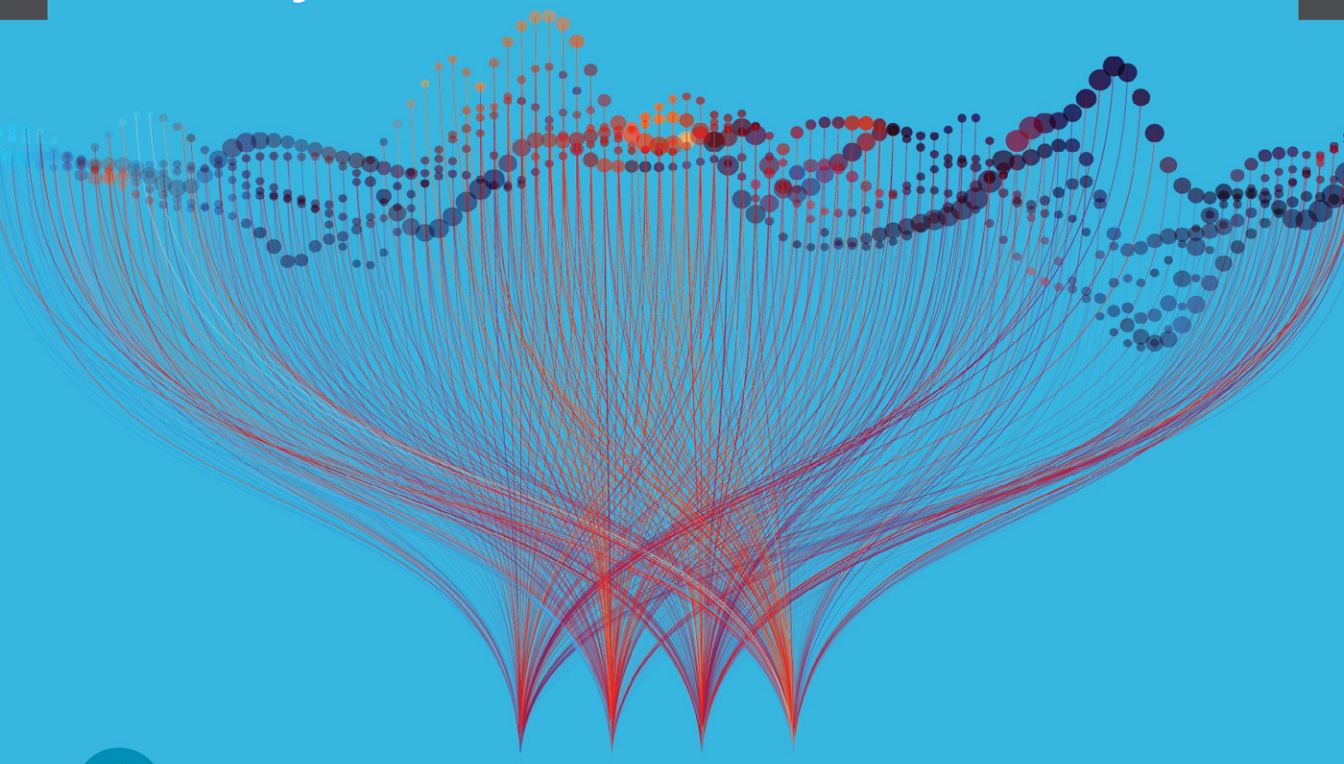
Addison Wesley Data & Analytics Series

PROGRAMMING SKILLS FOR DATA SCIENCE

Start Writing Code to Wrangle, Analyze, and Visualize Data with R



MICHAEL FREEMAN | JOEL ROSS

Programming Skills for Data Science

8

Lists

This chapter covers an additional R data type called a **list**. Lists are somewhat similar to vectors, but can store more types of data and usually include more details about that data (with some cost). Lists are R's version of a map, which is a common and extremely useful way of organizing data in a computer program. Moreover, lists are used to create *data frames*, which are the primary data storage type used for working with sets of real data in R. This chapter covers how to create and access elements in a list, as well as how to apply functions to lists.

8.1 What Is a List?

A **list** is a lot like a vector, in that it is a *one-dimensional collection of data*. However, unlike a vector, you can store elements of *different types* in a list; for example, a list can contain numeric data *and* character string data. Lists can also contain more complex data types—including vectors and even other lists!

Elements in a list can also be **tagged** with names that you can use to easily refer to them. For example, rather than talking about the list's "element #1," you can talk about the list's "`first_name` element." This feature allows you to use lists to create a type of map. In computer programming, a **map** (or "mapping") is a way of associating one value with another. The most common real-world example of a map is a dictionary or encyclopedia. A dictionary associates each word with its definition: you can "look up" a definition by using the word itself, rather than needing to look up the 3891st definition in the book. In fact, this same data structure is called a `dictionary` in the Python programming language!

