Data Frames and Lists

Chapter 7

Stats 20: Introduction to Statistical Programming with R

UCLA

Contents

Acknowledgements: Vivian Lew and Juana Sanchez

Learning Objectives

After studying this chapter, you should be able to:

- Install and load packages in R.
- Access and interpret the R Help Documentation for built-in objects and functions.
- Load datasets from packages.
- Create data frames and lists.
- Differentiate between matrices and data frames.
- Extract and assign values to data frames and lists.
- Understand the difference between the mode and the class of an object.
- Summarize an R object with `str()` and `summary()`.
- Understand how and when to use the `apply` family of functions: `apply()`, `lapply()`, `sapply()`, `vapply()`, `tapply()`.

1 Using R Packages

1.1 Installing and Loading R Packages

A **package** in R is a collection of functions, data, and documentation encapsulated into a single bundle. The initial download of R contains a few standard packages, collectively known as **base R**, that are loaded and available to use when you open a new R session. Some of the main packages in base R are the `base`, `stats`, `graphics`, and `datasets` packages. Other packages are stored on your computer in a **library**, a directory of the installed packages on your computer.

To load and access an installed package in an R session, we use the **`library()`** function and input the name of the package we want to use (without quotations). For example, to load the `MASS` package:

```
library(MASS)
```

The `library()` function will throw an error if you try to load a package that has not been installed on your computer.

```
library(whoops)
```

```
Error in library(whoops): there is no package called 'whoops'
```

The **`search()`** function outputs R's current **search path**, which allows us to see what packages are currently loaded.

```
search()
```

```
 [1] ".GlobalEnv"        "package:MASS"      "package:stats"    
 [4] "package:graphics"  "package:grDevices" "package:utils"    
 [7] "package:datasets"  "package:methods"   "Autoloads"        
[10] "package:base"     
```

Note: The packages and environments in the search path are where R will look into when trying to use objects and functions. If R tries to run a command and is unable to find an object or function in the search path, it will throw an error. The order of the search path is the order of the packages and environments in which R will search for objects. For example, the global environment `".GlobalEnv"` is first in the search path, so R will always look for objects in the global environment first before trying to find objects in other packages. This is why assigning `pi` to a value will mask the built-in `pi` object in base R.

Many people have written functions and added datasets that expand on the functions and datasets initially downloaded when installing R. These contributions are encapsulated into R packages. Most of these specialized packages are not included in the initial download of R and need to be installed separately.

The biggest **repository** of R packages online is the Comprehensive R Archive Network (CRAN). The `install.packages()` function allows us to install packages from CRAN. Input the name of the package you want to install, either in single or double quotations. For example, to install the `boot` package:

```
install.packages("boot") # Single quotes '' are okay too
```

You have to specify the CRAN mirror from which to download the package. The mirror at "USA (CA 1)" is at UC Berkeley.

You can also install packages in R or RStudio from the menu bar.

- In the R console, click on "Packages & Data" and then "Package Installer". Click on "Get List", select the CRAN mirror, select the package to install, and click on "Install Selected".
- In RStudio, click on "Tools" in the menu bar and then "Install Packages…".

Note: Packages only need to be installed once (per computer). Once the package is installed on your computer, you need to tell R that you want to access the functions and data from it by using the `library()` function.

Caution: To use a function or dataset from a given package, you have to use `library()` *every time* you open a new R console. If you quit an R session and reopen R, you need to load the package again.

1.2 Getting Help

For help on a built-in function in R, use `?` followed by the name of the function, or apply the `help()` function. For example:

```
?mean
help(mean) # Same thing as ?
```

Help files in R, collectively called **R documentation**, are not always user friendly, but they are usually a great place to start understanding syntax and functionality.

If you do not know the name of the function, you can do a search with a double question mark `??` followed by the search term, or apply the `help.search()` function. The "fuzzy" search will search over all the available help files and return a list of any documentation that has an alias, concept, or title that matches the search term. For example:

```
??regression
help.search("regression") # Same thing as ??
```

Note: The single question mark `?` will search for functions in the packages that are currently loaded. The double question mark `??` will search for any documentation in all of the packages installed on your computer.

To receive help on a specific package (that is already installed), use the `help` argument in the `library()` function, like in the example below:

```
library(help = "MASS")
```

1.3 The `data()` Function

Both built-in and contributed packages in R contain datasets. The `data()` function loads datasets from an available package currently in the search path and saves a copy to the workspace.

