Data Structures

Overview

Teaching: 40 min
Exercises: 15 min
Questions

- How can I read data in R?
- What are the basic data types in R?
- How do I represent categorical information in R?

Objectives

- To be able to identify the 5 main data types.
- To begin exploring data frames, and understand how they are related to vectors, factors and lists.
- To be able to ask questions from R about the type, class, and structure of an object.

One of R's most powerful features is its ability to deal with tabular data - such as you may already have in a spreadsheet or a CSV file. Let's start by making a toy dataset in your `data/` directory, called `feline-data.csv`:

R

```
cats <- data.frame(coat = c("calico", "black", "tabby"),
                   weight = c(2.1, 5.0, 3.2),
                   likes_string = c(1, 0, 1))
```

We can now save `cats` as a CSV file. It is good practice to call the argument names explicitly so the function knows what default values you are changing. Here we are setting `row.names = FALSE`. Recall you can use `?write.csv` to pull up the help file to check out the argument names and their default values.

R

```
write.csv(x = cats, file = "data/feline-data.csv", row.names = FALSE)
```

The contents of the new file, `feline-data.csv`:

R

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

Tip: Editing Text files in R

Alternatively, you can create `data/feline-data.csv` using a text editor (Nano), or within RStudio with the **File -> New File -> Text File** menu item.

We can load this into R via the following:

R

```
cats <- read.csv(file = "data/feline-data.csv", stringsAsFactors = TRUE)
cats
```

Output

```
    coat weight likes_string
1 calico    2.1            1
2  black    5.0            0
3  tabby    3.2            1
```

The `read.table` function is used for reading in tabular data stored in a text file where the columns of data are separated by punctuation characters such as CSV files (csv = comma-separated values). Tabs and commas are the most common punctuation characters used to separate or delimit data points in csv files. For convenience R provides 2 other versions of `read.table`. These are: `read.csv` for files where the data are separated with commas and `read.delim` for files where the data are separated with tabs. Of these three functions `read.csv` is the most commonly used. If needed it is possible to override the default delimiting punctuation marks for both `read.csv` and `read.delim`.

We can begin exploring our dataset right away, pulling out columns by specifying them using the `$` operator:

R

```
cats$weight
```

Output

```
[1] 2.1 5.0 3.2
```

R

```
cats$coat
```

Output

```
[1] calico black  tabby
Levels: black calico tabby
```

We can do other operations on the columns:

R

```
## Say we discovered that the scale weighs two Kg light:
cats$weight + 2
```

Output

```
[1] 4.1 7.0 5.2
```

R

```
paste("My cat is", cats$coat)
```

Output

```
[1] "My cat is calico" "My cat is black"  "My cat is tabby"
```

But what about

R

```
cats$weight + cats$coat
```

Warning

```
Warning in Ops.factor(cats$weight, cats$coat): '+' not meaningful for factors
```

Output

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

Understanding what happened here is key to successfully analyzing data in R.

Data Types

If you guessed that the last command will return an error because `2.1` plus `"black"` is nonsense, you're right - and you already have some intuition for an important concept in programming called *data types*. We can ask what type of data something is:

R

```
typeof(cats$weight)
```

Output

```
[1] "double"
```

There are 5 main types: `double`, `integer`, `complex`, `logical` and `character`.

R

```
typeof(3.14)
```

Output

```
[1] "double"
```

R

```
typeof(1L) # The L suffix forces the number to be an integer, since by default R uses float numbers
```

Output

```
[1] "integer"
```

R

```
typeof(1+1i)
```

Output

```
[1] "complex"
```

R

```
typeof(TRUE)
```

Output

```
[1] "logical"
```

R

```
typeof('banana')
```

Output

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

No matter how complicated our analyses become, all data in R is interpreted as one of these basic data types. This strictness has some really important consequences.

A user has added details of another cat. This information is in the file `data/feline-data_v2.csv`.

