Lecture 4: R Basics, Part 2

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Rutgers University – New Jersey Medical School

2024-04-30

Data Types

Variables in R can be of different types. For example: numbers, character strings, tables simple lists. The function `class` helps us determine what type of object we have:

```r
a <- 2
class(a)
```

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

To work efficiently in R, it is important to learn the different types of variables and what we can do with these.

Data Frames

Up to now, the variables we have defined are just one number, which is not very useful for storing data. The most common way of storing a dataset in R is in a **data frame**.

Conceptually, we can think of a data frame as a table with rows representing observations and the different variables reported for each observation defining the columns.

Data frames are particularly useful for datasets because we can combine different data types into one object.

Data Frames

A large proportion of data analysis use data stored in a data frame.

For example, we stored the data for our motivating example in a data frame. You can access this dataset by loading the **dslabs** library and loading the `murders` dataset using the `data` function:

```r
library(dslabs)
data(murders)
```

Data Frames

To see that this is in fact a data frame, we type:

```
class(murders)
```

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

Examining an Object

The function `str` is useful for finding out more about the structure of an object:

```
str(murders)
```

```
## 'data.frame':    51 obs. of  5 variables:
## $ state : chr "Alabama" "Alaska" "Arizona" "Arkansas" ..
## $ abb : chr "AL" "AK" "AZ" "AR" ...
## $ region : Factor w/ 4 levels "Northeast","South",..: 2
## $ population: num 4779736 710231 6392017 2915918 3725395
## $ total : num 135 19 232 93 1257 ...
```

This tells us much more about the object. We see that the table has 51 rows (50 states plus DC) and five variables.

Data Frames

We can show the first six lines using the function `head`:

```r
head(murders)
```

```
##        state abb region population total
## 1    Alabama  AL  South    4779736   135
## 2     Alaska  AK   West     710231    19
## 3    Arizona  AZ   West    6392017   232
## 4   Arkansas  AR  South    2915918    93
## 5 California  CA   West   37253956  1257
## 6   Colorado  CO   West    5029196    65
```

Data Frames

In this dataset, each state is considered an observation and five variables are reported for each state.

Before we go any further in answering our original question about different states, let's learn more about the components of this object.

The Accessor: $

For our analysis, we will need to access the different variables represented by columns included in this data frame. To do this, we use the accessor operator $ in the following way:

```
murders$population
```

```
##  [1]  4779736   710231  6392017  2915918 37253956  5029196  3574097
##  [9]   601723 19687653  9920000  1360301  1567582 12830632  6483802
## [17]  2853118  4339367  4533372  1328361  5773552  6547629  9883640
## [25]  2967297  5988927   989415  1826341  2700551  1316470  8791894
## [33] 19378102  9535483   672591 11536504  3751351  3831074 12702379
## [41]  4625364   814180  6346105 25145561  2763885   625741  8001024
## [49]  1852994  5686986   563626
```

The Accessor: $

But how did we know to use population? Previously, by applying the function str to the object murders, we revealed the names for each of the five variables stored in this table. We can quickly access the variable names using:

```
names(murders)
```

```
## [1] "state"      "abb"        "region"     "population" "total"
```

The Accessor: $

Important: Note the order of the entries in `murders$population` preserves the order of the rows in our data table. This will later permit us to manipulate one variable based on the results of another. For example, we will be able to order the state names by the number of murders.

Pro Tip: R comes with a very nice auto-complete functionality that saves us the trouble of typing out all the names. Try typing `murders$p` then hitting the **tab** key on your keyboard. This functionality and many other useful auto-complete features are available when working in RStudio.

Vectors: Numerics, Characters, and Logical

The object `murders$population` is not one number but several. We call these types of objects **vectors**. A single number is technically a vector of length 1, but in general we use the term vectors to refer to objects with several entries. The function `length` tells you how many entries are in the vector:

```
pop <- murders$population
length(pop)
```

```
## [1] 51
```

Vectors: Numerics, Characters, and Logical

This particular vector is **numeric** since population sizes are numbers:

```r
class(pop)
```

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

In a numeric vector, every entry must be a number.

