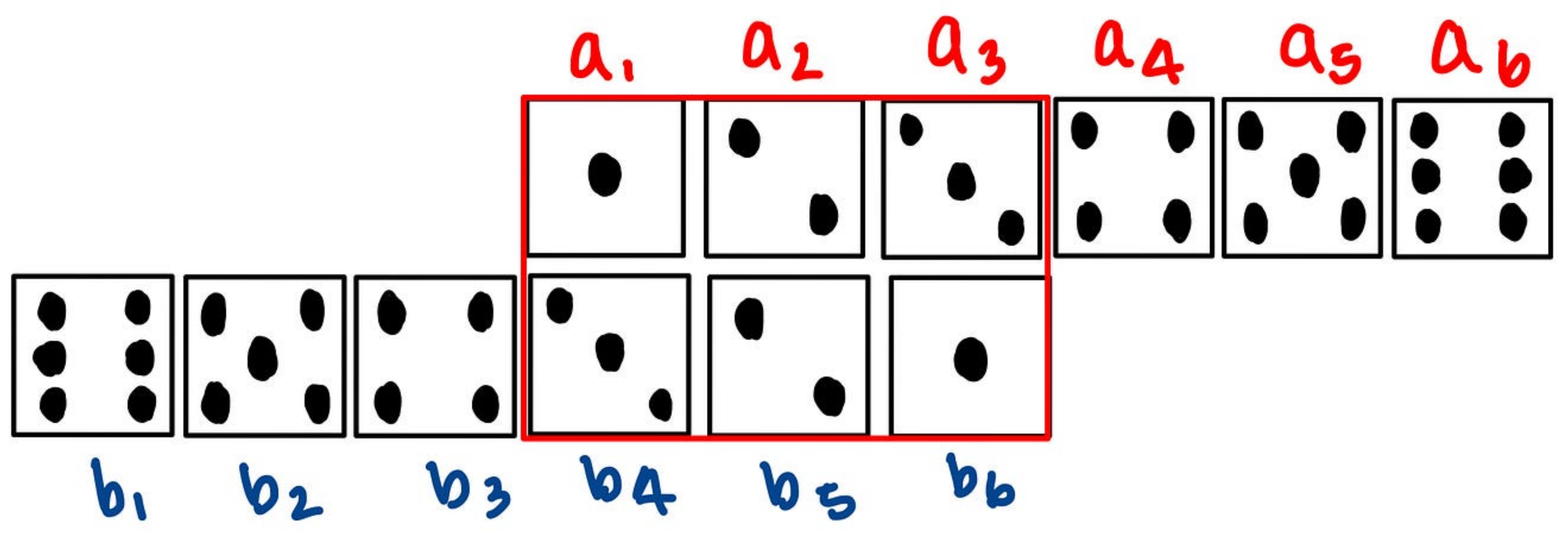
Packages

Check accuracy of your dice

- How can you use a histogram to check the accuracy of your dice?
- If you roll your dice many times and keep track of the results, you would expect some numbers to occur more than others. This is because there are more ways to get some numbers by adding two dice together than to get other numbers.
- See what convolution is.

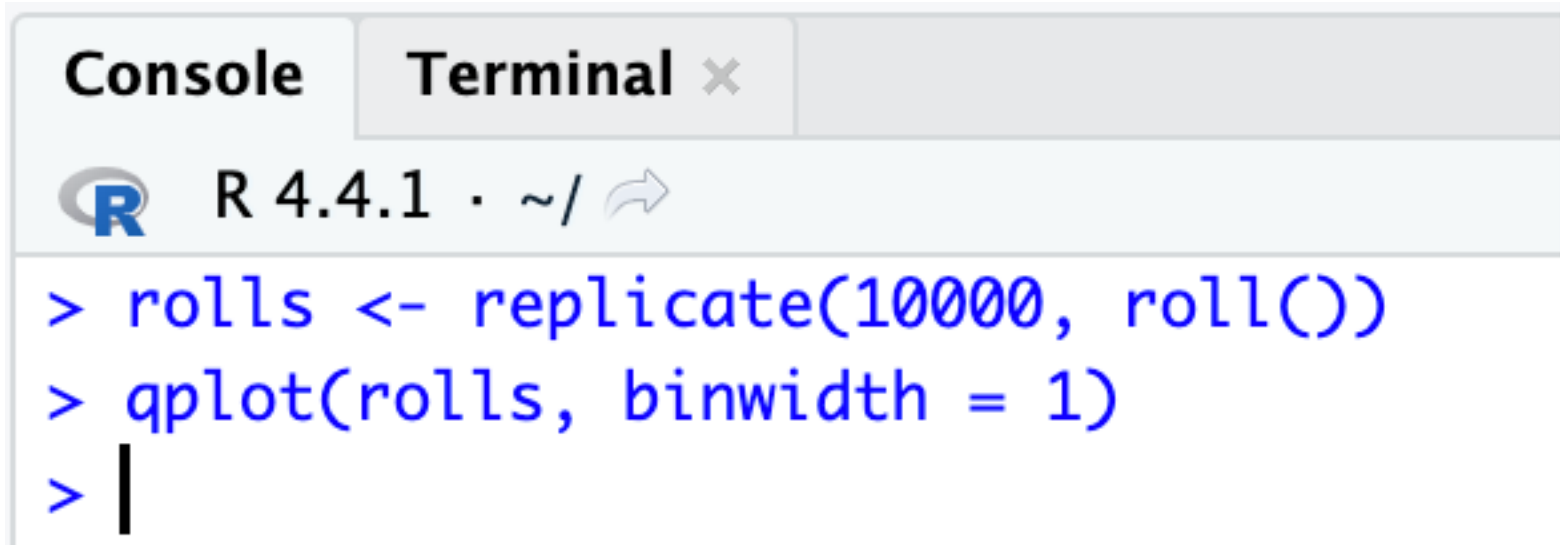
$$P(4) = (a_1 \cdot b_4) + (a_2 \cdot b_5) + (a_3 \cdot b_6)$$



Packages

Check accuracy of your dice

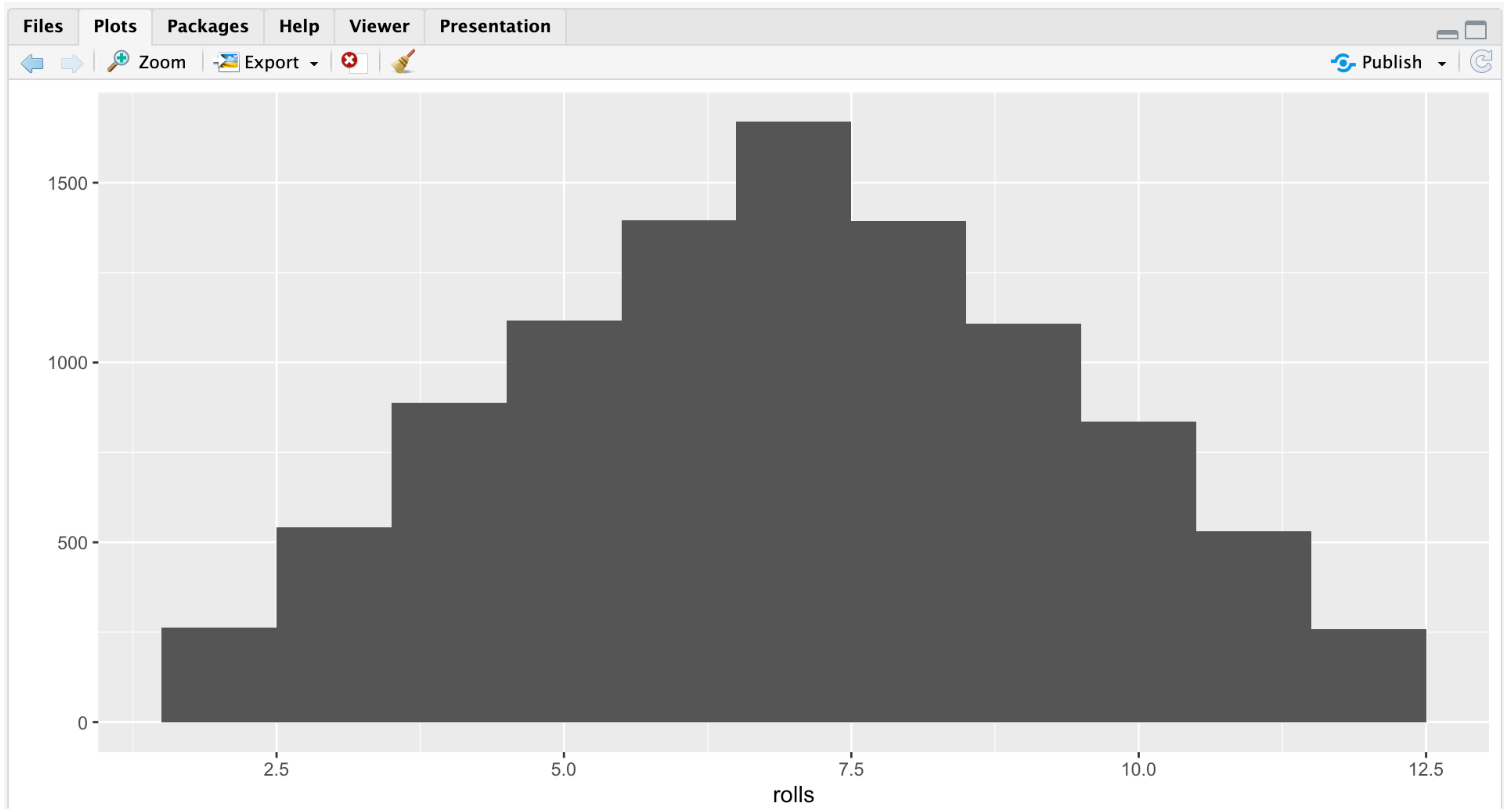
- If you roll your dice many times and plot the results with `qplot`, the histogram will show you how often each sum appeared. The sums that occurred most often will have the highest bars.
- This is where `replicate` comes in. `replicate` provides an easy way to repeat an R command many times.
- To use it, first give `replicate` the number of times you wish to repeat an R command, and then give it the command you wish to repeat.



```
> rolls <- replicate(10000, roll())
> qplot(rolls, binwidth = 1)
> 
```

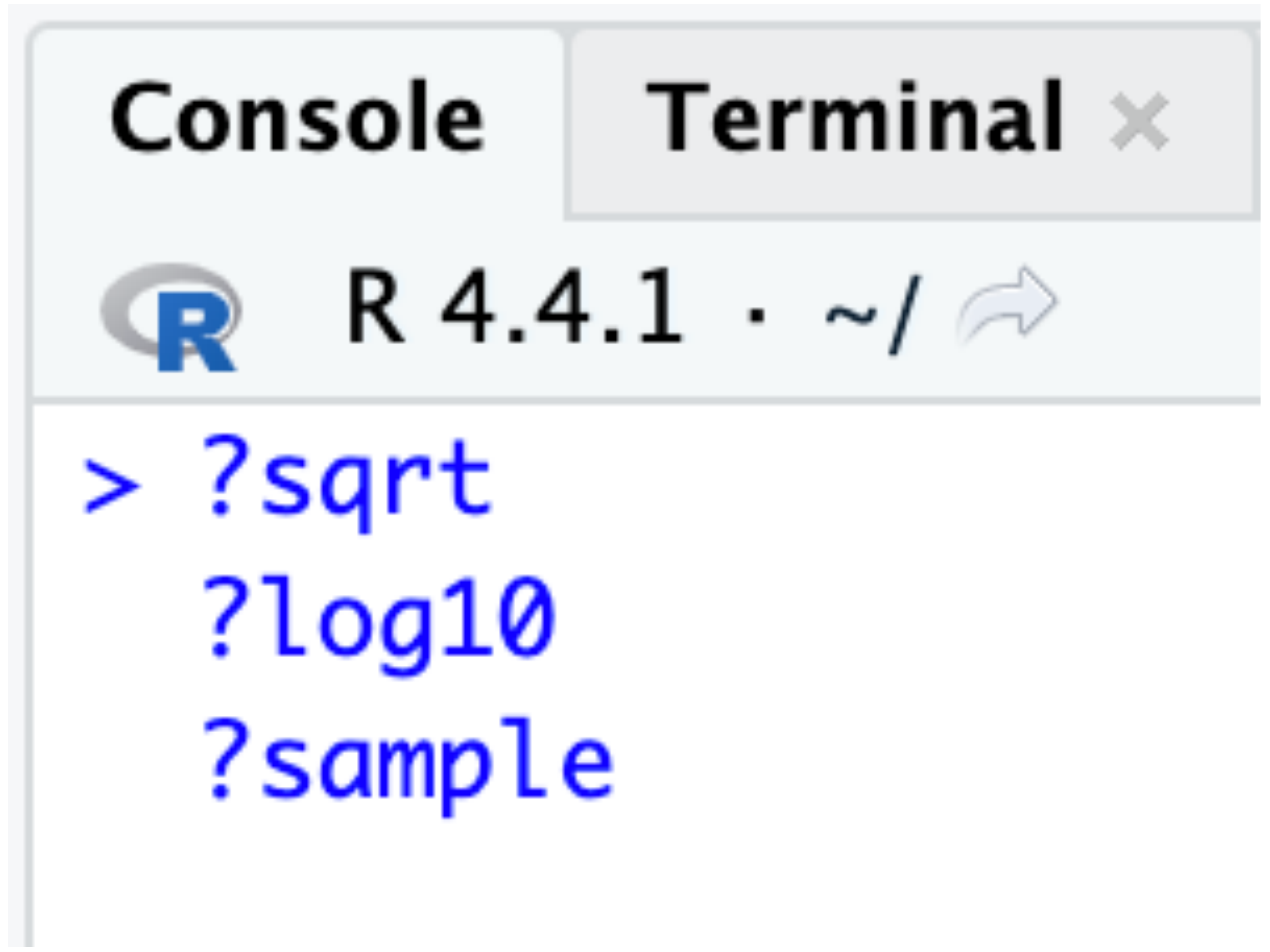
Packages

Plot from the previous R code



Getting help
Help pages

- There are over 1,000 functions at the core of R.
- This can be a lot of material to memorize and learn!
- Luckily, each R function comes with its own help page, which you can access by typing the function's name after a question mark.
- Help pages contain useful information about what each function does.



```
> ?sqrt
  ?log10
  ?sample
```

