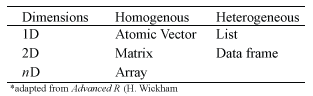
Managing Data Structures in R

*"Smart data structures and dumb code works a lot better than the other way around"* - Eric S. Raymond

In the previous section I illustrated how to work with different types of data; however, we primarily focused on data in a one-dimensional structure. In typical data analyses you often need more than one dimension. Many datasets can contain variables of different length and or types of values (i.e. numeric vs character). Furthermore, many statistical and mathematical calculations are based on matrices. R provides multiple types of data structures to deal with these different needs.

The basic data structures in R can be organized by their dimensionality (1D, 2D, ..., *n*D) and their "likeness" (homogenous vs. heterogeneous). This results in five data structure types most often used in data analysis; and almost all other objects in R are built from these foundational types:



Basic Data Structures in R

In this section I will cover the basics of these data structures. I have not had the need to use multi-dimensional arrays, therefore, the topics I will go into details on will include [vectors](#managing_vectors), [lists](#managing_lists), [matrices](#managing_matrices), and [data frames](#managing_dataframes). These types represent the most commonly used data structures for day-to-day analyses. For each data structure I will illustrate how to create the structure, add additional elements to a pre-existing structure, add attributes to structures, and how to subset the various data structures. Lastly, I will cover how to [deal with missing values](#managing_missing_values) in data structures. Consequently, this section will provide a robust understanding of managing various forms of datasets depending on dimensionality needs.

# Data Structure Basics

Prior to jumping into the data structures, it's beneficial to understand two components of data structures - the structure and attributes.

## Identifying the Structure

Given an object, the best way to understand what data structure it represents is to use the structure function str(). str() stands for structure and provides a compact display of the internal **str**ucture of an R object.

# different data structures  
vector <- 1:10  
list <- list(item1 = 1:10, item2 = LETTERS[1:18])  
matrix <- matrix(1:12, nrow = 4)   
df <- data.frame(item1 = 1:18, item2 = LETTERS[1:18])  
  
# identify the structure of each object  
str(vector)  
## int [1:10] 1 2 3 4 5 6 7 8 9 10  
  
str(list)  
## List of 2  
## $ item1: int [1:10] 1 2 3 4 5 6 7 8 9 10  
## $ item2: chr [1:18] "A" "B" "C" "D" ...  
  
str(matrix)  
## int [1:4, 1:3] 1 2 3 4 5 6 7 8 9 10 ...  
  
str(df)  
## 'data.frame': 18 obs. of 2 variables:  
## $ item1: int 1 2 3 4 5 6 7 8 9 10 ...  
## $ item2: Factor w/ 18 levels "A","B","C","D",..: 1 2 3 4 5 6 7 8 9 10 ...

## Attributes

R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are:

* names, dimnames
* dimensions (e.g. matrices, arrays)
* class (e.g. integer, numeric)
* length
* other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the attributes() function. Not all R objects contain attributes, in which case the attributes() function returns NULL.

# assess attributes of an object  
attributes(df)  
## $names  
## [1] "item1" "item2"  
##   
## $row.names  
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18  
##   
## $class  
## [1] "data.frame"  
  
attributes(matrix)  
## $dim  
## [1] 4 3  
  
# assess names of an object  
names(df)  
## [1] "item1" "item2"  
  
# assess the dimensions of an object  
dim(matrix)  
## [1] 4 3  
  
# assess the class of an object  
class(list)  
## [1] "list"  
  
# access the length of an object  
length(vector)  
## [1] 10  
  
# note that length will measure the number of items in  
# a list or number of columns in a data frame  
length(list)  
## [1] 2  
  
length(df)  
## [1] 2

This chapter only shows you functions to assess these attributes. In the chapters that follow more details are provided on how to view and create attributes for each type of data structure.

# Managing Vectors

The basic structure in R is the vector. A vector is a sequence of data elements of the same basic type: [integer](#integer_vs_double), [double](#integer_vs_double), logical, or [character](#character_basics).[[1]](#footnote-26) The one-dimensional examples illustrated in the previous section are considered vectors. In this chapter I will illustrate how to [create vectors](#vector_create), [add additional elements to pre-existing vectors](#vector_add), [add attributes to vectors](#vector_attributes), and [subset vectors](#vector_subset).

