Data types and structures

EDH7916 | Spring 2020

Benjamin Skinner

R has a number of data types and structures. I do not expect you to memorize the technical differences between them all. As you become more familiar with R, however, you'll gain a working knowledge of their uses and differences well enough to meet your data analysis needs. Use this resource as reference in the meantime.

Types

Broadly, **type** refers to the way the data will be used — is it a number? a character? a Boolean (true/false)? While we don't give a second thought to understanding that `3` and `"3"` represent the same underlying construct, a computer will understand `3` and `"3"` as two very distinct things. We can add `3` and `"3"` to get `6`, but a computer will balk as if you had tried to add `3` and `"a"`. The good news is that R is only weakly or loosely typed, meaning that it will try to convert data types as necessary to guess what you mean and make your life easier. The price is slightly slower run times.

There are three primary data types in R that you will regularly use:
- `logical`
- `numeric` (`integer` & `double`)
- `character`

Logical

Logical vectors can be `TRUE`, `FALSE`, or `NA`. They can be assigned to objects or returned by logical operators (*e.g.*, `==`, `!=`, `<`, `>`, etc), which makes them useful for control flow in loops and functions.

NB In R, you can shorten `TRUE` to `T` and `FALSE` to `F`, but both the short and long versions must be capitalized.

```
## assignment
x <- TRUE
x
```

```
[1] TRUE
```

```
## ! == NOT
!x
```

```
[1] FALSE
```

```
## check: is it a logical type?
is.logical(x)
```

```
[1] TRUE
```

```
## evaluate (notice the double `==`)
1 + 1 == 2
```

```
[1] TRUE
```

Numeric: Integer and Double

Numeric values can be both **integers** and double precision floating point values, or just **doubles**. R automatically converts between the two data types for you, so knowing the difference between the two isn't really important for most analyses.

If you want to use an integer, place a capital `L` after the number like `1L`. If a number is stored as an integer, some R output will place an `L` behind the digits to let you know that. Mostly, R defaults to using doubles, but if you see a number with an `L` behind it, know that it's still a number.

```
## use 'L' after digit to store as integer
x <- 1L
is.integer(x)
```

```
[1] TRUE
```

```
## R stores as double by default
y <- 1
is.double(y)
```

```
[1] TRUE
```

```
## both are numeric
is.numeric(x)
```

```
[1] TRUE
```

```
is.numeric(y)
```

```
[1] TRUE
```

Character

Character values are stored as strings, which means you need to place either single `'` or double `"` quotes around them (stylistically, double quotes are usually preferred). Numeric values can also be stored as strings (sometimes useful if you must store leading zeroes), but they have to be converted back to numbers before you can perform numeric operations on them (like adding or subtracting) or use them in a statistical model.

```
## store a string using quotation marks
x <- "The quick brown fox jumps over the lazy dog."
x
```

```
[1] "The quick brown fox jumps over the lazy dog."
```

```
## store a number with leading zeros
x <- "00001"
x
```

```
[1] "00001"
```

> **Quick exercise** Try to add a string digit to a numeric value. What happens? Can you convert the string version on the fly so that the equation works? (HINT: in R, you can change a vector type using `as.<type>()`, where `<type>` is the name of what you want.)

Structures

Building on these data types, R relies on four primary data structures:

- `vector`
- `matrix`[1]

- list
- dataframe

Vector

A vector in R is just a collection of the data types discussed. In fact, a single value is a vector of one. Vectors do not have dimensions (`dim()`), but do have `length()`, which is good to remember when inspecting your data or writing loops and functions.

You combine multiple values using the concatenate, `c()`, function. We will use `c()` a lot.

```
## create vector
x <- 1

## check
is.vector(x)
```

```
[1] TRUE
```

```
## add to vector
## NB: can do so recursively meaning old x can help make new x
x <- c(x, 5, 8)
x
```

```
[1] 1 5 8
```

```
## no dim...
dim(x)
```

```
NULL
```

```
## ...but length
length(x)
```

```
[1] 3
```

Subsetting using indices: `[]` You can access the elements of a vector using brackets, `[]`, after the object name. If you think of each element in the vector as having an address, that is, a way to access it specifically, then its address is its position number in the vector. This position number is called its index, and in R, the index always starts with `1`.

In our current vector, we have three items, `1`, `5`, and `8`, which in turn have indices of `1`, `2`, and `3`. To access `5` specifically, we can call it using the brackets and its index: `x[2]`.

```
## get the second element
x[2]
```

```
[1] 5
```

> **Quick exercise** Since you know how to access a specific element in a vector and how to assign new values, try to change the 3rd element of the `x` vector to 4.