R

```
file.show("data/feline-data_v2.csv")
```

R

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
tabby,2.3 or 2.4,1
```

Load the new cats data like before, and check what type of data we find in the `weight` column:

R

```
cats <- read.csv(file="data/feline-data_v2.csv", stringsAsFactors = TRUE)
typeof(cats$weight)
```

Output

```
[1] "integer"
```

Oh no, our weights aren't the double type anymore! If we try to do the same math we did on them before, we run into trouble:

R

```
cats$weight + 2
```

Warning

```
Warning in Ops.factor(cats$weight, 2): '+' not meaningful for factors
```

Output

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

What happened? The `cats` data we are working with is something called a *data frame*. Data frames are one of the most common and versatile types of *data structures* we will work with in R. In this example, the columns that make up the data frame cannot be composed of different data types. In this case, R does not read everything in the data frame as a *double*, therefore the entire column data type changes to something that is suitable for everything in the column.

When R reads a csv file, it reads it in as a *data frame*. Thus, when we loaded the `cats` csv file, it is stored as a data frame. We can check this by using the function `class()`.

R

```
class(cats)
```

Output

```
[1] "data.frame"
```

Data frames are composed of rows and columns, where each column has the same number of rows. Different columns in a data frame can be made up of different data types (this is what makes them so versatile), but everything in a given column needs to be the same type (e.g., vector, factor, or list).

Let's explore more about different data structures and how they behave. For now, let's remove that extra line from our cats data and reload it, while we investigate this behavior further:

feline-data.csv:

Code

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

And back in RStudio:

R

```
cats <- read.csv(file="data/feline-data.csv", stringsAsFactors = TRUE)
```

Vectors and Type Coercion

To better understand this behavior, let's meet another of the data structures: the *vector*.

R

```
my_vector <- vector(length = 3)
my_vector
```

Output

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

A vector in R is essentially an ordered list of things, with the special condition that *everything in the vector must be the same basic data type*. If you don't choose the datatype, it'll default to `logical`; or, you can declare an empty vector of whatever type you like.

R

```
another_vector <- vector(mode='character', length=3)
another_vector
```

Output

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

You can check if something is a vector:

R

```
str(another_vector)
```

Output

```
 chr [1:3] "" "" ""
```

The somewhat cryptic output from this command indicates the basic data type found in this vector - in this case `chr`, character; an indication of the number of things in the vector - actually, the indexes of the vector, in this case `[1:3]`; and a few examples of what's actually in the vector - in this case empty character strings. If we similarly do

R

```
str(cats$weight)
```

Output

```
 num [1:3] 2.1 5 3.2
```

we see that `cats$weight` is a vector, too - *the columns of data we load into R data.frames are all vectors*, and that's the root of why R forces everything in a column to be the same basic data type.

Discussion 1

Why is R so opinionated about what we put in our columns of data? How does this help us?

Discussion 1

You can also make vectors with explicit contents with the combine function:

R

```
combine_vector <- c(2,6,3)
combine_vector
```

Output

```
[1] 2 6 3
```

Given what we've learned so far, what do you think the following will produce?

R

```
quiz_vector <- c(2,6,'3')
```

This is something called *type coercion*, and it is the source of many surprises and the reason why we need to be aware of the basic data types and how R will interpret them. When R encounters a mix of types (here numeric and character) to be combined into a single vector, it will force them all to be the same type. Consider:

R

```
coercion_vector <- c('a', TRUE)
coercion_vector
```

Output

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

R

```
another_coercion_vector <- c(0, TRUE)
another_coercion_vector
```

Output

```
[1] 0 1
```

The coercion rules go: `logical -> integer -> numeric -> complex -> character`, where -> can be read as *are transformed into*. You can try to force coercion against this flow using the `as.` functions:

R

```
character_vector_example <- c('0','2','4')
character_vector_example
```

Output

```
[1] "0" "2" "4"
```

R

```
character_coerced_to_numeric <- as.numeric(character_vector_example)
character_coerced_to_numeric
```

Output

```
[1] 0 2 4
```

R

```
numeric_coerced_to_logical <- as.logical(character_coerced_to_numeric)
numeric_coerced_to_logical
```

Output