## Creating

The colon : operator can be used to create a vector of integers between two specified numbers or the c() function can be used to create vectors of objects by concatenating elements together:

# integer vector  
w <- 8:17  
w  
## [1] 8 9 10 11 12 13 14 15 16 17  
  
# double vector  
x <- c(0.5, 0.6, 0.2)  
x  
## [1] 0.5 0.6 0.2  
  
# logical vector  
y1 <- c(TRUE, FALSE, FALSE)  
y1  
## [1] TRUE FALSE FALSE  
  
# logical vector in shorthand  
y2 <- c(T, F, F)   
y2  
## [1] TRUE FALSE FALSE  
  
# Character vector  
z <- c("a", "b", "c")   
z  
## [1] "a" "b" "c"

You can also use the as.vector() function to initialize vectors or change the vector type:

v <- as.vector(8:17)  
v  
## [1] 8 9 10 11 12 13 14 15 16 17  
  
# turn numerical vector to character  
as.vector(v, mode = "character")  
## [1] "8" "9" "10" "11" "12" "13" "14" "15" "16" "17"

All elements of a vector must be the same type, so when you attempt to combine different types of elements they will be coerced to the most flexible type possible:

# numerics are turned to characters  
str(c("a", "b", "c", 1, 2, 3))  
## chr [1:6] "a" "b" "c" "1" "2" "3"  
  
# logical are turned to numerics...  
str(c(1, 2, 3, TRUE, FALSE))  
## num [1:5] 1 2 3 1 0  
  
# or character  
str(c("A", "B", "C", TRUE, FALSE))  
## chr [1:5] "A" "B" "C" "TRUE" "FALSE"

## Adding on to

To add additional elements to a pre-existing vector we can continue to leverage the c() function. Also, note that vectors are always flat so nested c() functions will not add additional dimensions to the vector:

v1 <- 8:17  
  
c(v1, 18:22)  
## [1] 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22  
  
# same as  
c(v1, c(18, c(19, c(20, c(21:22)))))  
## [1] 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

## Adding attributes

The attributes that you can add to vectors includes names and comments. If we continue with our vector v1 we can see that the vector currently has no attributes:

attributes(v1)  
## NULL

We can add names to vectors using two approaches. The first uses names() to assign names to each element of the vector. The second approach is to assign names when creating the vector.

# assigning names to a pre-existing vector  
names(v1) <- letters[1:length(v1)]  
v1  
## a b c d e f g h i j   
## 8 9 10 11 12 13 14 15 16 17  
attributes(v1)  
## $names  
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"  
  
# adding names when creating vectors  
v2 <- c(name1 = 1, name2 = 2, name3 = 3)  
v2  
## name1 name2 name3   
## 1 2 3  
attributes(v2)  
## $names  
## [1] "name1" "name2" "name3"

We can also add comments to vectors to act as a note to the user. This does not change how the vector behaves; rather, it simply acts as a form of metadata for the vector.

comment(v1) <- "This is a comment on a vector"  
v1  
## a b c d e f g h i j   
## 8 9 10 11 12 13 14 15 16 17  
attributes(v1)  
## $names  
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"  
##   
## $comment  
## [1] "This is a comment on a vector"

## Subsetting

The four main ways to subset a vector include combining square brackets [ ] with:

* [Positive integers](#vector_positive)
* [Negative integers](#vector_negative)
* [Logical values](#vector_logical)
* [Names](#vector_names)

You can also subset with double brackets [[ ]] for [simplifying](#vector_simplify) subsets.

### Subsetting with positive integers

Subsetting with positive integers returns the elements at the specified positions:

v1  
## a b c d e f g h i j   
## 8 9 10 11 12 13 14 15 16 17  
  
v1[2]  
## b   
## 9  
  
v1[2:4]  
## b c d   
## 9 10 11  
  
v1[c(2, 4, 6, 8)]  
## b d f h   
## 9 11 13 15  
  
# note that you can duplicate index positions  
v1[c(2, 2, 4)]  
## b b d   
## 9 9 11

### Subsetting with negative integers

Subsetting with negative integers will omit the elements at the specified positions:

v1[-1]  
## b c d e f g h i j   
## 9 10 11 12 13 14 15 16 17

v1[-c(2, 4, 6, 8)]  
## a c e g i j   
## 8 10 12 14 16 17

### Subsetting with logical values

Subsetting with logical values will select the elements where the corresponding logical value is TRUE:

v1[c(TRUE, FALSE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE)]  
## a c e f g j   
## 8 10 12 13 14 17  
  
v1[v1 < 12]  
## a b c d   
## 8 9 10 11  
  
v1[v1 < 12 | v1 > 15]  
## a b c d i j   
## 8 9 10 11 16 17  
  
# if logical vector is shorter than the length of the vector being  
# subsetted, it will be recycled to be the same length  
v1[c(TRUE, FALSE)]  
## a c e g i   
## 8 10 12 14 16

### Subsetting with names

Subsetting with names will return the elements with the matching names specified:

v1["b"]  
## b   
## 9  
  
v1[c("a", "c", "h")]  
## a c h   
## 8 10 15

### Simplifying vs. Preserving

Its also important to understand the difference between simplifying and preserving when subsetting. **Simplifying** subsets returns the simplest possible data structure that can represent the output. **Preserving** subsets keeps the structure of the output the same as the input.