Getting help

Parts of a Help page

- **Description** - A short summary of what the function does.
- **Usage** - An example of how you would type the function.
- **Arguments** - A list of each argument the function takes, what type of information R expects you to supply for the argument, and what the function will do with the information.
- **Details** - A more in-depth description of the function and how it operates.
- **Value** - A description of what the function returns when you run it.
- **See Also** - A short list of related R functions

Playing with cards

Dealing with data

What you’ll learn

- In this section, you’ll design a deck of playing cards that you can shuffle and deal from. Best of all, the deck will remember which cards have been dealt–just like a real deck.
- You can use the deck to play card games, tell fortunes, and test card-counting strategies.

Along the way, you will learn how to:

- Save new types of data, like character strings and logical values.
- Save a data set as a vector, matrix, array, list, or data frame.
- Load and save your own data sets with R
- Extract individual values from a data set
- Change individual values within a data set
- Write logical tests

R Objects

Create a deck of 52 cards

- You’ll start by building simple R objects that represent playing cards and then work your way up to a full-blown table of data.
- In short, you’ll build the equivalent of an Excel spreadsheet from scratch.
- When you are finished, your deck of cards will look something like this

```
  face   suit value
  king spades    13
 queen spades    12
  jack spades    11
   ten spades    10
  nine spades     9
 eight spades     8
...
```

R Objects

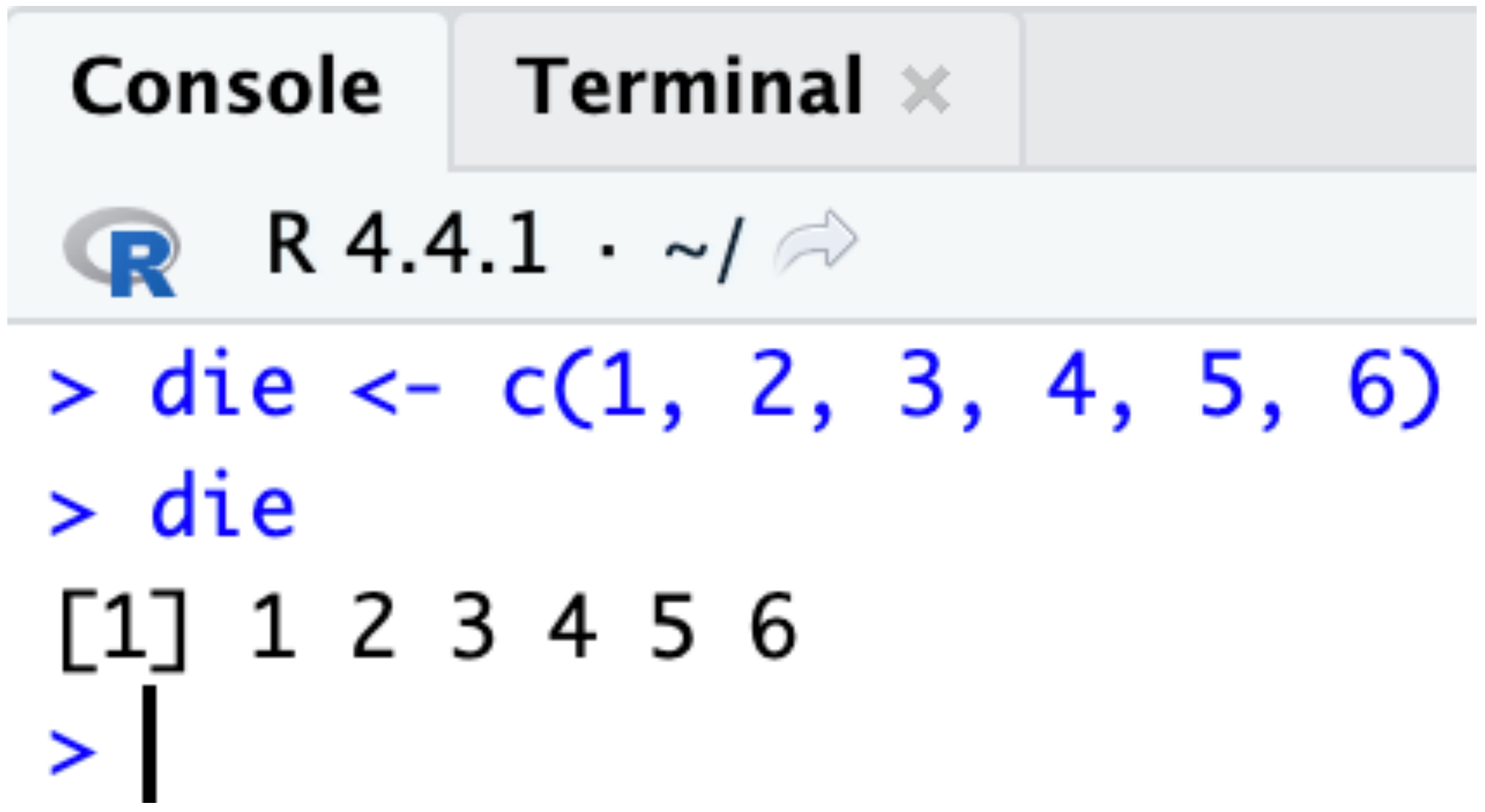
Create a deck of 52 cards

- Do you need to build a data set from scratch to use it in R?
- Not at all.
- You can load most data sets into R with one simple step (we'll see it later).
- This exercise will teach you how R stores data, and how you can assemble your own data sets.

```
 face    suit value
 king  spades    13
queen  spades    12
 jack  spades    11
  ten  spades    10
 nine  spades     9
eight  spades     8
...
```

R Objects
Atomic vectors

- An atomic vector is just a simple vector of data.
- The `die` object from the previous section was an atomic vector.
- You can make an atomic vector by grouping some values of data together with `c`

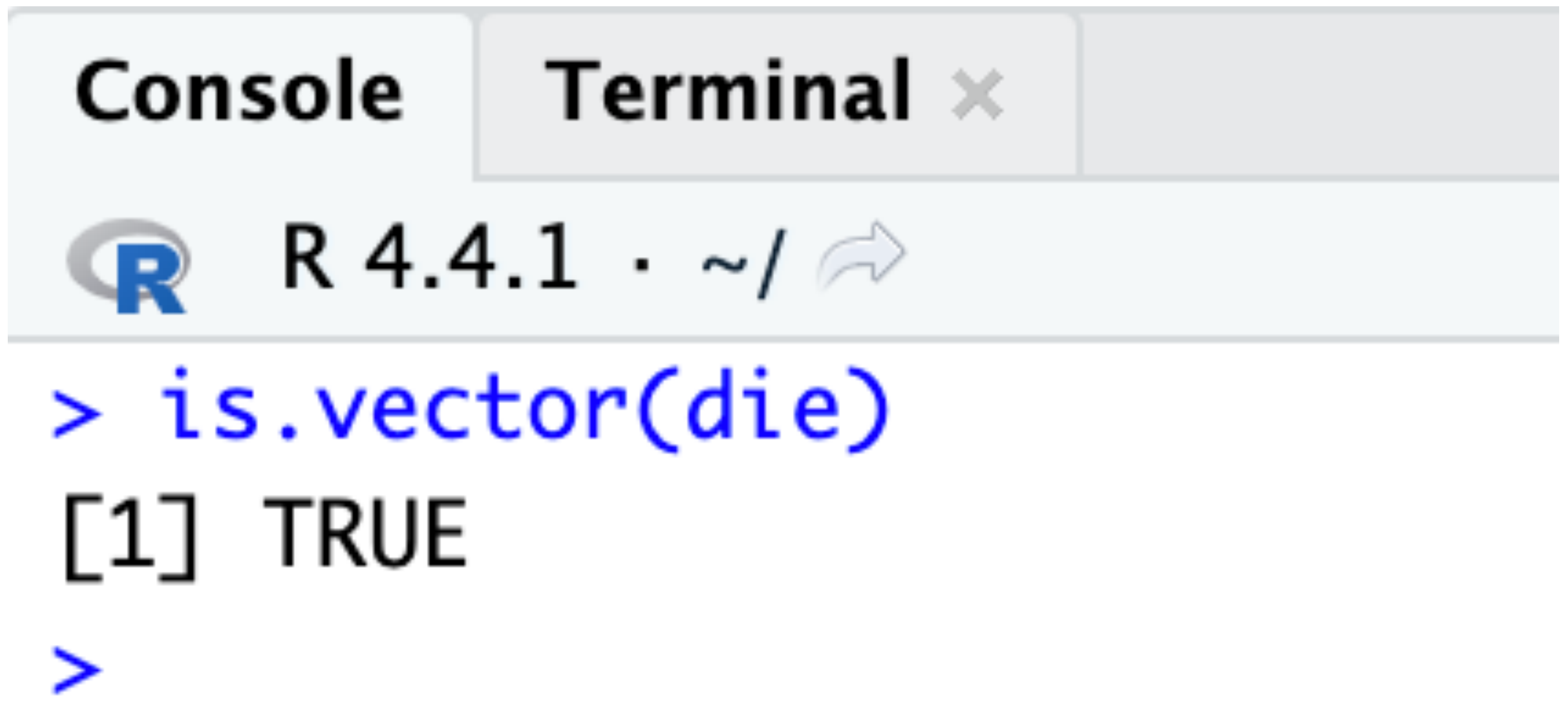


```
> die <- c(1, 2, 3, 4, 5, 6)
> die
[1] 1 2 3 4 5 6
> 
```