For example, there are many examples of datasets in the `datasets` package. The `datasets` package is part of base R, so the data objects actually can be used as if they are built-in objects in R (like `pi`). In particular, the objects can be called and used without loading them with the `data()` function. Other packages need to be loaded first with the `library()` function before data objects can be used.

```
data(trees) # Load the trees object
ls() # The trees object has been added to the workspace
```

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

Question: How can we find out what type of trees were measured for this dataset?

The `data()` function has a second functionality that allows us to list the available datasets in a specific package. We can type the name of the package in the `package` argument of the `data()` function.

```
data(package = "MASS") # List the available datasets in the MASS package
```

The `MASS` package contains a dataset called `geyser`. We first load the package (if it has not yet been loaded for the current R session), then load the dataset.

```
library(MASS) # Load the MASS package (if it was not loaded already)
data(geyser) # Load the geyser object
```

Question: Which geyser was measured for this dataset? When was this data collected?

1.4 The `head()` and `tail()` Functions

It is generally helpful to print/return a dataset to get an idea of how the data is organized. For objects with many values (or datasets with many observations), it may not be useful to print the entire object. The **`head()`** function outputs the first few values of the input object. For vectors, `head()` will output the first few elements. For two-dimensional objects (like data frames and matrices), `head()` will output the first few rows.

```
head(trees) # Return the first few values of the trees object
```

```
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
```

The second argument in `head()` is the size `n`, which controls how many values to output. By default, `n=6`, so `head()` returns the first six values (or rows). A negative `n` argument will return all but the last `n` values.

```
head(trees, n = 3) # Return the first 3 rows
```

```
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
```

```
head(1:20, n = -8) # Return all values except the last 8
```

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

Similarly, the **`tail()`** function outputs the last few values (or rows) of the input object. The syntax is analogous to `head()`: A positive `n` argument returns the last `n` values, and a negative `n` argument returns all but the first `n` values.

```
tail(geyser) # Return the last few (default is 6) rows
```

```
    waiting duration
294      87 2.133333
295      52 4.083333
296      85 2.066667
297      58 4.000000
298      88 4.000000
299      79 2.000000
```

```
tail(1:20, n = -5) # Return all values except the first 5
```

```
 [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

2 Data Frames

2.1 Basic Definitions and Functions

Recall that all the values in a matrix object must be of the same type (i.e., all numeric, character, logical). Many datasets in statistics involve both numeric and categorical variables, so storing data in a matrix is often too restrictive.

Like a matrix, a **data frame** is also a two-dimensional array of values. However, data frames are more flexible objects in that each column of a data frame can be of a different type. Like how most data tables are organized in statistics, each column of a data frame generally corresponds to variables, and each row corresponds to observations.

Consider the table of data on the employees at the Pawnee Parks and Recreation Department, introduced in the previous chapter.

Name	Height (inches)	Weight (pounds)	Income ($/month)
Leslie	62	115	4000
Ron	71	201	(Redacted)
April	66	119	2000

Recall that we used the `matrix()` function to create a matrix of the numeric values in the table.

```
parks_mat <- cbind(c(62, 71, 66), c(115, 201, 119), c(4000, NA, 2000))
rownames(parks_mat) <- c("Leslie", "Ron", "April")
colnames(parks_mat) <- c("Height", "Weight", "Income")
parks_mat
```

```
       Height Weight Income
Leslie     62    115   4000
Ron        71    201     NA
April      66    119   2000
```

The `data.frame()` function inputs multiple vectors of the same length and outputs a data frame with each column corresponding to the vectors (in order). We can set column (variable) names by typing the name of the column in quotation marks.

```
parks_df <- data.frame(
  "Name" = c("Leslie", "Ron", "April"), "Height" = c(62, 71, 66),
  "Weight" = c(115, 201, 119), "Income" = c(4000, NA, 2000)
)
parks_df
```

```
    Name Height Weight Income
1 Leslie     62    115   4000
2    Ron     71    201     NA
3  April     66    119   2000
```

For the `parks_df` object, the `Name` variable is a column in the data frame, not the row name. The 'Name column has a different type than the other columns.

We can also use `data.frame()` to convert (coerce) matrices into data frames. By converting `parks_mat` into a data frame, the row and column names are preserved.

```
data.frame(parks_mat)
```

```
       Height Weight Income
Leslie     62    115   4000
Ron        71    201     NA
April      66    119   2000
```

Many of the same basic functions for matrices also work for data frames.

- The `dim()` function outputs the dimension of the input data frame.

