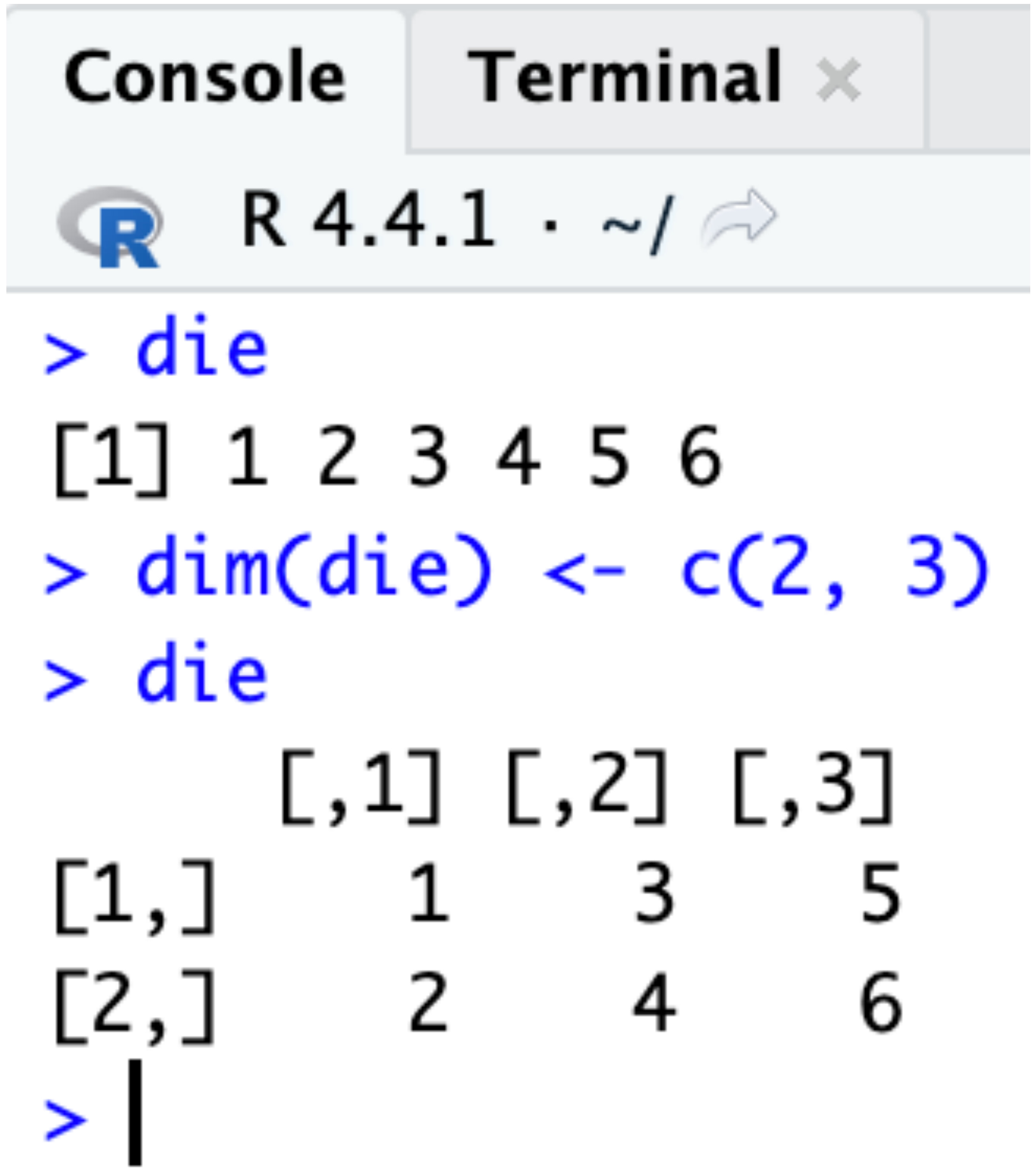
R Objects

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.

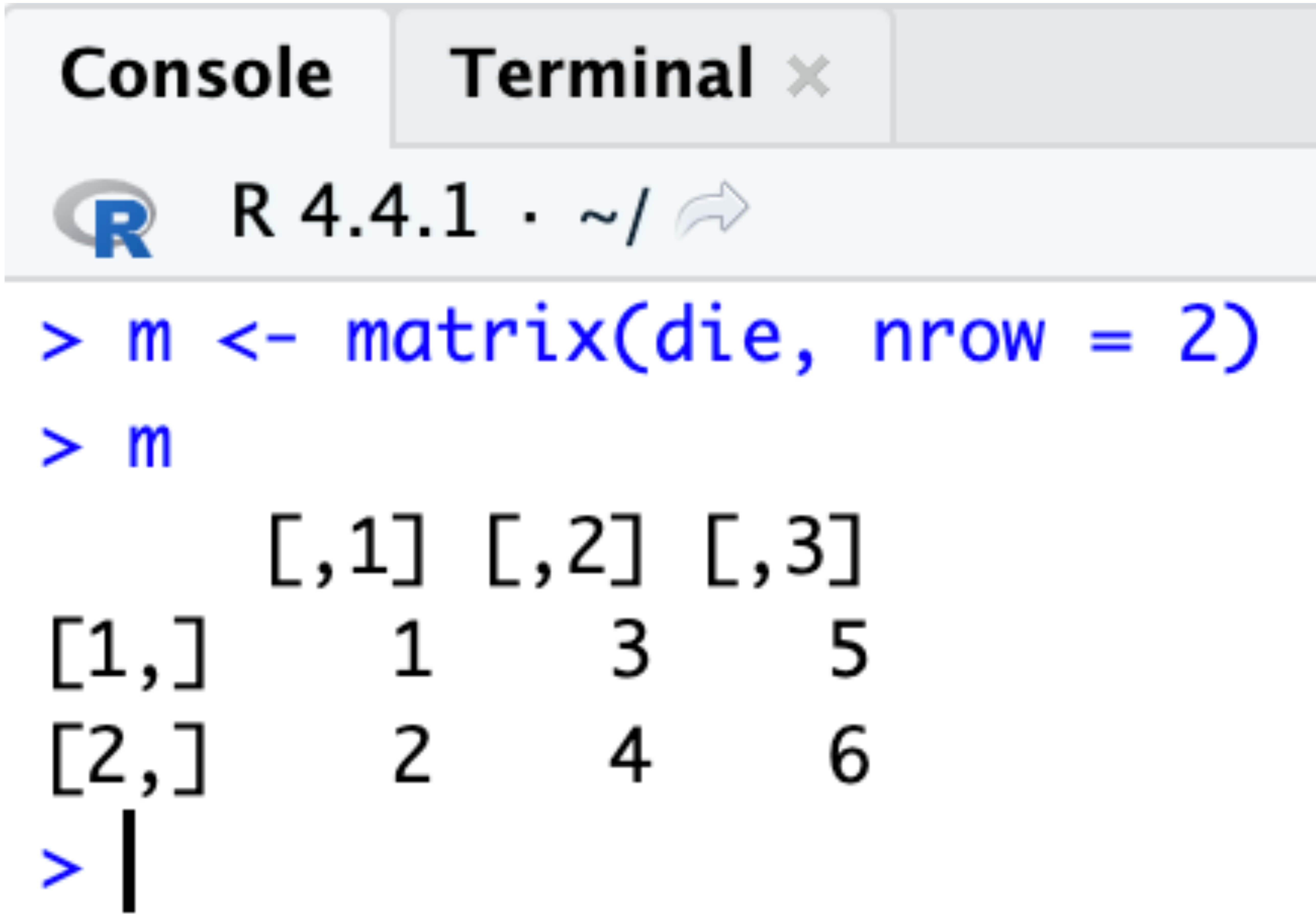


```
> die
[1] 1 2 3 4 5 6
> dim(die) <- c(2, 3)
> die
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> 
```

R Objects

Matrices

- Matrices store values in a two-dimensional array, just like a matrix from linear algebra.
- To create one, first give `matrix` an atomic vector to reorganize into a matrix.
- Then, define how many rows should be in the matrix by setting the `nrow` argument to a number. `matrix` will organize your vector of values into a matrix with the specified number of rows.
- Alternatively, you can set the `ncol` argument, which tells R how many columns to include in the matrix.

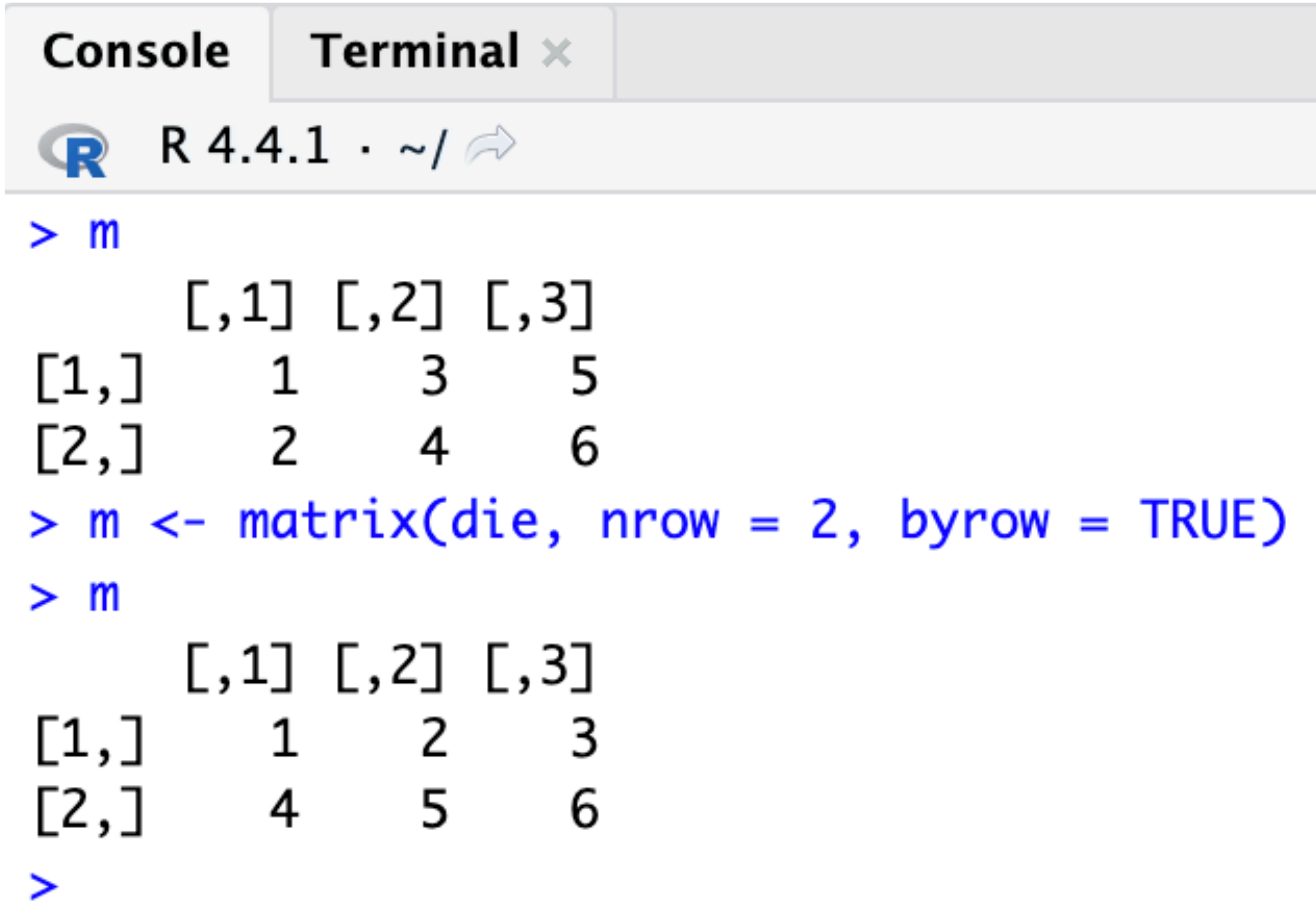


```
> m <- matrix(die, nrow = 2)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> 
```

R Objects

Matrices

- `matrix` will fill up the matrix column by column by default, but you can fill the matrix row by row if you include the argument `byrow = TRUE`
- `matrix` also has other default arguments that you can use to customize your matrix. You can read about them at `matrix`'s help page (accessible by `?matrix`).

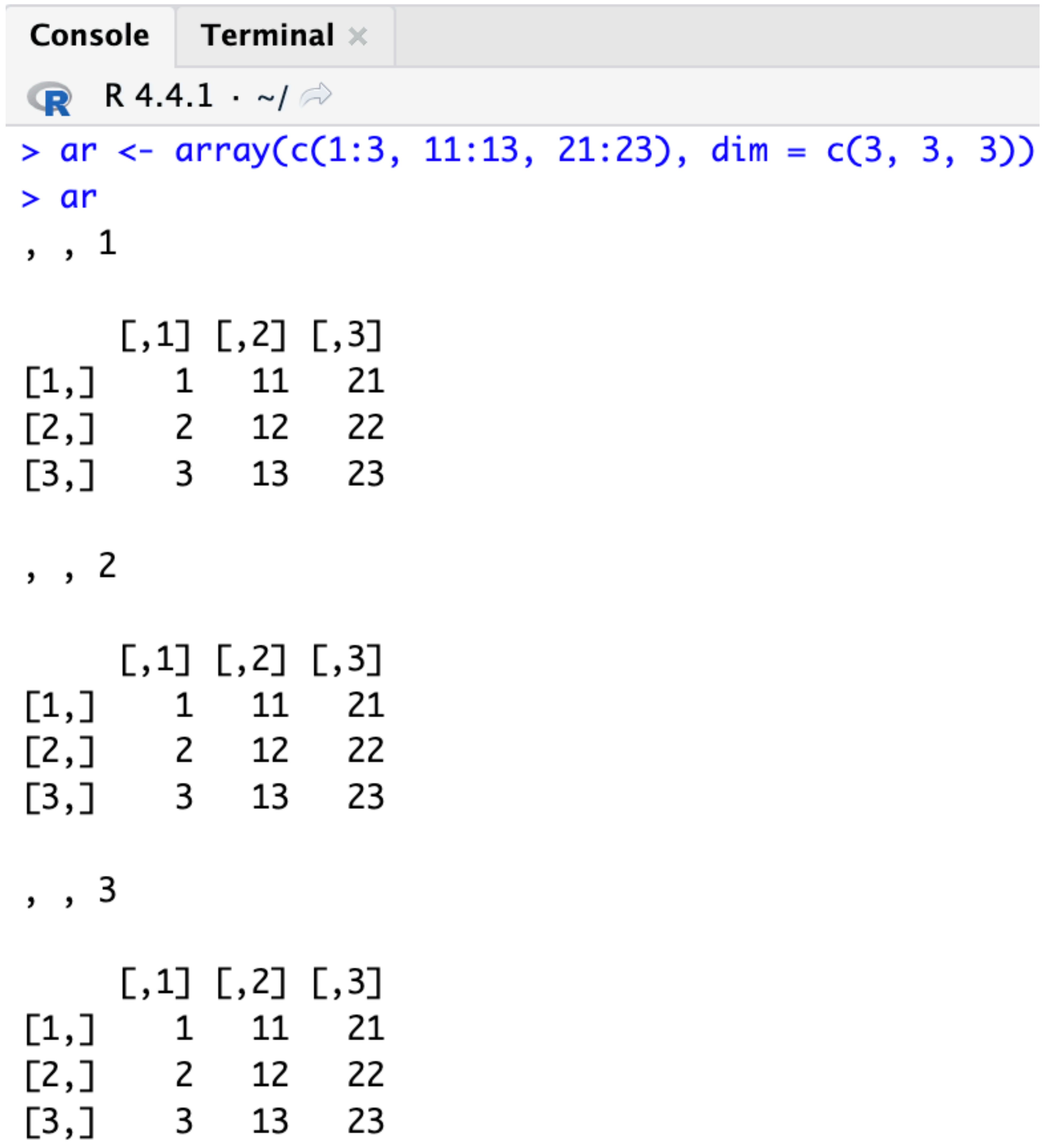


```
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> m <- matrix(die, nrow = 2, byrow = TRUE)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
>
```