R Objects

Atomic vectors

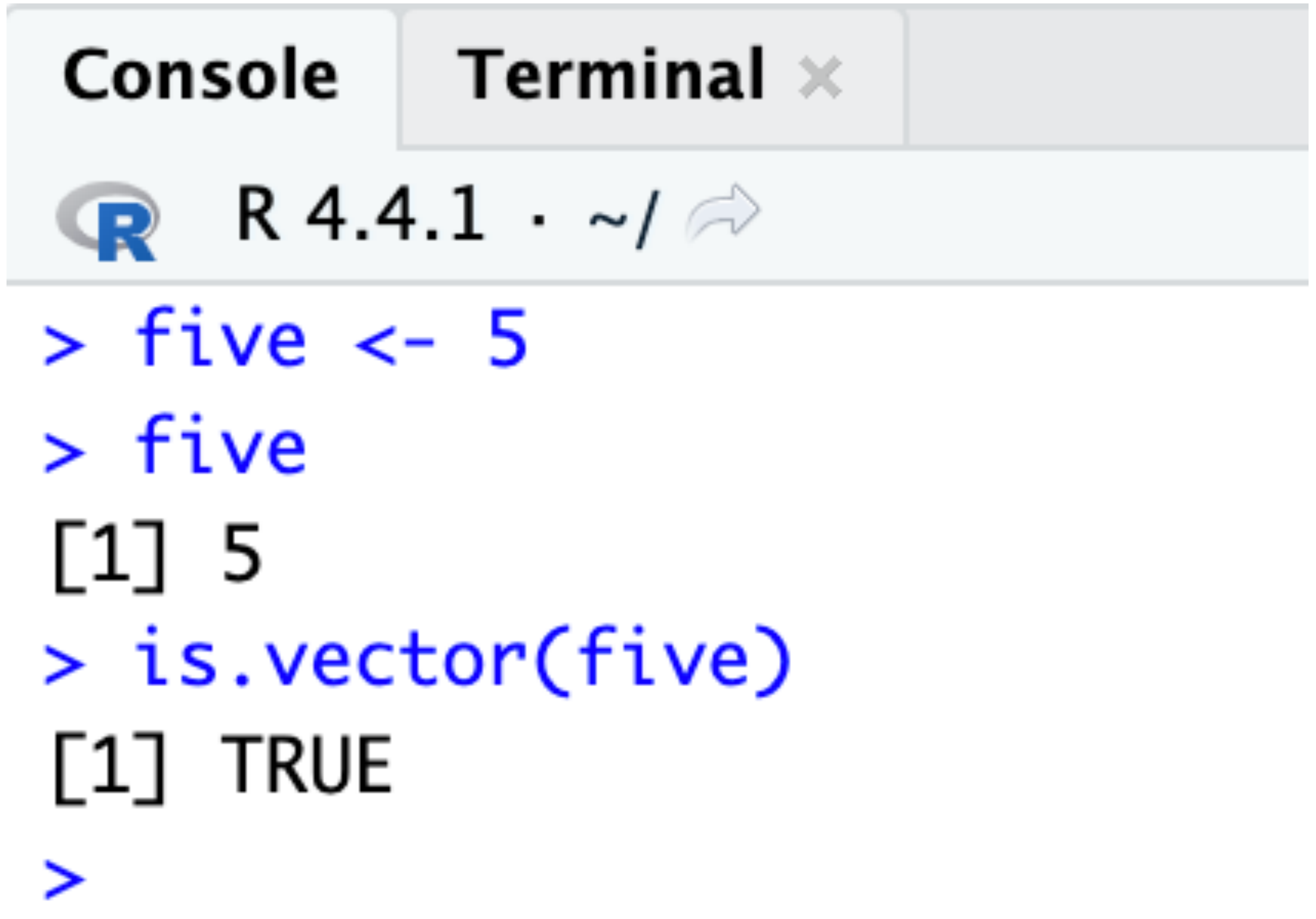
- You can test whether an object is an atomic vector or not using `is.vector`
- It returns TRUE if the object is an atomic vector and FALSE otherwise.



```
> is.vector(die)
[1] TRUE
>
```

R Objects
Atomic vectors

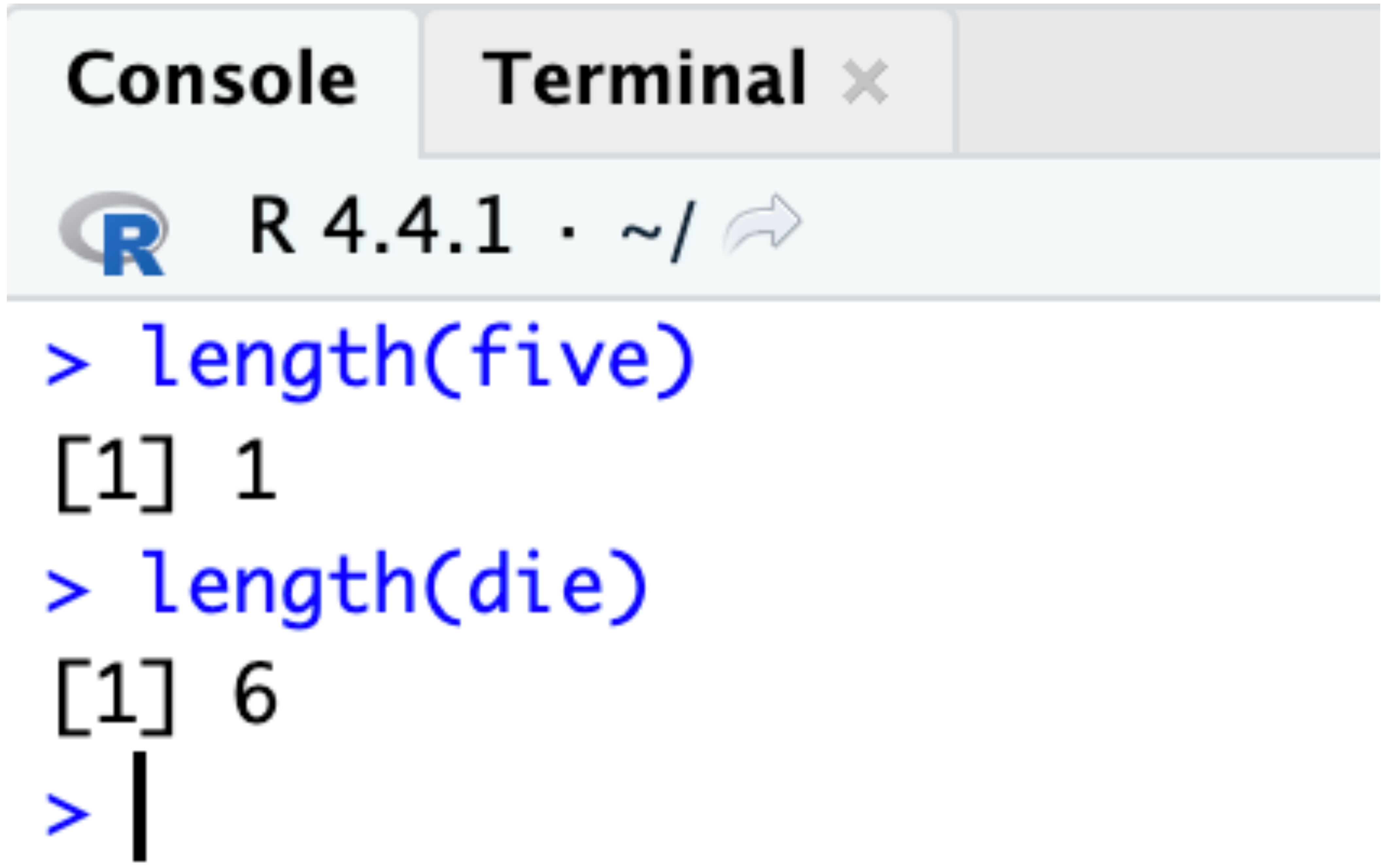
- You can also make an atomic vector with just one value.
- R saves single values as an atomic vector of length 1.



```
> five <- 5
> five
[1] 5
> is.vector(five)
[1] TRUE
>
```

R Objects
Atomic vectors

- You can check the length of an atomic vector with `length` .

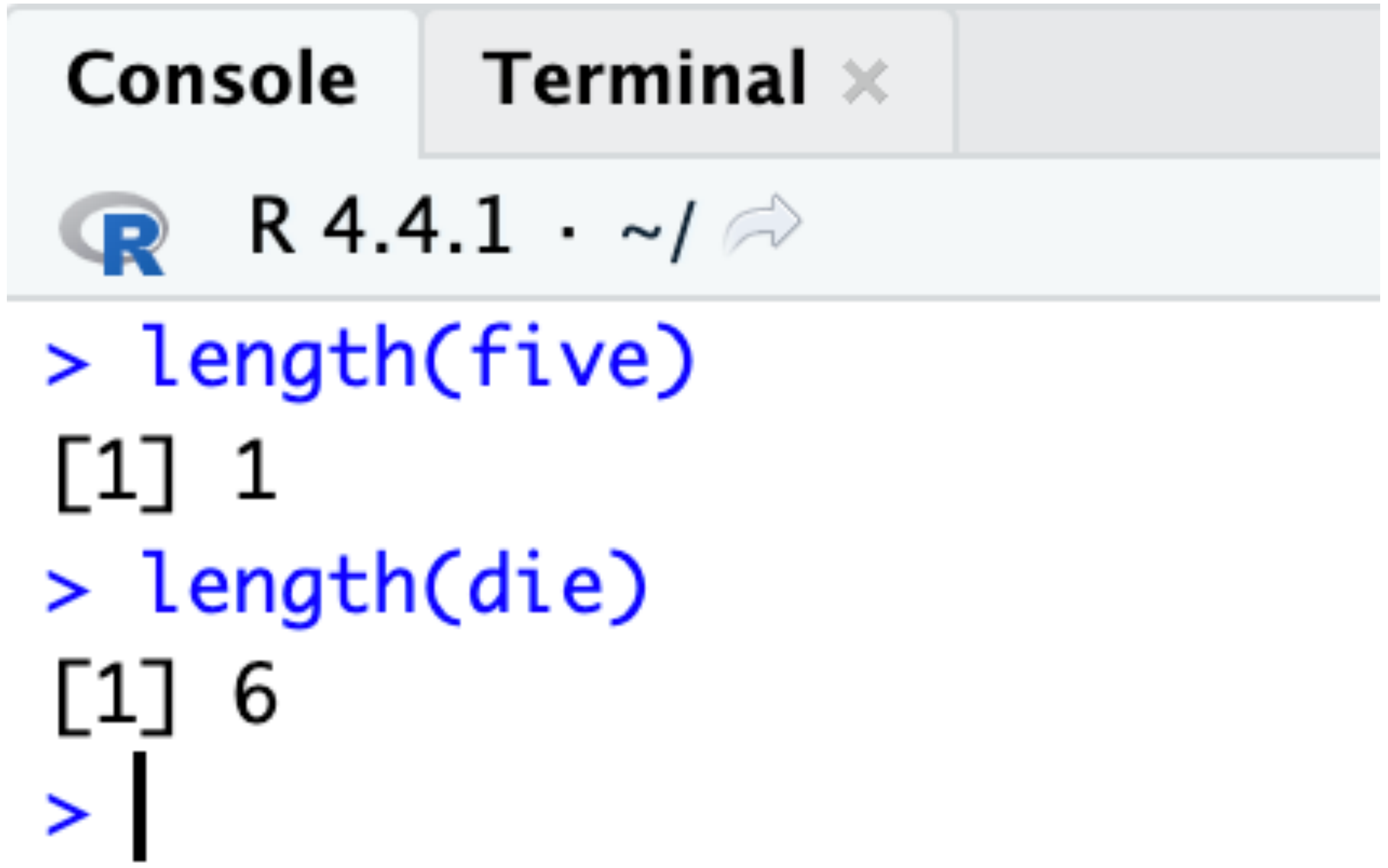


```
> length(five)
[1] 1
> length(die)
[1] 6
>
```

R Objects

Atomic vectors

- Each atomic vector stores its values as a one-dimensional vector, and each atomic vector can only store one type of data. You can save different types of data in R by using different types of atomic vectors. Altogether, R recognizes six basic types of atomic vectors: *doubles*, *integers*, *characters*, *logicals*, *complex*, and *raw*.

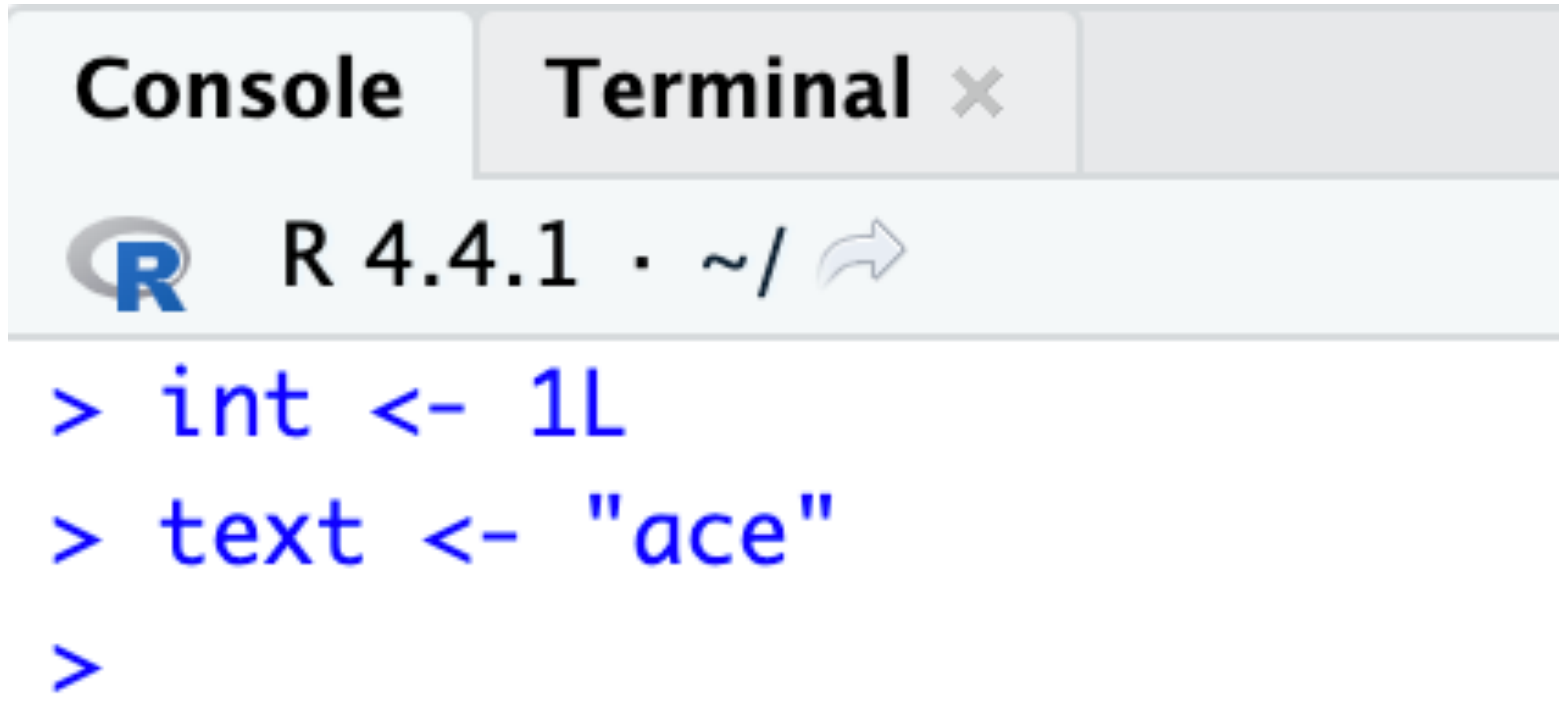


```
> length(five)
[1] 1
> length(die)
[1] 6
>
```