```
dim(parks_df)
```

```
[1] 3 4
```

- The `rownames()`, `colnames()`, and `dimnames()` functions return row and column names.

```
rownames(parks_df)
```

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

```
colnames(parks_df)
```

```
[1] "Name"   "Height" "Weight" "Income"
```

```
dimnames(parks_df)
```

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

[[2]]
[1] "Name"   "Height" "Weight" "Income"
```

Note: Unlike the `matrix()` function that did not assign row or column names, notice that the default row names from `data.frame()` are the row numbers.

- The `cbind()` function combines (binds) columns of data frames together. The vectors or data frames should contain the same number of rows/observations.

  Side Note: Recycling with `cbind()` works differently for data frames as it does for matrices. Column binding with a vector will automatically recycle only if it will be completely recycled. If incomplete recycling might occur, then it will throw an error.

```
cbind(parks_df, "Age" = c(34, 49, 20))
```

```
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20
```

- The `rbind()` combines rows of data frames together. Since different values in rows are allowed to be different types, added rows are typically either data frames or lists. Merging rows from two data frames can get complicated, though, because the names of the columns in each data frame should correspond to the names in the other.

```
# Create a data frame with a new observation
ron_dunn <- data.frame("Name" = "Ron", "Height" = 74, "Weight" = 194, "Income" = 5000)
rbind(parks_df, ron_dunn)
```

```
    Name Height Weight Income
1 Leslie     62    115   4000
2    Ron     71    201     NA
3  April     66    119   2000
4    Ron     74    194   5000
```

```
rbind(parks_df, list("Ron", 74, 194, 5000)) # Same thing
```

```
    Name Height Weight Income
1 Leslie     62    115   4000
2    Ron     71    201     NA
3  April     66    119   2000
4    Ron     74    194   5000
```

Question: What is different about the command `rbind(parks_df, c("Ron", 74, 194, 5000))`?

2.2 Extracting Data from Data Frames

2.2.1 Square Brackets

Since data frames are two-dimensional objects, we can use the same methods for extracting and reassigning values from matrices on data frames. In particular, we can use square brackets with an ordered pair of indices, corresponding to the row index and the column index, separated by a comma. For example, an index of `[i, j]` means to extract the entry in the `i`th row and `j`th column, also called the (i, j)th entry. Logical and named indices will also work as expected.

```
parks_df[1, ] # Extract the first row
```

```
    Name Height Weight Income
1 Leslie     62    115   4000
```

```
parks_df[, -1] # Remove the first column
```

```
  Height Weight Income
1     62    115   4000
2     71    201     NA
3     66    119   2000
```

```
parks_df[-2, 3] # Remove the second row and extract the third column
```

```
[1] 115 119
```

```
parks_df[, "Name"] # Extract the Names column
```

```
[1] "Leslie" "Ron"    "April"
```

```
parks_df[c(FALSE, FALSE, TRUE), "Income"] # Extract the third entry from the Income column
```

```
[1] 2000
```

Note: Data frames consist of columns of vectors. When the output contains multiple columns, the output remains a data frame (so it can still allow for columns of different types). When the output contains only one column, the output becomes a vector. To preserve the data structure when subsetting, include the argument `drop = FALSE` in the square brackets.

```
parks_df[, "Name", drop = FALSE] # The output will stay as a data frame
```

```
    Name
1 Leslie
2    Ron
3  April
```

The `drop = FALSE` argument also works when subsetting single rows or columns from matrix objects.

Note: The `data.frame()` function has an optional argument called `stringsAsFactors` that controls whether to coerce characters (also called strings) into factors. In R Version 4.0.2 or later, the argument is set to `FALSE` by default. In Version 3.6.3 or earlier, the argument was set to `TRUE`. To ensure that the `data.frame()` function does not coerce columns of characters into factors (regardless of version), we can explicitly set the argument `stringsAsFactors = FALSE`.

Caution: Be careful to notice whether a data frame is storing a column as a character vector or a factor. In order to reassign a value in a factor column, we need to use the methods that we use for factors. We cannot just reassign a value with the assignment `<-` operator as we would for a character vector.

2.2.2 Double Square Brackets

There are many ways to extract data from objects in R, depending on the type of object. Data frames are internally stored in R as list objects whose components are the column vectors.