Vectors: Numerics, Characters, and Logical

To store character strings, vectors can also be of class **character**. For example, the state names are characters:

```r
class(murders$state)
```

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

As with numeric vectors, all entries in a character vector need to be a character.

Vectors: Numerics, Characters, and Logical

Another important type of vectors are **logical vectors**. These must be either `TRUE` or `FALSE`.

```
z <- 3 == 2
z
```

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

```
class(z)
```

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

Here the == is a relational operator asking if 3 is equal to 2. In R, if you just use one =, you actually assign a variable, but if you use two == you test for equality.

Vectors: Numerics, Characters, and Logical

You can see the other **relational operators** by typing:

```
?Comparison
```

In the future, you will see how useful relational operators can be. We discuss more important features of vectors later.

Vectors: Numerics, Characters, and Logical

Advanced: Mathematically, the values in `pop` are integers and there is an integer class in R. However, by default, numbers are assigned class numeric even when they are round integers. For example, `class(1)` returns numeric. You can turn them into class integer with the `as.integer()` function or by adding an `L` like this: `1L`. Note the class by typing: `class(1L)`

Factors

In the `murders` dataset, we might expect the region to also be a character vector. However, it is not:

```
class(murders$region)
```

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

It is a **factor**. Factors are useful for storing categorical data.

Factors

We can see that there 4 regions by using the `levels` function:

```
levels(murders$region)
```

```
## [1] "Northeast"     "South"         "North Central" "West"
```

Factors

In the background, R stores these **levels** as integers and keeps a map to keep track of the labels. This is more memory efficient than storing all the characters.

Note that the levels have an order that is different from the order of appearance in the factor object. The default in R is for the levels to follow alphabetical order. However, often we want the levels to follow a different order.

You can specify an order through the `levels` argument when creating the factor with the `factor` function. For example, in the murders dataset regions are ordered from east to west. The function `reorder` lets us change the order of the levels of a factor variable based on a summary computed on a numeric vector.

Factors

Suppose we want the levels of the region by the total number of murders rather than alphabetical order. If there are values associated with each level, we can use the `reorder` and specify a data summary to determine the order. The following code takes the sum of the total murders in each region, and reorders the factor following these sums.

```r
region <- murders$region
value <- murders$total
region <- reorder(region, value, FUN = sum)
levels(region)
```

```
## [1] "Northeast"     "North Central" "West"          "South"
```

Factors

The new order is in agreement with the fact that the Northeast has the least murders and the South has the most.

Warning: Factors can be a source of confusion since sometimes they behave like characters and sometimes they do not. As a result, confusing factors and characters are a common source of bugs.

Lists

Data frames are a special case of **lists**. Lists are useful because you can store any combination of different types. You can create a list using the `list` function like this:

```r
record <- list(name = "John Doe",
               student_id = 1234,
               grades = c(95, 82, 91, 97, 93),
               final_grade = "A")
```

The function `c` (for concatenate) is described later.

Lists

The list on the prior slide includes a character, a number, a vector with five numbers, and another character.

```
class(record)
```

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

```
record
```

```
## $name
## [1] "John Doe"
##
## $student_id
## [1] 1234
##
## $grades
## [1] 95 82 91 97 93
##
## $final_grade
## [1] "A"
```

Lists

As with data frames, you can extract the components of a list with the accessor $.

```
record$student_id
```

```
## [1] 1234
```

Lists

We can also use double square brackets (`[[`) like this:

```r
record[["student_id"]]
```

```
## [1] 1234
```

You should get used to the fact that in R, there are often several ways to do the same thing, such as accessing entries.

Lists

You might also encounter lists without variable names.

```r
record2 <- list("John Doe", 1234)
record2
```

```
## [[1]]
## [1] "John Doe"
##
## [[2]]
## [1] 1234
```

Lists

If a list does not have names, you cannot extract the elements with $, but you can still use the brackets method and instead of providing the variable name, you provide the list index, like this:

```
record2[[1]]
```

```
## [1] "John Doe"
```

We won't be using lists until later, but you might encounter one in your own exploration of R. For this reason, we showed you some basics here.

Matrices

Matrices are another type of object that are common in R. Matrices are similar to data frames in that they are two-dimensional: they have rows and columns. However, like numeric, character and logical vectors, entries in matrices have to be all the same type. For this reason data frames are much more useful for storing data, since we can have characters, factors, and numbers in them.