R Objects

Arrays

- The `array` function creates an n-dimensional array.
- `array` is not as customizeable as `matrix` and basically does the same thing as setting the `dim` attribute.
- To use `array`, provide an atomic vector as the first argument, and a vector of dimensions as the second argument, called `dim`

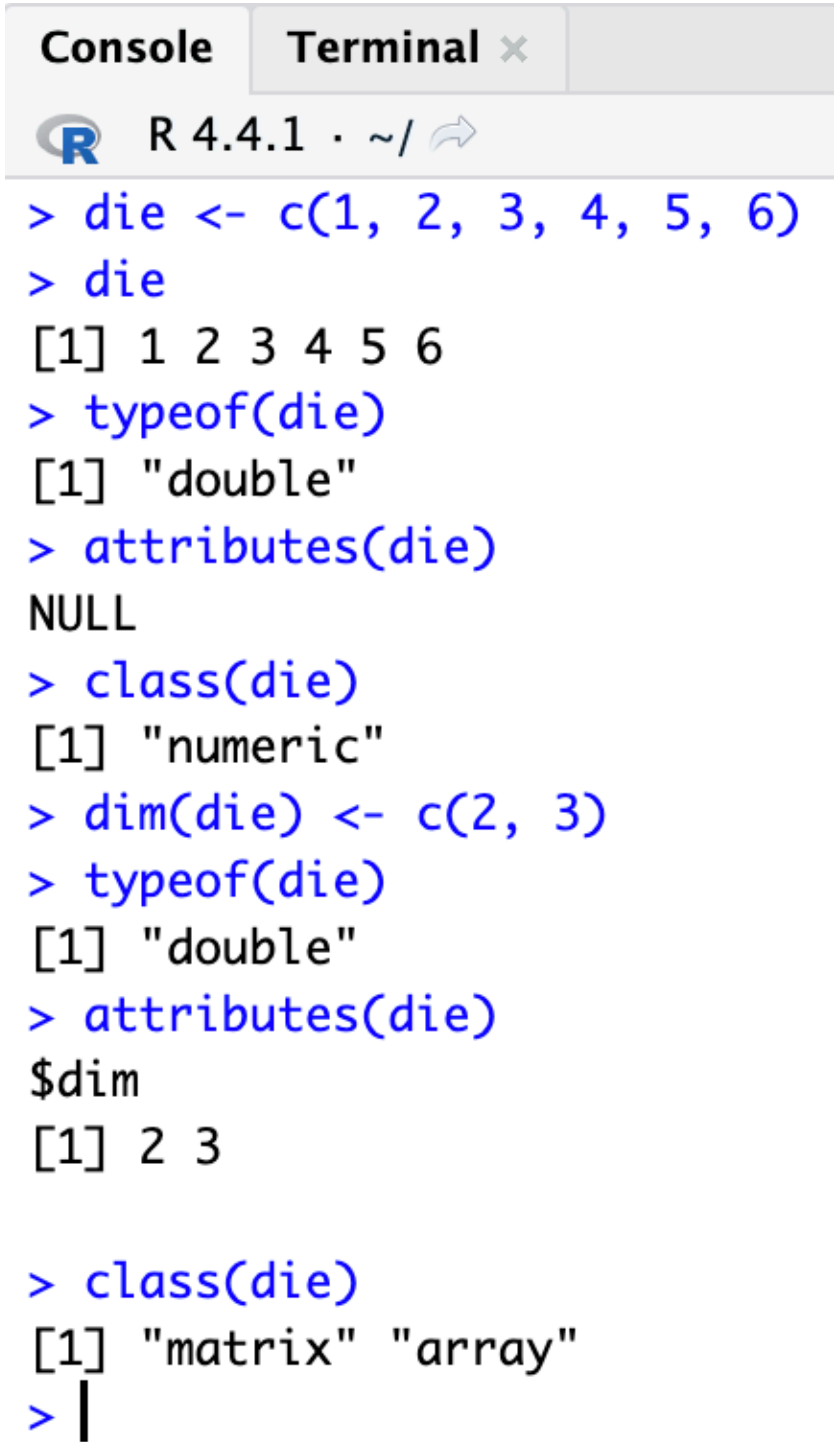


```
> ar <- array(c(1:3, 11:13, 21:23), dim = c(3, 3, 3))
> ar
, , 1

     [,1] [,2] [,3]
[1,]    1   11   21
[2,]    2   12   22
[3,]    3   13   23

, , 2

     [,1] [,2] [,3]
[1,]    1   11   21
[2,]    2   12   22
[3,]    3   13   23

, , 3

     [,1] [,2] [,3]
[1,]    1   11   21
[2,]    2   12   22
[3,]    3   13   23
```

R Objects

Class

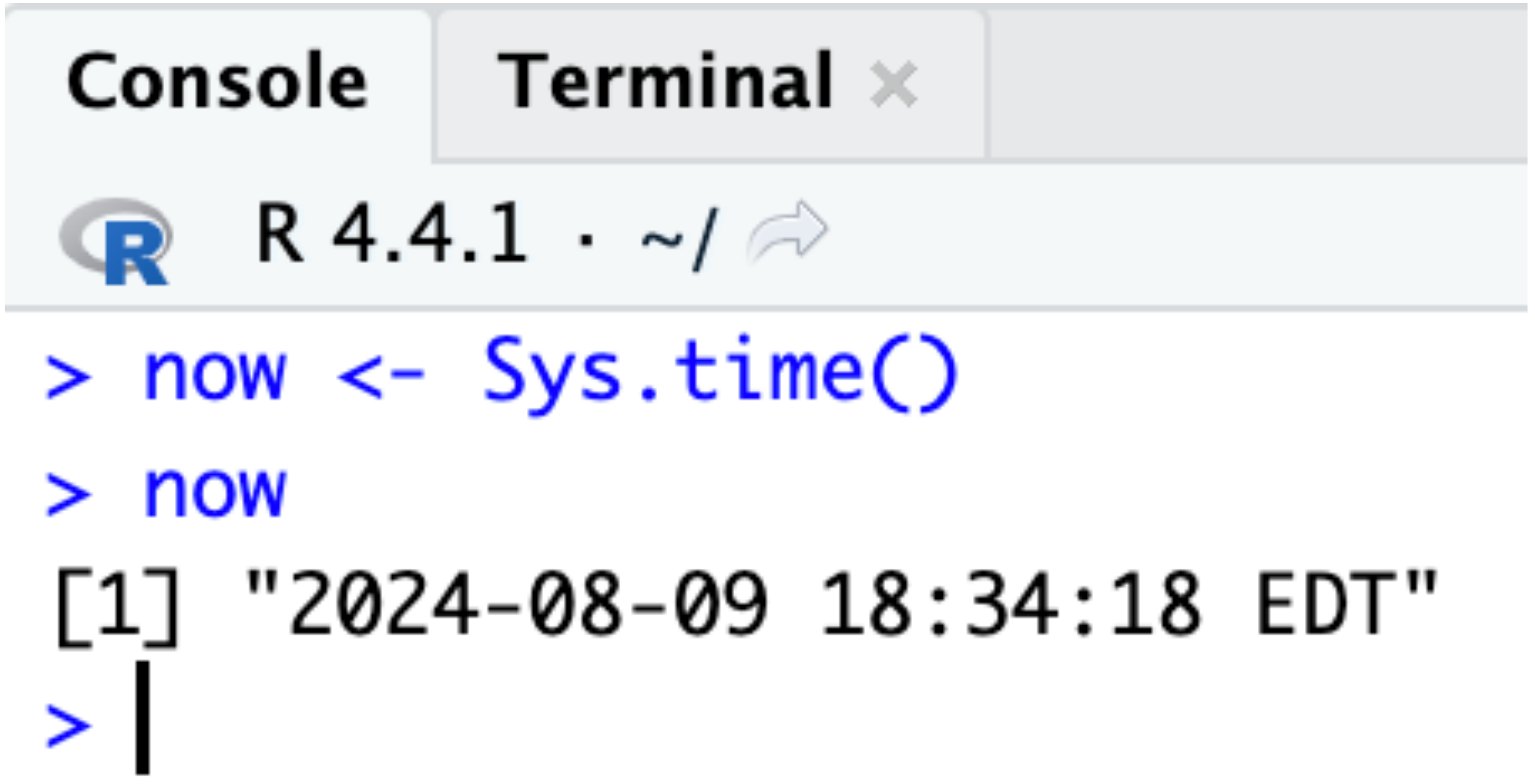
- Notice that changing the dimensions of your object will not change the type of the object, but it *will* change the object's `class` attribute.
- A matrix is a special case of an atomic vector.
- Every element in the matrix is still a double, but the elements have been arranged into a new structure.
- R added a `class` attribute to `die` when you changed its dimensions. Many R functions will specifically look for an object's `class` attribute.
- Note that an object's `class` attribute will not always appear when you run `attributes`; you may need to specifically search for it with `class`



```
> die <- c(1, 2, 3, 4, 5, 6)
> die
[1] 1 2 3 4 5 6
> typeof(die)
[1] "double"
> attributes(die)
NULL
> class(die)
[1] "numeric"
> dim(die) <- c(2, 3)
> typeof(die)
[1] "double"
> attributes(die)
$dim
[1] 2 3

> class(die)
[1] "matrix" "array"
> 
```

R Objects
Dates and Times

- The attribute system lets R represent more types of data than just doubles, integers, characters, logicals, complexes, and raws. The time looks like a character string when you display it, but its data type is actually `"double"`, and its class is `"POSIXct" "POSIXt"` (it has two classes)

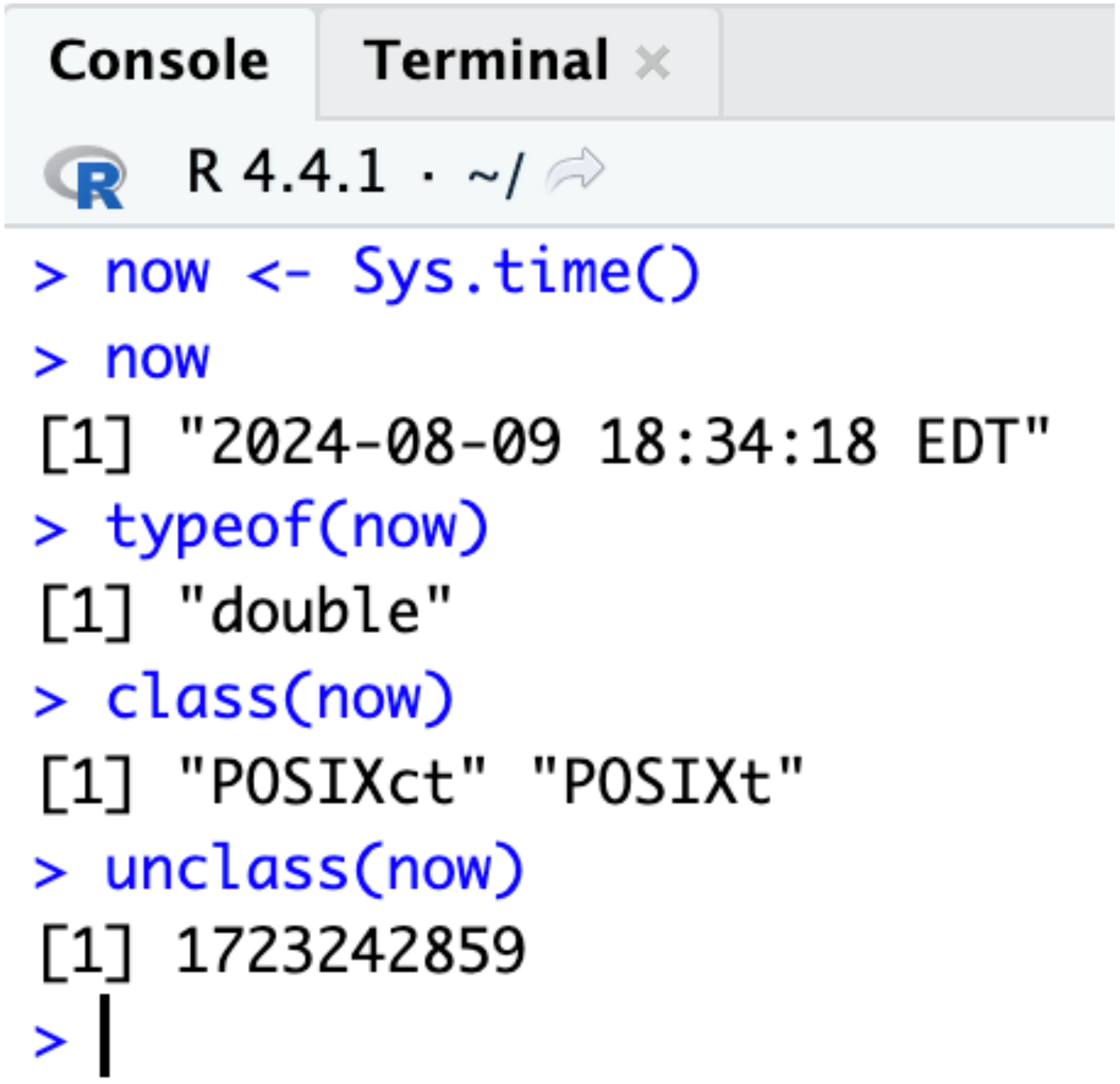


```
> now <- Sys.time()
> now
[1] "2024-08-09 18:34:18 EDT"
>
```

