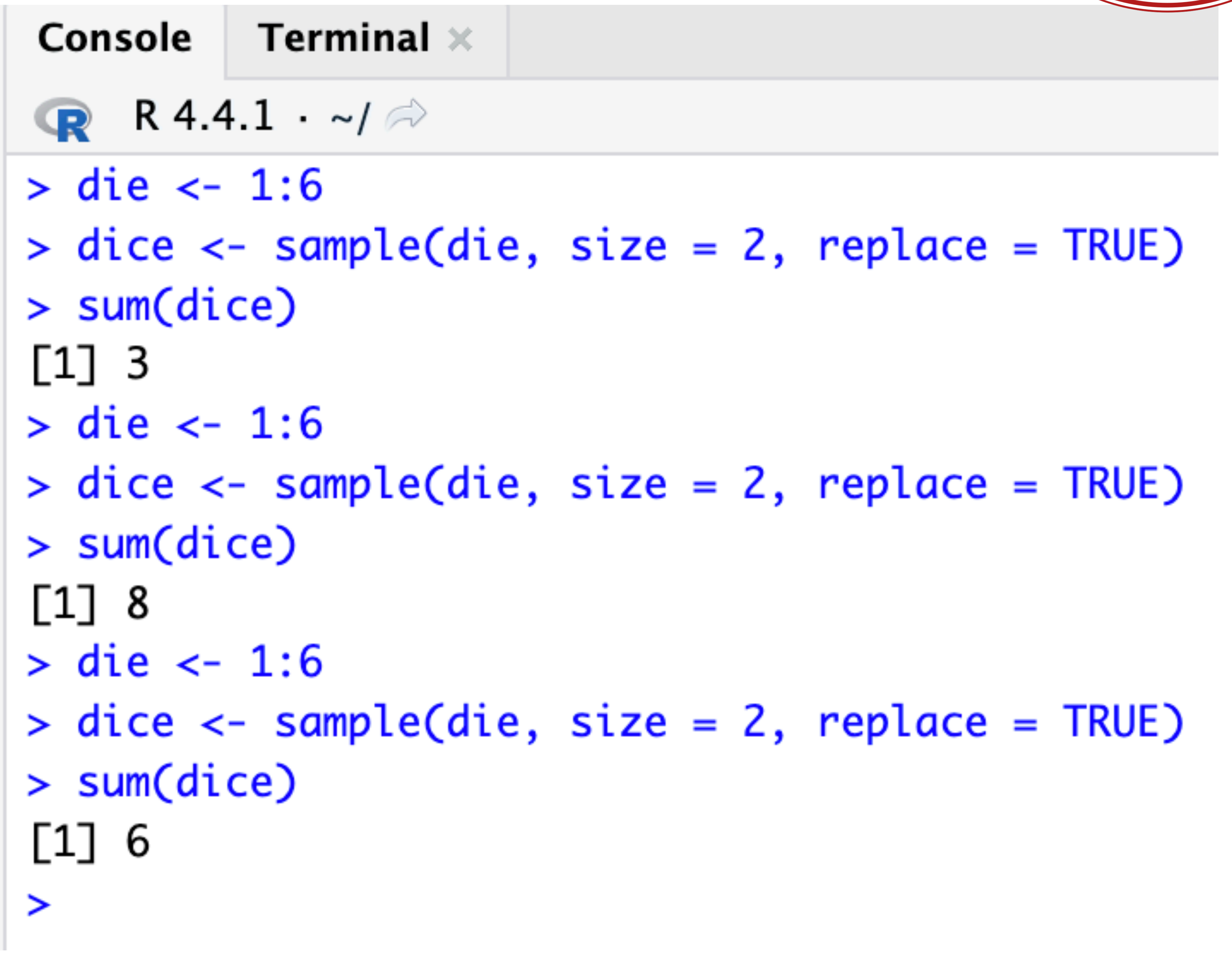


```
> sample(die, size = 2, replace = TRUE)
[1] 2 1
> sample(die, size = 2, replace = TRUE)
[1] 3 4
> sample(die, size = 2, replace = TRUE)
[1] 5 4
> sample(die, size = 2, replace = TRUE)
[1] 5 1
> sample(die, size = 2, replace = TRUE)
[1] 4 5
> sample(die, size = 2, replace = TRUE)
[1] 6 5
> sample(die, size = 2, replace = TRUE)
[1] 2 2
> sample(die, size = 2, replace = TRUE)
[1] 5 1
> sample(die, size = 2, replace = TRUE)
[1] 1 5
> sample(die, size = 2, replace = TRUE)
[1] 3 1
> sample(die, size = 2, replace = TRUE)
[1] 3 2
> sample(die, size = 2, replace = TRUE)
[1] 1 2
> sample(die, size = 2, replace = TRUE)
[1] 3 4
```

Basics

Writing your own functions

- You already have working R code that simulates rolling a pair of dice and summing the result.
- You can retype this code into the console anytime you want to re-roll your dice.
- However, this is an awkward way to work with the code.
- It would be easier to use your code if you wrapped it into its own function, which is exactly what we'll do now.

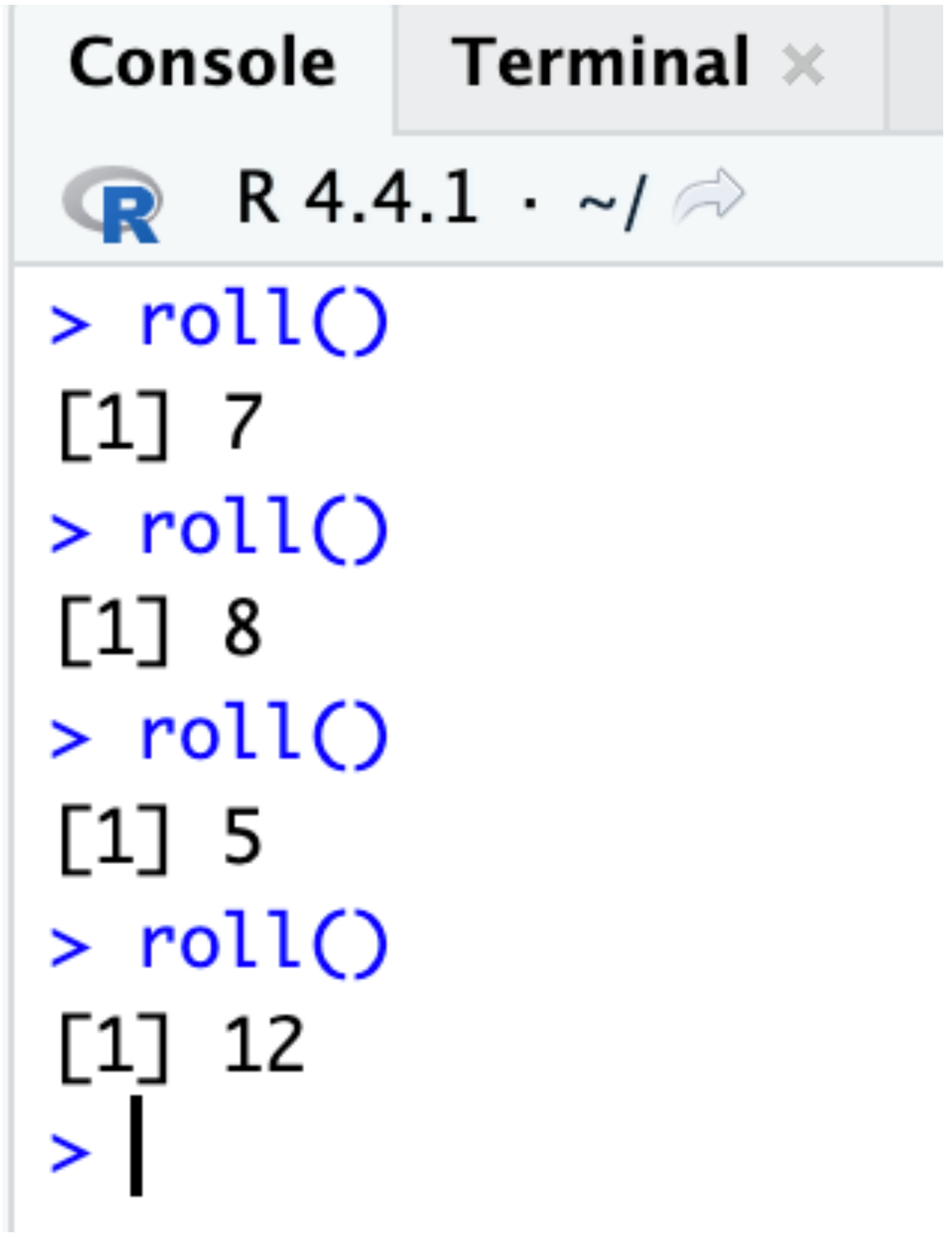


```
> die <- 1:6
> dice <- sample(die, size = 2, replace = TRUE)
> sum(dice)
[1] 3
> die <- 1:6
> dice <- sample(die, size = 2, replace = TRUE)
> sum(dice)
[1] 8
> die <- 1:6
> dice <- sample(die, size = 2, replace = TRUE)
> sum(dice)
[1] 6
>
```

Basics

Writing your own functions

- We’re going to write a function named `roll` that you can use to roll your virtual dice.
- When you’re finished, the function will work like this: each time you call `roll()`, R will return the sum of rolling two dice.

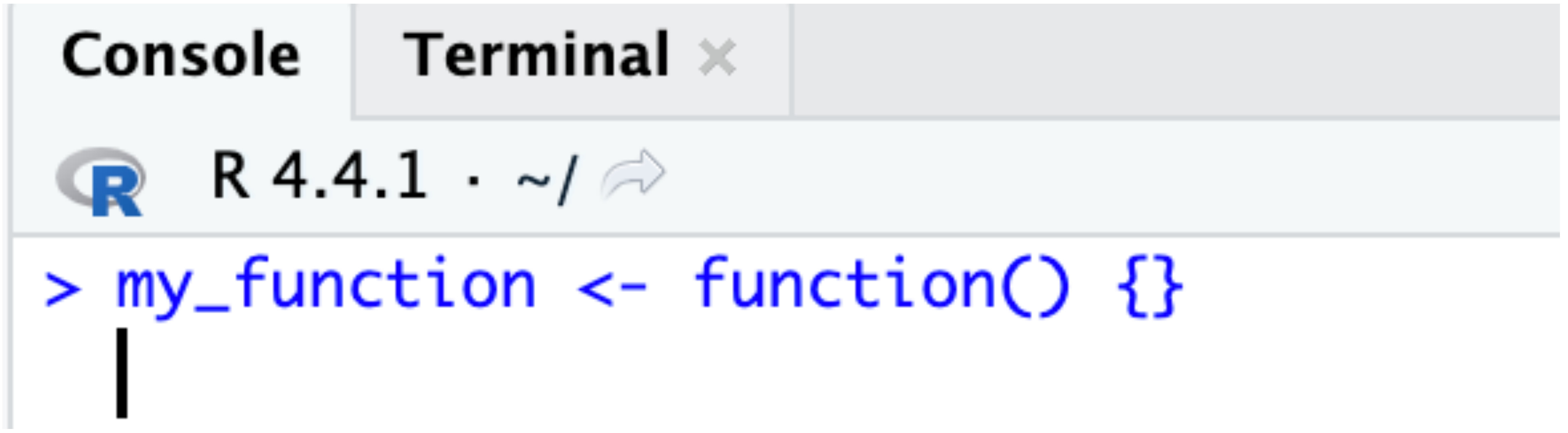


```
> roll()
[1] 7
> roll()
[1] 8
> roll()
[1] 5
> roll()
[1] 12
> 
```

Basics

Function constructor

- Every function in R has three basic parts: a name, a body of code, and a set of arguments.
- To make your own function, you need to replicate these parts and store them in an R object, which you can do with the `function` function.
- To do this, call `function()` and follow it with a pair of braces, `{}.`
- `function` will build a function out of whatever R code you place between the braces.