R Objects

Atomic vectors

- To create your card deck, you will need to use different types of atomic vectors to save different types of information.
- You can do this by using some simple conventions when you enter your data.
- For example, you can create an integer vector by including a capital L with your input. You can create a character vector by surrounding your input in quotation marks

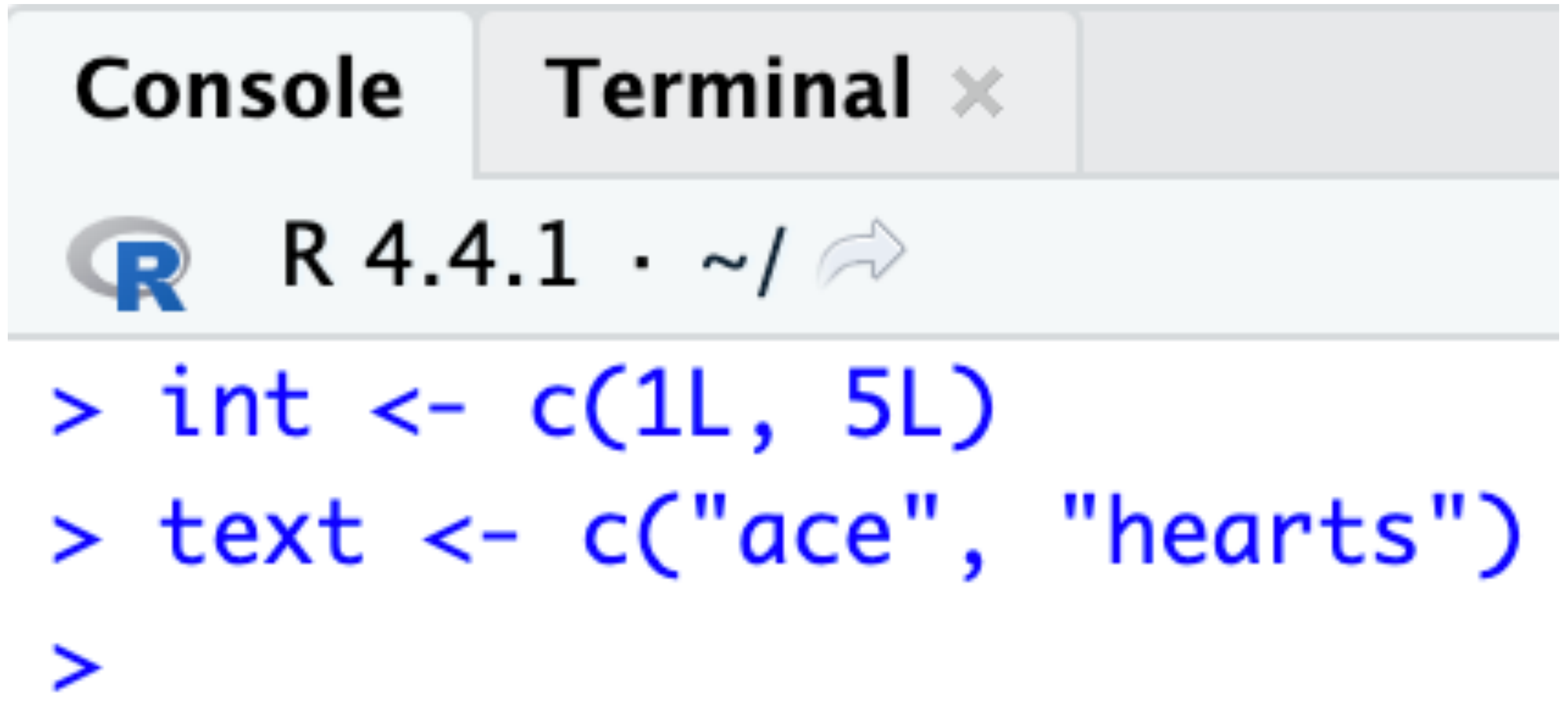


```
> int <- 1L
> text <- "ace"
>
```

R Objects

Atomic vectors

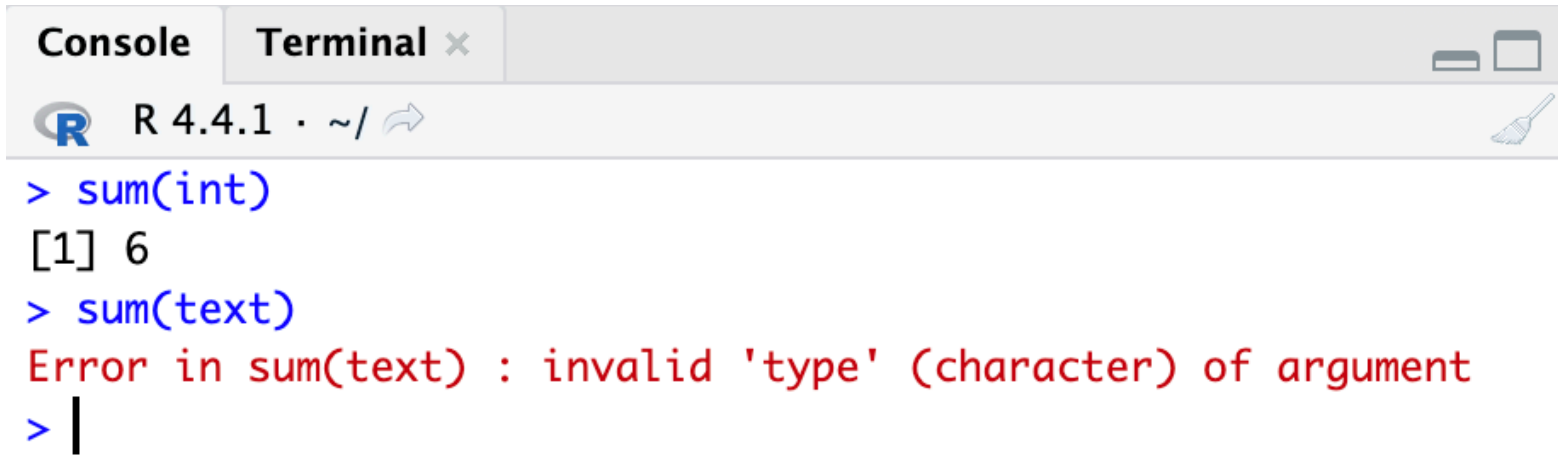
- Each type of atomic vector has its own convention.
- R will recognize the convention and use it to create an atomic vector of the appropriate type.
- If you'd like to make atomic vectors that have more than one element in them, you can combine an element with the `c` function.



```
> int <- c(1L, 5L)
> text <- c("ace", "hearts")
>
```

R Objects
Atomic vectors

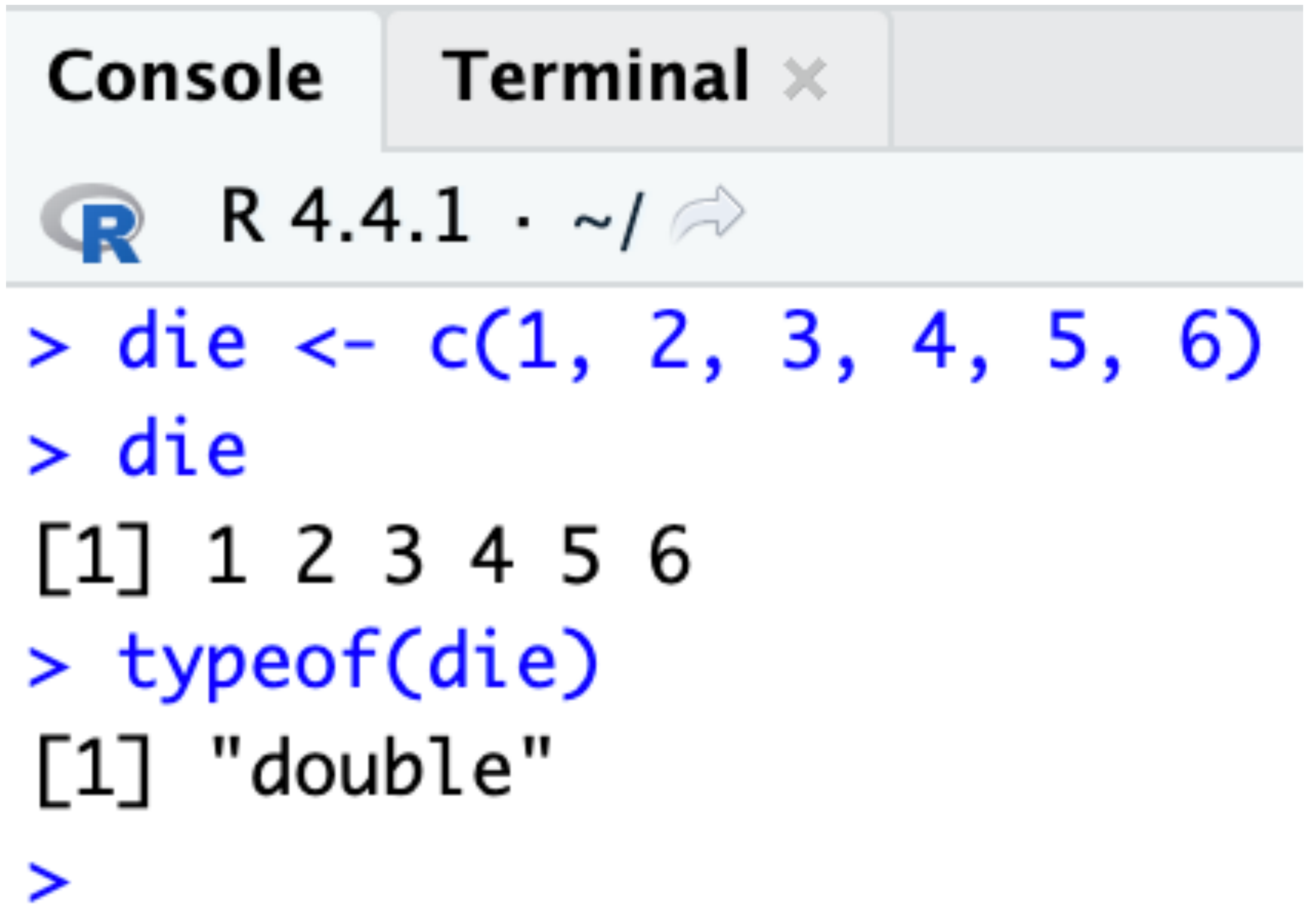
- You may wonder why R uses multiple types of vectors.
- Vector types help R behave as you would expect.
- For example, R will do math with atomic vectors that contain numbers, but not with atomic vectors that contain character strings.



```
> sum(int)
[1] 6
> sum(text)
Error in sum(text) : invalid 'type' (character) of argument
> |
```