All values in a vector must be of the same type. If you concatenate values of different data types, R will automatically promote all values to least ambiguous type. We can check this with `class()`.

```
## check class of x
class(x)
```

```
[1] "numeric"
```

```
## add character
x <- c(x, "a")
x
```

```
[1] "1" "5" "8" "a"
```

```
## check class
class(x)
```

```
[1] "character"
```

Notice how R converted everything to a character? That's because R's internal rules say that for users, it makes more sense to convert a number to a string version of itself (`1` to `"1"`) than to figure out a number for a string that would make as much sense (`"a"` to `1` since it's the first letter in the alphabet? But what would `"A"` be then?).

Matrix

A matrix is a 2D arrangement of data types. Instead of length, it has dimensions. Like vectors, all data elements must be of the same type.

```
## create 3 x 3 matrix that is the sequence of numbers between 1 and 9
x <- matrix(1:9, nrow = 3, ncol = 3)
x
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
## ...fill by row this time
y <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
y
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

```
## a matrix has two dimensions
dim(x)
```

```
[1] 3 3
```

Use `nrow()` and `ncol()` to get the number of rows and columns, respectively.

```
## # of rows
nrow(x)
```

```
[1] 3
```

```
## # of columns
ncol(x)
```

```
[1] 3
```

Like a vector, you can access parts of a matrix. Since it has two dimensions, use a comma in the bracket to separate row indices from column indices.

When using brackets with objects that have two dimensions, a good rule of thumb is to add your comma first: `x[ , ]`. Numbers or objects you put between the first bracket and the comma will affect the rows; numbers between the comma and the closing bracket will affect the columns.

If you don't put anything in either of those spaces (a blank space doesn't count), R will assume you want all rows or columns, depending on which side of the comma is blank.

```
## show the values in the first row
x[1, ]
```

```
[1] 1 4 7
```

```
## show the values in the third column
x[ ,3]
```

```
[1] 7 8 9
```

```
## this is the same as just calling x by itself
x[ , ]
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Quick exercise Return the middle value of the `x` matrix. Next assign the middle value the character value 'a'. What happens to the rest of the values in the matrix?

List

Lists are a catch all objects that can hold an assortment of other objects of different data types. They can be flat, meaning that all values are at the same level, or nested, with lists holding other lists.

```
## create single-level list
x <- list(1, "a", TRUE)

## show
x
```

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

[[2]]
[1] "a"

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

```
## check
is.list(x)
```

```
[1] TRUE
```

```
## create blank list
y <- list()

## add to first list, creating nested list
z <- list(x, y)

## show
z
```

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

[[1]][[2]]
[1] "a"

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


[[2]]
list()
```

You access items in lists like you do vectors and matrices. You may, however, need to use double brackets, `[[]]`, and multiple pairs, `[[]][[]]`, to reach the item you need.

```
## the first item in list z is list x
z[[1]]
```

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

[[2]]
[1] "a"

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

```
## to get to "a" in list x, need to add more brackets
z[[1]][[2]]
```

```
[1] "a"
```

Lists can get pretty complex pretty quickly. Sometimes, the best approach to finding what you want is just trial and error!

Data frame

Data frames are really just an organized collection of lists / vectors that are the same length. That quick description, however, belies the importance of data frames: you will use them all the time in your data work.

Most of the time, you will be reading in data frames, but you can also create them.

```
## create data frame where col_* are the column (variable) names
df <- data.frame(col_a = c(1,2,3),
                 col_b = c(4,5,6),
                 col_c = c(7,8,9))

## show
df
```

```
  col_a col_b col_c
1     1     4     7
2     2     5     8
3     3     6     9
```

```
## check
is.data.frame(df)
```

```
[1] TRUE
```

Like matrices, data frames have a `dim()` and the number of rows and columns can be recovered using `nrow()` and `ncol()`. The column names, which are needed when estimating models and making graphics, are accessed using `names()`.

```
## get column names
names(df)
```

```
[1] "col_a" "col_b" "col_c"
```

To access a column, you need to give R the data frame's name followed by a `$` and then the variable name.

```
## get col_a
df$col_a
```

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

You can also use the `df[["<var name>"]]` construction, which comes in handy in loops and functions.

```
## get col_a (note the quotation marks this time)
df[["col_a"]]
```

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

> **Quick exercise** Create two or three equal length vectors. Next, combine to create a data frame. Finally, change one value in the data frame (HINT: think about how you changed vector and matrix values before).

Notes

1. R also supports arrays, which can take on more than two dimensions.