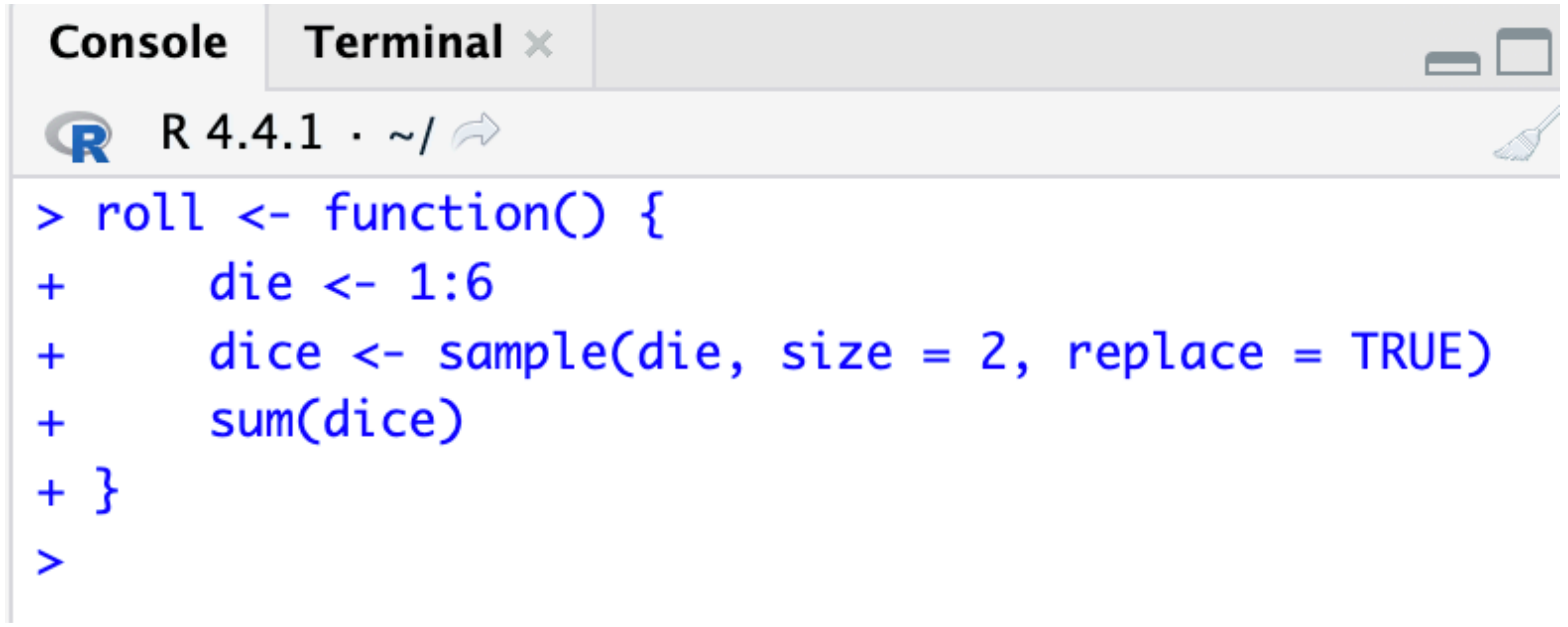


```
> my_function <- function() {}
```

Basics

Function constructor

- You can turn your dice code into a function by calling `roll()`
- Notice that I indented each line of code between the braces. This makes the code easier for you and me to read but has no impact on how the code runs. R ignores spaces and line breaks and executes one complete expression at a time.
- Just hit the Enter key between each line after the first brace, `{`. R will wait for you to type the last brace, `}`, before it responds.

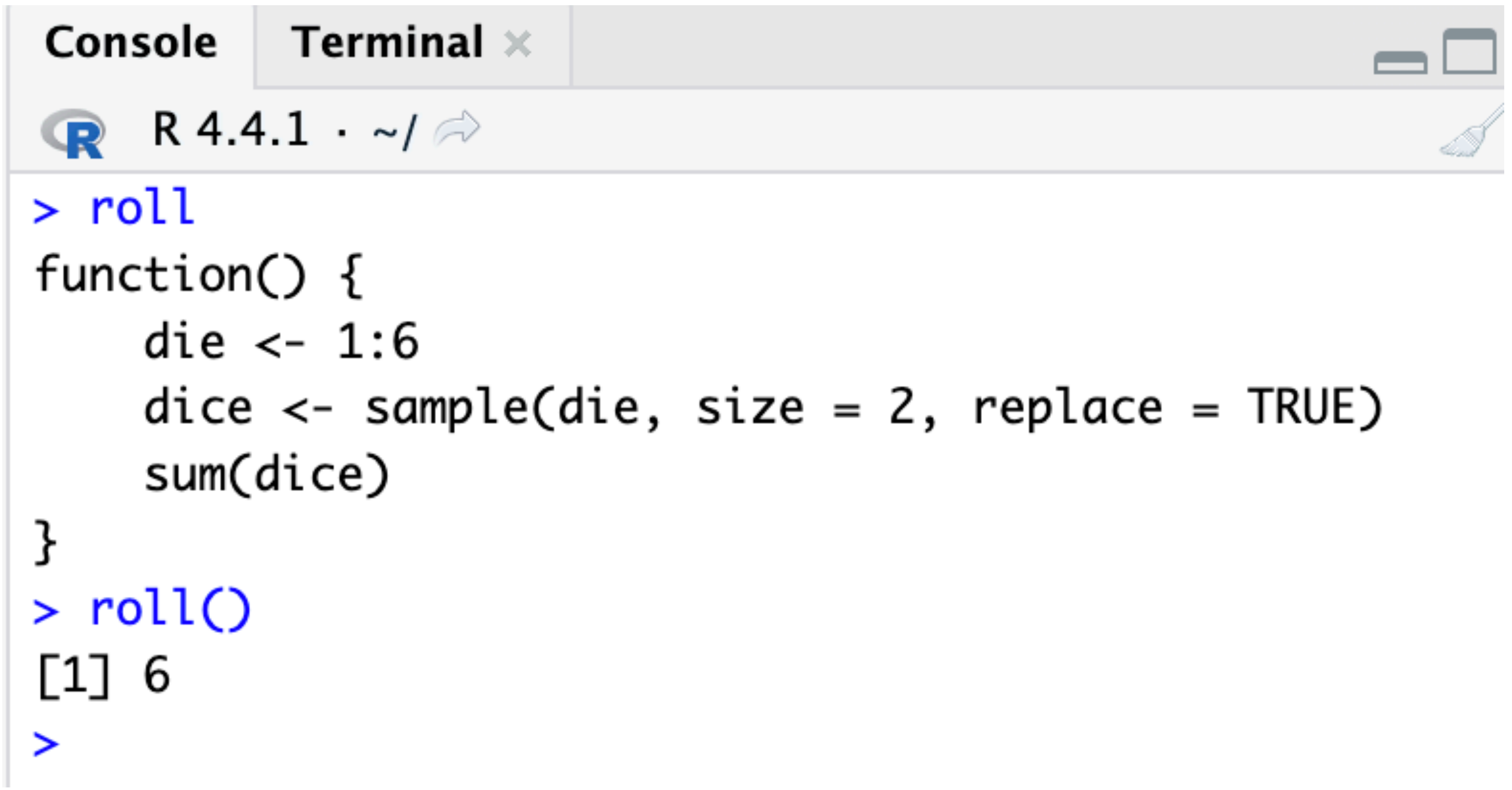


```
> roll <- function() {
+     die <- 1:6
+     dice <- sample(die, size = 2, replace = TRUE)
+     sum(dice)
+ }
>
```

Basics

Function constructor

- Don’t forget to save the output of `function` to an R object.
- To use it, write the object’s name followed by an open and closed parenthesis.
- You can think of the parentheses as the “trigger” that causes R to run the function.
- If you type in a function’s name *without* the parentheses, R will show you the code that is stored inside the function. If you type in the name *with* the parentheses, R will run that code.



```
> roll
function() {
    die <- 1:6
    dice <- sample(die, size = 2, replace = TRUE)
    sum(dice)
}
> roll()
[1] 6
>
```