R Objects

POSIXct

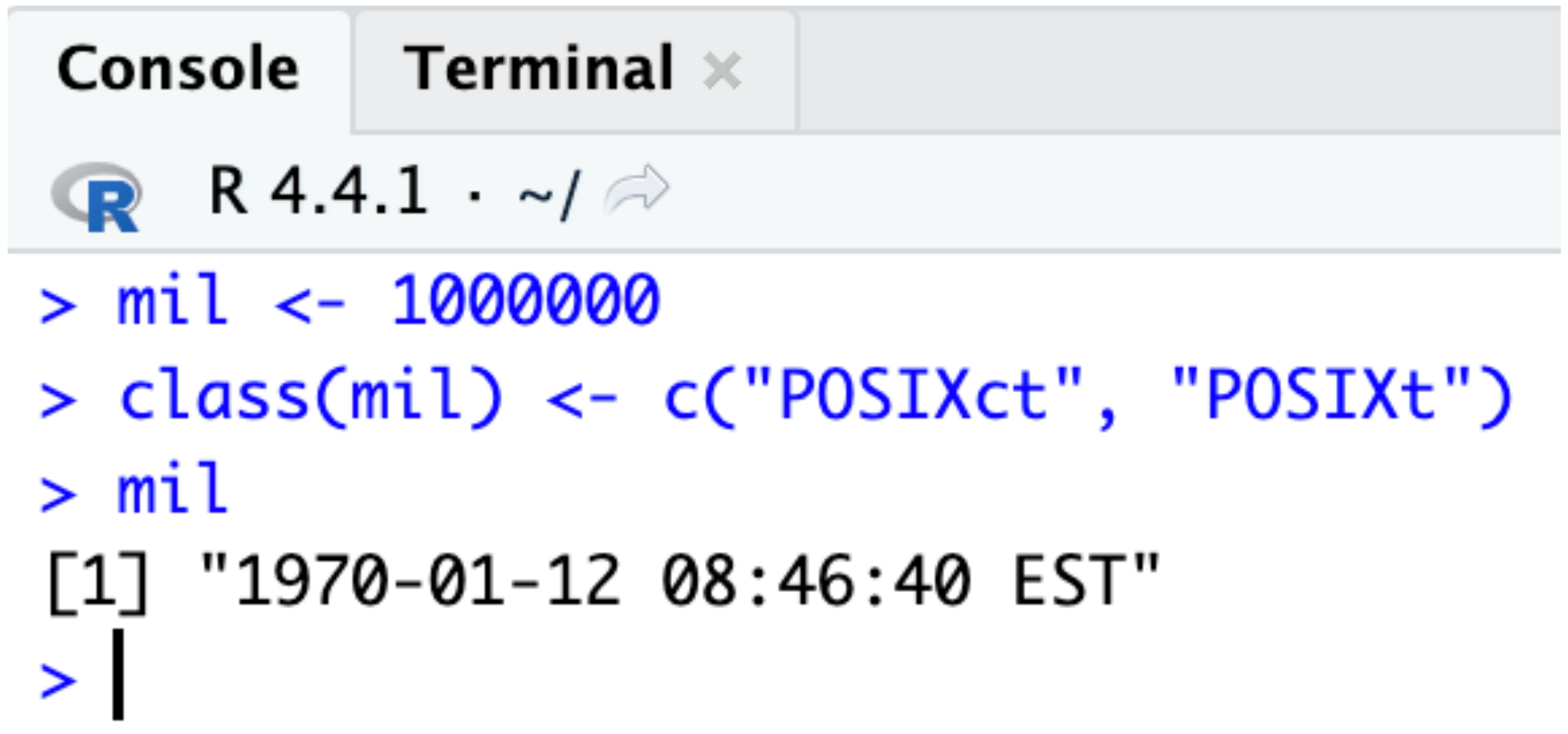
- POSIXct is a widely used framework for representing dates and times.
- In the POSIXct framework, each time is represented by the number of seconds that have passed between the time and 12:00 AM January 1st 1970 (UTC).
- R creates the time object by building a double vector with one element, `1723242859`. You can see this vector by removing the `class` attribute of `now`, or by using the `unclass` function, which does the same thing



```
> now <- Sys.time()
> now
[1] "2024-08-09 18:34:18 EDT"
> typeof(now)
[1] "double"
> class(now)
[1] "POSIXct" "POSIXt"
> unclass(now)
[1] 1723242859
>
```

R Objects
POSIXct

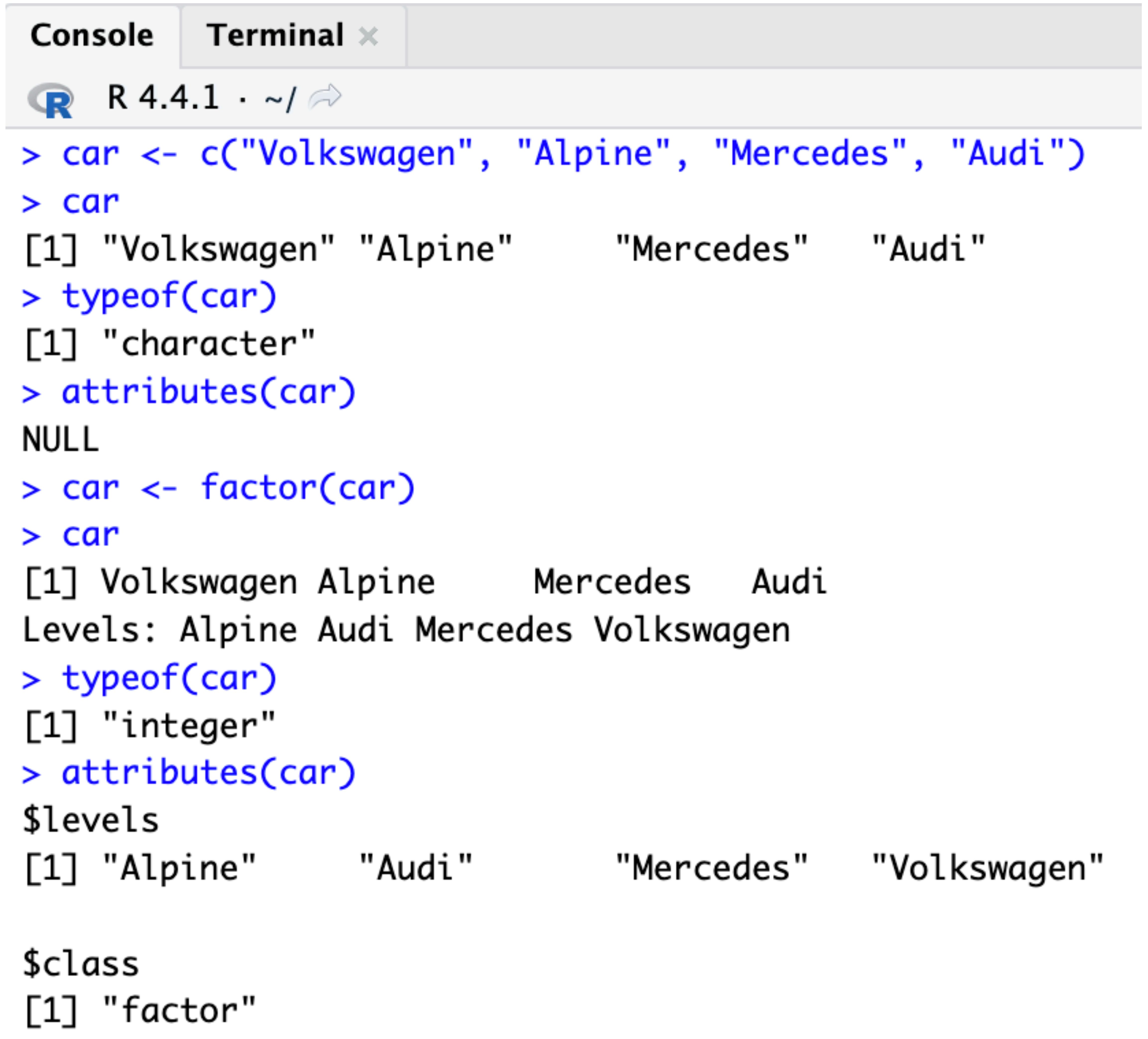
- You can take advantage of this system by giving the `POSIXct` class to random R objects. For example, have you ever wondered what day it was a million seconds after 12:00 a.m. Jan. 1, 1970?
- Jan. 12, 1970. A million seconds goes by faster than you would think. This conversion worked well because the `POSIXct` class does not rely on any additional attributes, but in general, forcing the class of an object is a bad idea.



```
> mil <- 1000000
> class(mil) <- c("POSIXct", "POSIXt")
> mil
[1] "1970-01-12 08:46:40 EST"
> 
```

R Objects
Factors

- Factors are R's way of storing categorical information, like ethnicity or eye color.
- A factor can only have certain values and these values may have their own idiosyncratic order.
- This arrangement makes factors very useful for recording the treatment levels of a study and other categorical variables.

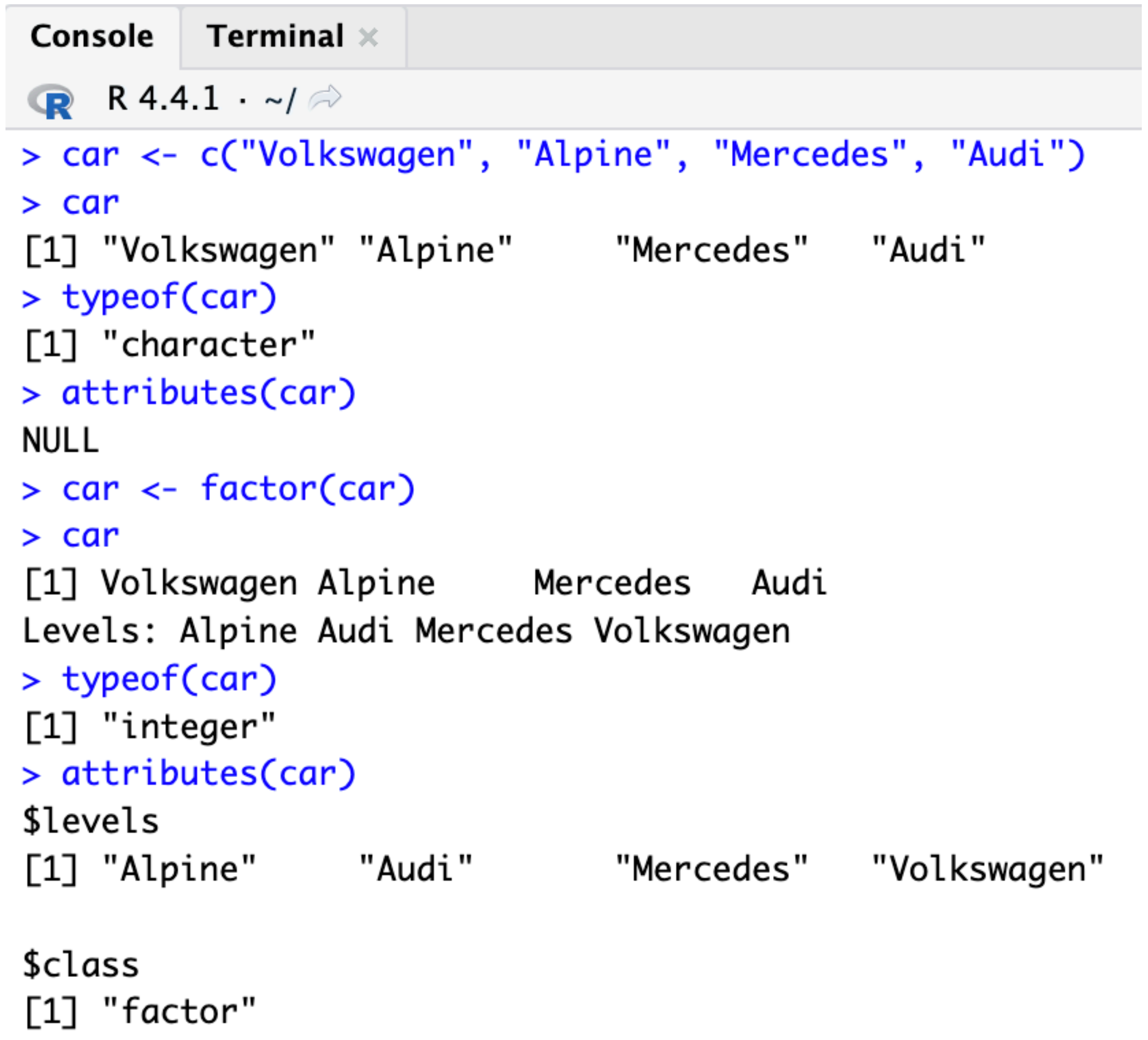


```
> car <- c("Volkswagen", "Alpine", "Mercedes", "Audi")
> car
[1] "Volkswagen" "Alpine"     "Mercedes"   "Audi"
> typeof(car)
[1] "character"
> attributes(car)
NULL
> car <- factor(car)
> car
[1] Volkswagen Alpine     Mercedes   Audi
Levels: Alpine Audi Mercedes Volkswagen
> typeof(car)
[1] "integer"
> attributes(car)
$levels
[1] "Alpine"     "Audi"       "Mercedes"   "Volkswagen"

$class
[1] "factor"
```

R Objects

Factors

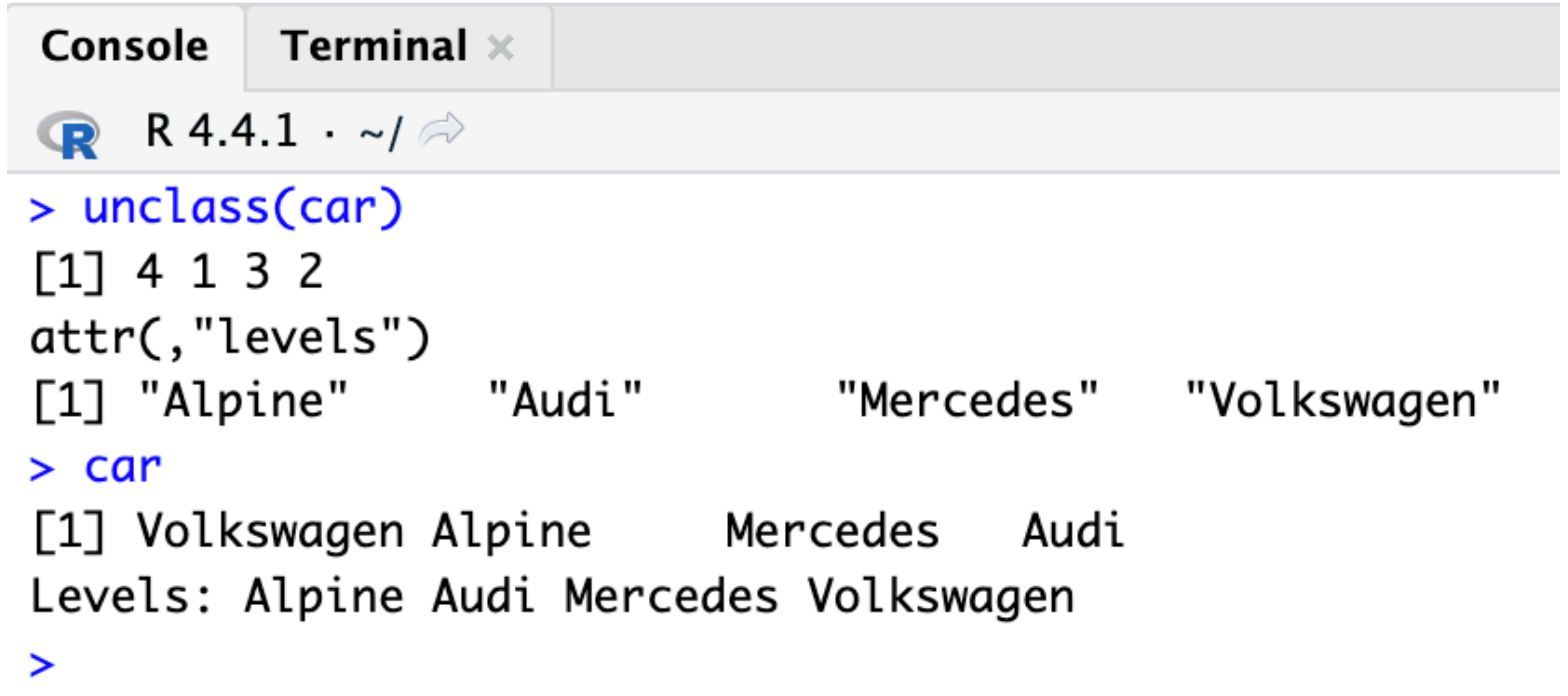
- To make a factor, pass an atomic vector into the `factor` function.
- R will recode the data in the vector as integers and store the results in an integer vector.
- R will also add a `levels` attribute to the integer, which contains a set of labels for displaying the factor values, and a `class` attribute, which contains the class `factor`



```
> car <- c("Volkswagen", "Alpine", "Mercedes", "Audi")
> car
[1] "Volkswagen" "Alpine"     "Mercedes"   "Audi"
> typeof(car)
[1] "character"
> attributes(car)
NULL
> car <- factor(car)
> car
[1] Volkswagen Alpine     Mercedes   Audi
Levels: Alpine Audi Mercedes Volkswagen
> typeof(car)
[1] "integer"
> attributes(car)
$levels
[1] "Alpine"     "Audi"       "Mercedes"   "Volkswagen"

$class
[1] "factor"
```

R Objects
Factors

- You can see exactly how R is storing your factor with `unclass`
- R uses the levels attribute when it displays the factor. R will display each `1` as `Alpine`, the first label in the levels vector, each `2` as `Audi`, the second label etc.