Matrices

Yet matrices have a major advantage over data frames: we can perform matrix algebra operations, a powerful type of mathematical technique. We will cover matematical computions on matrices in more detail later!

Matrices

We can define a matrix using the `matrix` function. We need to specify the number of rows and columns.

```
mat <- matrix(1:12, 4, 3)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

Matrices

You can access specific entries in a matrix using square brackets (`[`). If you want the second row, third column, you use:

```
mat[2, 3]
```

```
## [1] 10
```

Matrices

If you want the entire second row, you leave the column spot empty:

```
mat[2, ]
```

```
## [1]  2  6 10
```

Notice that this returns a vector, not a matrix.

Matrices

Similarly, if you want the entire third column, you leave the row spot empty:

```
mat[, 3]
```

```
## [1]  9 10 11 12
```

This is also a vector, not a matrix.

Matrices

You can access more than one column or more than one row if you like. This will give you a new matrix.

```r
mat[, 2:3]
```

```
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
## [3,]    7   11
## [4,]    8   12
```

Matrices

You can subset both rows and columns:

```
mat[1:2, 2:3]
```

```
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
```

Matrices

We can convert matrices into data frames using the function `as.data.frame`:

```r
as.data.frame(mat)
```

```
##   V1 V2 V3
## 1  1  5  9
## 2  2  6 10
## 3  3  7 11
## 4  4  8 12
```

Matrices

You can also use single square brackets ([) to access rows and columns of a data frame:

```
data("murders")
murders[25, 1]
```

```
## [1] "Mississippi"
```

```
murders[2:3, ]
```

```
##     state abb region population total
## 2  Alaska  AK   West     710231    19
## 3 Arizona  AZ   West    6392017   232
```

Exercises

Now open the **R Basics Exercises** file and complete Exercises 6-11.

Vectors

In R, the most basic objects available to store data are **vectors**. As we have seen, complex datasets can usually be broken down into components that are vectors. For example, in a data frame, each column is a vector. Here we learn more about this important class.

Creating Vectors

We can create vectors using the function c, which stands for **concatenate**. We use c to concatenate entries in the following way:

```
codes <- c(380, 124, 818)
codes
```

```
## [1] 380 124 818
```

Creating Vectors

We can also create character vectors. We use the quotes to denote that the entries are characters rather than variable names.

```r
country <- c("italy", "canada", "egypt")
```

Creating Vectors

In R you can also use single quotes:

```
country <- c('italy', 'canada', 'egypt')
```

But be careful not to confuse the single quote ’ with the **back quote** ‘.

Creating Vectors

By now you should know that if you type:

```
country <- c(italy, canada, egypt)
```

you receive an error because the variables `'italy'`, `'canada'`, and `'egypt'` are not defined. If we do not use the quotes, R looks for variables with those names and returns an error.

Names

Sometimes it is useful to name the entries of a vector. For example, when defining a vector of country codes, we can use the names to connect the two:

```
codes <- c(italy = 380, canada = 124, egypt = 818)
codes
```

```
##   italy canada  egypt
##     380    124    818
```

Names

The object codes continues to be a numeric vector:

```
class(codes)
```

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

but with names:

```
names(codes)
```

```
## [1] "italy"  "canada" "egypt"
```

Names

If the use of strings without quotes looks confusing, know that you can use the quotes as well:

```
codes <- c("italy" = 380, "canada" = 124, "egypt" = 818)
codes
```

```
##  italy canada  egypt
##    380    124    818
```

There is no difference between this function call and the previous one. This is one of the many ways in which R is quirky compared to other languages.

Names

We can also assign names using the `names` functions:

```r
codes <- c(380, 124, 818)
country <- c("italy","canada","egypt")
names(codes) <- country
codes
```

```
##  italy canada  egypt
##    380    124    818
```

Sequences

Another useful function for creating vectors generates sequences:

```r
seq(1, 10)
```

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

Sequences

The first argument defines the start, and the second defines the end which is included. The default is to go up in increments of 1, but a third argument lets us tell it how much to jump by:

```r
seq(1, 10, 2)
```

```
## [1] 1 3 5 7 9
```

Sequences

If we want consecutive integers, we can use the following shorthand:

```
1:10
```

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

Sequences

When we use these functions, R produces integers, not numerics, because they are typically used to index something:

```
class(1:10)
```

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

However, if we create a sequence including non-integers, the class changes:

```
class(seq(1, 10, 0.5))
```

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

Subsetting

We use square brackets to access specific elements of a vector. For the vector `codes` we defined above, we can access the second element using:

```
codes[2]
```

```
## canada
##    124
```

You can get more than one entry by using a multi-entry vector as an index:

```
codes[c(1,3)]
```

```
## italy egypt
##   380   818
```

Subsetting

The sequences defined above are particularly useful if we want to access, say, the first two elements:

```
codes[1:2]
```

```
##  italy canada
##    380    124
```

Subsetting

If the elements have names, we can also access the entries using these names. Below are two examples.

```
codes["canada"]
```

```
## canada
##    124
```

```
codes[c("egypt","italy")]
```

```
## egypt italy
##   818   380
```

Coercion

In general, **coercion** is an attempt by R to be flexible with data types. When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error. This can also lead to confusion.

Failing to understand **coercion** can drive programmers crazy in R since it behaves quite differently from most other languages in this regard. Let's learn about it with some examples.

Coercion

We said that vectors must be all of the same type. So if we try to combine, say, numbers and characters, you might expect an error:

```
x <- c(1, "canada", 3)
```

But we don't get one, not even a warning! What happened? Look at `x` and its class:

```
x
```

```
## [1] "1"      "canada" "3"
```

```
class(x)
```

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

Coercion

R **coerced** the data into characters. It guessed that because you put a character string in the vector, you meant the 1 and 3 to actually be character strings `"1"` and “3”. The fact that not even a warning is issued is an example of how coercion can cause many unnoticed errors in R.

R also offers functions to change from one type to another. For example, you can turn numbers into characters with:

```r
x <- 1:5
y <- as.character(x)
y
```

```
## [1] "1" "2" "3" "4" "5"
```

Coercion

You can turn it back with `as.numeric`:

```r
as.numeric(y)
```

```
## [1] 1 2 3 4 5
```

This function is actually quite useful since datasets that include numbers as character strings are common.

Not Availables (NA)

When a function tries to coerce one type to another and encounters an impossible case, it usually gives us a warning and turns the entry into a special value called an NA for "not available". For example:

```
x <- c("1", "b", "3")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1 NA  3
```

R does not have any guesses for what number you want when you type b, so it does not try. As a data scientist you will encounter the NAs often as they are generally used for missing data, a common problem in real-world datasets.

Exercises

Now open the **R Basics Exercises** file and complete Exercises 12-23.

Session Info

```
sessionInfo()
```

```
## R version 4.4.0 (2024-04-24)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  LAPACK vers
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] dslabs_0.8.0      countrycode_1.6.0 ggflags_0.0.1     lubridate_1.9.3
##  [5] forcats_1.0.0     stringr_1.5.1     dplyr_1.1.4       purrr_1.0.2
##  [9] readr_2.1.5       tidyr_1.3.1       tibble_3.2.1      ggplot2_3.5.1
## [13] tidyverse_2.0.0
##
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.5      compiler_4.4.0    tidyselect_1.2.1  scales_1.3.0
##  [5] yaml_2.3.8        fastmap_1.1.1     R6_2.5.1          generics_0.1.3
##  [9] knitr_1.46        munsell_0.5.1     pillar_1.9.0      tzdb_0.4.0
## [13] rlang_1.1.3       utf8_1.2.4        stringi_1.8.3     xfun_0.43
## [17] timechange_0.3.0  cli_3.6.2         withr_3.0.0       magrittr_2.0.3
## [21] digest_0.6.35     grid_4.4.0        rstudioapi_0.16.0 hms_1.1.3
## [25] lifecycle_1.0.4   vctrs_0.6.5       evaluate_0.23     glue_1.7.0
## [29] fansi_1.0.6       colorspace_2.1-0  rmarkdown_2.26    tools_4.4.0
## [33] pkgconfig_2.0.3   htmltools_0.5.8.1
```