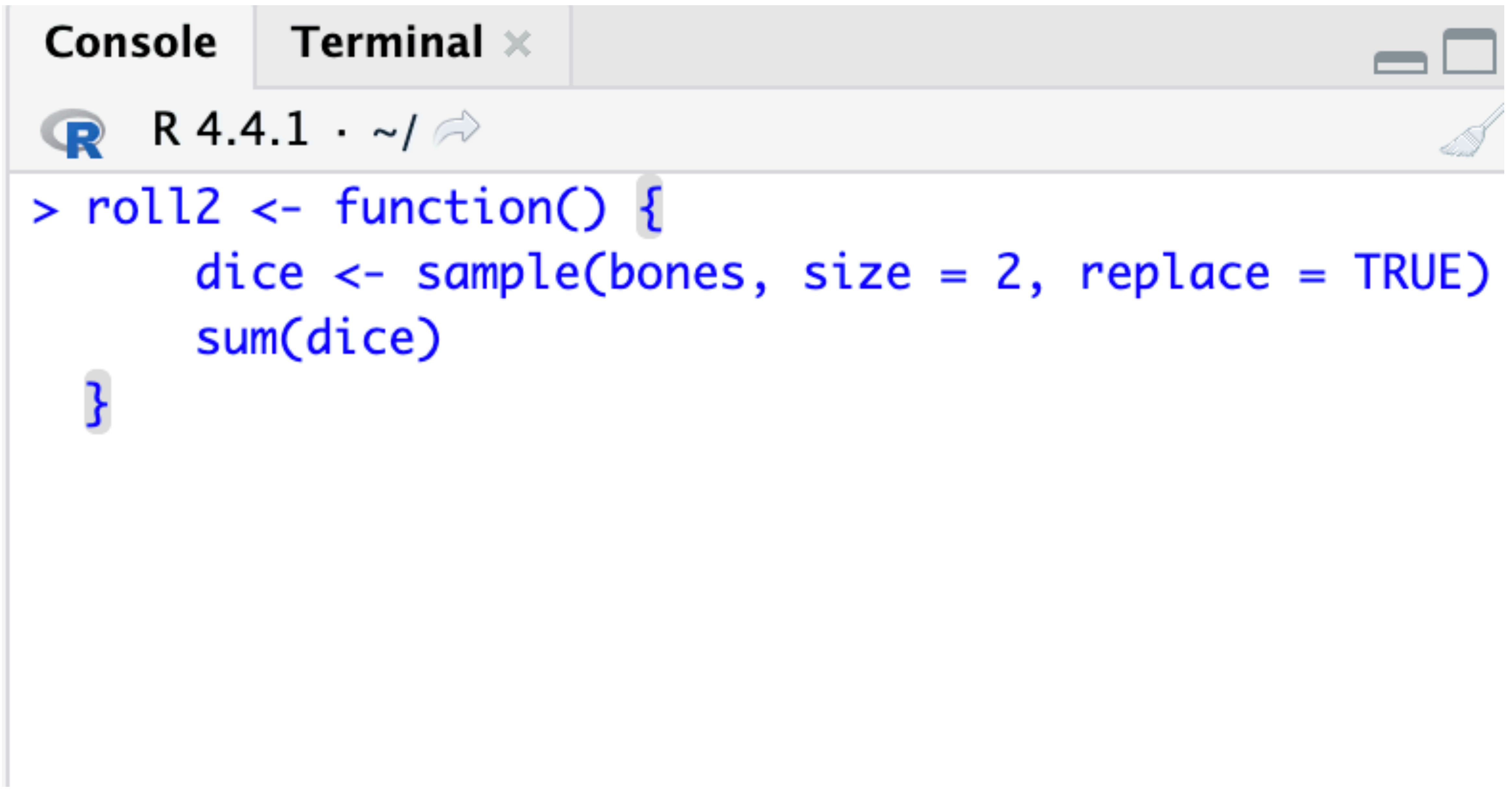
Basics
Function constructor

- The code that you place inside your function is known as the *body* of the function. When you run a function in R, R will execute all of the code in the body and then return the result of the last line of code. If the last line of code doesn’t return a value, neither will your function, so you want to ensure that your final line of code returns a value.

Basics

Arguments

- What if we removed one line of code from our function and changed the name `die` to `bones`, like this?

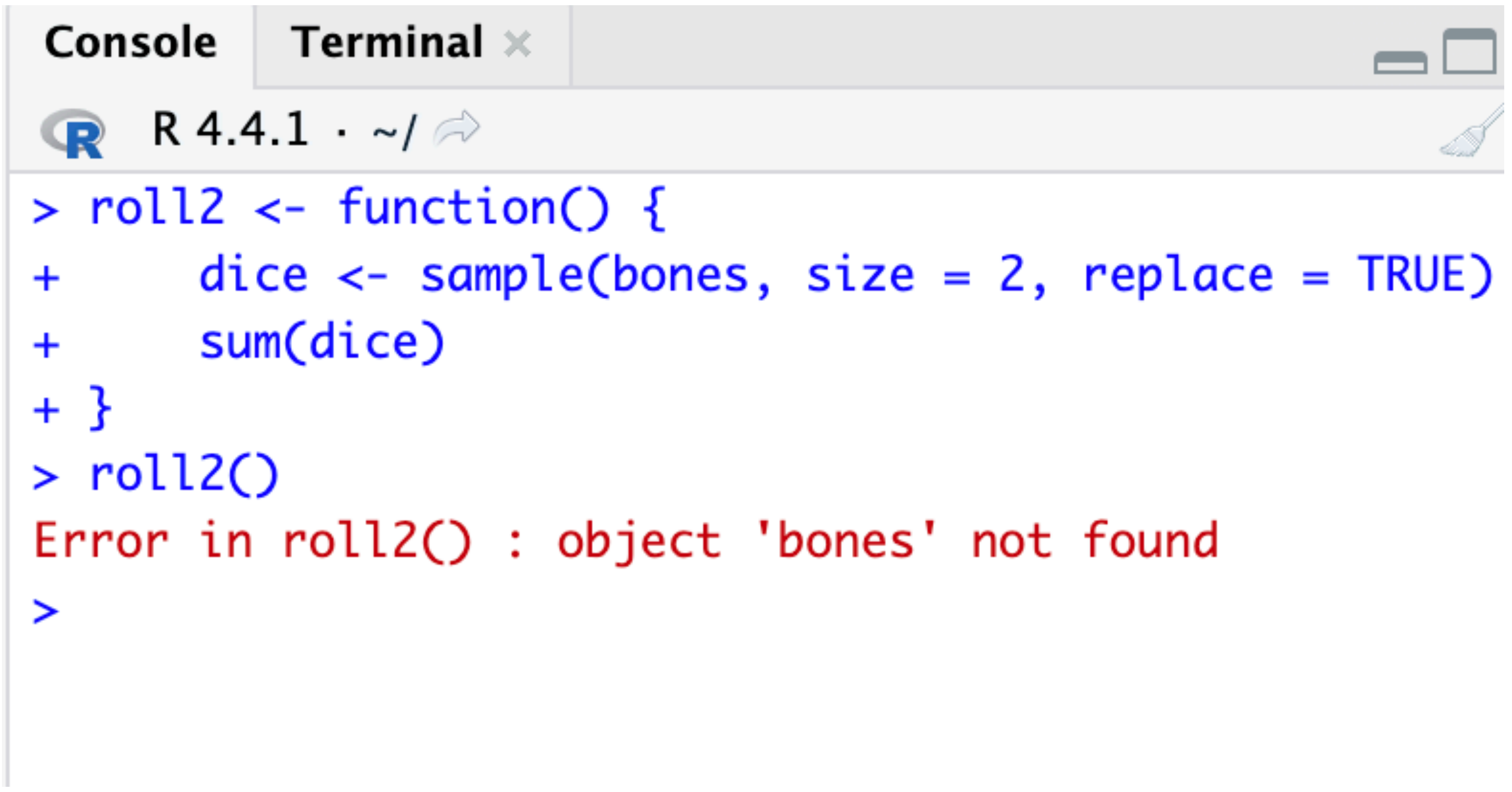


```
> roll2 <- function() {
      dice <- sample(bones, size = 2, replace = TRUE)
      sum(dice)
  }
```

Basics

Arguments

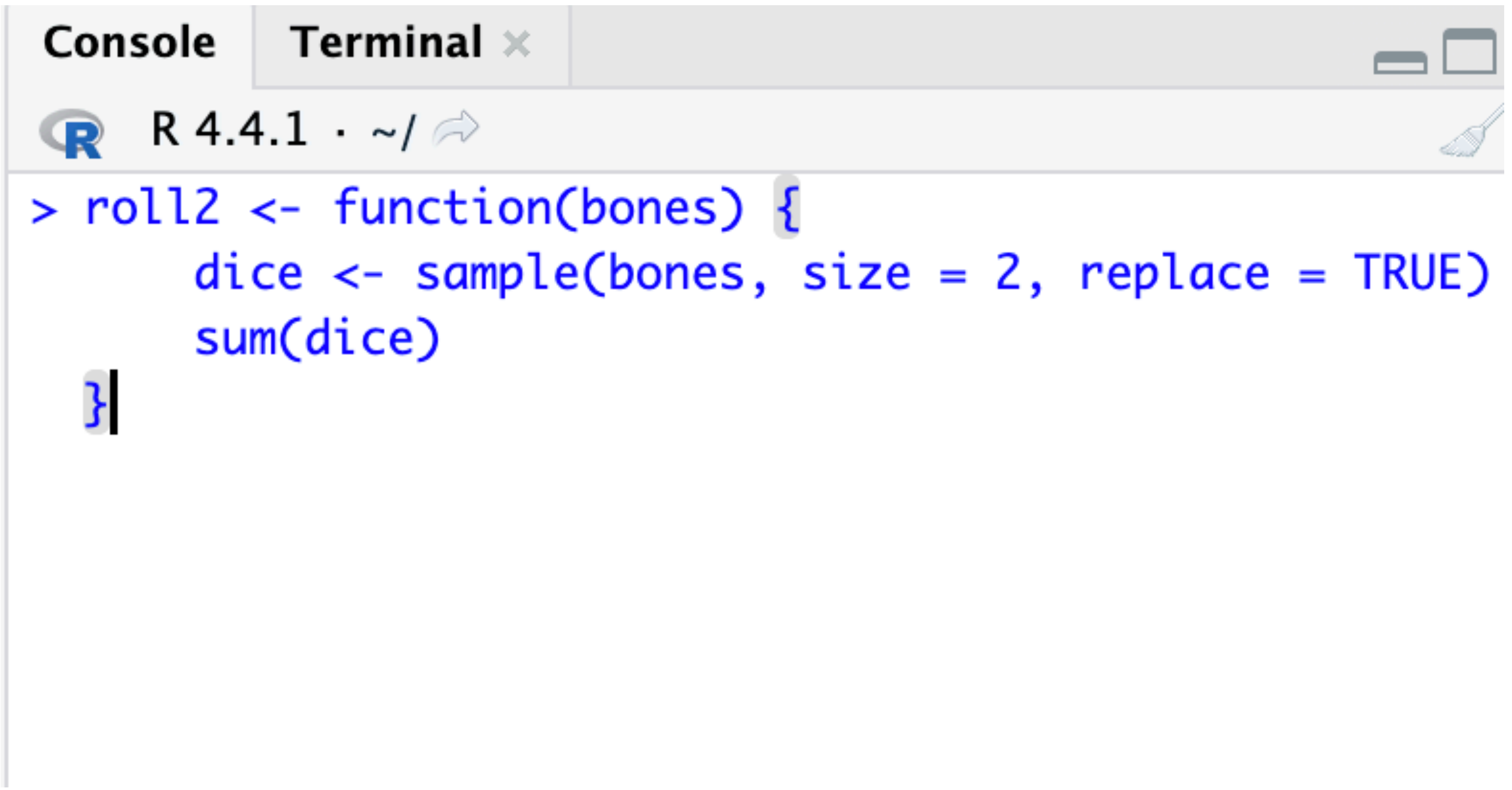
- Now I’ll get an error when I run the function. The function needs the object bones to do its job, but there is no object named bones to be found.



```
> roll2 <- function() {
+     dice <- sample(bones, size = 2, replace = TRUE)
+     sum(dice)
+ }
> roll2()
Error in roll2() : object 'bones' not found
>
```