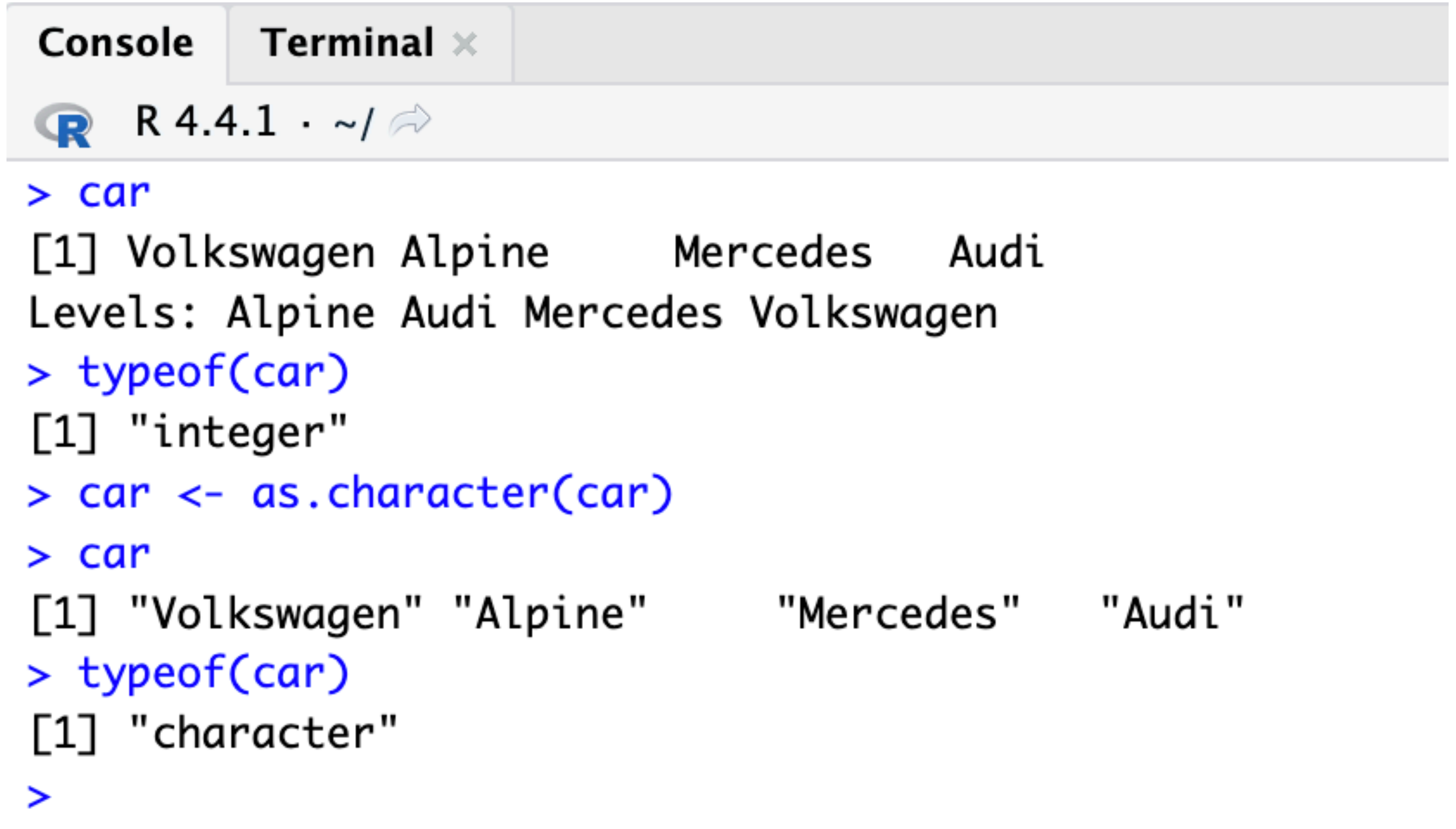


```
> unclass(car)
[1] 4 1 3 2
attr(,"levels")
[1] "Alpine"     "Audi"       "Mercedes"   "Volkswagen"
> car
[1] Volkswagen Alpine     Mercedes   Audi
Levels: Alpine Audi Mercedes Volkswagen
>
```

R Objects

Factors

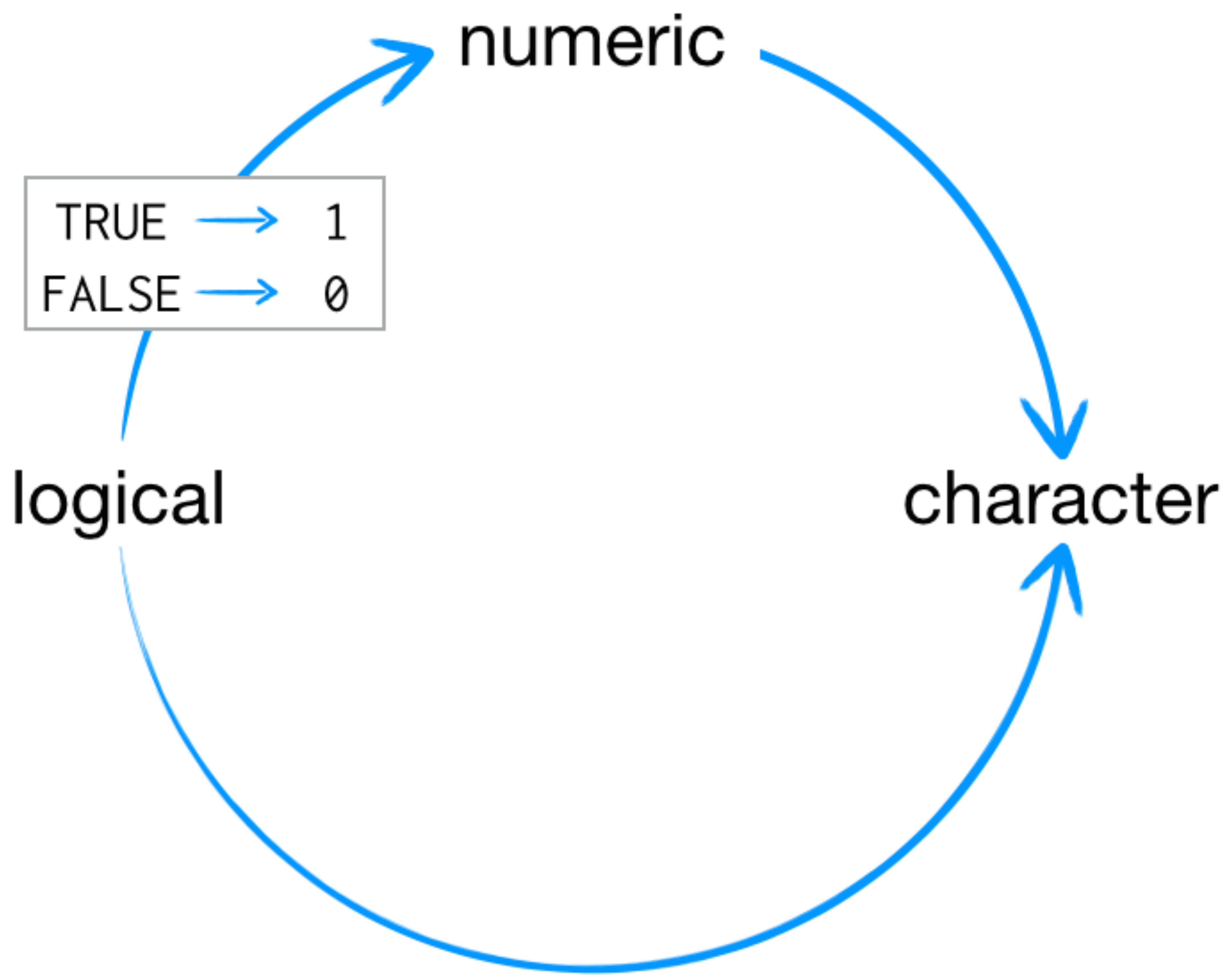
- Factors can be confusing since they look like character strings but behave like integers.
- R will often try to convert character strings to factors when you load and create data. In general, you will have a smoother experience if you do NOT let R make factors until you ask for them.
- You can convert a factor to a character string with the `as.character` function.



```
> car
[1] Volkswagen Alpine     Mercedes   Audi
Levels: Alpine Audi Mercedes Volkswagen
> typeof(car)
[1] "integer"
> car <- as.character(car)
> car
[1] "Volkswagen" "Alpine"     "Mercedes"   "Audi"
> typeof(car)
[1] "character"
>
```

R Objects
Coercion

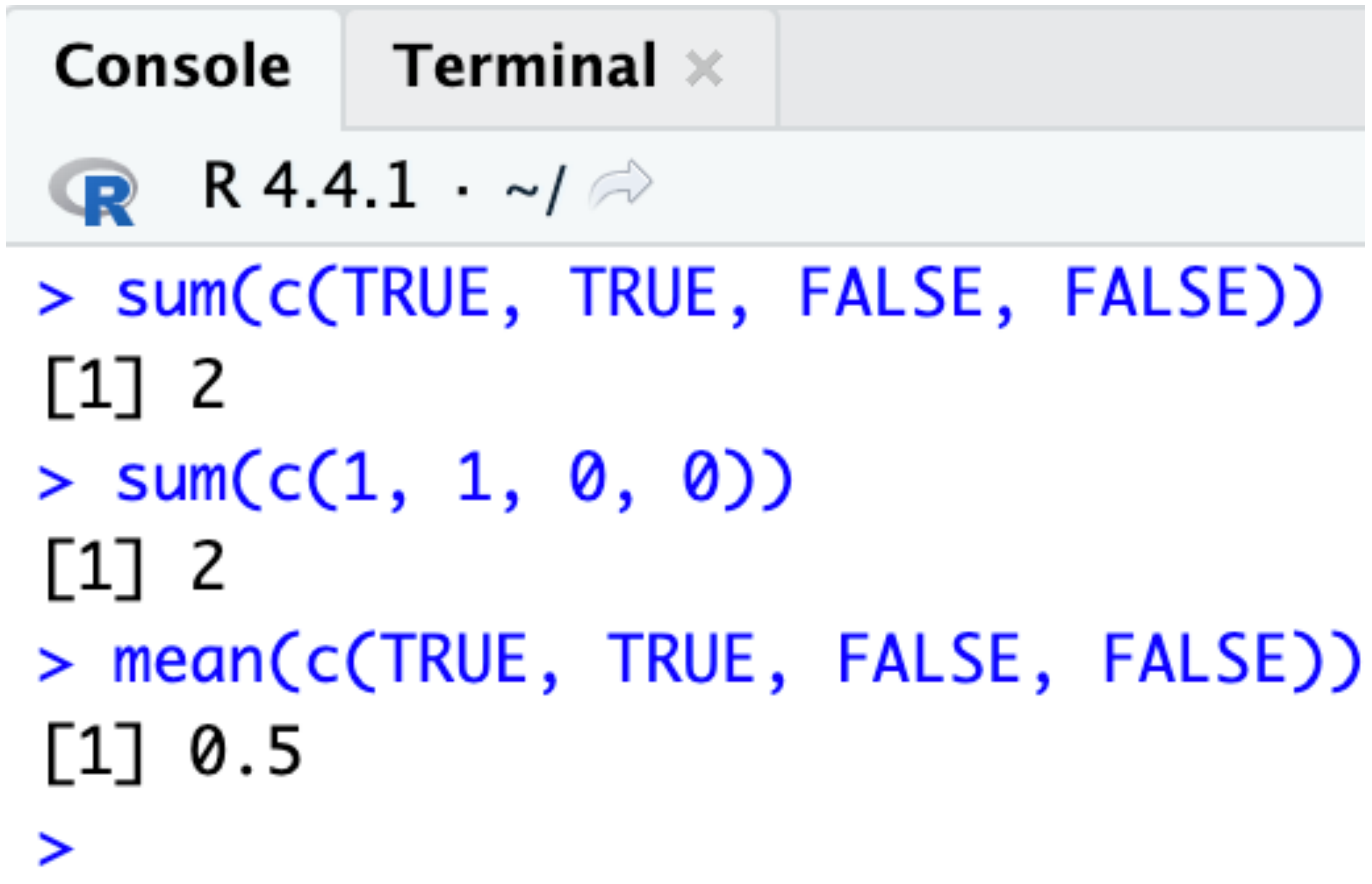
- So how does R coerce data types?
- If a character string is present in an atomic vector, R will convert everything else in the vector to character strings.
- If a vector only contains logicals and numbers, R will convert the logicals to numbers; every TRUE becomes a 1, and every FALSE becomes a 0.



R Objects

Coercion

- R uses the same coercion rules when you try to do math with logical values.
- This means that `sum` will count the number of `TRUE`s in a logical vector (and `mean` will calculate the proportion of `TRUE`s)



```
> sum(c(TRUE, TRUE, FALSE, FALSE))
[1] 2
> sum(c(1, 1, 0, 0))
[1] 2
> mean(c(TRUE, TRUE, FALSE, FALSE))
[1] 0.5
>
```

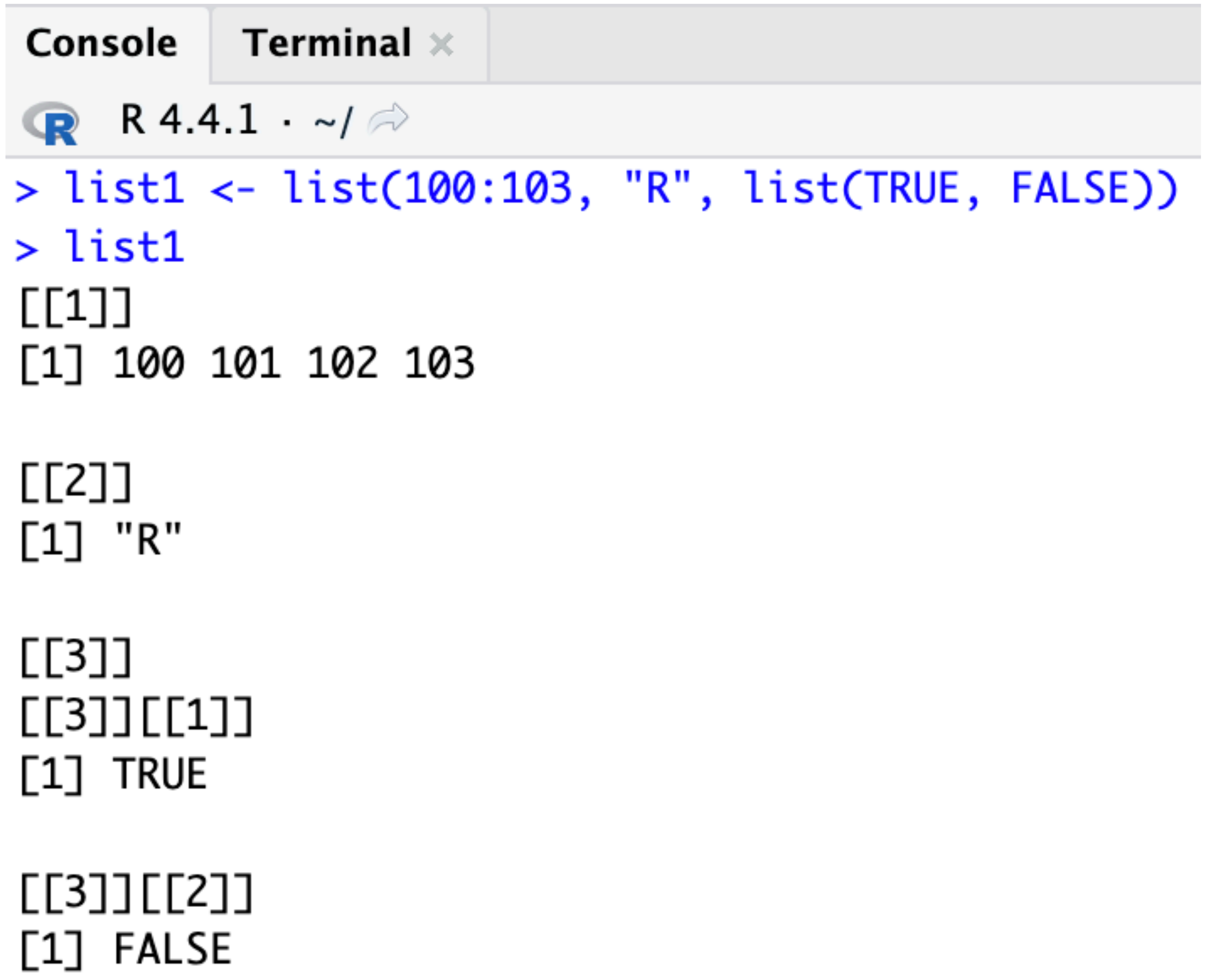
R Objects
Coercion

- Many data sets contain multiple types of information.
- The inability of vectors, matrices, and arrays to store multiple data types seems like a major limitation.
- So why bother with them?
- In some cases, using only a single type of data is a huge advantage. Vectors, matrices, and arrays make it very easy to do math on large sets of numbers because R knows that it can manipulate each value the same way.
- Operations with vectors, matrices, and arrays also tend to be fast because the objects are so simple to store in memory.