> **Caution**: The definition of a list in the R language is distinct from how some other languages use the term "list." When you begin to explore other languages, don't assume that the same terminology implies the same capabilities.

As a result, lists are extremely useful for organizing data. They allow you to group together data like a person's name (characters), job title (characters), salary (number), and whether the person is a member of a union (logical)—and you don't have to remember whether the person's name or title was the first element!

> **Remember**: If you want to label elements in a collection, *use a list*. While vector elements can also be tagged with names, that practice is somewhat uncommon and requires a more verbose syntax for accessing the elements.

8.2 Creating Lists

You create a list by using the **`list()`** function and passing it any number of **arguments** (separated by commas) that you want to make up that list—similar to the `c()` function for vectors.

However, you can (and should) specify the *tags* for each element in the list by putting the name of the tag (which is like a variable name), followed by an equals symbol (`=`), followed by the value you want to go in the list and be associated with that tag. This is similar to how named arguments are specified for functions (see Section 6.2.1). For example:

```
# Create a `person` variable storing information about someone
# Code is shown on multiple lines for readability (which is valid R code!)
person <- list(
  first_name = "Ada",
  job = "Programmer",
  salary = 78000,
  in_union = TRUE
)
```

This creates a list of four elements: `"Ada"`, which is tagged with `first_name`; `"Programmer"`, which is tagged with `job`; `78000`, which is tagged with `salary`; and `TRUE`, which is tagged with `in_union`.

> **Remember**: You can have vectors as elements of a list. In fact, each scalar value in the preceding example is really a vector (of length 1).

It is possible to create a list without tagging the elements:

```
# Create a list without tagged elements. NOT the suggested usage.
person_alt <- list("Ada", "Programmer", 78000, TRUE)
```

However, tags make it easier and less error-prone to access specific elements. In addition, tags help other programmers read and understand the code—tags let them know what each element in the list represents, similar to an informative variable name. Thus it is recommended to always tag lists you create.

> **Tip**: You can get a vector of the names of your list items using the `names()` function. This is useful for understanding the structure of variables that may have come from other data sources.

Because lists can store elements of different types, they can store values that are lists themselves. For example, consider adding a list of favorite items to the `person` list in the previous example:

```
# Create a `person` list that has a list of favorite items
person <- list(
  first_name = "Ada",
  job = "Programmer",
  salary = 78000,
  in_union = TRUE,
  favorites = list(
    music = "jazz",
    food = "pizza"
  )
)
```

This data structure (a *list of lists*) is a common way to represent data that is typically stored in *JavaScript Object Notation* (JSON). For more information on working with JSON data, see Chapter 14.

8.3 Accessing List Elements

Once you store information in a list, you will likely want to retrieve or reference that information in the future. Consider the output of printing the `person` list, as shown in Figure 8.1. Notice that the output includes each tag name prepended with a dollar sign (**$**) symbol, and then on the following line prints the element itself.

Because list elements are (usually) tagged, you can access them by their tag name rather than by the index number you used with vectors. You do this by using **dollar notation**: refer to the element

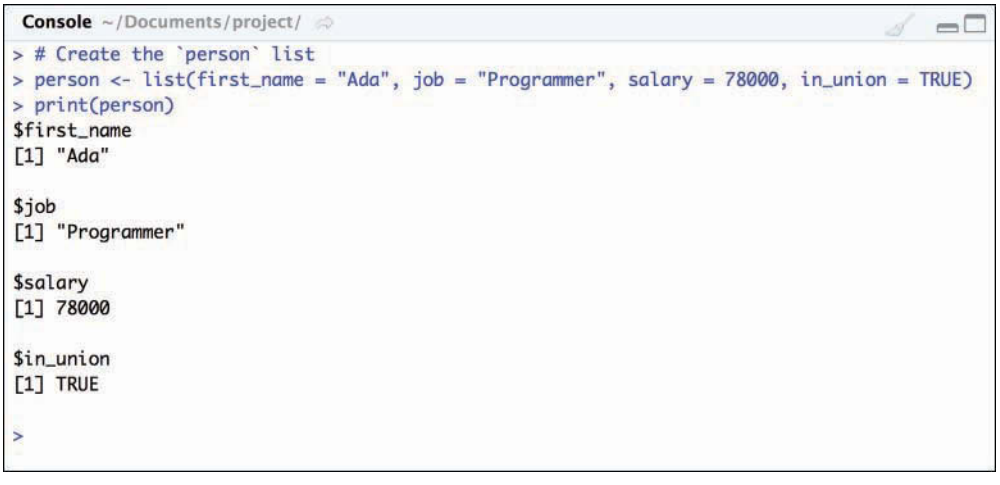


Figure 8.1 Creating and printing a list element in RStudio.

with a particular tag in a list by writing the name of the list, followed by a **$**, followed by the element's tag (a syntax unavailable to named vectors):

```
# Create the `person` list
person <- list(
  first_name = "Ada",
  job = "Programmer",
  salary = 78000,
  in_union = TRUE
)

# Reference specific tags in the `person` list
person$first_name # [1] "Ada"
person$salary     # [1] 78000
```

You can almost read the dollar sign as if it were an "apostrophe s" (possessive) in English. Thus, `person$salary` would mean "the `person` list's `salary` value."