R Objects
Doubles

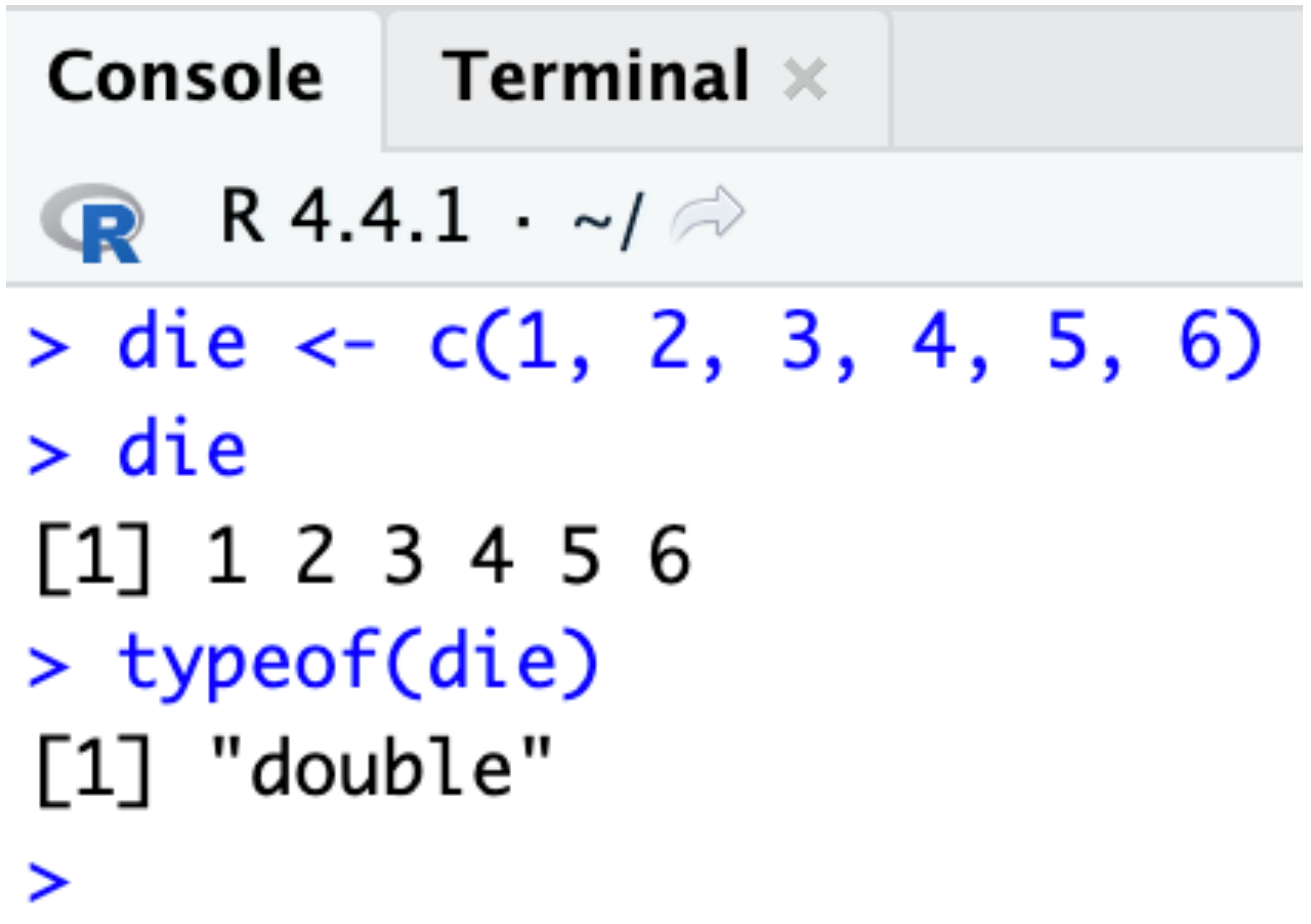
- A double vector stores regular numbers.
- The numbers can be positive or negative, large or small, and have digits to the right of the decimal place or not.
- In general, R will save any number that you type in R as a double.
- For example, the die you made previously was a double object.



```
> die <- c(1, 2, 3, 4, 5, 6)
> die
[1] 1 2 3 4 5 6
> typeof(die)
[1] "double"
>
```

R Objects
Doubles

- Some R functions refer to doubles as “numerics,” and I will often do the same.
- Double is a computer science term.
- It refers to the specific number of bytes your computer uses to store a number, but I find “numeric” to be much more intuitive when doing data science.

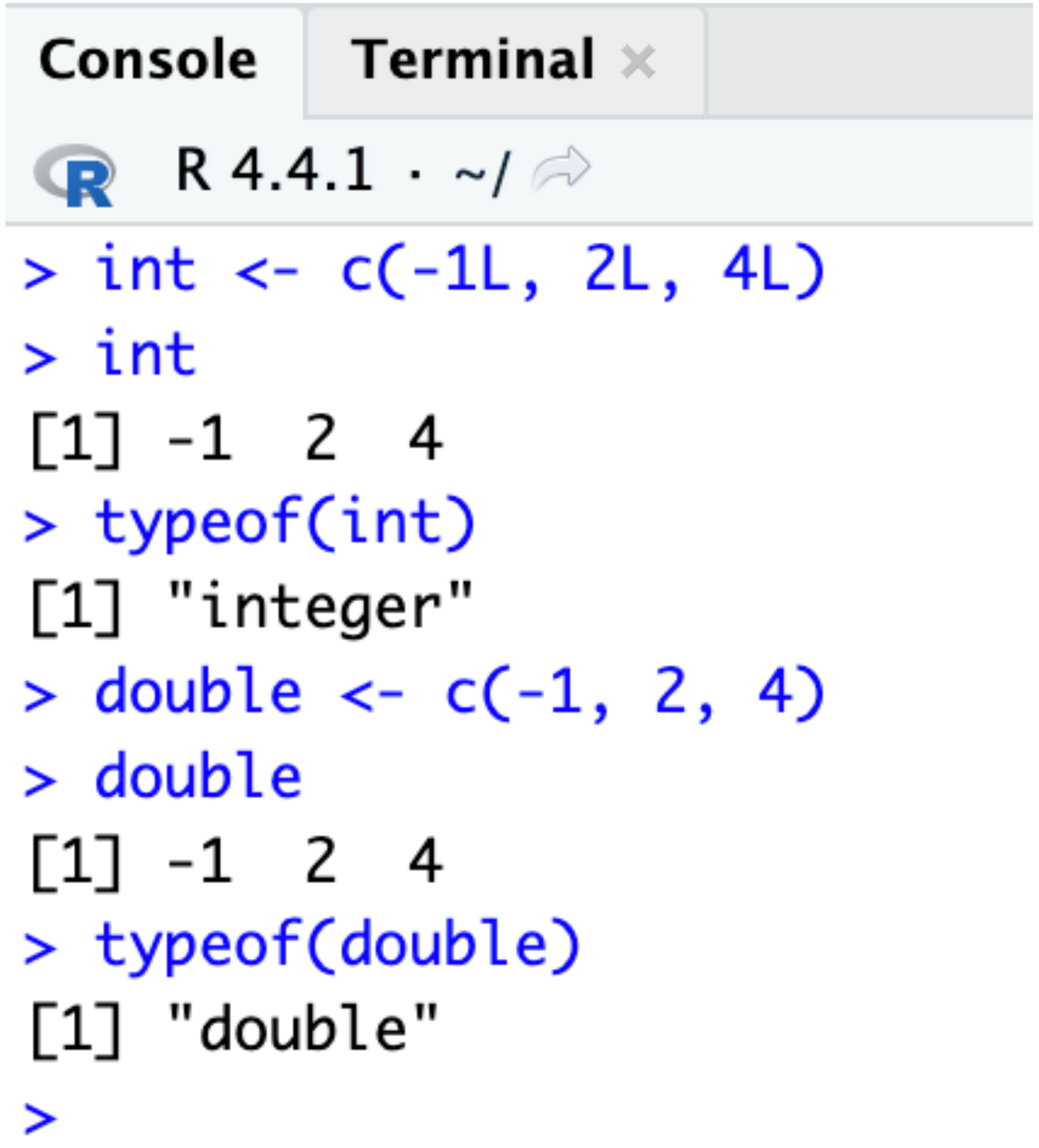


```
> die <- c(1, 2, 3, 4, 5, 6)
> die
[1] 1 2 3 4 5 6
> typeof(die)
[1] "double"
>
```

R Objects

Integers

- Integer vectors store integers, numbers that can be written without a decimal component.
- As a data scientist, you won't use the integer type very often because you can save integers as a double object.
- You can specifically create an integer in R by typing a number followed by an uppercase `L`.

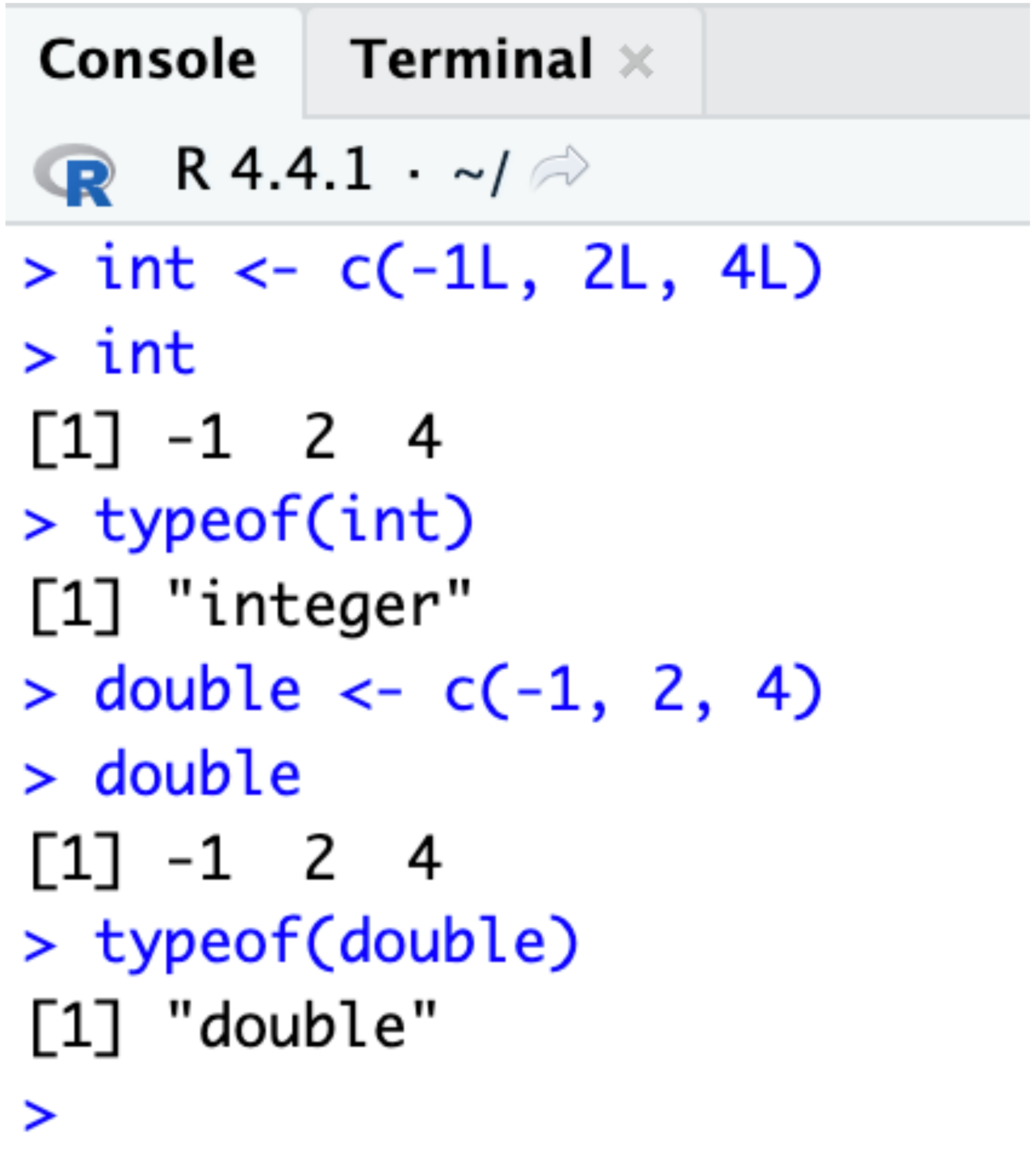


```
> int <- c(-1L, 2L, 4L)
> int
[1] -1  2  4
> typeof(int)
[1] "integer"
> double <- c(-1, 2, 4)
> double
[1] -1  2  4
> typeof(double)
[1] "double"
>
```

R Objects

Integers

- Note that R won't save a number as an integer unless you include the L.
- Integer numbers without the L will be saved as doubles.
- The only difference between 4 and 4L is how R saves the number in your computer's memory.
- Integers are defined more precisely in your computer's memory than doubles.



```
> int <- c(-1L, 2L, 4L)
> int
[1] -1  2  4
> typeof(int)
[1] "integer"
> double <- c(-1, 2, 4)
> double
[1] -1  2  4
> typeof(double)
[1] "double"
>
```