Basics
Arguments

- You can supply `bones` when you call `roll2` if you make `bones` an argument of the function. To do this, put the name `bones` in the parentheses that follow `function` when you define `roll2`:
- Now `roll2` will work as long as you supply `bones` when you call the function. You can take advantage of this to roll different types of dice each time you call `roll2`

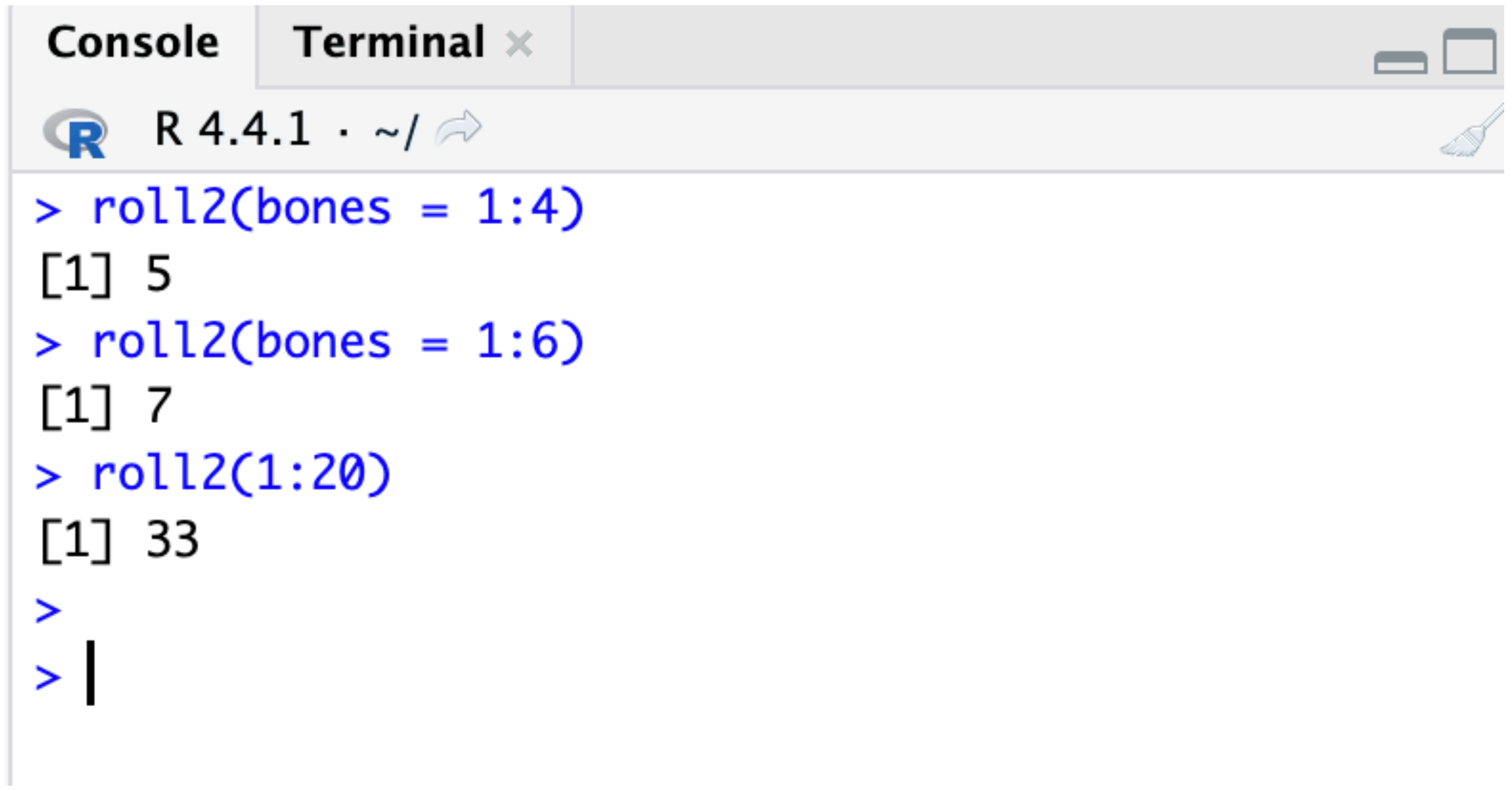


```
> roll2 <- function(bones) {
      dice <- sample(bones, size = 2, replace = TRUE)
      sum(dice)
  }
```

Basics

Arguments

- Remember, we're rolling pairs of dice.
- Now `roll2` will work as long as you supply `bones` when you call the function. You can take advantage of this to roll different types of dice each time you call `roll2`

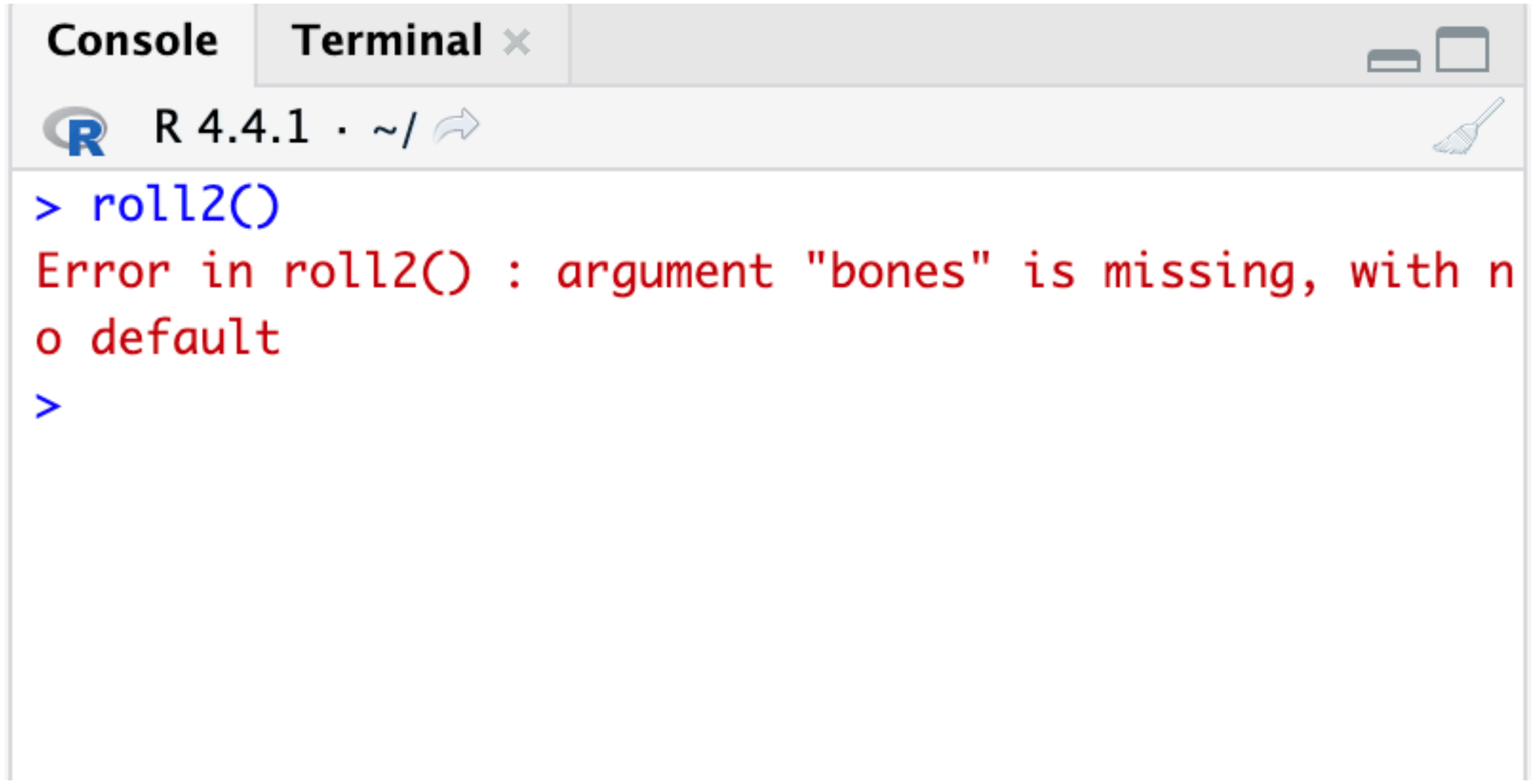


```
> roll2(bones = 1:4)
[1] 5
> roll2(bones = 1:6)
[1] 7
> roll2(1:20)
[1] 33
>
> |
```

Basics

Arguments

- Notice that `roll2` will still give an error if you do not supply a value for the `bones` argument when you call `roll2`

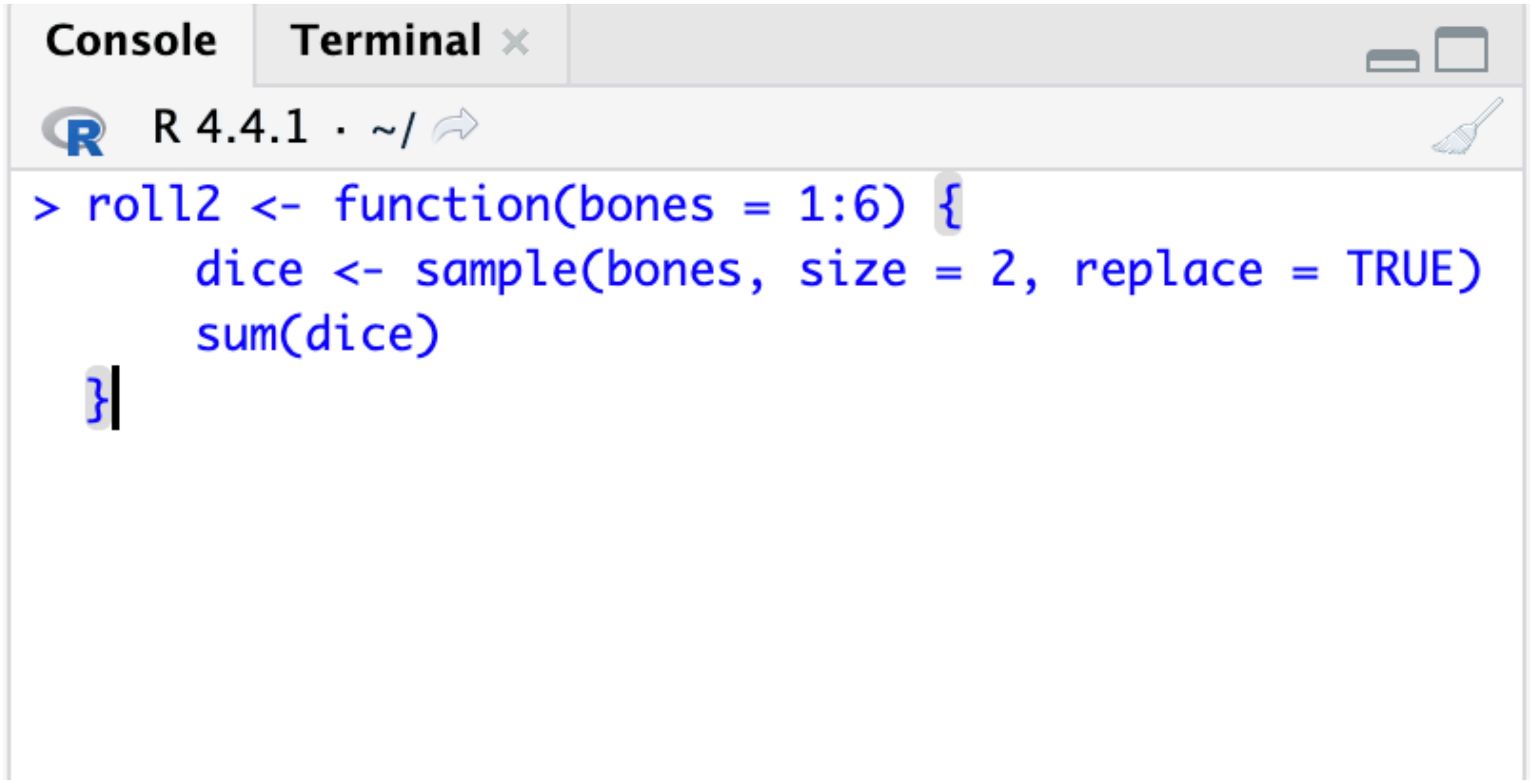


```
> roll2()
Error in roll2() : argument "bones" is missing, with n
o default
>
```