R Objects

Lists

- Lists group data into a one-dimensional set.
- However, lists do not group together individual values. They group together R objects.
- For example, you can make a list that contains a numeric vector of length 31 in its first element, a character vector of length 1 in its second element, and a new list of length 2 in its third element. To do this, use the `list` function.

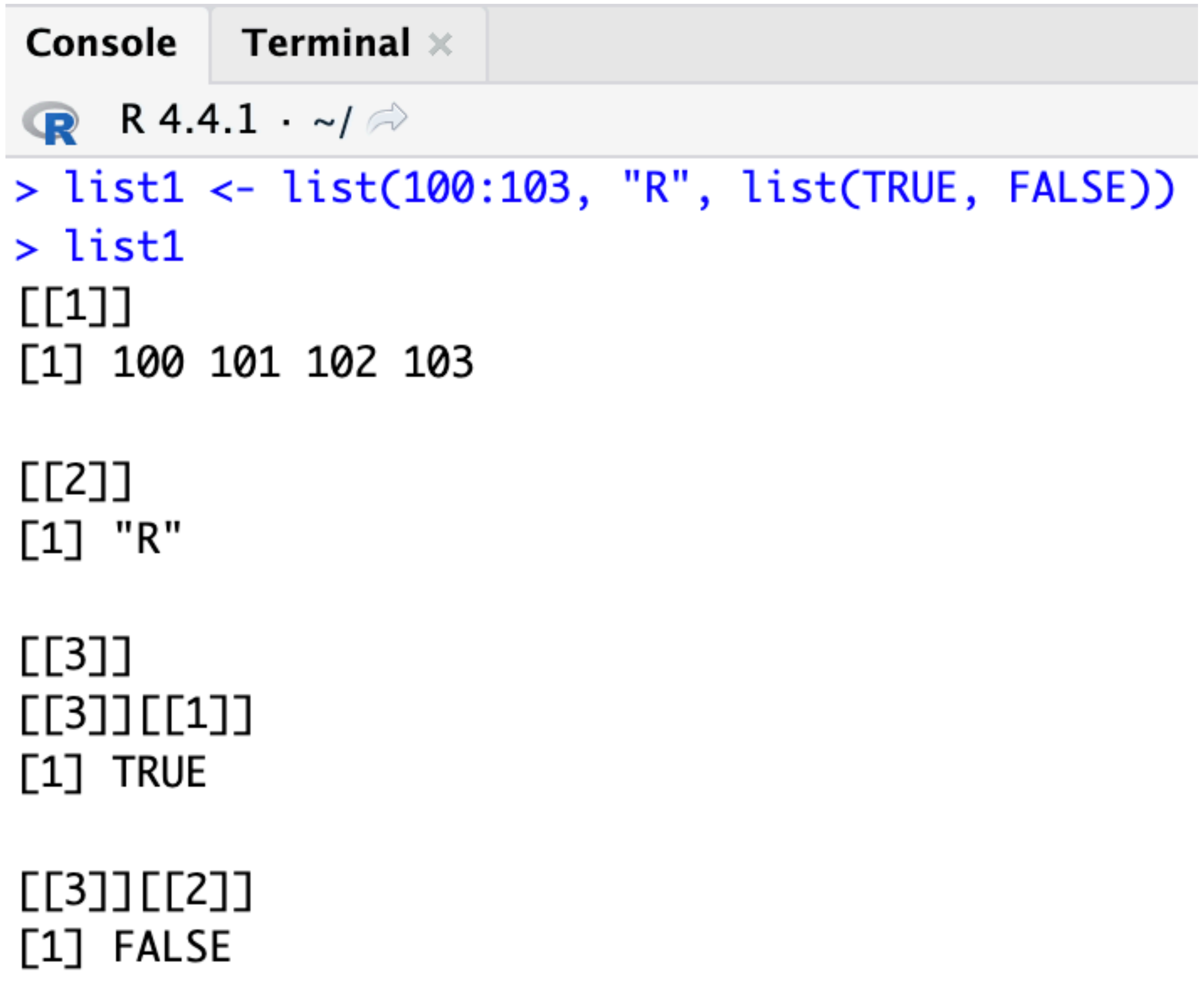


```
> list1 <- list(100:103, "R", list(TRUE, FALSE))
> list1
[[1]]
[1] 100 101 102 103

[[2]]
[1] "R"

[[3]]
[[3]][[1]]
[1] TRUE

[[3]][[2]]
[1] FALSE
```

R Objects

Lists

- The double-bracketed indexes tell you which element of the list is being displayed.
- The single-bracket indexes tell you which subelement of an element is being displayed.
- For example, `100` is the first subelement of the first element in the list. `"R"` is the first sub-element of the second element.
- This two-system notation arises because each element of a list can be *any* R object, including a new vector (or list) with its own indexes.

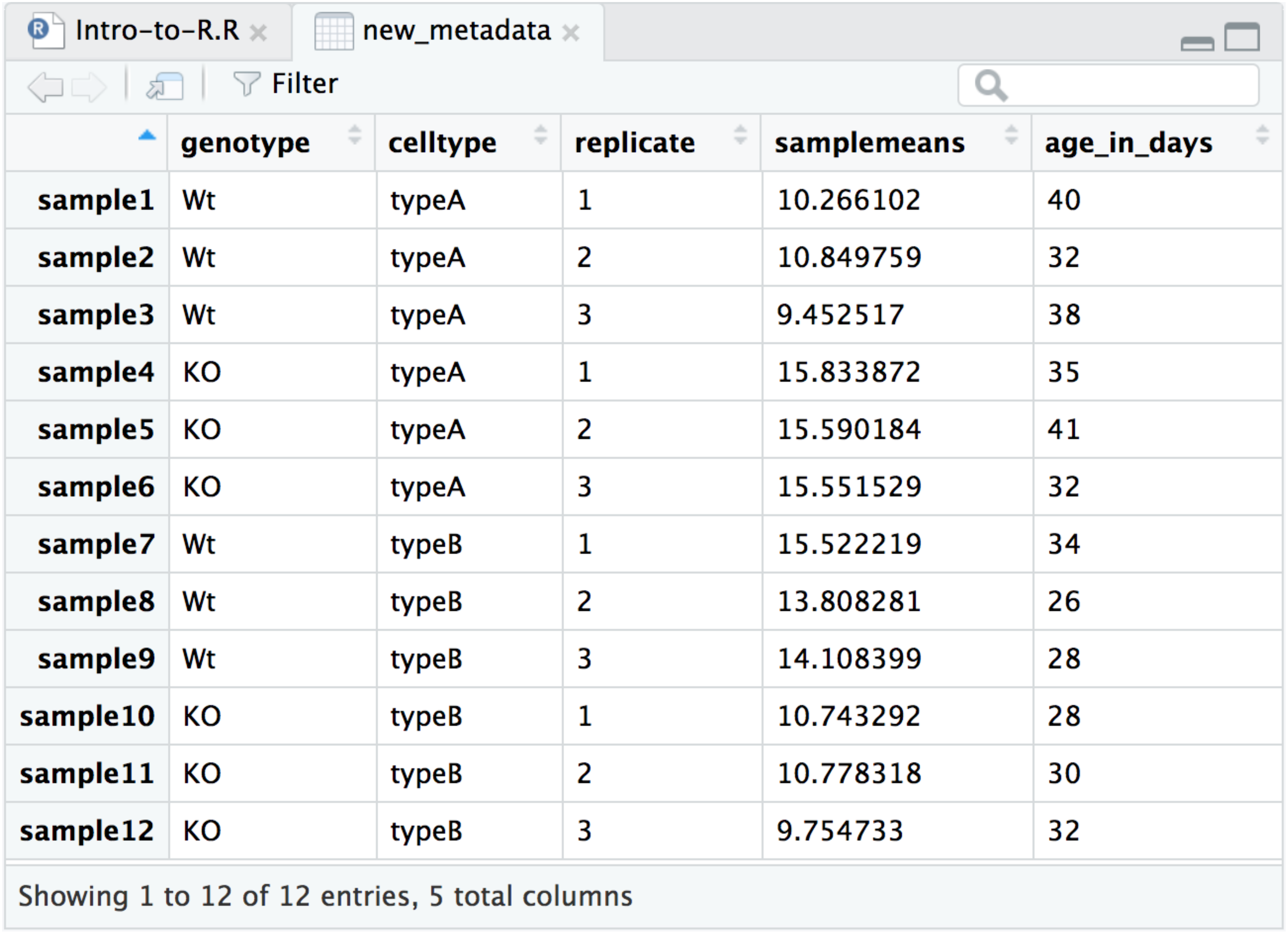


```
> list1 <- list(100:103, "R", list(TRUE, FALSE))
> list1
[[1]]
[1] 100 101 102 103

[[2]]
[1] "R"

[[3]]
[[3]][[1]]
[1] TRUE

[[3]][[2]]
[1] FALSE
```

R Objects

DataFrames

- Data frames are the two-dimensional version of a list.
- They are far and away the most useful storage structure for data analysis, and they provide an ideal way to store an entire deck of cards.
- You can think of a data frame as R's equivalent to the Excel spreadsheet because it stores data in a similar format.



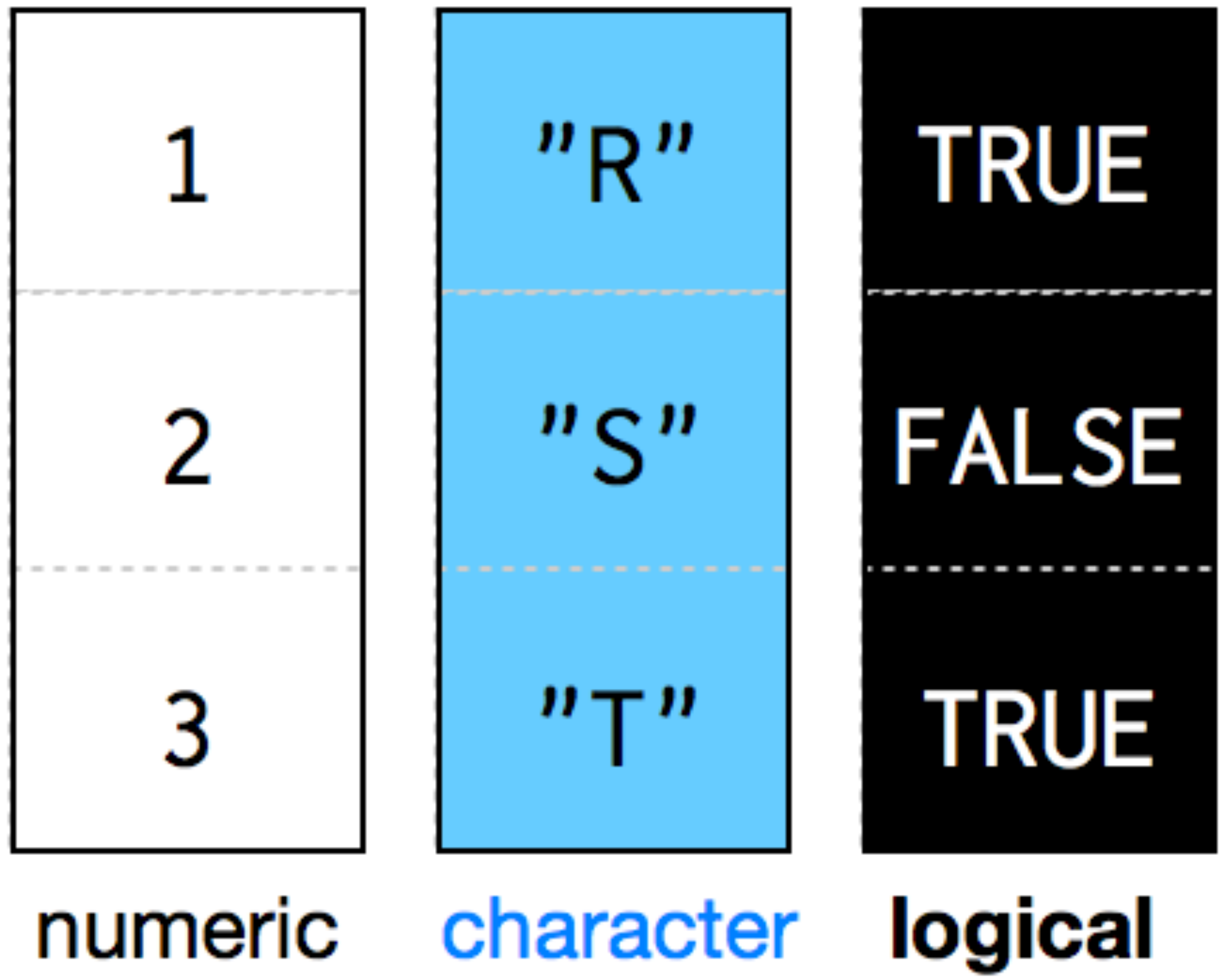
	genotype	celltype	replicate	samplemeans	age_in_days
sample1	Wt	typeA	1	10.266102	40
sample2	Wt	typeA	2	10.849759	32
sample3	Wt	typeA	3	9.452517	38
sample4	KO	typeA	1	15.833872	35
sample5	KO	typeA	2	15.590184	41
sample6	KO	typeA	3	15.551529	32
sample7	Wt	typeB	1	15.522219	34
sample8	Wt	typeB	2	13.808281	26
sample9	Wt	typeB	3	14.108399	28
sample10	KO	typeB	1	10.743292	28
sample11	KO	typeB	2	10.778318	30
sample12	KO	typeB	3	9.754733	32