For data frames and lists, the columns/components can be extracted using **double square brackets `[[]]`**, either referring to the components by numeric index or by name.

```
parks_df[[1]] # Extract the first column (which is Name)
```

```
[1] "Leslie" "Ron"    "April"
```

```
parks_df[["Height"]] # Extract the Height column
```

```
[1] 62 71 66
```

```
parks_df[[3]][1] # Extract the first element of the third column (Weight)
```

```
[1] 115
```

2.2.3 The $ Operator

For data frames (and lists) where the columns of the object typically have names, the `$` operator is an efficient way to extract a single column. The left side of the `$` contains the data frame we want to extract from, and the right side contains the name of the column to extract.

```
parks_df$Height # Extract the Height column from parks_df
```

```
[1] 62 71 66
```

```
parks_df$Income # Extract the Income column from parks_df
```

```
[1] 4000   NA 2000
```

When multiple data frames in the workspace have the same variable name inside, it becomes crucial to always know which variable you are using. The `$` operator is helpful in keeping track of which data frame the variable comes from.

Note: The `$` operator is also able to add a new column (of the same length) to an existing data frame. This can be an alternative to `cbind()`.

```
parks_df # Does not have the Age variable
```

```
    Name Height Weight Income
1 Leslie     62    115   4000
2    Ron     71    201     NA
3  April     66    119   2000
```

```
parks_df$Age <- c(34, 49, 20) # Add the Age variable to the parks_df object
```

```
parks_df
```

```
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20
```

2.2.4 The `with()` Function

When referring to a data frame many times, typing the name of the data frame every time may become too cumbersome.

The **`with()`** function allows us to reference variable names inside a data frame without brackets or the `$` operator. The first argument of `with()` is the data frame we want to use, and the second argument is the command we want to run using the input data frame.

```
with(parks_df, Height) # Output the Height variable from parks_df
```

```
[1] 62 71 66
```

```
with(parks_df, Weight > 110) # Which weights in parks_df are greater than 110?
```

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

```
with(parks_df, mean(Height)) # Compute the mean of the heights
```

```
[1] 66.33333
```

Side Note: Technically, the `with()` command evaluates expressions in a local environment constructed by the data frame we want to use. The local environment behaves in a similar way to the body of functions:

- Columns in the data frame will be accessible by name as objects in the local environment created inside `with()`.
- Using curly braces `{}`, it is possible to input multiple command lines inside the `with()` function, but only the last command line will return output.
- Objects created or reassigned inside the local environment inside `with()` will not appear in the global environment.

```
with(parks_df, {
  height_in_cm <- Height * 2.54 # Convert heights into cm
  tall_cm_index <- height_in_cm > 165 # Find the heights taller than 165 cm
  Name[tall_cm_index] # Output the names of the people who are taller than 165 cm
})
```

```
[1] "Ron"   "April"
```

3 Modes and Classes

The **class** of an object determines how R will present the output to you when you call the object. For example, typing `parks_df` will present the data as a two-dimensional array with 3 rows and 5 columns, since `parks_df` is a data frame. Typing `parks_df$Name` will produce output of a factor object, which displays the vector of levels and the possible levels for the factor.

The mode of an object is how R internally stores the object. *This is not the same as the class.* For example, a matrix object is stored in R as a long vector. Data frames are actually stored as lists, where each column of the data frame is stored as a separate vector in the list. This is why the columns of a data frame are allowed to have different types, but entries in a matrix must have the same type.

It can be important to know both the class and mode of objects in R. Many functions expect certain modes as inputs and will give an error if you input an object with an incorrect mode. Some of the syntax we use to work with data frames (the `$` notation, for example) is available to us because data frames are stored as lists. This is why the `$` notation can be used with data frames and not matrices, and it is also why the `$` notation will be used for other list objects with different classes (such as the `lm` object for linear regression models).

The `class()` function inputs any R object and outputs the class of the object. For vectors, the `class()` function will differentiate between integer and double (numeric) types.

```
# The class and mode of a data frame
class(parks_df)
```

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

```
mode(parks_df)
```

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

```
# The class and mode of a matrix
class(parks_mat)
```

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

```
mode(parks_mat)
```

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

```
# The class and mode of a factor
class(factor(parks_df$Name))
```

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

```
mode(factor(parks_df$Name))
```

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

```
# The class and mode of an integer vector
class(1:9)
```

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

```
mode(1:9)
```

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

Side Note: In R Version 4.0.2 or later, the output of the `class()` function for matrix objects is `c("matrix", "array")` instead of only `"matrix"` as it was in Version 3.6.3 or earlier. This will not affect anything in this class (except that you cannot assume the class of a matrix object has length 1). In the context of Stats 102A, it means that the matrix class will inherit class properties (and methods) from the array class (arrays are higher-dimensional generalizations of matrices).