Basics

Arguments

- You can prevent this error by giving the `bones` argument a default value. To do this, set `bones` equal to a value when you define `roll2`
- Now you can supply a new value for `bones` if you like, and `roll2` will use the default if you do not.



```
> roll2 <- function(bones = 1:6) {
      dice <- sample(bones, size = 2, replace = TRUE)
      sum(dice)
  }
```

Basics

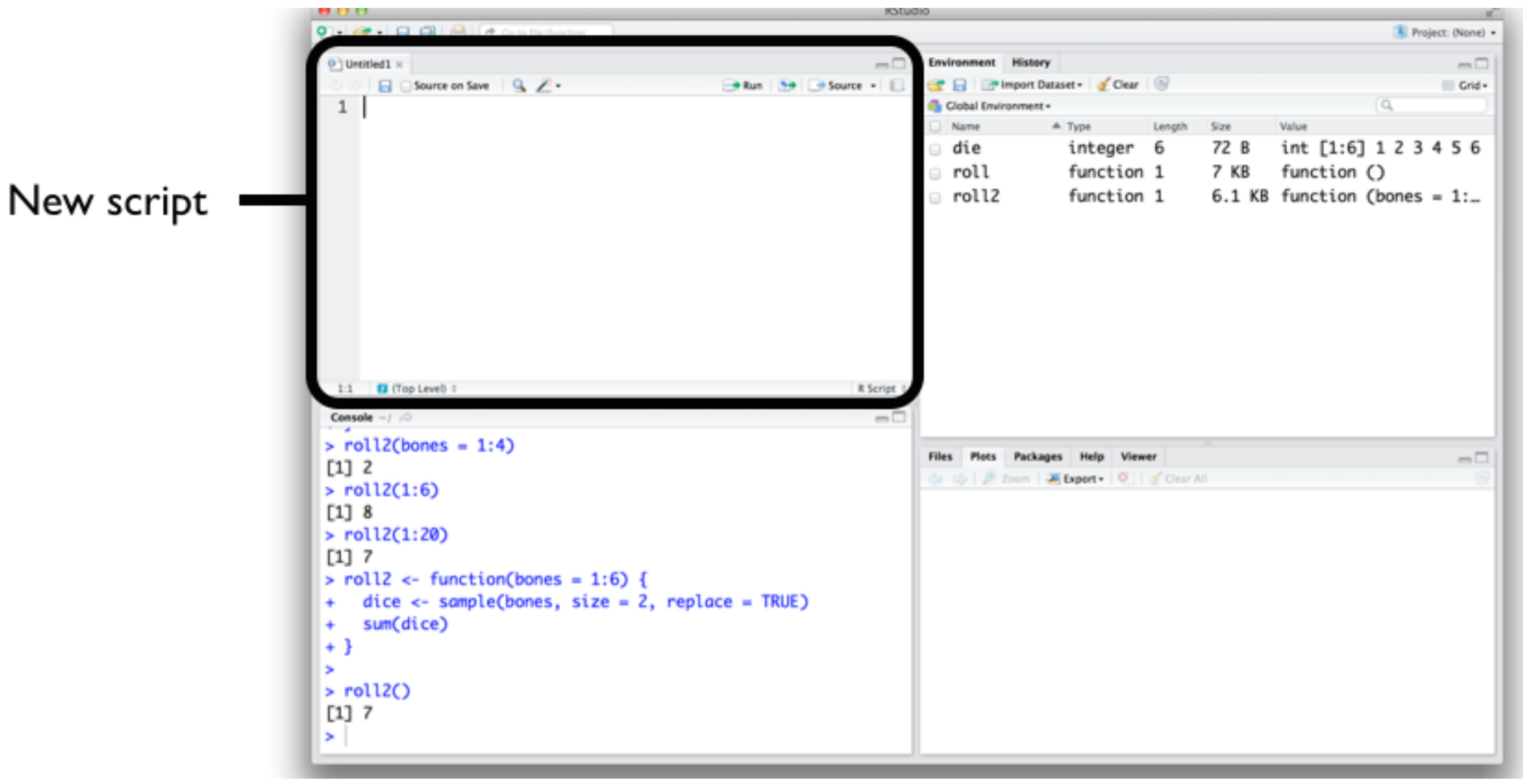
Arguments

- You can give your functions as many arguments as you like. Just list their names, separated by commas, in the parentheses that follow `function`. When the function is run, R will replace each argument name in the function body with the value that the user supplies for the argument. If the user does not supply a value, R will replace the argument name with the argument's default value (if you defined one).

Basics

Scripts

- What if you want to edit `roll2` again? You could go back and retype each line of code in `roll2`, but it would be so much easier if you had a draft of the code to start from. You can create a draft of your code as you go by using an R *script*. An R script is just a plain text file that you save R code in. You can open an R script in RStudio by going to `File > New File > R script` in the menu bar. RStudio will then open a fresh script above your console pane.

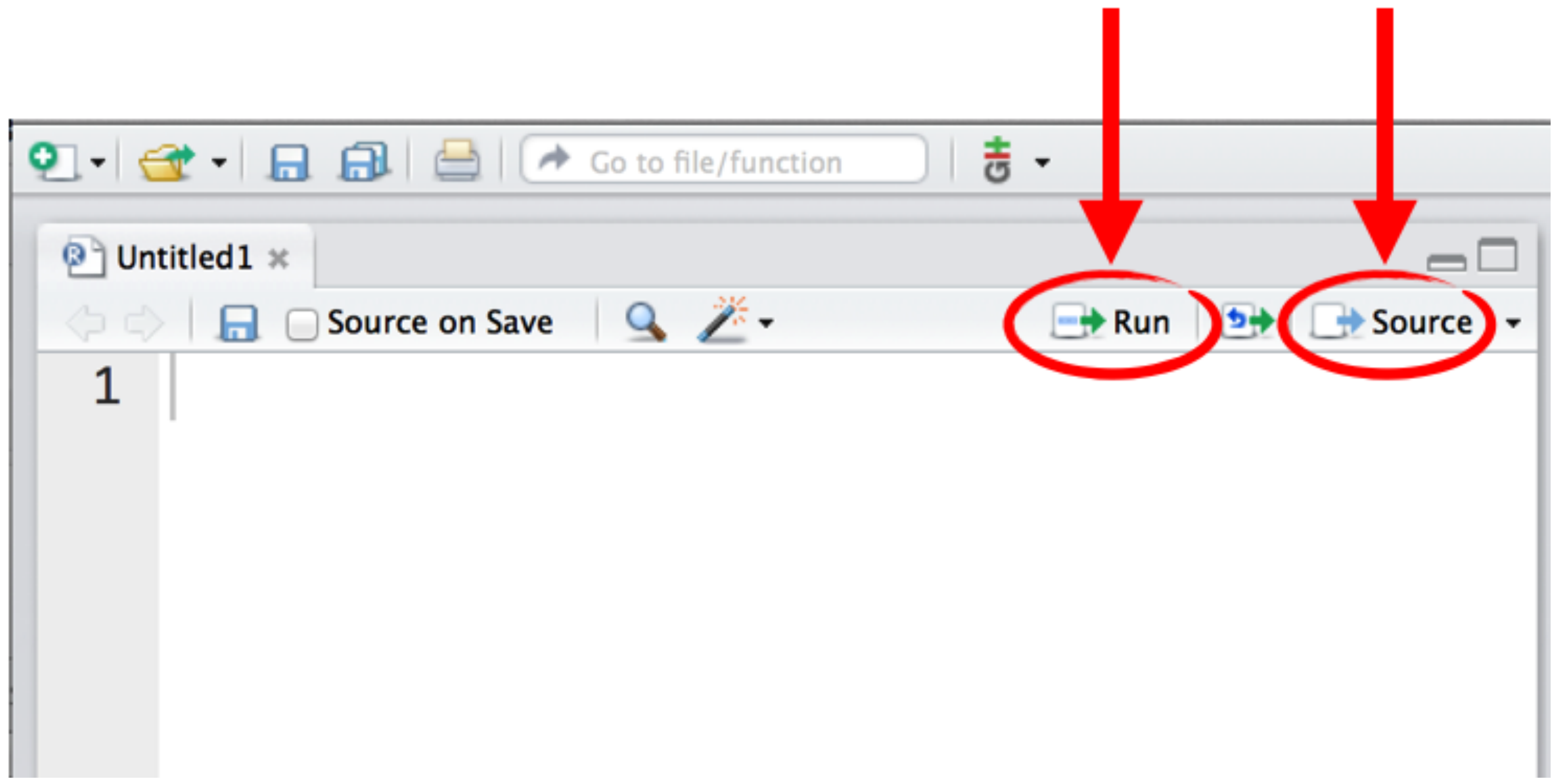


Name	Type	Length	Size	Value
die	integer	6	72 B	int [1:6] 1 2 3 4 5 6
roll	function	1	7 KB	function ()
roll2	function	1	6.1 KB	function (bones = 1:…

```
> roll2(bones = 1:4)
[1] 2
> roll2(1:6)
[1] 8
> roll2(1:20)
[1] 7
> roll2 <- function(bones = 1:6) {
+   dice <- sample(bones, size = 2, replace = TRUE)
+   sum(dice)
+ }
>
> roll2()
[1] 7
>
```

Basics
Scripts

- RStudio comes with many built-in features that make it easy to work with scripts. First, you can automatically execute a line of code in a script by clicking the Run button.
- R will run whichever line of code your cursor is on. If you have a whole section highlighted, R will run the highlighted code. Alternatively, you can run the entire script by clicking the Source button. Don't like clicking buttons? You can use Control + Return as a shortcut for the Run button. On Macs, that would be Command + Return.