```
[1] FALSE  TRUE  TRUE
```

As you can see, some surprising things can happen when R forces one basic data type into another! Nitty-gritty of type coercion aside, the point is: if your data doesn't look like what you thought it was going to look like, type coercion may well be to blame; make sure everything is the same type in your vectors and your columns of data.frames, or you will get nasty surprises!

But coercion can also be very useful! For example, in our `cats` data `likes_string` is numeric, but we know that the 1s and 0s actually represent `TRUE` and `FALSE` (a common way of representing them). We should use the `logical` datatype here, which has two states: `TRUE` or `FALSE`, which is exactly what our data represents. We can 'coerce' this column to be `logical` by using the `as.logical` function:

R

```
cats$likes_string
```

Output

```
[1] 1 0 1
```

R

```
cats$likes_string <- as.logical(cats$likes_string)
cats$likes_string
```

Output

```
[1]  TRUE FALSE  TRUE
```

The combine function, `c()`, will also append things to an existing vector:

R

```
ab_vector <- c('a', 'b')
ab_vector
```

Output

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

R

```
combine_example <- c(ab_vector, 'SWC')
combine_example
```

Output

```
[1] "a"   "b"   "SWC"
```

You can also make series of numbers:

R

```
mySeries <- 1:10
mySeries
```

Output

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

R

```
seq(10)
```

Output

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

R

```
seq(1,10, by=0.1)
```

Output

```
 [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3  2.4
[16]  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9
[31]  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1  5.2  5.3  5.4
[46]  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9
[61]  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9  8.0  8.1  8.2  8.3  8.4
[76]  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3  9.4  9.5  9.6  9.7  9.8  9.9
[91] 10.0
```

We can ask a few questions about vectors:

R

```
sequence_example <- seq(10)
head(sequence_example, n=2)
```

Output

```
[1] 1 2
```

R

```
tail(sequence_example, n=4)
```

Output

```
[1]  7  8  9 10
```

R

```
length(sequence_example)
```

Output

```
[1] 10
```

R

```
class(sequence_example)
```

Output

```
[1] "integer"
```

R

```
typeof(sequence_example)
```

Output

```
[1] "integer"
```

Finally, you can give names to elements in your vector:

R

```
my_example <- 5:8
names(my_example) <- c("a", "b", "c", "d")
my_example
```

Output

```
a b c d
5 6 7 8
```

R

```
names(my_example)
```

Output

```
[1] "a" "b" "c" "d"
```

Challenge 1

Start by making a vector with the numbers 1 through 26. Multiply the vector by 2, and give the resulting vector names A through Z (hint: there is a built in vector called `LETTERS`)

Solution to Challenge 1

R

```
x <- 1:26
x <- x * 2
names(x) <- LETTERS
```

Data Frames

We said that columns in data.frames were vectors:

R

```
str(cats$weight)
```

Output

```
 num [1:3] 2.1 5 3.2
```

R

```
str(cats$likes_string)
```

Output

```
 logi [1:3] TRUE FALSE TRUE
```

These make sense. But what about

R

```
str(cats$coat)
```

Output

```
 Factor w/ 3 levels "black","calico",..: 2 1 3
```

📌 Tip: Renaming data frame columns

Data frames have column names, which can be accessed with the `names()` function.

R

```
names(cats)
```

Output

```
[1] "coat"         "weight"       "likes_string"
```

If you want to rename the second column of `cats`, you can assign a new name to the second element of `names(cats)`.

R

```
names(cats)[2] <- "weight_kg"
cats
```

Output

```
    coat weight_kg likes_string
1 calico       2.1         TRUE
2  black       5.0        FALSE
3  tabby       3.2         TRUE
```

Factors

Another important data structure is called a *factor*. Factors usually look like character data, but are typically used to represent categorical information. For example, let's make a vector of strings labelling cat colorations for all the cats in our study:

R

```
coats <- c('tabby', 'tortoiseshell', 'tortoiseshell', 'black', 'tabby')
coats
```

Output

```
[1] "tabby"         "tortoiseshell" "tortoiseshell" "black"
[5] "tabby"
```

R

```
str(coats)
```

Output

```
 chr [1:5] "tabby" "tortoiseshell" "tortoiseshell" "black" "tabby"
```

We can turn a vector into a factor like so:

R