4 Lists

4.1 Basic Definitions and Functions

A **list** is an ordered collection of objects. Lists are possibly the most flexible objects in R. Each component in a list can be *any* other object in R, including vectors, matrices, data frames, functions, and even other lists.

```
L <- list(
  1:10,
  matrix(1:6, nrow = 2, ncol = 3),
  parks_df,
  list(1:5, matrix(1:9, nrow = 3, ncol = 3))
)
L
```

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

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

[[3]]
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20

[[4]]
[[4]][[1]]
[1] 1 2 3 4 5

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

Note: Conceptually, a vector is an ordered collection of values. In this sense, lists are vectors too, so lists are sometimes called **recursive** or **generic** vectors. The vector objects we have worked with so far are sometimes called **atomic** vectors, since their components cannot be broken down into smaller components.

Since lists are generic vectors, a few of the basic functions that work for vectors also work for lists.

- The concatenation function `c()` for vectors can also be used to concatenate lists together.

```
char_vec <- c("Pawnee Rules", "Eagleton Drools")
c(L, list(char_vec))
```

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

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

```
[[3]]
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20

[[4]]
[[4]][[1]]
[1] 1 2 3 4 5

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


[[5]]
[1] "Pawnee Rules"    "Eagleton Drools"
```

- The **`length()`** function, applied to a list, will return the number of (top level) components in the list.

```
length(L)
```

```
[1] 4
```

- The **`names()`** function can be used to assign or return the names of the components in a list.

```
names(L) <- c("Vector", "Matrix", "Data Frame", "List")
names(L)
```

```
[1] "Vector"     "Matrix"     "Data Frame" "List"
```

```
L
```

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

$Matrix
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

$`Data Frame`
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20

$List
$List[[1]]
[1] 1 2 3 4 5

$List[[2]]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

```
[3,]    3    6    9
```

The names can also be set when creating a list by typing the names of each component in quotation marks.

```r
list("Vector" = 1:10, "Matrix" = matrix(1:6, nrow = 2, ncol = 3))
```

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

$Matrix
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Note: The `names()` function can also be used to add names to elements of vectors. For data frames, `names()` is interchangeable with `colnames()`.

```r
first_five <- 1:5
names(first_five) <- c("One", "Two", "Three", "Four", "Five")
first_five
```

```
  One   Two Three  Four  Five 
    1     2     3     4     5 
```

```r
names(parks_df) # Same as colnames(parks_df)
```

```
[1] "Name"   "Height" "Weight" "Income" "Age"   
```

4.2 Extracting Data from Lists

The double square brackets `[[]]` and `$` operator are two ways of extracting data that are specific to list objects (and classes of objects stored as lists, like data frames).

4.2.1 Double Square Brackets

The double square brackets `[[]]` denote the index of the top level components in the list object. Double square brackets can thus be used to extract individual components from a list.

```r
L[[1]] # A vector of length 10
```

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

```r
L[[2]] # A 2x3 matrix
```

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

```r
L[[2]][, 1] # The first column of the 2x3 matrix
```

```
[1] 1 2
```

```r
L[[4]] # A list with two components
```

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

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

```
[3,]    3    6    9
```

Caution: The single index inside the double square brackets can be a single positive numeric value or a single character for a name of component. Double square brackets cannot be used to extract multiple top level components at a time.

```
L[[-1]]
```

```
Error in L[[-1]]: invalid negative subscript in get1index <real>
```

Note: Notice that `L[[4]]`, the fourth component of the list `L`, itself has a list nested inside. To access the components inside the nested list, we use *two* sets of double square brackets: The first set tells us which top level component object we are indexing, and the second set tells us which component of the inner list object to extract.

The first component of the `L[[4]]` list is a vector and the second component is a 3×3 matrix. To access the 3×3 matrix component, we would use `[[2]]`, applied to the `L[[4]]` object:

```
L[[4]][[2]] # The 3x3 matrix inside the L[[4]] list
```

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

Question: How can we extract the third column of the `L[[4]][[2]]` matrix?

If a list contains a list component inside, it is often called a **recursive list**. For recursive lists with many lists nested inside other lists, using multiple sets of double square brackets to access the nested list components can be confusing and cumbersome. We can instead use **recursive indexing** by inputting a vector index (of length greater than 1) in double square brackets. The ith element of the vector index will refer to the ith level component to extract.

```
# Extract the 2nd component of the 4th component of L
L[[c(4, 2)]] # Same as L[[4]][[2]]
```

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

```
L[[c(4, 2, 3)]] # Same as L[[4]][[2]][[3]]
```

```
[1] 3
```

Question: Why does `L[[c(4, 2, 3)]]` not output the third column of `L[[c(4, 2)]]`?