Regardless of whether a list element has a tag, you can also access it by its numeric index (i.e., if it is the first, second, and so on item in the list). You do this by using **double-bracket notation**. With this notation, you refer to the element at a particular index of a list by writing the name of the list, followed by double square brackets (**`[[]]`**) that contain the index of interest:

```
# This is a list (not a vector!), even though elements have the same type
animals <- list("Aardvark", "Baboon", "Camel")

animals[[1]] # [1] "Aardvark"
animals[[3]] # [1] "Camel"
animals[[4]] # Error: subscript out of bounds!
```

You can also use double-bracket notation to access an element by its tag if you put a character string of the tag name inside the brackets. This is particularly useful in cases when the tag name is stored in a variable:

```
# Create the `person` list with an additional `last_name` attribute
person <- list(
  first_name = "Ada",
  last_name = "Gomez",
  job = "Programmer",
  salary = 78000,
  in_union = TRUE
)

# Retrieve values stored in list elements using strings
person[["first_name"]] # [1] "Ada"
person[["salary"]]     # [1] 78000
```

```
# Retrieve values stored in list elements
# using strings that are stored in variables
name_to_use <- "last_name"  # choose name (i.e., based on formality)
person[[name_to_use]]       # [1] "Gomez"
name_to_use <- "first_name" # change name to use
person[[name_to_use]]       # [1] "Ada"

# You can use also indices for tagged elements
# (but they're difficult to keep track of)
person[[1]] # [1] "Ada"
person[[5]] # [1] TRUE
```

Remember that lists can contain complex values (including other lists). Accessing these elements with either dollar or double-bracket notation will return that "nested" list, allowing you to access its elements:

```
# Create a list that stores a vector and a list. `job_post` has
# a *list* of qualifications and a *vector* of responsibilities.
job_post <- list(
  qualifications = list(
    experience = "5 years",
    bachelors_degree = TRUE
  ),
  responsibilities = c("Team Management", "Data Analysis", "Visualization")
)

# Extract the `qualifications` elements (a list) and store it in a variable
job_qualifications <- job_post$qualifications

# Because `job_qualifications` is a list, you can access its elements
job_qualifications$experience # "5 years"
```

In this example, `job_qualifications` is a variable that refers to a list, so its elements can be accessed via dollar notation. But as with any operator or function, it is also possible to use dollar notation on an *anonymous value* (e.g., a literal value that has not been assigned to a variable). That is, because `job_post$qualifications` is a list, you can use bracket or dollar notation to refer to an element of that list without assigning it to a variable first:

```
# Access the `qualifications` list's `experience` element
job_post$qualifications$experience # "5 years"

# Access the `responsibilities` vector's first element
# Remember, `job_post$responsibilities` is a vector!
job_post$responsibilities[1] # "Team Management"
```

This example of "chaining" together dollar-sign operators allows you to directly access elements in lists with a complex structure: you can use a single expression to refer to the "`job-post`'s `qualification`'s `experience`" value.

8.4 Modifying Lists

As with vectors, you can add and modify list elements. List elements can be modified by *assigning a new value* to an existing list element. New elements can be added by assigning a value to a new tag (or index). Moreover, list elements can be removed by reassigning the value `NULL` to an existing list element. All of these operations are demonstrated in the following example:

```
# Create the `person` list
person <- list(
  first_name = "Ada",
  job = "Programmer",
  salary = 78000,
  in_union = TRUE
)

# There is currently no `age` element (it's NULL)
person$age # NULL

# Assign a value to the (new) `age` tag
person$age <- 40
person$age # [1] 40

# Reassign a value to list's `job` element
person$job <- "Senior Programmer" # a promotion!
print(person$job)
# [1] "Senior Programmer"

# Reassign a value to the `salary` element (using the current value!)
person$salary <- person$salary * 1.15 # a 15% raise!
print(person$salary)
# [1] 89700

# Remove the `first_name` tag to make the person anonymous
person$first_name <- NULL
```

`NULL` is a special value that means "undefined" (note that it is a special value `NULL`, not the character string `"NULL"`). `NULL` is somewhat similar to the term `NA`—the difference is that `NA` is used to refer to a value that is *missing* (such as an empty element in a vector)—that is, a "hole." Conversely, `NULL` is used to refer to a value that is not defined but doesn't necessarily leave a "hole" in the data. `NA` values usually result when you are creating or loading data that may have parts missing; `NULL` can be used to remove values. For more information on the difference between these values, see this R-Bloggers post.[1]

[1] **R: NA vs. NULL** post on R-Bloggers: https://www.r-bloggers.com/r-na-vs-null/