Showing 1 to 12 of 12 entries, 5 total columns

R Objects

DataFrames

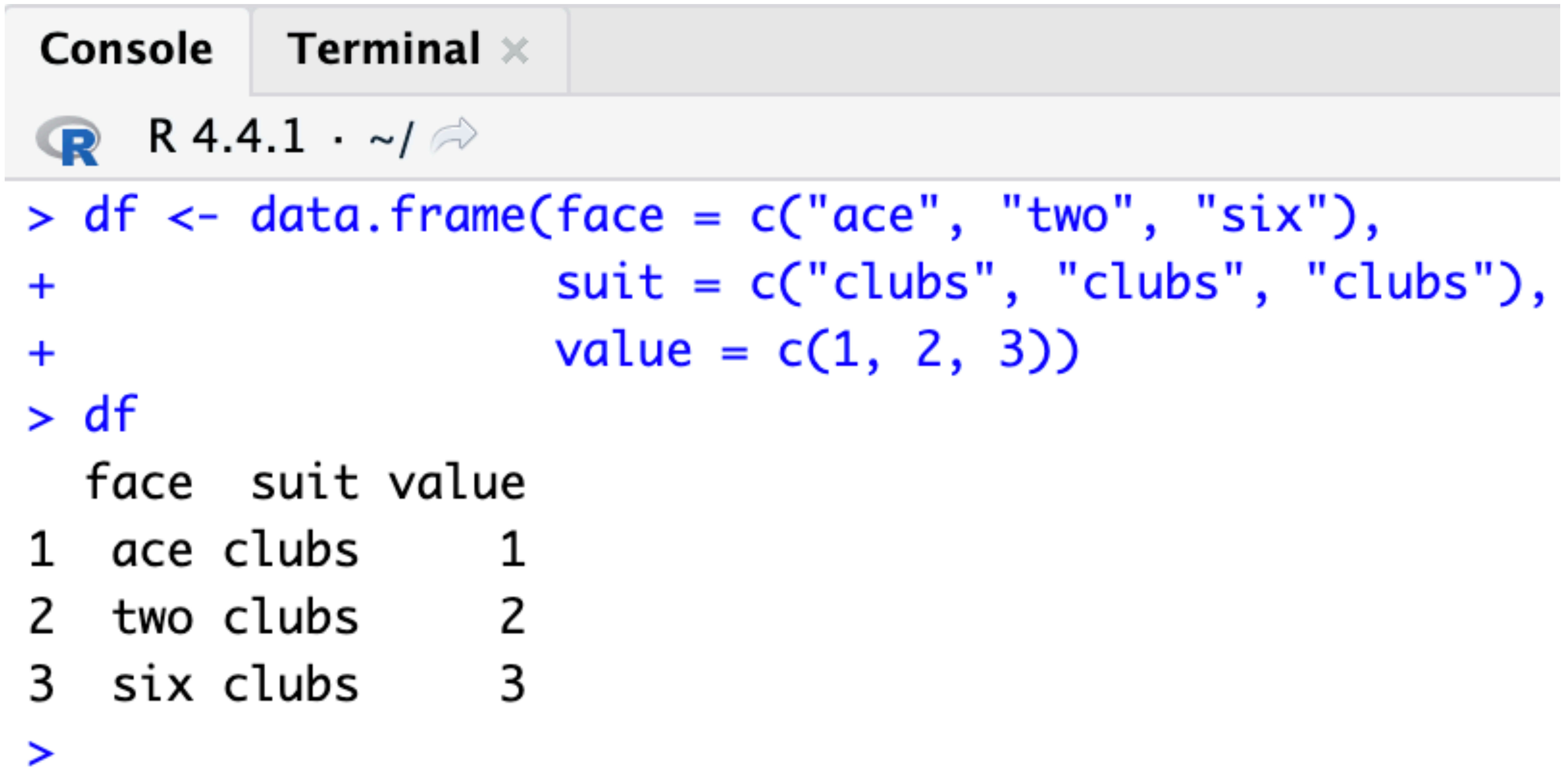
- Data frames group vectors together into a two-dimensional table. Each vector becomes a column in the table.
- As a result, each column of a data frame can contain a different type of data; but within a column, every cell must be the same type of data.



R Objects

DataFrames

- Creating a data frame by hand takes a lot of typing, but you can do it with the `data.frame` function.
- Give `data.frame` any number of vectors, each separated with a comma.
- Each vector should be set equal to a name that describes the vector. `data.frame` will turn each vector into a column of the new data frame.

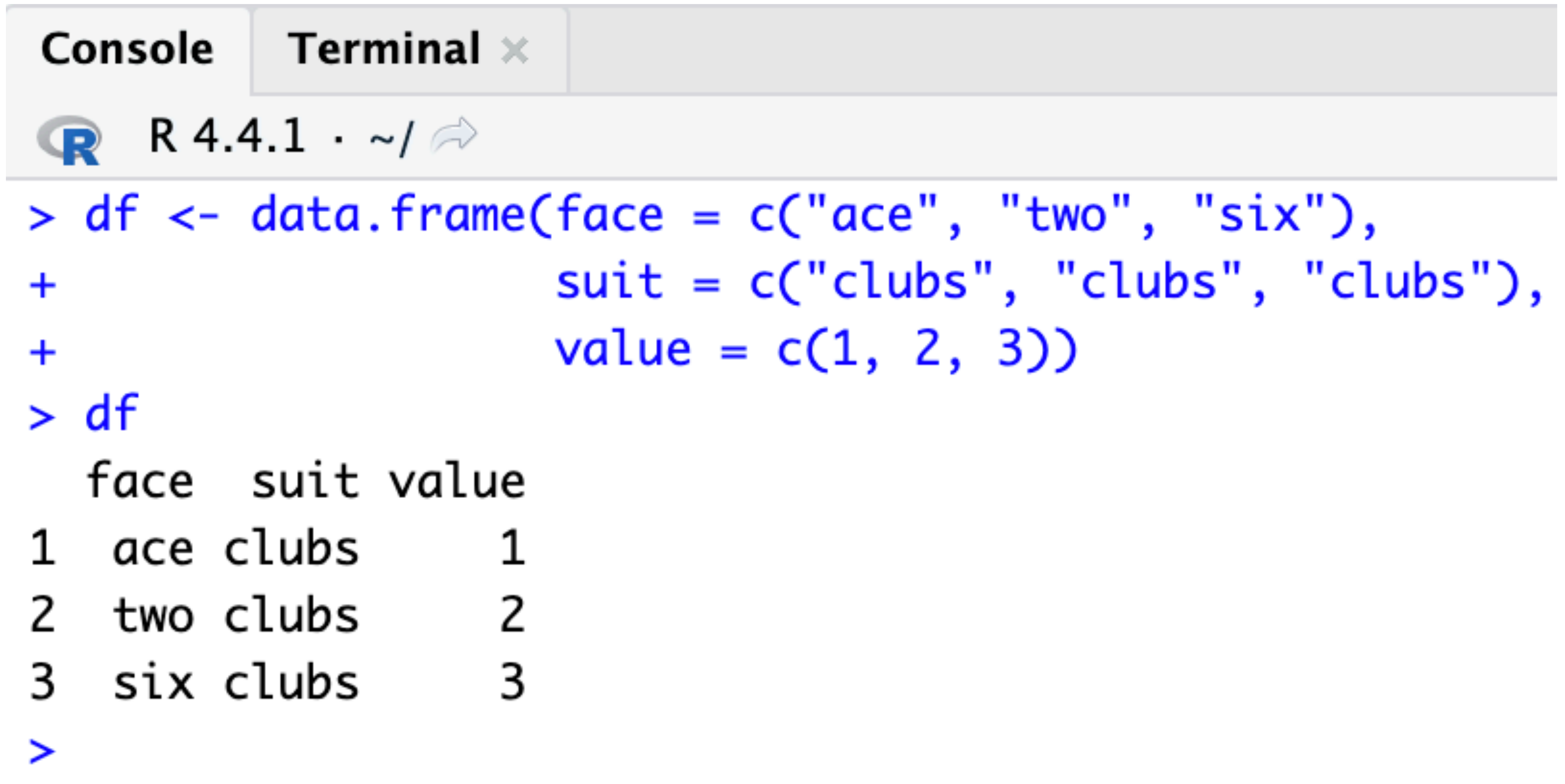


```
> df <- data.frame(face = c("ace", "two", "six"),
+                  suit = c("clubs", "clubs", "clubs"),
+                  value = c(1, 2, 3))
> df
  face  suit value
1  ace clubs     1
2  two clubs     2
3  six clubs     3
>
```

R Objects

DataFrames

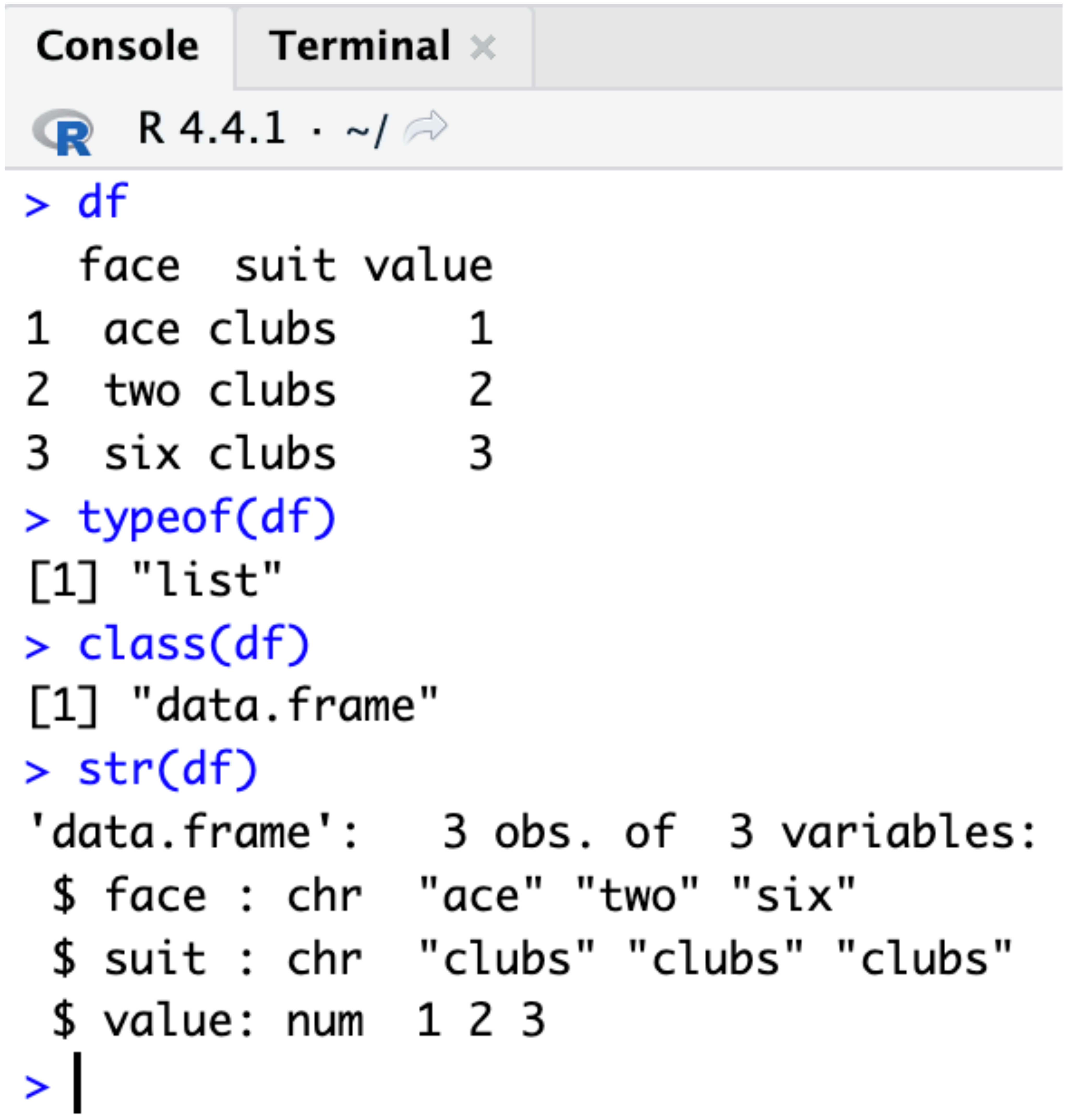
- You'll need to make sure that each vector is the same length.
- In the previous code, I named the arguments in `data.frame` `face`, `suit`, and `value`, but you can name the arguments whatever you like.
- `data.frame` will use your argument names to label the columns of the data frame.



```
> df <- data.frame(face = c("ace", "two", "six"),
+                  suit = c("clubs", "clubs", "clubs"),
+                  value = c(1, 2, 3))
> df
  face  suit value
1  ace clubs     1
2  two clubs     2
3  six clubs     3
>
```

R Objects
DataFrames

- If you look at the type of a data frame, you will see that it is a list.
- In fact, each data frame is a list with class `data.frame`.
- You can see what types of objects are grouped together by a list with the `str` function.

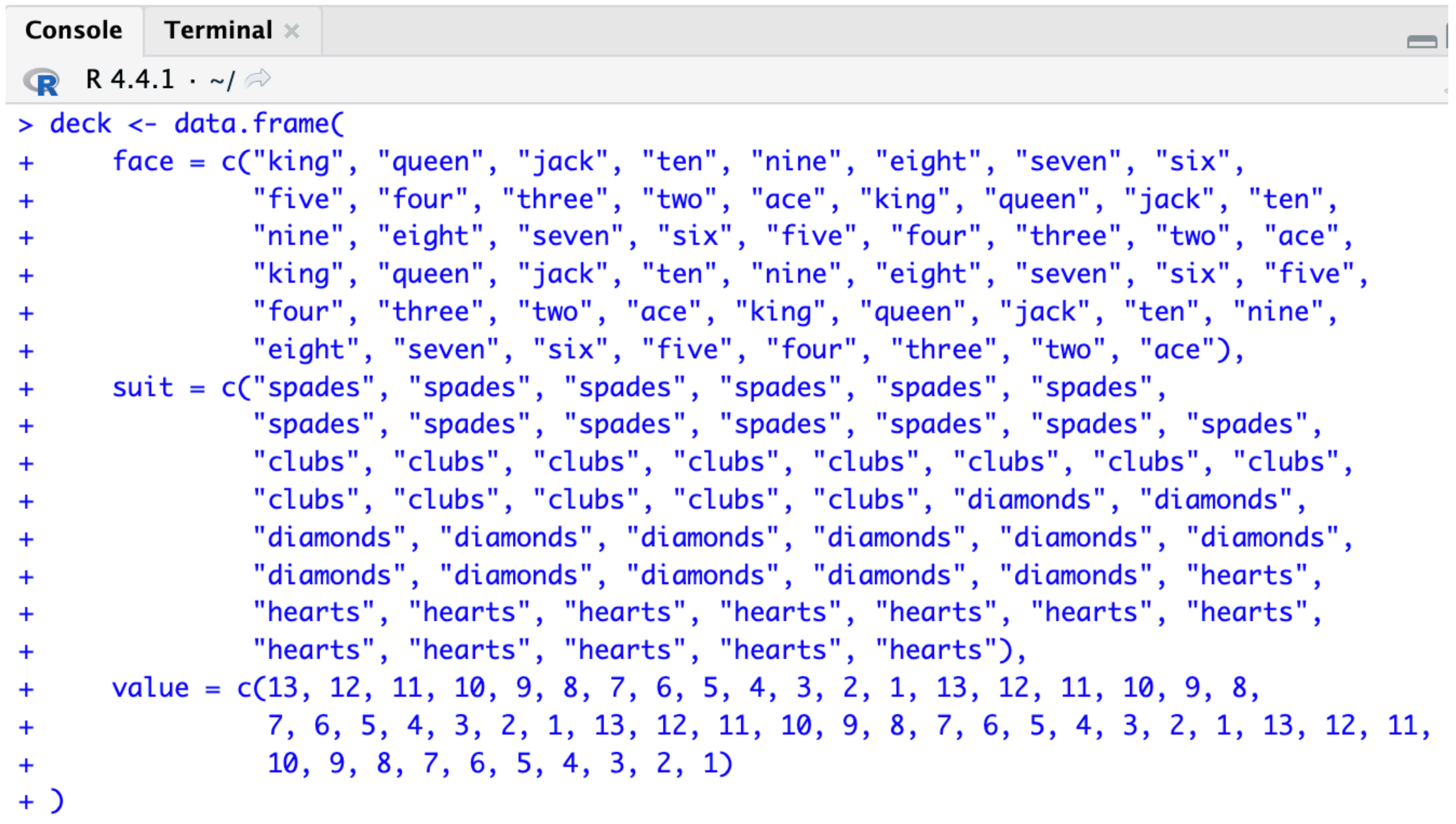


```
> df
  face  suit value
1  ace clubs     1
2  two clubs     2
3  six clubs     3
> typeof(df)
[1] "list"
> class(df)
[1] "data.frame"
> str(df)
'data.frame':	3 obs. of  3 variables:
 $ face : chr  "ace" "two" "six"
 $ suit : chr  "clubs" "clubs" "clubs"
 $ value: num  1 2 3
> 
```

R Objects

DataFrames

- A data frame is a great way to build an entire deck of cards.
- You can make each row in the data frame a playing card, and each column a type of value—each with its own appropriate data type.
- You could create this data frame with `data.frame`, but look at the typing involved! You need to write three vectors, each with 52 elements.