Using a recursive index with too many indices will result in an error.

```
L[[c(4, 2, 3, 1)]]
```

```
Error in L[[c(4, 2, 3, 1)]]: recursive indexing failed at level 3
```

4.2.2 The $ Operator

When the components of a list have names, the `$` operator can be used to extract a single component. The left side of the `$` contains the list we want to extract from, and the right side contains the name of the component to extract.

```
L$Vector
```

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

```
L$Matrix
```

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

```
L$`Data Frame`
```

```
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20
```

```
L$List
```

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

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

Note: Notice that the name `"Data Frame"` contains a space, so using the `$` with the full name requires backticks (or quotation marks) around the name.

For lists with many components, or components with long names, the first few letters of the component name can be used, as long as there is no ambiguity in which component is being referenced.

Since the name of every component of the `L` list starts with a different letter, then we only need to type the first letter for the `$` operator to know which component to extract.

```
L$D # Data Frame
```

```
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20
```

```
L$L # List
```

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

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

Caution: The two `L`'s in `L$L` refer to different things. The left `L` refers to the list object `L`. The right `L` refers to the first letter of the component inside `L` called `"List"`. In general, even if it is technically possible to use a single letter to reference a component, you should never shorten a component name more than is necessary. Clarity is more important than brevity.

Note: Just like for data frames (which are lists), the `$` operator is also able to add a new component to an existing list.

```
L$Function <- mean
names(L) # Function has been added to the list
```

```
[1] "Vector"     "Matrix"     "Data Frame" "List"       "Function"
```

```
L$Function(L$Vector) # Compute mean of the Vector component using the Function component
```

```
[1] 5.5
```

To remove a component from a list (or a column from a data frame), set the component to `NULL`.

```
L$Function <- NULL
L
```

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

$Matrix
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

$`Data Frame`
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20

$List
$List[[1]]
[1] 1 2 3 4 5

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

4.2.3 Single Square Brackets

As lists are generic vectors, single square brackets `[]` can also be used to subset from lists. One key difference between single square brackets `[]` and double square brackets `[[]]` is that the single square bracket always outputs a list object while the double square bracket outputs the component object inside.

```
L[1]
```

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

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

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

Single square brackets behave the same way for lists as you would expect with (atomic) vectors. They allow you to subset multiple components of a list with numeric, character, or logical indices.

```
L[-c(1, 4)]
```

```
$Matrix
```

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

$`Data Frame`
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20
```

```
L[c("Vector", "List")]
```

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

$List
$List[[1]]
[1] 1 2 3 4 5

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

```
L[c(TRUE, FALSE)]
```

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

$`Data Frame`
    Name Height Weight Income Age
1 Leslie     62    115   4000  34
2    Ron     71    201     NA  49
3  April     66    119   2000  20
```

5 Vectorized Functions for Data Frames and Lists

Recall that a function in R is vectorized if applying the function to an object will automatically apply the function to individual components of the object. For (atomic) vectors, vector arithmetic is implements operations element-by-element.

```
c(1, 2, 3) + c(2, 3, 4)
```

```
[1] 3 5 7
```

For more complex data structures (like matrices, data frames, and lists), we may be interested in applying a function to each row, column, or component.

5.1 Vectorized Summary Functions

We will start with some generic vectorized functions that provide useful summaries for columns or components of R objects.

5.1.1 The str() Function

For a quick overview of any object in R, the **str()** function returns a compact display of the internal **structure** of the input object. As an example, we will apply this function to the trees data in the datasets package.

```
str(trees) # Display the structure of the trees object
```

```
'data.frame':   31 obs. of  3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
 $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
 $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

The output of the str(trees) command shows that trees is a data frame with 31 observations and 3 variables. A brief summary of each component (column) in trees is given: Each component of trees is numeric (num), and the first few values from each component are printed.

The str() function is well suited for displaying the contents of nested lists (lists inside lists).