Packages

Introduction

- You now have a function that simulates rolling a pair of dice. Let's make things a little more interesting by weighting the dice in your favor. Let's make the dice roll high numbers slightly more often than it rolls low numbers.
- Before we weight the dice, we should make sure that they are fair to begin with. Two tools will help you do this: *repetition* and *visualization*.
- We will repeat our dice rolls with a function called `replicate`, and we will visualize our rolls with a function called `qplot`. `qplot` does not come with R when you download it; `qplot` comes in a standalone R package. Many of the most useful R tools come in R packages, so let's take a moment to look at what R packages are and how you can use them.

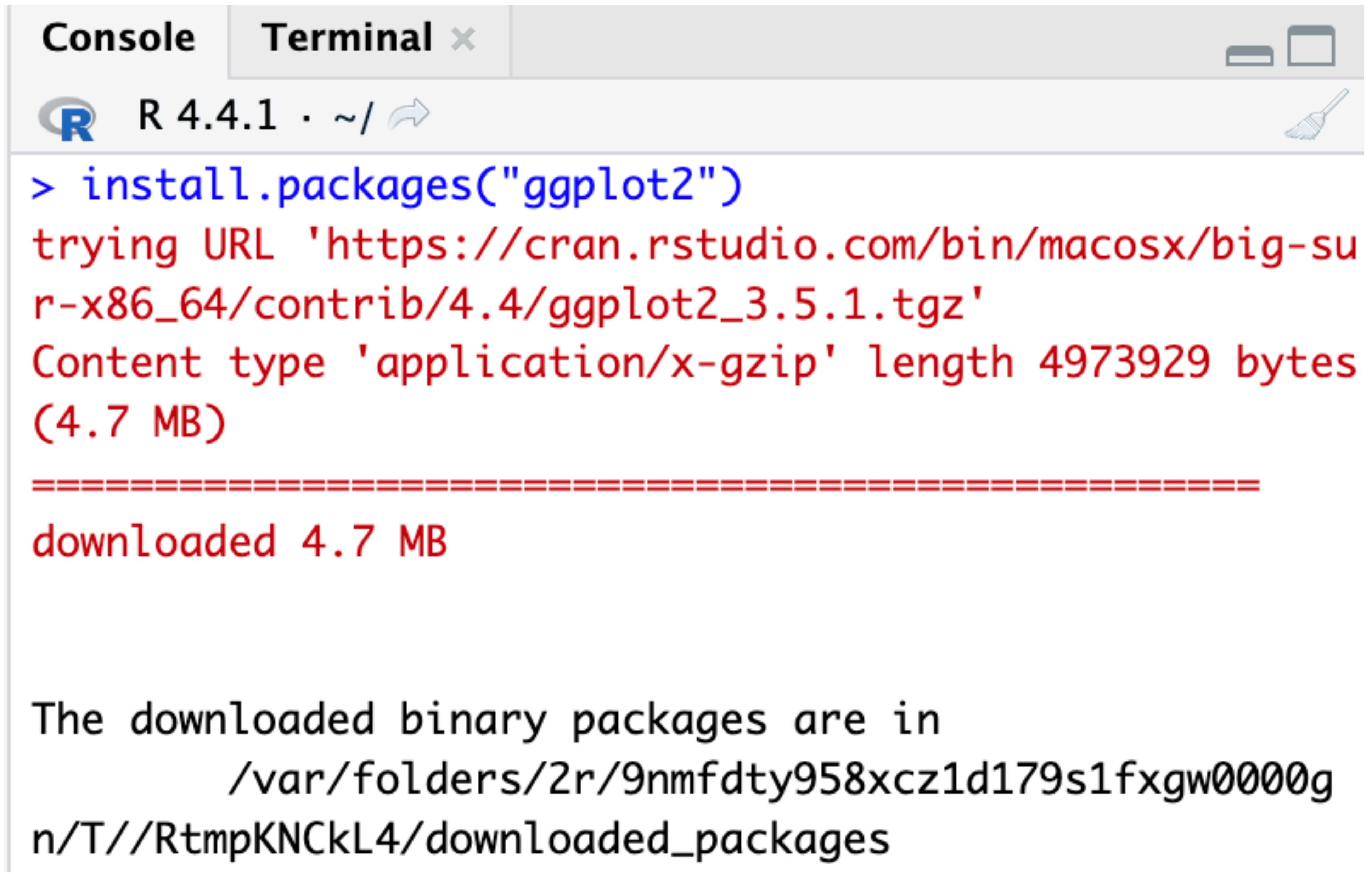
Packages
Introduction

- You’re not the only person writing your own functions with R. Many professors, programmers, and statisticians use R to design tools that can help people analyze data. They then make these tools free for anyone to use. To use these tools, you just have to download them.
- They come as preassembled collections of functions and objects called packages.
- We’re going to use the `qplot` function to make some quick plots. `qplot` comes in the *ggplot2* package, a popular package for making graphs. Before you can use `qplot`, or anything else in the ggplot2 package, you need to download and install it.

Packages

Install.packages

- Each R package is hosted at http://cran.r-project.org, the same website that hosts R.
- However, you don't need to visit the website to download an R package; you can download packages straight from R's command line.
- Open RStudio.
- Make sure you are connected to the Internet.
- Run `install.packages("ggplot2")`



```
> install.packages("ggplot2")
trying URL 'https://cran.rstudio.com/bin/macosx/big-su
r-x86_64/contrib/4.4/ggplot2_3.5.1.tgz'
Content type 'application/x-gzip' length 4973929 bytes
(4.7 MB)
==================================================
downloaded 4.7 MB


The downloaded binary packages are in
	/var/folders/2r/9nmfdty958xcz1d179s1fxgw0000g
n/T//RtmpKNCkL4/downloaded_packages
```

Packages

Install.packages

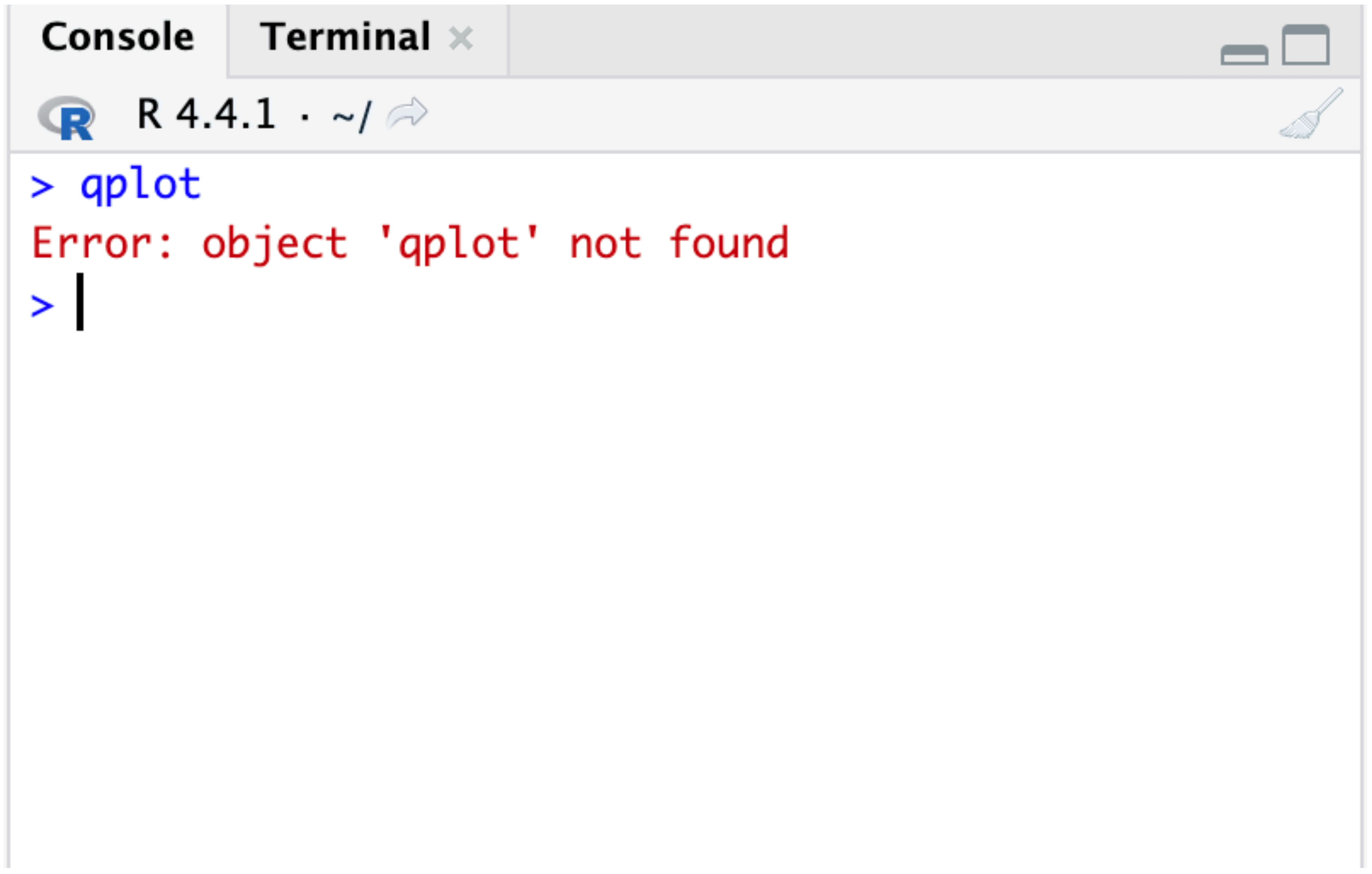
- That’s it. R will have your computer visit the website, download ggplot2, and install the package in your hard drive right where R wants to find it. You now have the ggplot2 package. If you would like to install another package, replace ggplot2 with your package name in the code.

Packages
Library

- Installing a package doesn't place its functions at your fingertips just yet: it simply places them in your hard drive. To use an R package, you next have to load it in your R session with the command `library("ggplot2")`.
- If you would like to load a different package, replace ggplot2 with your package name in the code.