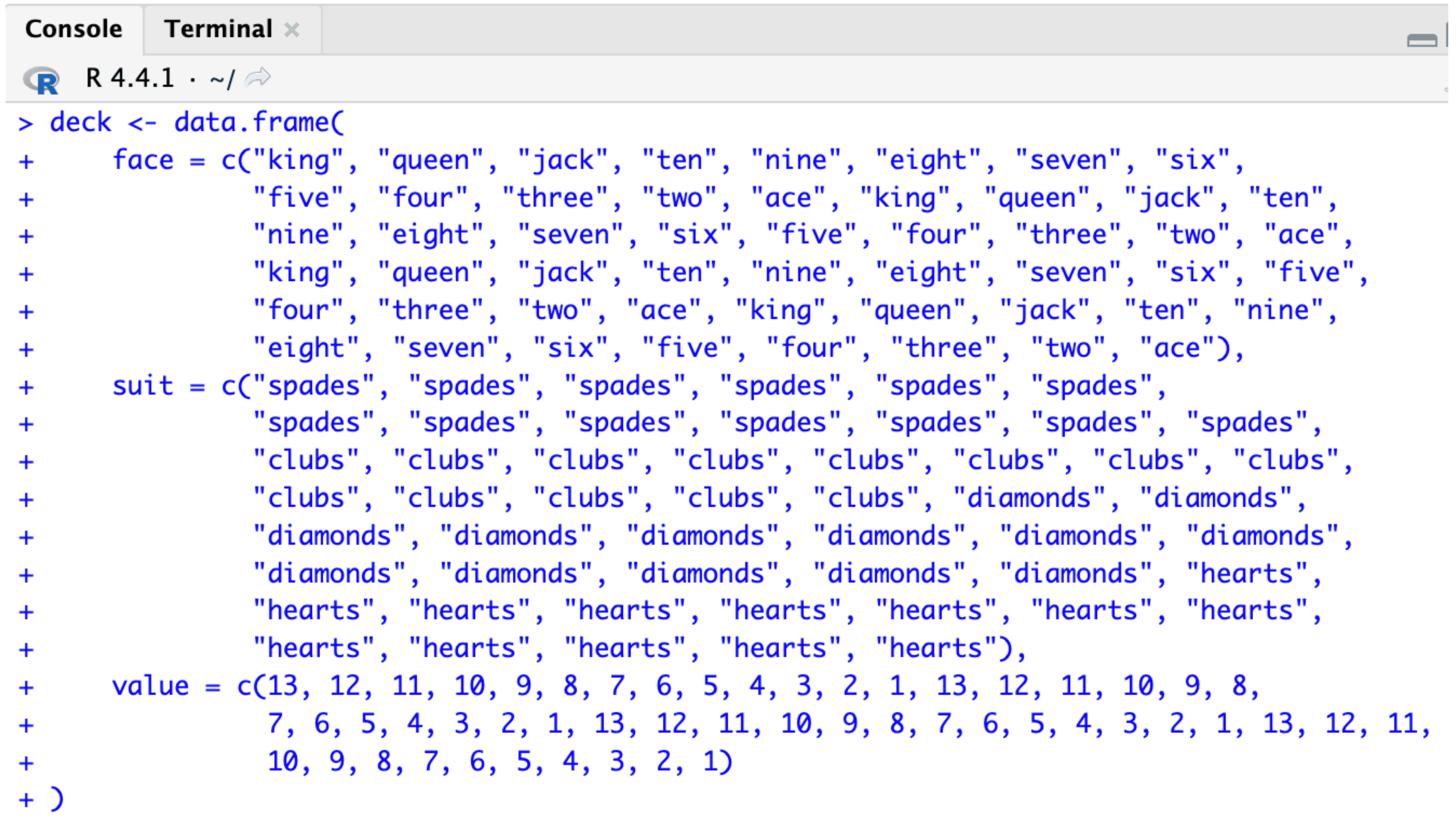


```
> deck <- data.frame(
+     face = c("king", "queen", "jack", "ten", "nine", "eight", "seven", "six",
+              "five", "four", "three", "two", "ace", "king", "queen", "jack", "ten",
+              "nine", "eight", "seven", "six", "five", "four", "three", "two", "ace",
+              "king", "queen", "jack", "ten", "nine", "eight", "seven", "six", "five",
+              "four", "three", "two", "ace", "king", "queen", "jack", "ten", "nine",
+              "eight", "seven", "six", "five", "four", "three", "two", "ace"),
+     suit = c("spades", "spades", "spades", "spades", "spades", "spades",
+              "spades", "spades", "spades", "spades", "spades", "spades", "spades",
+              "clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs",
+              "clubs", "clubs", "clubs", "clubs", "clubs", "diamonds", "diamonds",
+              "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "diamonds",
+              "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "hearts",
+              "hearts", "hearts", "hearts", "hearts", "hearts", "hearts", "hearts",
+              "hearts", "hearts", "hearts", "hearts", "hearts"),
+     value = c(13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8,
+               7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11,
+               10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
+ )
```

R Objects

Loading data

- You should avoid typing large data sets in by hand whenever possible.
- Typing invites typos and errors.
- It is always better to acquire large data sets as a computer file.
- You can then ask R to read the file and store the contents as an object.
- I'll send you a file that contains a data frame of playing-card information, so don't worry about typing in the code.

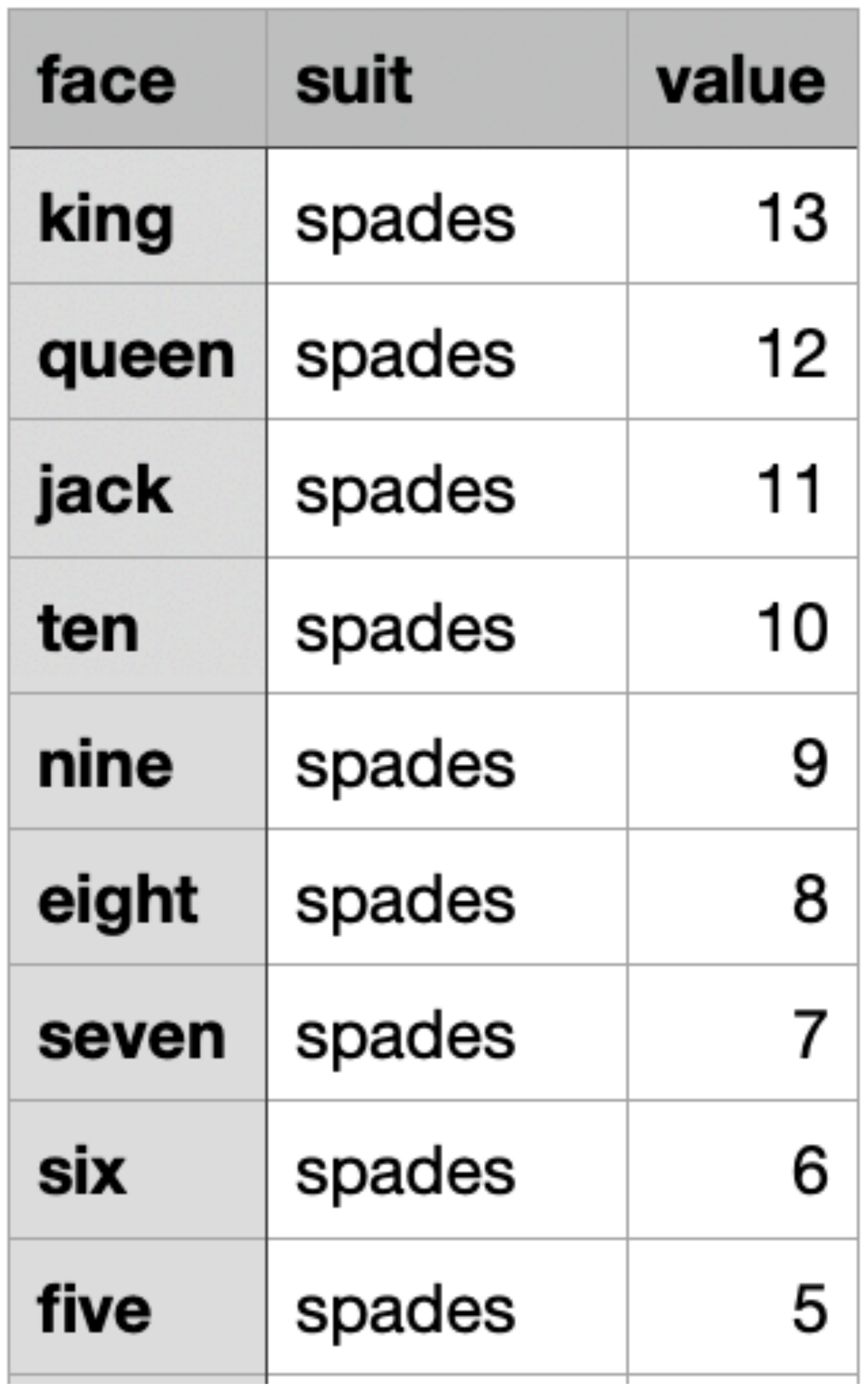


```
> deck <- data.frame(
+     face = c("king", "queen", "jack", "ten", "nine", "eight", "seven", "six",
+              "five", "four", "three", "two", "ace", "king", "queen", "jack", "ten",
+              "nine", "eight", "seven", "six", "five", "four", "three", "two", "ace",
+              "king", "queen", "jack", "ten", "nine", "eight", "seven", "six", "five",
+              "four", "three", "two", "ace", "king", "queen", "jack", "ten", "nine",
+              "eight", "seven", "six", "five", "four", "three", "two", "ace"),
+     suit = c("spades", "spades", "spades", "spades", "spades", "spades",
+              "spades", "spades", "spades", "spades", "spades", "spades", "spades",
+              "clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs",
+              "clubs", "clubs", "clubs", "clubs", "clubs", "diamonds", "diamonds",
+              "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "diamonds",
+              "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "hearts",
+              "hearts", "hearts", "hearts", "hearts", "hearts", "hearts", "hearts",
+              "hearts", "hearts", "hearts", "hearts", "hearts"),
+     value = c(13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8,
+               7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11,
+               10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
+ )
```

R Objects
Loading data

- You can load the `deck` data frame from the file Data on the page course.
- *deck.csv* is a comma-separated values file, or CSV for short.
- CSVs are plain-text files, which means you can open them in a text editor.

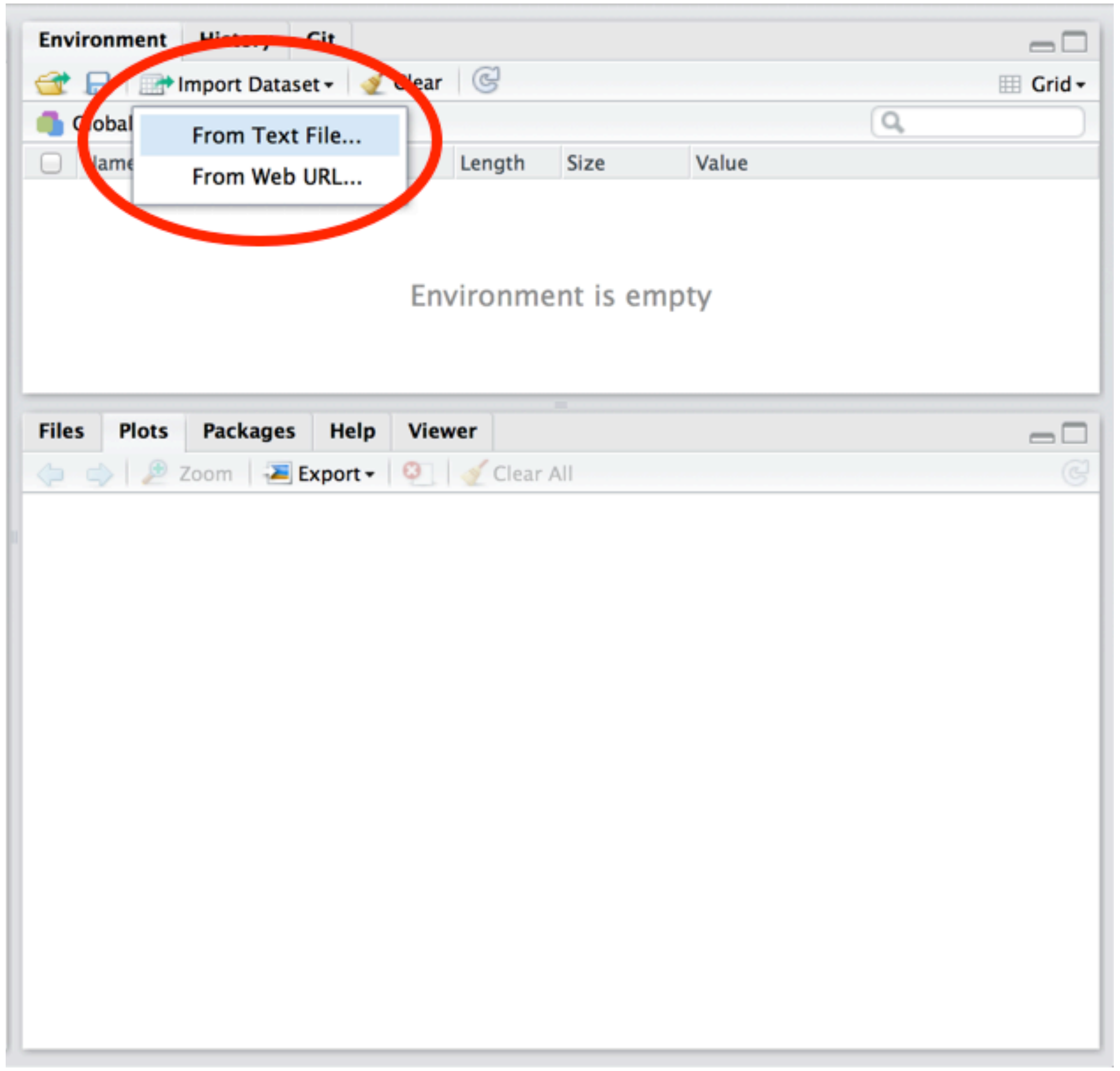
deck



face	suit	value
king	spades	13
queen	spades	12
jack	spades	11
ten	spades	10
nine	spades	9
eight	spades	8
seven	spades	7
six	spades	6
five	spades	5