```
str(L)
```

```
List of 4
 $ Vector    : int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ Matrix    : int [1:2, 1:3] 1 2 3 4 5 6
 $ Data Frame:'data.frame': 3 obs. of  5 variables:
  ..$ Name  : chr [1:3] "Leslie" "Ron" "April"
  ..$ Height: num [1:3] 62 71 66
  ..$ Weight: num [1:3] 115 201 119
  ..$ Income: num [1:3] 4000 NA 2000
  ..$ Age   : num [1:3] 34 49 20
 $ List      :List of 2
  ..$ : int [1:5] 1 2 3 4 5
  ..$ : int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
```

5.1.2 The summary() Function

We previously used the **summary()** function to compute a few standard summary statistics on numeric vectors.

```
summary(trees$Volume)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  10.20   19.40   24.20   30.17   37.30   77.00 
```

The summary() function is an example of a **polymorphic** function in that it changes its output based on the type of input. Specifically, the output of summary() will depend on the class of the input object.

For data frames, the summary() function will compute summary statistics for each column in the data frame. If the column is a character or factor vector, the summary() output will adapt and return frequencies. For lists, the summary() function will return the length, class attribute, and mode of each component.

```
summary(trees)
```

```
     Girth           Height       Volume     
 Min.   : 8.30   Min.   :63   Min.   :10.20  
 1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40  
 Median :12.90   Median :76   Median :24.20  
 Mean   :13.25   Mean   :76   Mean   :30.17  
 3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30  
 Max.   :20.60   Max.   :87   Max.   :77.00  
```

```
summary(parks_df)
```

```
     Name               Height          Weight        Income    
 Length:3           Min.   :62.00   Min.   :115   Min.   :2000  
 Class :character   1st Qu.:64.00   1st Qu.:117   1st Qu.:2500  
 Mode  :character   Median :66.00   Median :119   Median :3000  
                    Mean   :66.33   Mean   :145   Mean   :3000  
                    3rd Qu.:68.50   3rd Qu.:160   3rd Qu.:3500  
                    Max.   :71.00   Max.   :201   Max.   :4000  
                                                  NA's   :1     
      Age       
 Min.   :20.00  
 1st Qu.:27.00  
 Median :34.00  
 Mean   :34.33  
 3rd Qu.:41.50  
 Max.   :49.00  
```

```
summary(L)
```

```
           Length Class      Mode   
Vector     10     -none-     numeric
Matrix      6     -none-     numeric
Data Frame  5     data.frame list   
List        2     -none-     list   
```

5.2 The apply Family of Functions

One of the most widely used features of R is the **apply** family of functions. The `apply` family consists of vectorized functions that minimize the need to use loops or repetitive code. The most common `apply` functions are `apply()`, `lapply()`, `sapply()`, `vapply()`, and `tapply()`, some of which we have covered in previous chapters. There are other functions in the same family (`mapply()`, `rapply()`, and `eapply()`), but these will not be covered.

5.2.1 The apply() Function

Recall that the **apply()** function is used to apply a function to the rows or columns (the **margins**) of matrices or data frames.

```
# Compute the mean of every column of the trees data frame
apply(trees, 2, mean)
```

```
   Girth   Height   Volume 
13.24839 76.00000 30.17097 
```

```
# Compute the mean of every row of the trees data frame
apply(trees, 1, mean)
```

```
 [1] 29.53333 27.96667 27.33333 32.96667 36.83333 37.83333 30.86667 34.73333
 [9] 37.90000 35.36667 38.16667 36.13333 36.26667 34.00000 35.36667 36.36667
[17] 43.90000 42.23333 36.80000 34.23333 42.16667 41.96667 41.60000 42.10000
[25] 45.30000 51.23333 51.73333 52.06667 49.83333 49.66667 61.53333
```

```
# Compute the range (min and max) of every column of the trees data frame
apply(trees, 2, range)
```

```
     Girth Height Volume
[1,]   8.3     63   10.2
[2,]  20.6     87   77.0
```

Note: Remember that the output of `apply()` will be a matrix if the applied function returns a vector with more than one element.

Caution: Use caution when using `apply()` to a data frame. Ideally, the columns of the data frame should all be of the same type. The `apply()` function is intended for matrices (and arrays, which are higher dimensional versions of matrices). Using `apply()` on a data frame will first coerce the data frame into a matrix with `as.matrix()` before applying the function in the `FUN` argument.

Question: What does `apply(parks_df, 2, mean)` output? Why does this command not give the results we intended? How can we find the mean of each of the numeric columns in `parks_df` using `apply()`?

Question: How is `summary(trees)` different from `apply(trees, 2, summary)`?

5.2.2 The `lapply()` Function

The `lapply()` function is used to apply a function to each component of a list (`lapply` is short for "list apply"). The output of `lapply()` will be a list.

The syntax of `lapply()` is `lapply(X, FUN, ...)`, where the arguments are:

- `X`: A list
- `FUN`: The function to be applied.
- `...`: Any optional arguments to be passed to the `FUN` function.

Note that there is no margin argument like in `apply()`, as lists have a single index.