R Objects

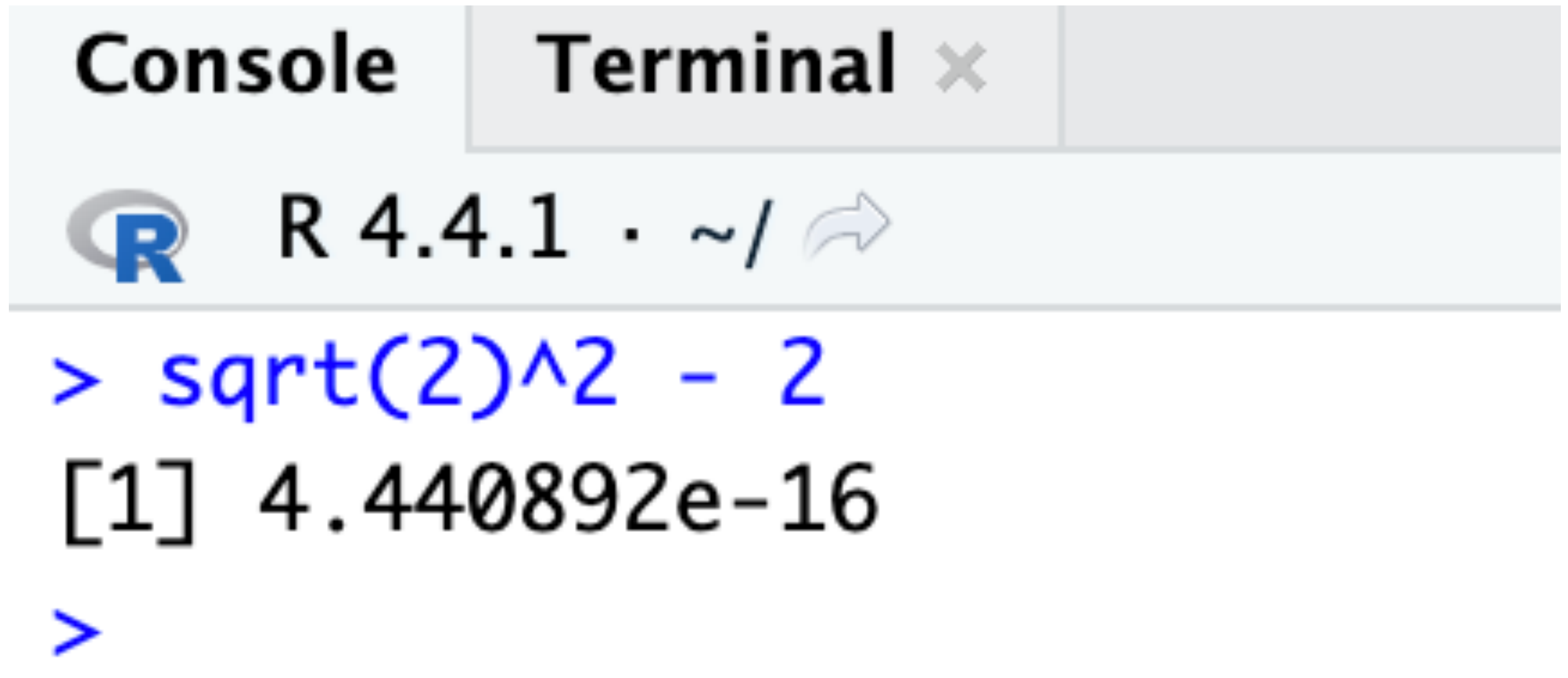
Integers vs Doubles

- Why would you save your data as an integer instead of a double?
- Sometimes a difference in precision can have surprising effects.
- Your computer allocates 64 bits of memory to store each double in an R program. This allows a lot of precision, but some numbers cannot be expressed exactly in 64 bits. For example, the number π contains an endless sequences of digits to the right of the decimal place. Your computer must round π to something close to, but not exactly equal to π to store π in its memory. Many decimal numbers share a similar fate.
- As a result, each double is accurate to about 16 significant digits. This introduces a little bit of error. In most cases, this rounding error will go unnoticed. However, in some situations, the rounding error can cause surprising results.

R Objects

A problem

- For example, you may expect the result of the expression on the right to be zero, but it is not.
- The square root of two cannot be expressed exactly in 16 significant digits.
- As a result, R has to round the quantity, and the expression resolves to something very close to—but not quite—zero.

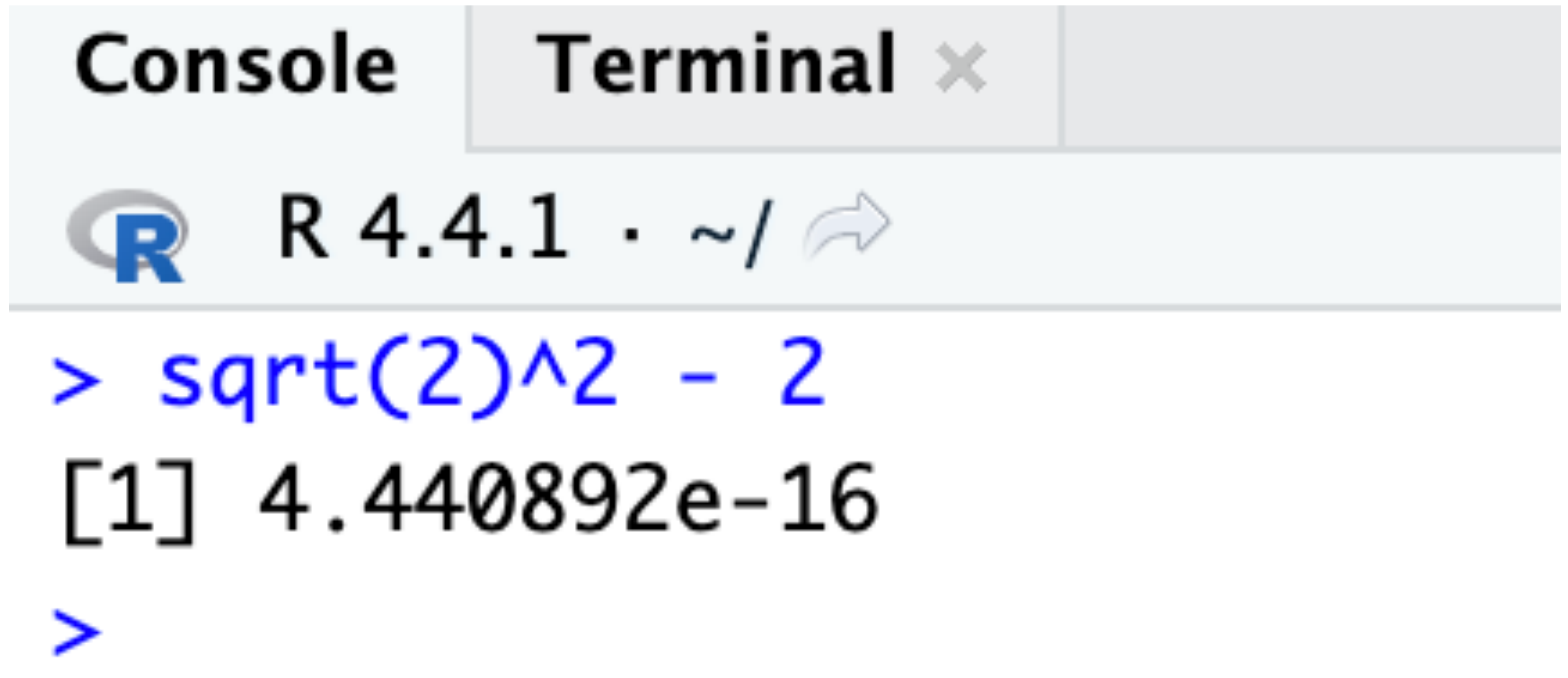


```
> sqrt(2)^2 - 2
[1] 4.440892e-16
>
```

R Objects

Floating point

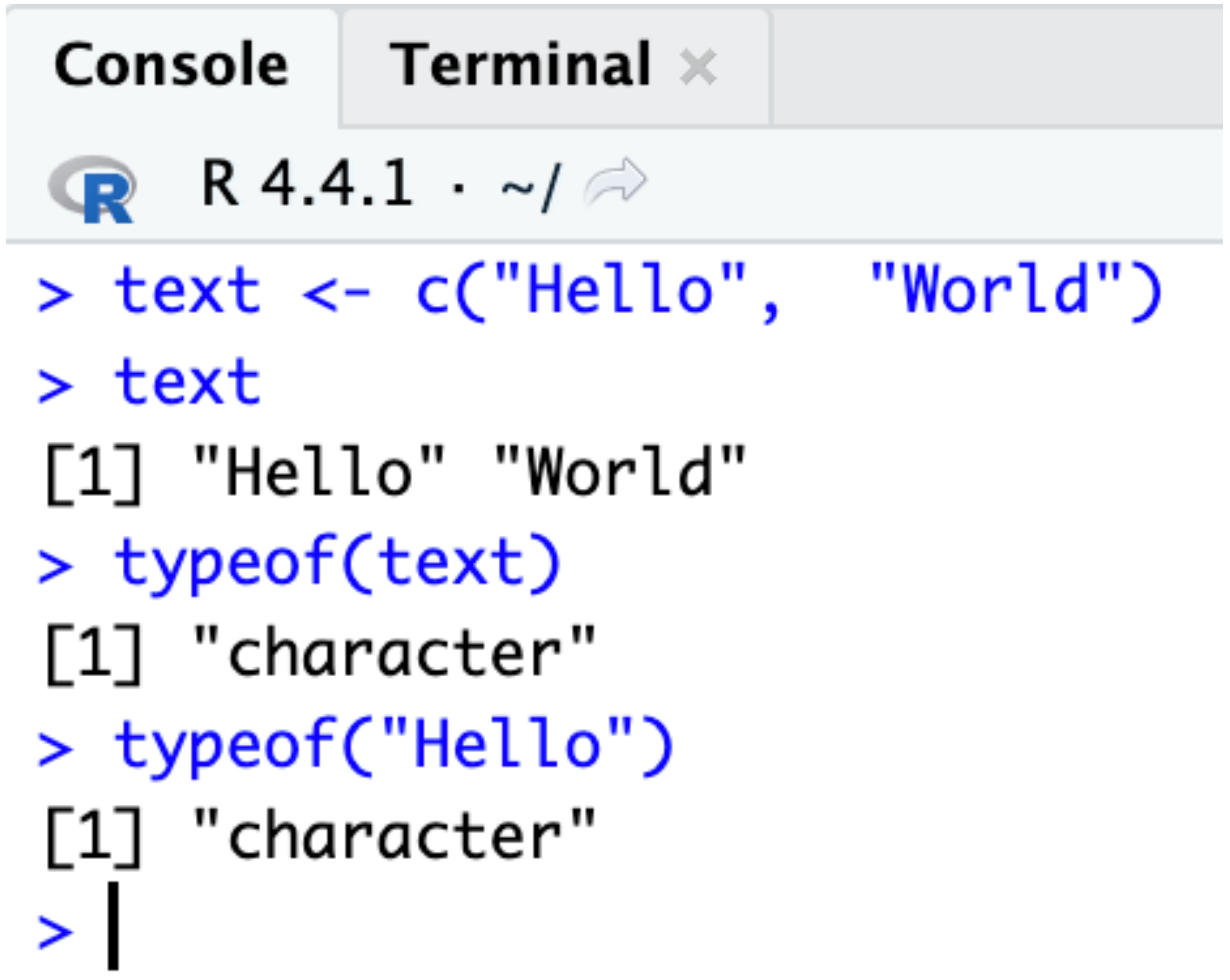
- These errors are known as *floating-point* errors, and doing arithmetic in these conditions is known as *floating-point arithmetic*.
- Floating-point arithmetic is not a feature of R; it is a feature of computer programming. Usually floating-point errors won't be enough to ruin your day.
- Just keep in mind that they may be the cause of surprising results.



```
> sqrt(2)^2 - 2
[1] 4.440892e-16
>
```