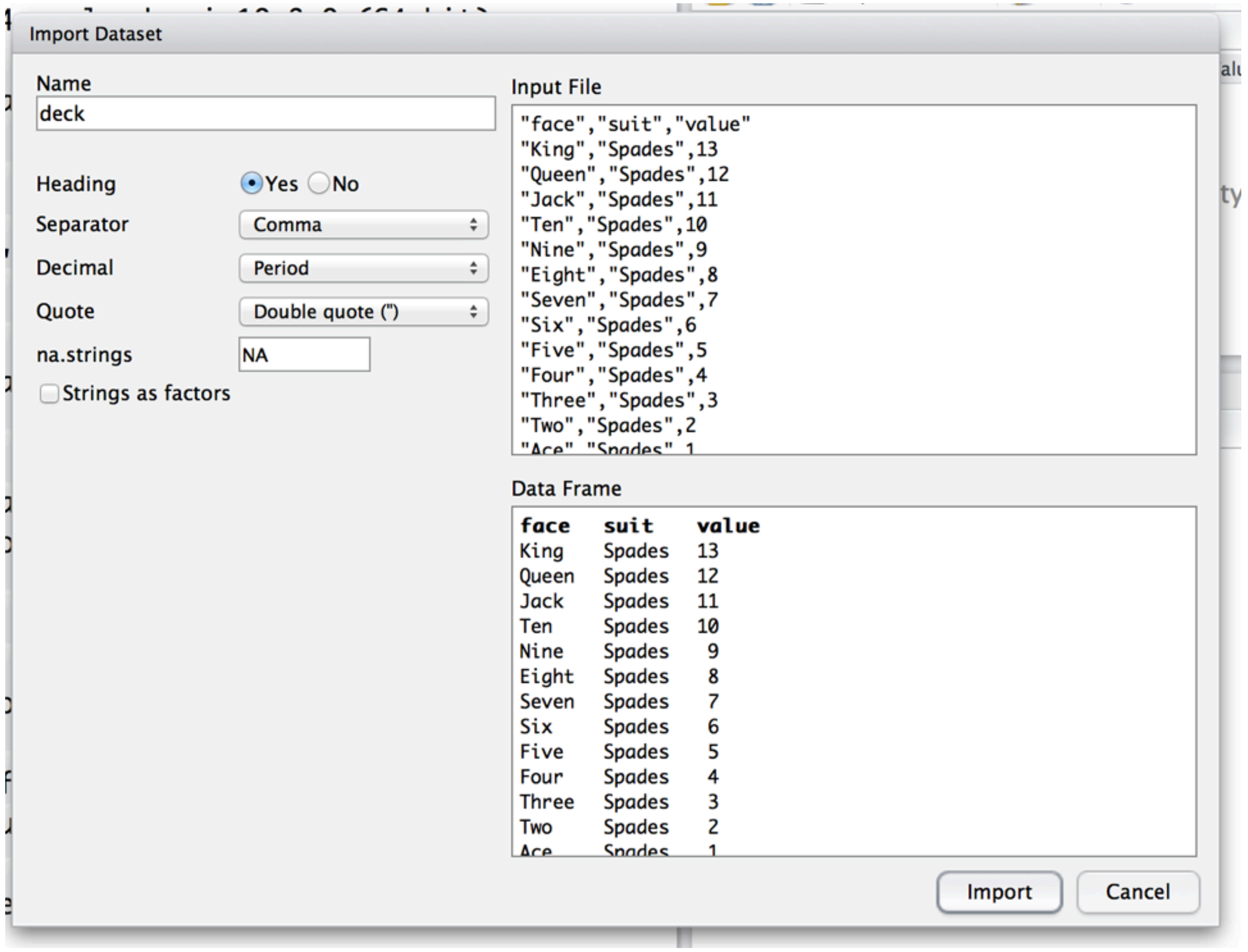
R Objects
Loading data

- To load a plain-text file into R, click the Import Dataset icon in RStudio
- RStudio will ask you to select the file you want to import, then it will open a wizard to help you import the data
- Use the wizard to tell RStudio what name to give the data set.
- Tell RStudio which character the data set uses as a separator, which character represents decimals, whether the data set comes with a row of column names.



R Objects
Loading data

- To load a plain-text file into R, click the Import Dataset icon in RStudio
- RStudio will ask you to select the file you want to import, then it will open a wizard to help you import the data
- Use the wizard to tell RStudio what name to give the data set.
- Tell RStudio which character the data set uses as a separator, which character represents decimals, whether the data set comes with a row of column names.



Name	deck
Heading	[✓ Yes] No
Separator	Comma
Decimal	Period
Quote	Double quote (")
na.strings	NA
Strings as factors	☐

Input File

```
"face","suit","value"
"King","Spades",13
"Queen","Spades",12
"Jack","Spades",11
"Ten","Spades",10
"Nine","Spades",9
"Eight","Spades",8
"Seven","Spades",7
"Six","Spades",6
"Five","Spades",5
"Four","Spades",4
"Three","Spades",3
"Two","Spades",2
"Ace","Spades",1
```

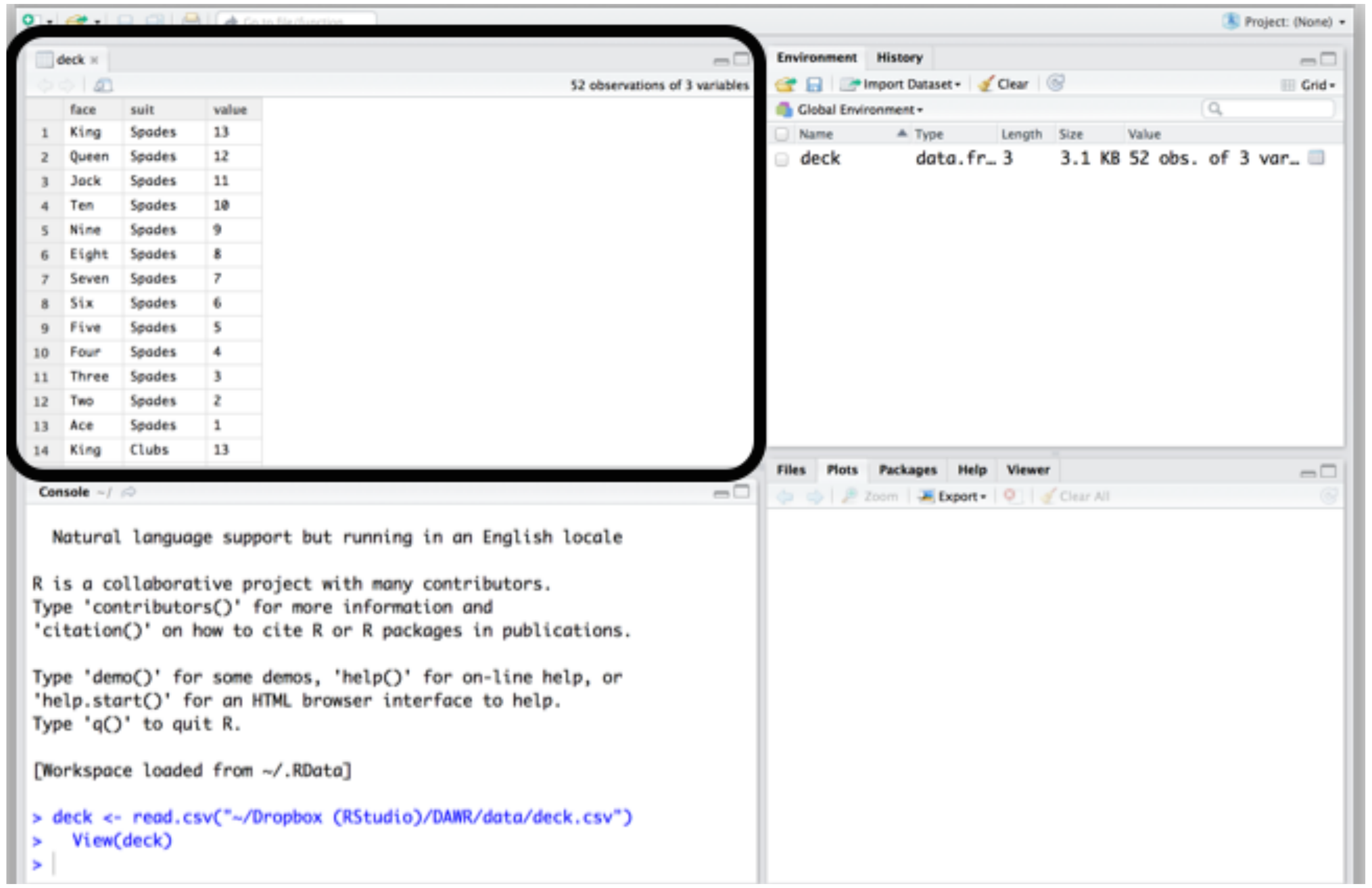
Data Frame

```
face    suit    value
King    Spades   13
Queen   Spades   12
Jack    Spades   11
Ten     Spades   10
Nine    Spades    9
Eight   Spades    8
Seven   Spades    7
Six     Spades    6
Five    Spades    5
Four    Spades    4
Three   Spades    3
Two     Spades    2
Ace     Spades    1
```

R Objects

Loading data

- RStudio will read in the data and save it to a data frame.
- RStudio will also open a data viewer, so you can see your new data in a spreadsheet format.
- If all worked well, your file should appear in a View tab of RStudio.



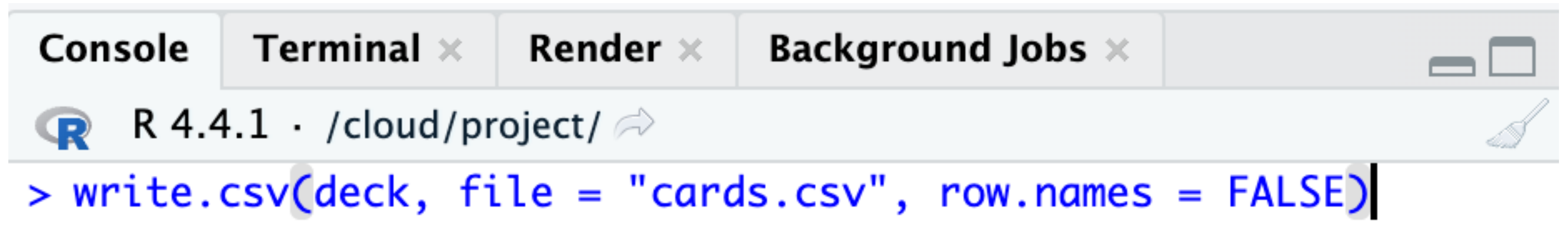
	face	suit	value
1	King	Spades	13
2	Queen	Spades	12
3	Jack	Spades	11
4	Ten	Spades	10
5	Nine	Spades	9
6	Eight	Spades	8
7	Seven	Spades	7
8	Six	Spades	6
9	Five	Spades	5
10	Four	Spades	4
11	Three	Spades	3
12	Two	Spades	2
13	Ace	Spades	1
14	King	Clubs	13

Name	Type	Length	Size	Value
deck	data.fr…	3	3.1 KB	52 obs. of 3 var…

```
 Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> deck <- read.csv("~/Dropbox (RStudio)/DAWR/data/deck.csv")
>   View(deck)
> 
```

R Objects
Saving data

- Before we go any further, let's save a copy of `deck` as a new *.csv* file.
- That way you can email it to a colleague, store it on a thumb drive, or open it in a different program.
- You can save any data frame in R to a *.csv* file with the command `write.csv`

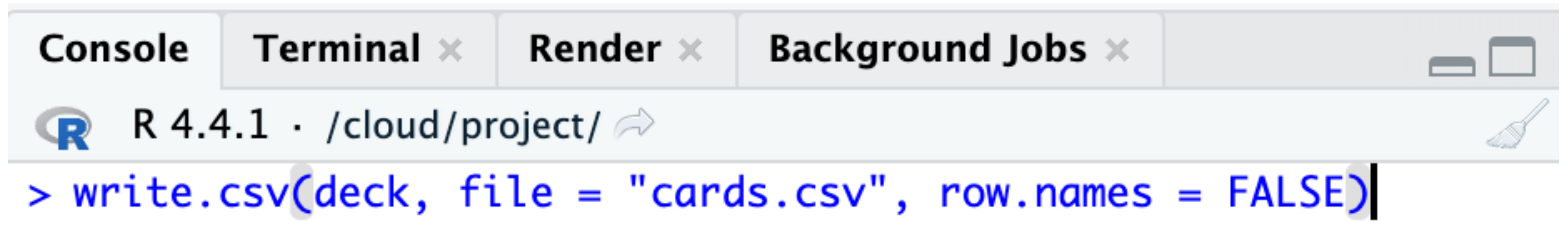


```
> write.csv(deck, file = "cards.csv", row.names = FALSE)
```

R Objects

Saving data

- To see where your working directory is, run `getwd()`
- To change the location of your working directory, visit Session > Set Working Directory > Choose Directory in the RStudio menu bar.
- You can customize the save process with `write.csv`’s large set of optional arguments (see `?write.csv` for details).

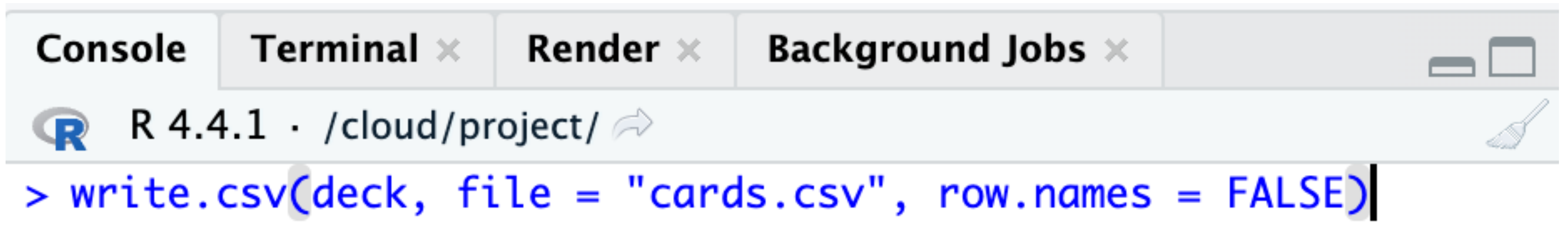


```
> write.csv(deck, file = "cards.csv", row.names = FALSE)
```

R Objects

Saving data

- there are three arguments that you should use *every* time you run `write.csv`.
- add the argument `row.names = FALSE`. This will prevent R from adding a column of numbers at the start of your data frame.
- You now have a virtual deck of cards to work with.



```
> write.csv(deck, file = "cards.csv", row.names = FALSE)
```