```
# Return the length of each component in the L list
lapply(L, length)
```

```
$Vector
[1] 10

$Matrix
[1] 6

$`Data Frame`
[1] 5

$List
[1] 2
```

Note: Since data frames are (stored as) lists, `lapply()` also works for data frames.

```
# Compute the range (min and max) of every column of the trees data frame
lapply(trees, range)
```

```
$Girth
[1]  8.3 20.6

$Height
[1] 63 87

$Volume
[1] 10.2 77.0
```

Question: How is `apply(trees, 2, range)` different from `lapply(trees, range)`?

The list output from `lapply()` is particularly useful when the result from each component may have a different length (or even a different dimension or class).

```
which_median <- function(x) {
  which(x == median(x))
}
lapply(trees, which_median)
```

```
$Girth
[1] 16 17

$Height
[1] 12 13

$Volume
[1] 11
```

5.2.3 The `sapply()` Function

The output that is returned from `lapply()` is always a list, with the same number of components as the input list. In many cases, the output could be simplified to a vector or matrix.

The **`sapply()`** function is a **wrapper** function for `lapply()`, meaning that `sapply()` actually internally calls `lapply()` to apply a function to each component of a list. The only difference is that `sapply()` will try to simplify the output from `lapply()` whenever possible (`sapply` is short for "simplified [l]apply"). In particular:

- If the result is a list where every component is a vector of length 1 (i.e., a scalar), then `sapply()` will return a vector.
- If the result is a list where every component is a vector of the same length (greater than 1), then `sapply()` will return a matrix.
- If the result is a list where every component is not a vector of the same length, then `sapply()` will return a list (i.e., the same output as from `lapply()`.)

By using `lapply()`, we found the length of each component of list `L`. Notice the difference when using `sapply()`.

```
sapply(L, length)
```

```
    Vector     Matrix Data Frame       List 
        10          6          5          2 
```

If the output of each application of the `length()` function is a single character value, so `sapply()` returns a vector.

```
sapply(trees, range)
```

```
     Girth Height Volume
[1,]   8.3     63   10.2
[2,]  20.6     87   77.0
```

Note: Notice that `sapply(trees, range)` gives the same output as `apply(trees, 2, range)`. Since a data frame is stored as a list with the column vectors as its components, `sapply()` applies functions to the components of `trees` as a list, and `apply()` with `MARGIN = 2` applies functions to the columns of the coerced matrix version of `trees` (`as.matrix(trees)`). The output is the same in this case. However, since lapply() and sapply() do not coerce data frames into having columns of the same type, certain functions may produce different results.

Question: How is apply(parks_df, 2, mean) different from sapply(parks_df, mean)?

5.2.4 The vapply() Function

Recall that the **vapply()** function applies a function to each element of an atomic vector. Since lists are generic vectors, the vapply() function can also be used to apply a function to each component of a list.

The vapply() function is similar to sapply(), except that it requires the FUN.VALUE argument that specifies the type of return value you expect the FUN function to output.

For example, since we know the class() function returns a single character value for trees, we would set FUN.VALUE = character(1).

```
vapply(trees, class, character(1))
```

```
    Girth    Height    Volume
"numeric" "numeric" "numeric"
```

Since the range() function returns a numeric vector of length 2, we would set FUN.VALUE = numeric(2).

```
vapply(trees, range, numeric(2))
```

```
     Girth Height Volume
[1,]   8.3     63   10.2
[2,]  20.6     87   77.0
```

Remember that vapply() will throw an error if the FUN.VALUE is set to a return type that is not what we are expecting.

```
vapply(trees, mean, numeric(2))
```

```
Error in vapply(trees, mean, numeric(2)): values must be length 2,
 but FUN(X[[1]]) result is length 1
```

A natural question that arises may be to ask why one would prefer vapply() over sapply(). At first glance, it appears that sapply() is more flexible and easier to use than vapply(). However, the flexibility of sapply() makes it dangerous when trying to ensure that your output has a specific length, dimension, and/or type.

The strictness of vapply() imposed by requiring the FUN.VALUE argument helps in making sure that your output has exactly the structure you expect. The error that vapply() can throw is meant to alert the user of unexpected results. When a function has a predictable output structure, it is generally safer and thus often preferred to use vapply() over sapply().