R Objects
Characters

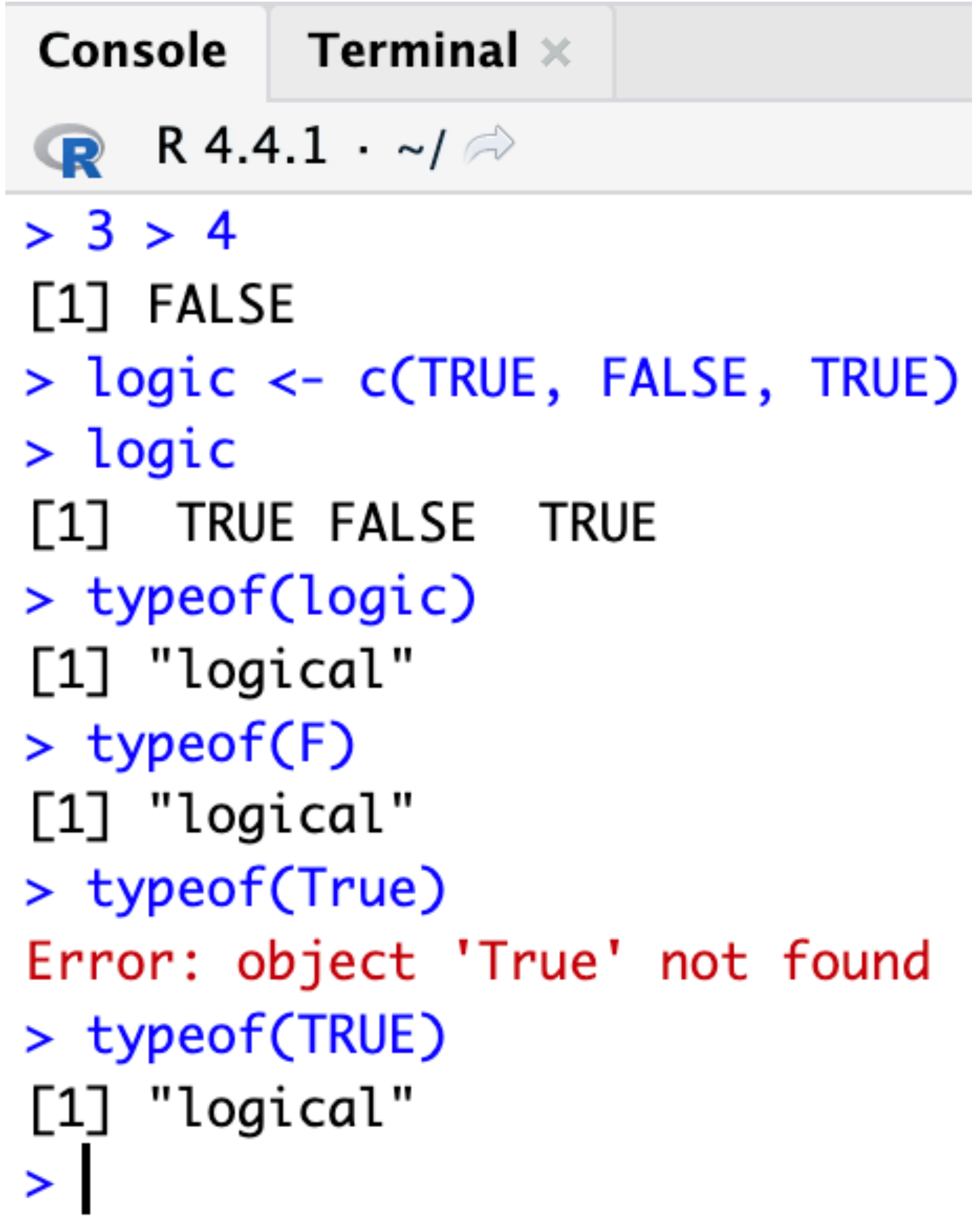
- A character vector stores small pieces of text.
- You can create a character vector in R by typing a character or string of characters surrounded by quotes.
- The individual elements of a character vector are known as *strings*.
- Note that a string can contain more than just letters. You can assemble a character string from numbers or symbols as well.



```
> text <- c("Hello",  "World")
> text
[1] "Hello" "World"
> typeof(text)
[1] "character"
> typeof("Hello")
[1] "character"
> 
```

R Objects
Logicals

- Logical vectors store TRUEs and FALSEs, R's form of Boolean data. Logicals are very helpful for doing things like comparisons.
- Any time you type TRUE or FALSE in capital letters (without quotation marks), R will treat your input as logical data.
- R also assumes that T and F are shorthand for TRUE and FALSE, unless they are defined elsewhere (e.g. `T <- 500`).

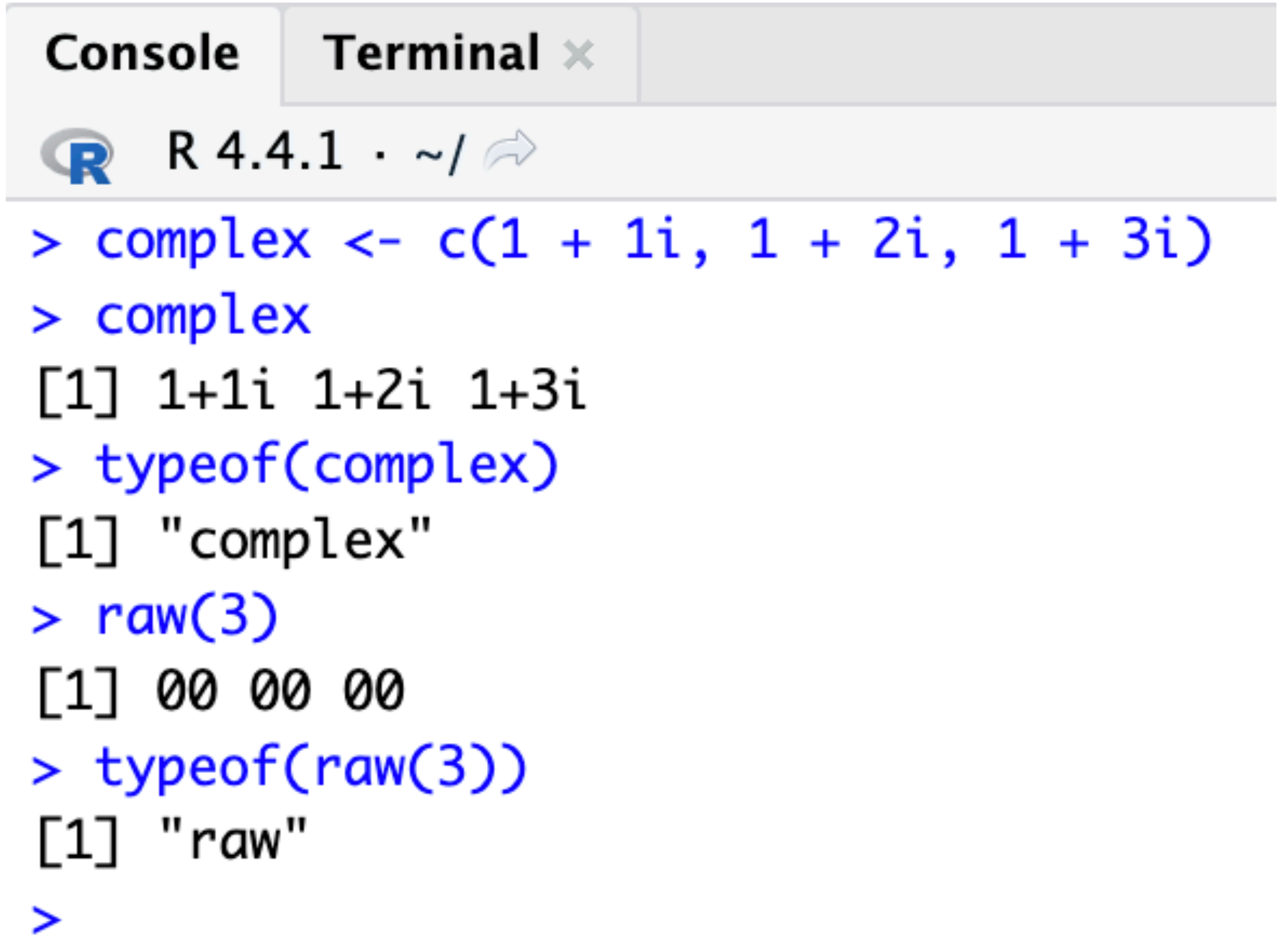


```
> 3 > 4
[1] FALSE
> logic <- c(TRUE, FALSE, TRUE)
> logic
[1]  TRUE FALSE  TRUE
> typeof(logic)
[1] "logical"
> typeof(F)
[1] "logical"
> typeof(True)
Error: object 'True' not found
> typeof(TRUE)
[1] "logical"
> 
```

R Objects

Complex and Raw

- Doubles, integers, characters, and logicals are the most common types of atomic vectors.
- R also recognizes two more types: complex and raw. It is doubtful that you will ever use these to analyze data.
- Complex vectors store complex numbers. To create a complex vector, add an imaginary term to a number with i
- Raw vectors store raw bytes of data. Making raw vectors gets complicated, but you can make an empty raw vector of length *n* with `raw(n)`



```
> complex <- c(1 + 1i, 1 + 2i, 1 + 3i)
> complex
[1] 1+1i 1+2i 1+3i
> typeof(complex)
[1] "complex"
> raw(3)
[1] 00 00 00
> typeof(raw(3))
[1] "raw"
>
```

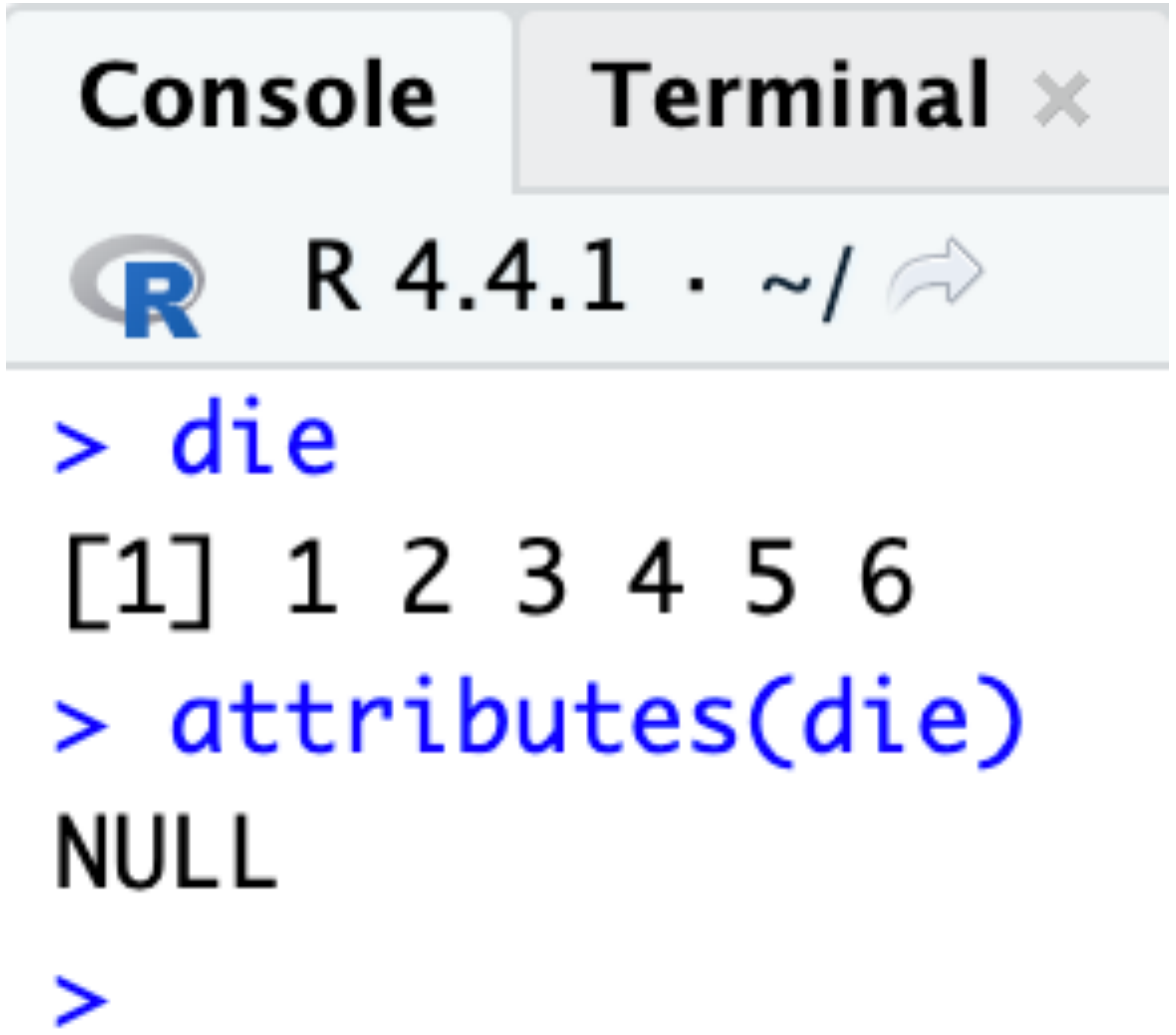
R Objects
Attributes

- An attribute is a piece of information that you can attach to an atomic vector.
- The attribute won't affect any of the values in the object, and it will not appear when you display your object.
- You can think of an attribute as "metadata"; it is just a convenient place to put information associated with an object.
- R will normally ignore this metadata, but some R functions will check for specific attributes.
- These functions may use the attributes to do special things with the data.

R Objects

Attributes

- You can see which attributes an object has with `attributes`. `attributes` will return `NULL` if an object has no attributes. An atomic vector, like `die`, won't have any attributes unless you give it some.