Packages
Library

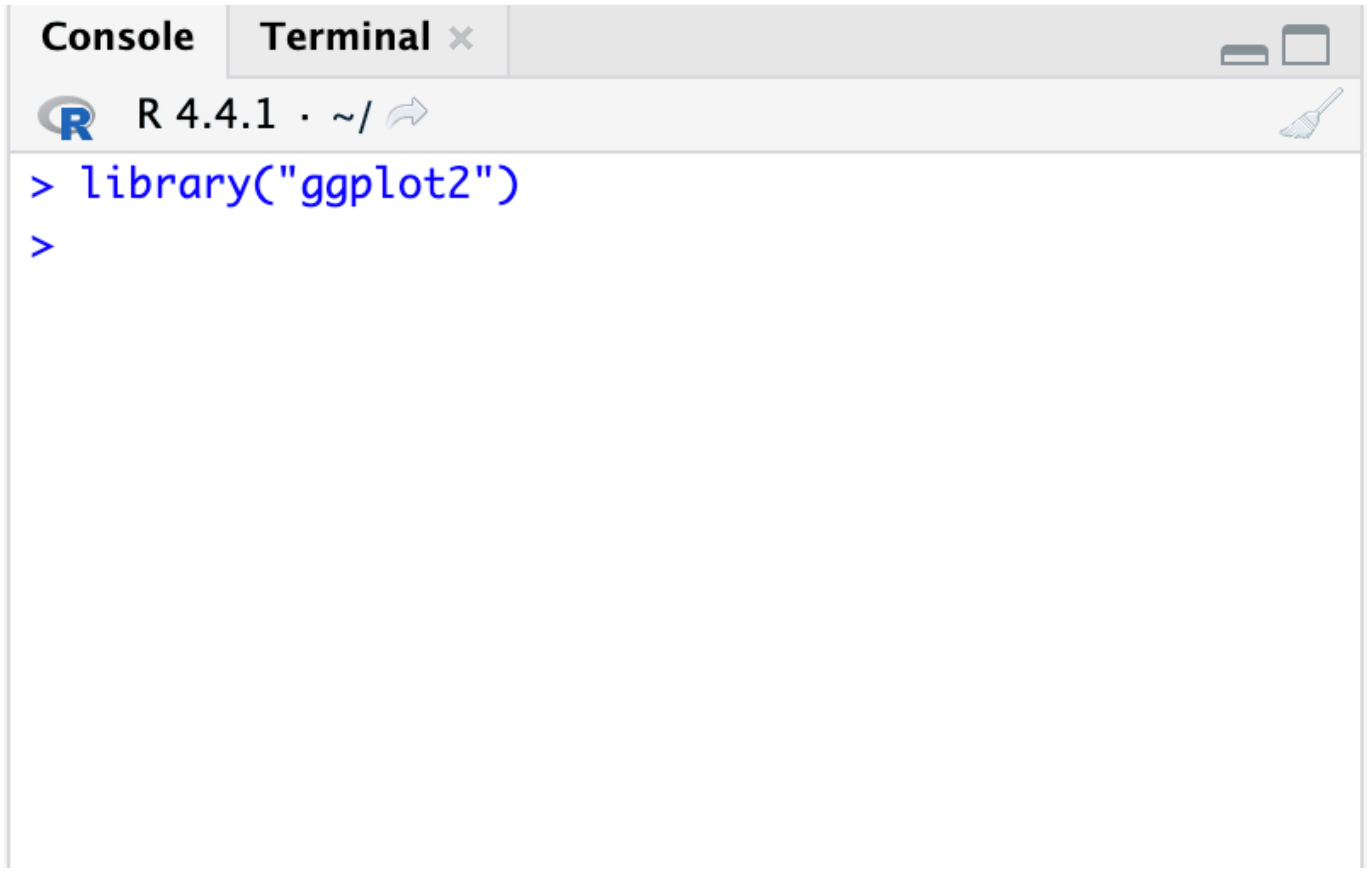
- To see what this does, try an experiment. First, ask R to show you the `qplot` function.
- R won't be able to find `qplot` because `qplot` lives in the ggplot2 package, which you haven't loaded.



```
> qplot
Error: object 'qplot' not found
> |
```

Packages
Library

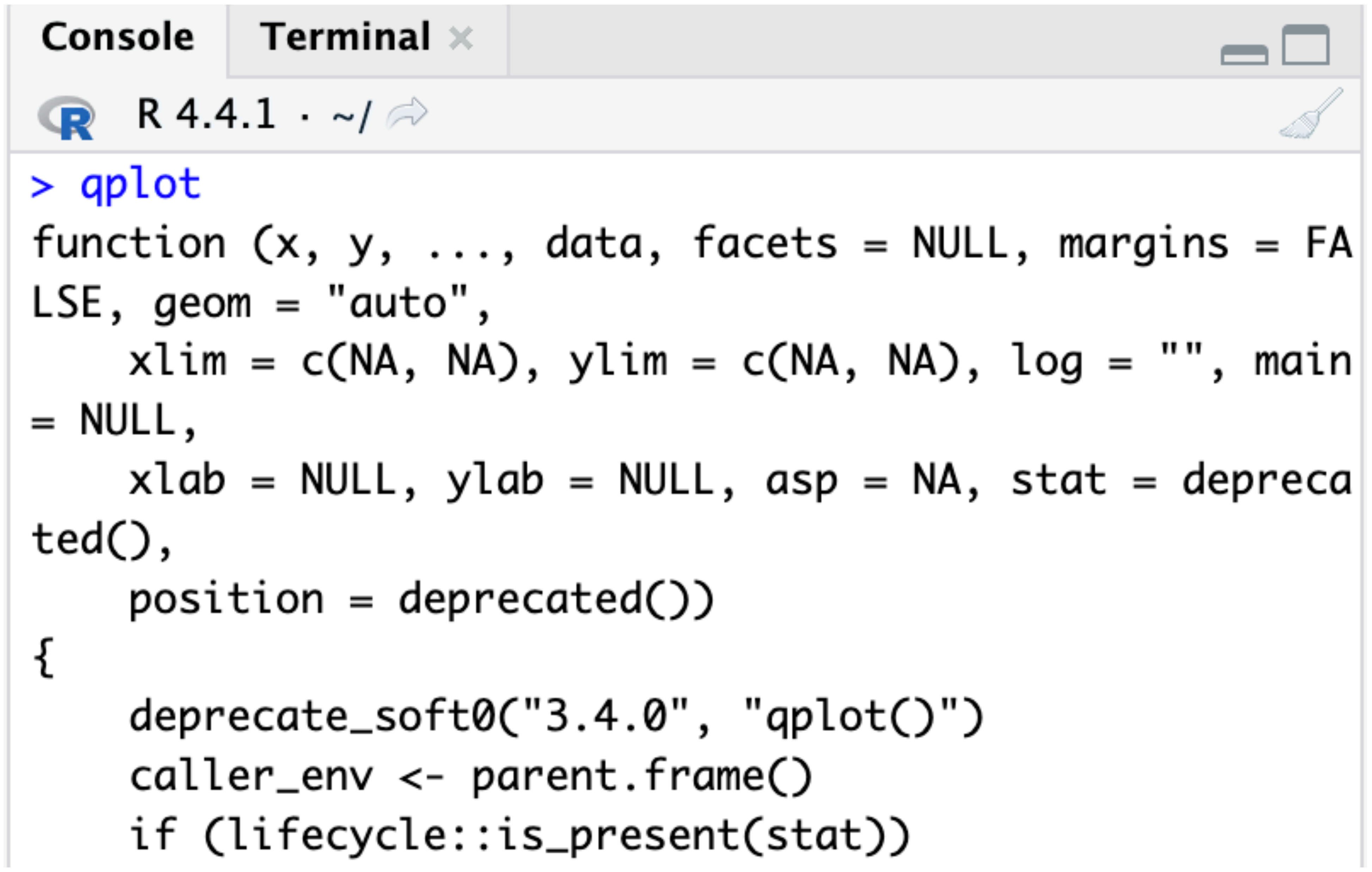
- Now load the ggplot2 package.
- If you installed the package with `install.packages` as instructed, everything should go fine.
- Don't worry if you don't see any results or messages. No news is fine news when loading a package.



```
> library("ggplot2")
>
```

Packages
Library

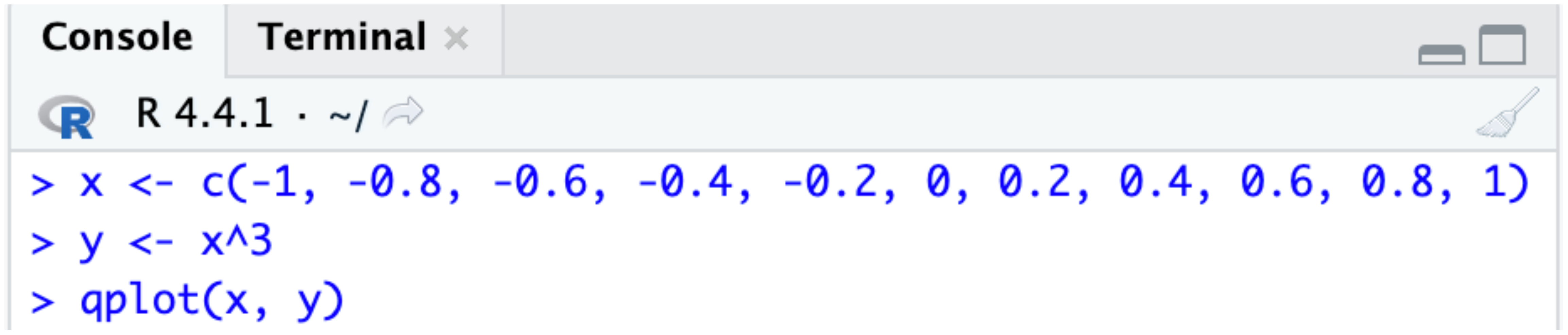
- Now if you ask to see `qplot`, R will show you quite a bit of code.
- The main thing to remember is that you only need to install a package once, but you need to load it with `library` each time you wish to use it in a new R session.
- R will unload all of its packages each time you close RStudio.



```
> qplot
function (x, y, ..., data, facets = NULL, margins = FA
LSE, geom = "auto",
    xlim = c(NA, NA), ylim = c(NA, NA), log = "", main
= NULL,
    xlab = NULL, ylab = NULL, asp = NA, stat = depreca
ted(),
    position = deprecated())
{
    deprecate_soft0("3.4.0", "qplot()")
    caller_env <- parent.frame()
    if (lifecycle::is_present(stat))
```