```
CATegories <- factor(coats)
class(CATegories)
```

Output

```
[1] "factor"
```

R

```
str(CATegories)
```

Output

```
 Factor w/ 3 levels "black","tabby",..: 2 3 3 1 2
```

Now R has noticed that there are three possible categories in our data - but it also did something surprising; instead of printing out the strings we gave it, we got a bunch of numbers instead. R has replaced our human-readable categories with numbered indices under the hood, this is necessary as many statistical calculations utilise such numerical representations for categorical data:

R

```
typeof(coats)
```

Output

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

R

```
typeof(CATegories)
```

Output

```
[1] "integer"
```

Challenge 2

Is there a factor in our `cats` data.frame? what is its name? Try using `?read.csv` to figure out how to keep text columns as character vectors instead of factors; then write a command or two to show that the factor in `cats` is actually a character vector when loaded in this way.

Solution to Challenge 2

One solution is use the argument `stringAsFactors` :

R

```
cats <- read.csv(file="data/feline-data.csv", stringsAsFactors=FALSE)
str(cats$coat)
```

Another solution is use the argument `colClasses` that allow finer control.

R

```
cats <- read.csv(file="data/feline-data.csv", colClasses=c(NA, NA, "character"))
str(cats$coat)
```

Note: new students find the help files difficult to understand; make sure to let them know that this is typical, and encourage them to take their best guess based on semantic meaning, even if they aren't sure.

In modelling functions, it's important to know what the baseline levels are. This is assumed to be the first factor, but by default factors are labelled in alphabetical order. You can change this by specifying the levels:

R

```
mydata <- c("case", "control", "control", "case")
factor_ordering_example <- factor(mydata, levels = c("control", "case"))
str(factor_ordering_example)
```

Output

```
 Factor w/ 2 levels "control","case": 2 1 1 2
```

In this case, we've explicitly told R that "control" should be represented by 1, and "case" by 2. This designation can be very important for interpreting the results of statistical models!

Lists

Another data structure you'll want in your bag of tricks is the `list` . A list is simpler in some ways than the other types, because you can put anything you want in it. Remember *everything in the vector must be the same basic data type*, but a list can have different data types:

R

```
list_example <- list(1, "a", TRUE, 1+4i)
list_example
```

Output

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

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

R

```
another_list <- list(title = "Numbers", numbers = 1:10, data = TRUE )
another_list
```

Output

```
$title
[1] "Numbers"

$numbers
 [1]  1  2  3  4  5  6  7  8  9 10

$data
[1] TRUE
```

We can now understand something a bit surprising in our data.frame; what happens if we run:

R

```
typeof(cats)
```

Output

```
[1] "list"
```

We see that data.frames look like lists 'under the hood' - this is because a data.frame is really a list of vectors and factors, as they have to be - in order to hold those columns that are a mix of vectors and factors, the data.frame needs something a bit more flexible than a vector to put all the columns together into a familiar table. In other words, a `data.frame` is a special list in which all the vectors must have the same length.

In our `cats` example, we have an integer, a double and a logical variable. As we have seen already, each column of data.frame is a vector.

R

```
cats$coat
```

Output

```
[1] calico black  tabby
Levels: black calico tabby
```

R

```
cats[,1]
```

Output

```
[1] calico black  tabby
Levels: black calico tabby
```

R

```
typeof(cats[,1])
```

Output

```
[1] "integer"
```

R

```
str(cats[,1])
```

Output

```
 Factor w/ 3 levels "black","calico",..: 2 1 3
```

Each row is an *observation* of different variables, itself a data.frame, and thus can be composed of elements of different types.

R

```
cats[1,]
```

Output

```
    coat weight likes_string
1 calico    2.1            1
```

R

```
typeof(cats[1,])
```

Output

```
[1] "list"
```

R

```
str(cats[1,])
```

Output

```
'data.frame':   1 obs. of  3 variables:
 $ coat        : Factor w/ 3 levels "black","calico",..: 2
 $ weight      : num 2.1
 $ likes_string: num 1
```

Challenge 3

There are several subtly different ways to call variables, observations and elements from data.frames:

- `cats[1]`
- `cats[[1]]`
- `cats$coat`
- `cats["coat"]`
- `cats[1, 1]`
- `cats[, 1]`
- `cats[1, ]`

Try out these examples and explain what is returned by each one.

Hint: Use the function `typeof()` to examine what is returned in each case.

Solution to Challenge 3

Matrices

Last but not least is the matrix. We can declare a matrix full of zeros:

R