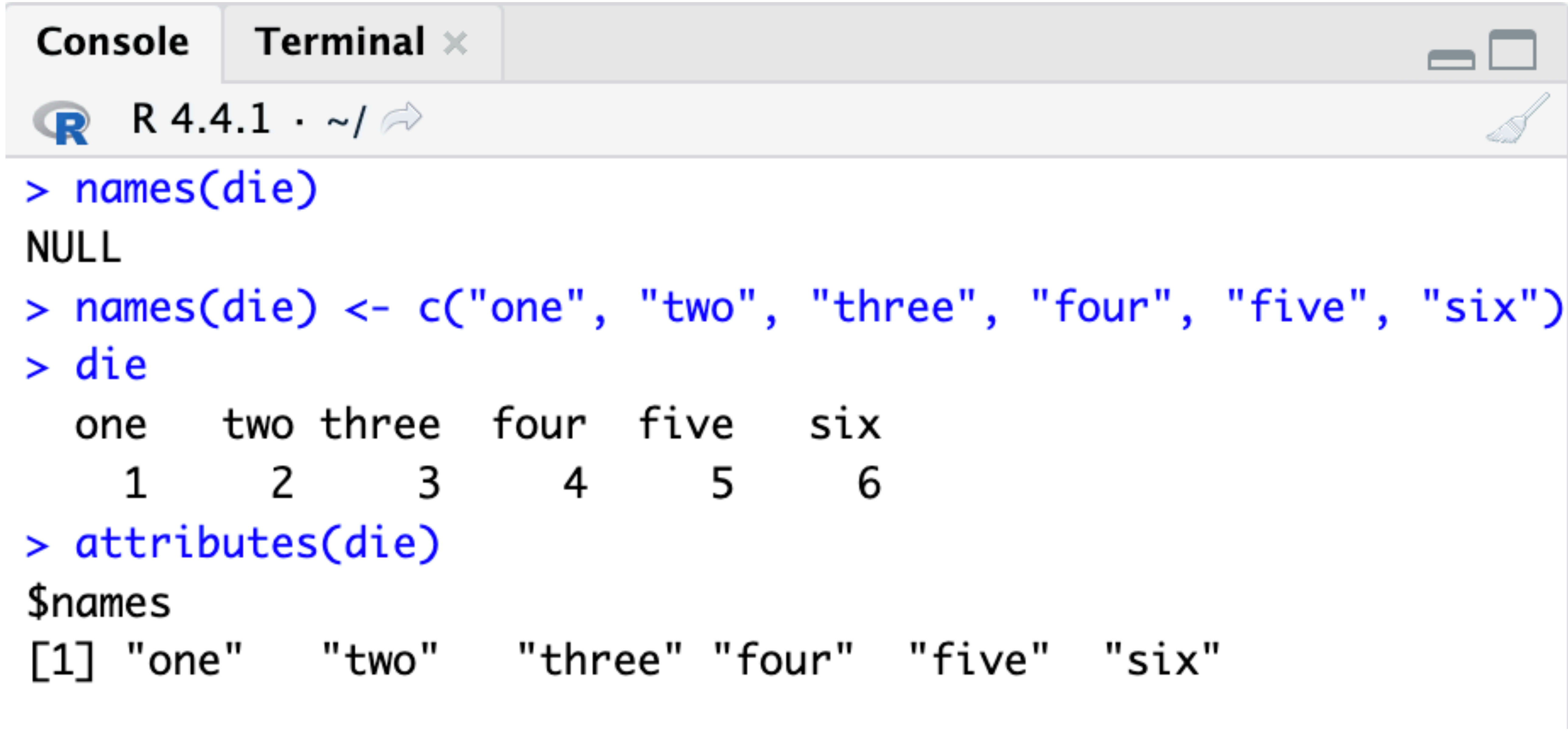


```
> die
[1] 1 2 3 4 5 6
> attributes(die)
NULL
>
```

R Objects

Names

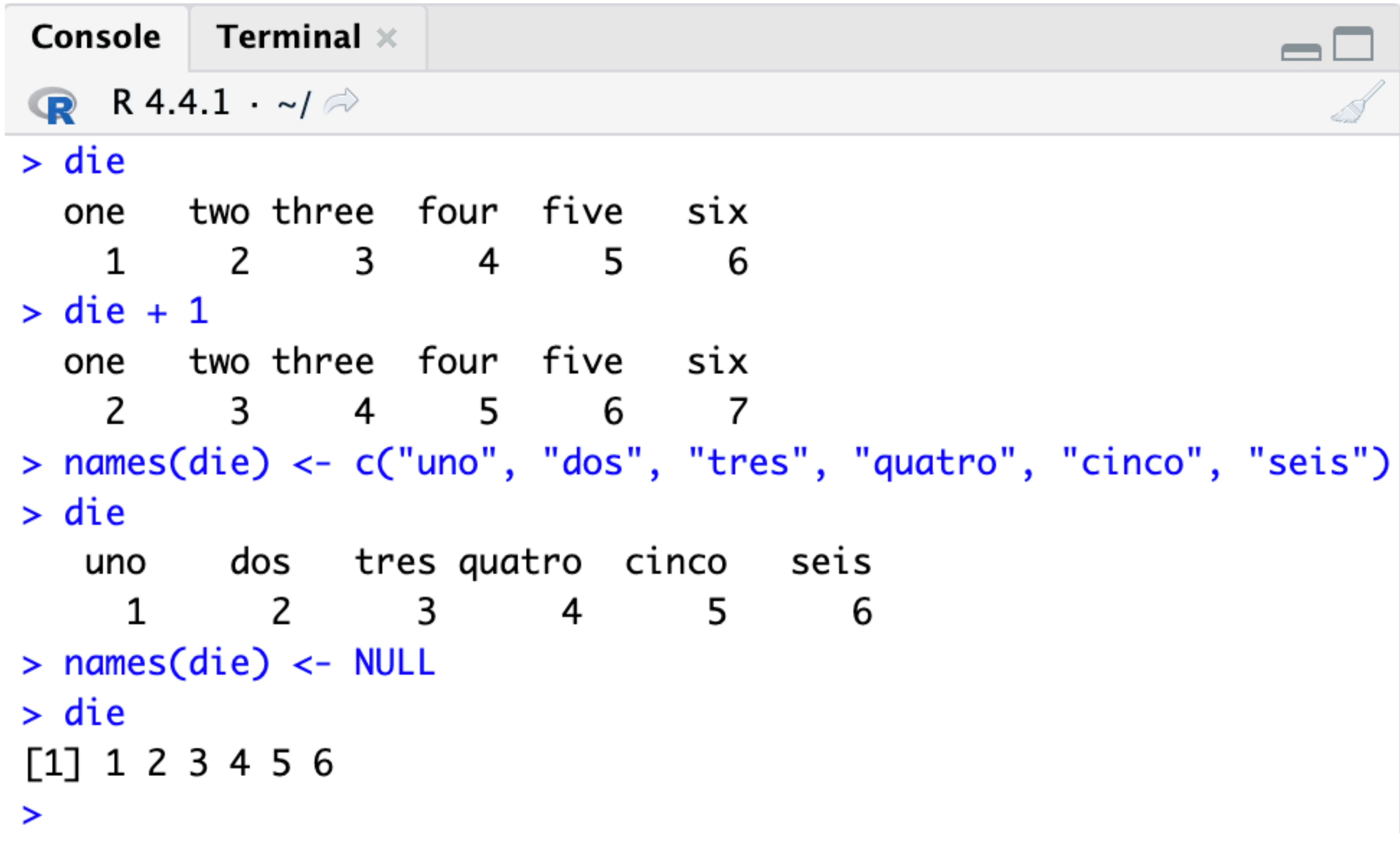
- The most common attributes to give an atomic vector are names, dimensions (dim), and classes.
- Each of these attributes has its own helper function that you can use to give attributes to an object.
- `NULL` means that `die` does not have a names attribute. You can give one to `die` by assigning a character vector to the output of `names`. The vector should include one name for each element in `die`


Console Terminal
R 4.4.1 · ~/

```
> names(die)
NULL
> names(die) <- c("one", "two", "three", "four", "five", "six")
> die
  one   two three  four  five   six 
    1     2     3     4     5     6 
> attributes(die)
$names
[1] "one"   "two"   "three" "four"  "five"  "six"
```

R Objects
Names

- Be careful, the names won't affect the actual values of the vector, nor will the names be affected when you manipulate the values of the vector.
- You can also use `names` to change the names attribute or remove it all together. To change the names, assign a new set of labels to `names`
- To remove the names attribute, set it to `NULL`



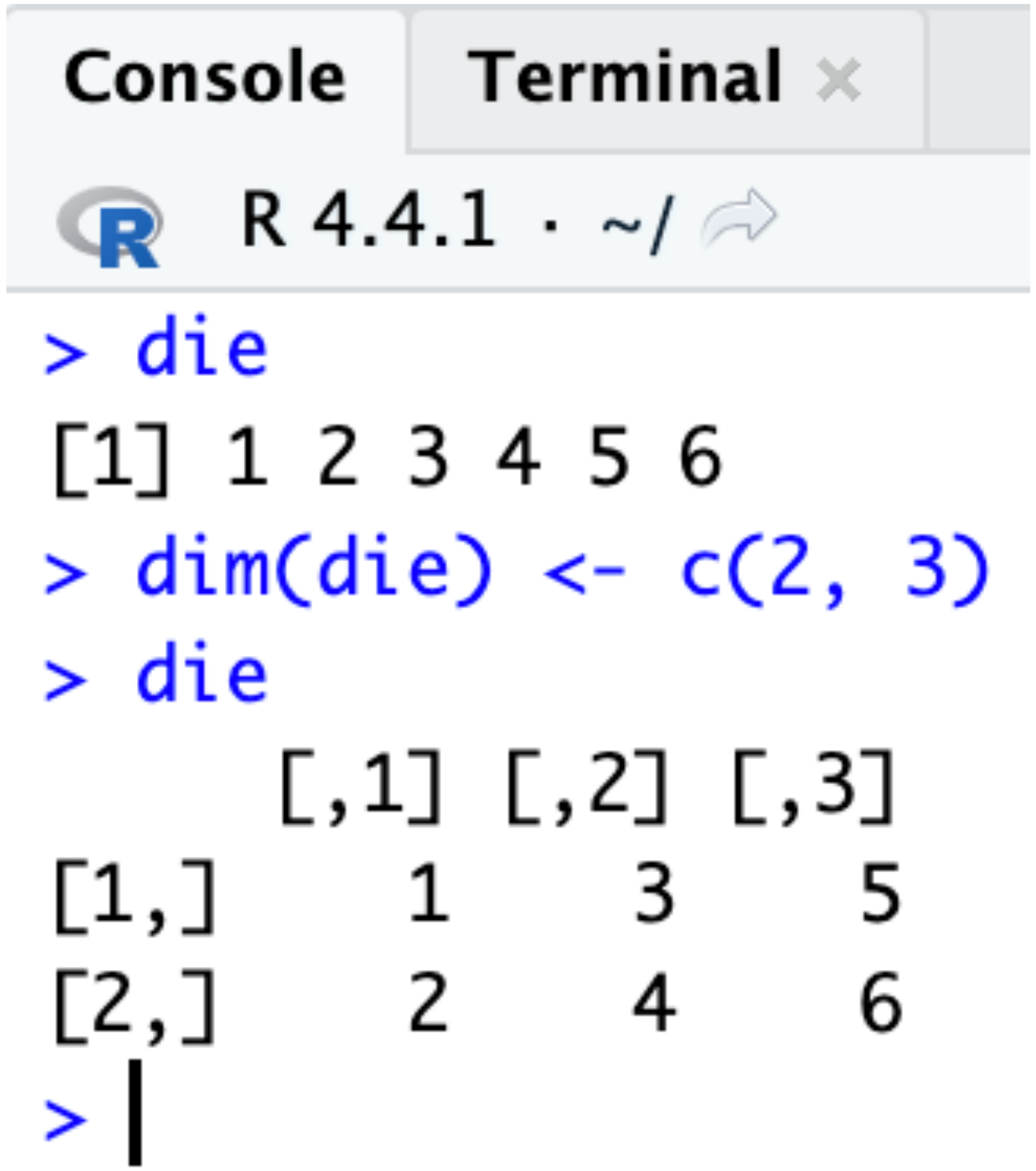
Console Terminal

R 4.4.1 · ~/

```
> die
  one   two three  four  five   six
    1     2     3     4     5     6
> die + 1
  one   two three  four  five   six
    2     3     4     5     6     7
> names(die) <- c("uno", "dos", "tres", "quatro", "cinco", "seis")
> die
   uno    dos   tres quatro  cinco   seis
     1      2      3      4      5      6
> names(die) <- NULL
> die
[1] 1 2 3 4 5 6
>
```

R Objects
Dim

- You can transform an atomic vector into an *n*-dimensional array by giving it a dimensions attribute with `dim`.
- To do this, set the `dim` attribute to a numeric vector of length *n*.
- R will reorganize the elements of the vector into *n* dimensions.
- Each dimension will have as many rows (or columns, etc.) as the *nth* value of the `dim` vector.



```
> die
[1] 1 2 3 4 5 6
> dim(die) <- c(2, 3)
> die
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> 
```