Packages
qplot

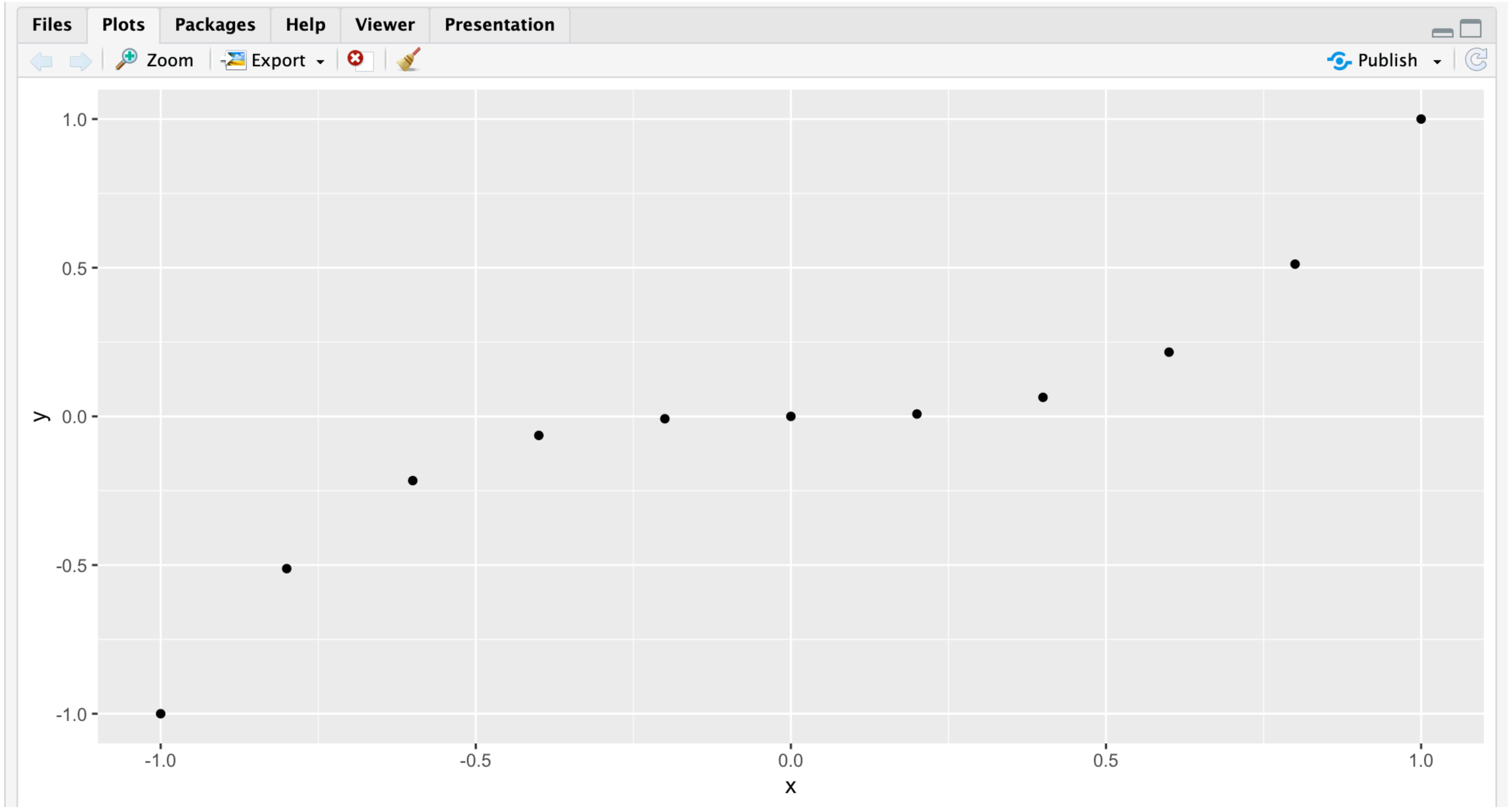
- Now that you've loaded `qplot`, let's take it for a spin. `qplot` makes "quick plots."
- If you give `qplot` two vectors of equal lengths, `qplot` will draw a scatterplot for you. `qplot` will use the first vector as a set of x values and the second vector as a set of y values.



```
> x <- c(-1, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1)
> y <- x^3
> qplot(x, y)
```

Packages

qplot

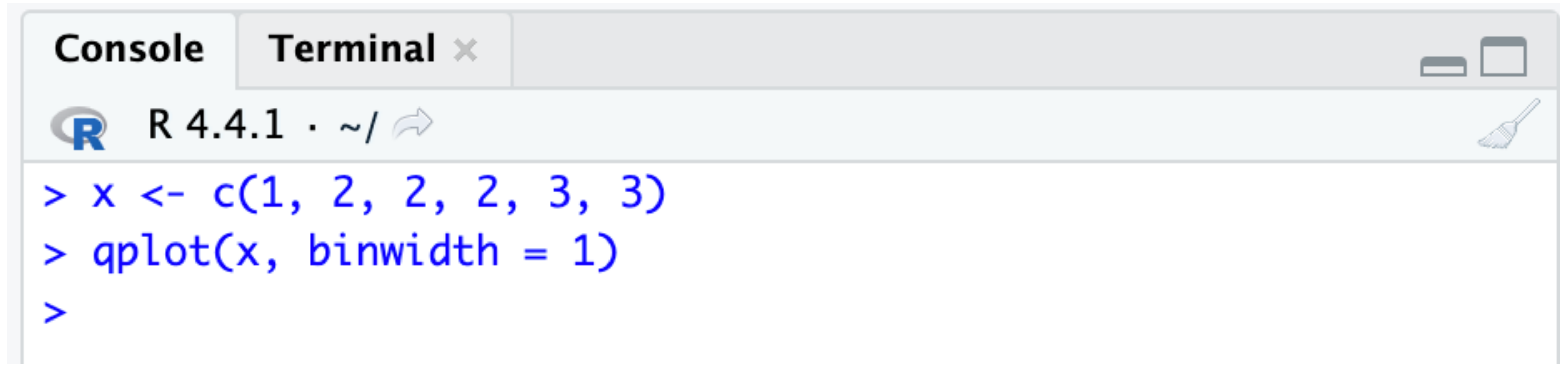


Packages
qplot

- You don’t need to name your vectors `x` and `y`. I just did that to make the example clear.
- How did R match up the values in `x` and `y` to make these points? With element-wise execution.

Packages
qplot

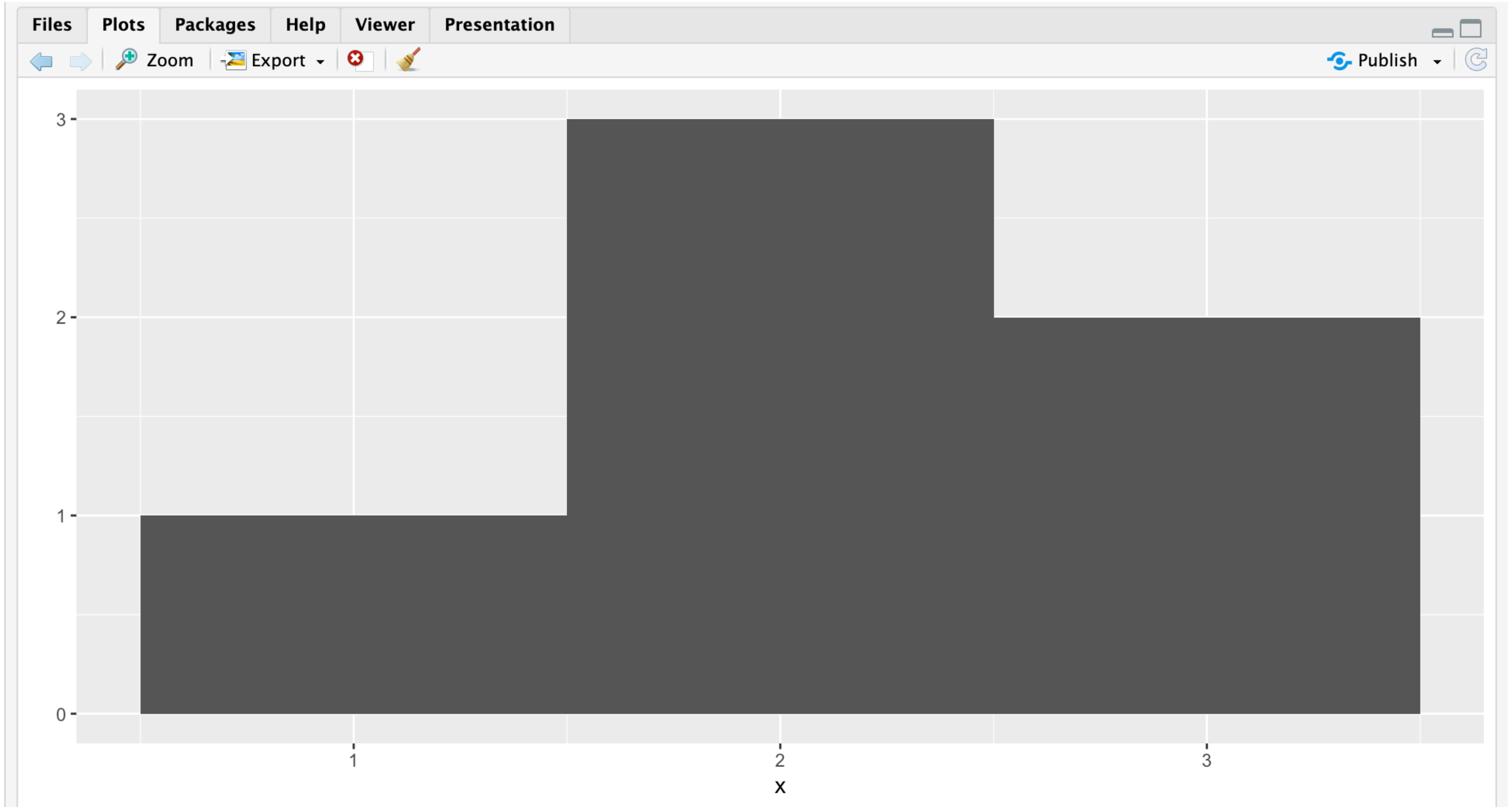
- Scatterplots are useful for visualizing the relationship between two variables. However, we're going to use a different type of graph, a *histogram*.
- A histogram visualizes the distribution of a single variable; it displays how many data points appear at each value of x.
- `qplot` will make a histogram whenever you give it only *one* vector to plot.



```
> x <- c(1, 2, 2, 2, 3, 3)
> qplot(x, binwidth = 1)
>
```

Packages

Plot from the previous R code



Packages

Introduction

- This plot shows that our vector contains one value in the interval `[1, 2)` by placing a bar of height 1 above that interval. Similarly, the plot shows that the vector contains three values in the interval `[2, 3)` by placing a bar of height 3 in that interval. It shows that the vector contains two values in the interval `[3, 4)` by placing a bar of height 2 in that interval. In these intervals, the hard bracket, `[`, means that the first number is included in the interval. The parenthesis, `)`, means that the last number is *not* included.