```
matrix_example <- matrix(0, ncol=6, nrow=3)
matrix_example
```

Output

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    0    0    0    0    0    0
[3,]    0    0    0    0    0    0
```

And similar to other data structures, we can ask things about our matrix:

R

```
class(matrix_example)
```

Output

```
[1] "matrix" "array"
```

R

```
typeof(matrix_example)
```

Output

```
[1] "double"
```

R

```
str(matrix_example)
```

Output

```
 num [1:3, 1:6] 0 0 0 0 0 0 0 0 0 0 ...
```

R

```
dim(matrix_example)
```

Output

```
[1] 3 6
```

R

```
nrow(matrix_example)
```

Output

```
[1] 3
```

R

```
ncol(matrix_example)
```

Output

```
[1] 6
```

Challenge 4

What do you think will be the result of `length(matrix_example)` ? Try it. Were you right? Why / why not?

Solution to Challenge 4

What do you think will be the result of `length(matrix_example)` ?

R

```
matrix_example <- matrix(0, ncol=6, nrow=3)
length(matrix_example)
```

Output

```
[1] 18
```

Because a matrix is a vector with added dimension attributes, `length` gives you the total number of elements in the matrix.

Challenge 5

Make another matrix, this time containing the numbers 1:50, with 5 columns and 10 rows. Did the `matrix` function fill your matrix by column, or by row, as its default behaviour? See if you can figure out how to change this. (hint: read the documentation for `matrix` !)

Solution to Challenge 5

Make another matrix, this time containing the numbers 1:50, with 5 columns and 10 rows. Did the `matrix` function fill your matrix by column, or by row, as its default behaviour? See if you can figure out how to change this. (hint: read the documentation for `matrix` !)

R

```
x <- matrix(1:50, ncol=5, nrow=10)
x <- matrix(1:50, ncol=5, nrow=10, byrow = TRUE) # to fill by row
```

Challenge 6

Create a list of length two containing a character vector for each of the sections in this part of the workshop:

- Data types
- Data structures

Populate each character vector with the names of the data types and data structures we've seen so far.

Solution to Challenge 6

R

```
dataTypes <- c('double', 'complex', 'integer', 'character', 'logical')
dataStructures <- c('data.frame', 'vector', 'factor', 'list', 'matrix')
answer <- list(dataTypes, dataStructures)
```

Note: it's nice to make a list in big writing on the board or taped to the wall listing all of these types and structures - leave it up for the rest of the workshop to remind people of the importance of these basics.

Challenge 7

Consider the R output of the matrix below:

Output

```
     [,1] [,2]
[1,]    4    1
[2,]    9    5
[3,]   10    7
```

What was the correct command used to write this matrix? Examine each command and try to figure out the correct one before typing them. Think about what matrices the other commands will produce.

1. `matrix(c(4, 1, 9, 5, 10, 7), nrow = 3)`
2. `matrix(c(4, 9, 10, 1, 5, 7), ncol = 2, byrow = TRUE)`
3. `matrix(c(4, 9, 10, 1, 5, 7), nrow = 2)`
4. `matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)`

Solution to Challenge 7

Consider the R output of the matrix below:

Output

```
     [,1] [,2]
[1,]    4    1
[2,]    9    5
[3,]   10    7
```

What was the correct command used to write this matrix? Examine each command and try to figure out the correct one before typing them. Think about what matrices the other commands will produce.

R

```
matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)
```

Key Points

- Use `read.csv` to read tabular data in R.
- The basic data types in R are double, integer, complex, logical, and character.
- Use factors to represent categories in R.

‹ (/r-novice-gapminder/03-seeking-help/index.html)

› (/r-novice-gapmin data-structu part2/in