For vectors, subsetting with single brackets [ ] preserves while subsetting with double brackets [[ ]] simplifies. The change you will notice when simplifying vectors is the removal of names.

v1[1]  
## a   
## 8  
  
v1[[1]]  
## [1] 8

# Managing Lists

A list is an R structure that allows you to combine elements of different types, including lists embedded in a list, and length. Many statistical outputs are provided as a list as well; therefore, its critical to understand how to work with lists. In this chapter I will illustrate how to [create lists](#lists_create), [add additional elements to pre-existing lists](#lists_add), [add attributes to lists](#lists_attribute), and [subset lists](#lists_subset).

## Creating

To create a list we can use the list() function. Note how each of the four list items are of different classes (integer, character, logical, and numeric) and different length.

l <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.5, 4.2))  
str(l)  
## List of 4  
## $ : int [1:3] 1 2 3  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE  
## $ : num [1:2] 2.5 4.2  
  
# a list containing a list  
l <- list(1:3, list(letters[1:5], c(TRUE, FALSE, TRUE)))  
str(l)  
## List of 2  
## $ : int [1:3] 1 2 3  
## $ :List of 2  
## ..$ : chr [1:5] "a" "b" "c" "d" ...  
## ..$ : logi [1:3] TRUE FALSE TRUE

## Adding on to

To add additional list components to a list we can leverage the list() and append() functions. We can illustrate with the following list.

l1 <- list(1:3, "a", c(TRUE, FALSE, TRUE))  
str(l1)  
## List of 3  
## $ : int [1:3] 1 2 3  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE

If we add the new elements with list() it will create a list of two components, component 1 will be a nested list of the original list and component 2 will be the new elements added:

l2 <- list(l1, c(2.5, 4.2))  
str(l2)  
## List of 2  
## $ :List of 3  
## ..$ : int [1:3] 1 2 3  
## ..$ : chr "a"  
## ..$ : logi [1:3] TRUE FALSE TRUE  
## $ : num [1:2] 2.5 4.2

To simply add a 4th list component without creating nested lists we use the append() function:

l3 <- append(l1, list(c(2.5, 4.2)))  
str(l3)  
## List of 4  
## $ : int [1:3] 1 2 3  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE  
## $ : num [1:2] 2.5 4.2

Alternatively, we can also add a new list component by utilizing the '$' sign and naming the new item:

l3$item4 <- "new list item"  
str(l3)  
## List of 5  
## $ : int [1:3] 1 2 3  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE  
## $ : num [1:2] 2.5 4.2  
## $ item4: chr "new list item"

To add individual elements to a specific list component we need to introduce some subsetting which is further discussed later in the chapter in the [Subsetting section](#lists_subset). We'll continue with our original l1 list:

str(l1)  
## List of 3  
## $ : int [1:3] 1 2 3  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE

To add additional values to a list item you need to subset for that specific list item and then you can use the c() function to add the additional elements to that list item:

l1[[1]] <- c(l1[[1]], 4:6)  
str(l1)  
## List of 3  
## $ : int [1:6] 1 2 3 4 5 6  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE  
  
l1[[2]] <- c(l1[[2]], c("dding", "to a", "list"))  
str(l1)  
## List of 3  
## $ : int [1:6] 1 2 3 4 5 6  
## $ : chr [1:4] "a" "dding" "to a" "list"  
## $ : logi [1:3] TRUE FALSE TRUE

## Adding attributes

The attributes that you can add to lists include names, general comments, and specific list item comments. Currently, our l1 list has no attributes:

attributes(l1)  
## NULL

We can add names to lists in two ways. First, we can use names() to assign names to list items in a pre-existing list. Second, we can add names to a list when we are creating a list.

# adding names to a pre-existing list  
names(l1) <- c("item1", "item2", "item3")  
str(l1)  
## List of 3  
## $ item1: int [1:6] 1 2 3 4 5 6  
## $ item2: chr [1:4] "a" "dding" "to a" "list"  
## $ item3: logi [1:3] TRUE FALSE TRUE  
attributes(l1)  
## $names  
## [1] "item1" "item2" "item3"  
  
# adding names when creating lists  
l2 <- list(item1 = 1:3, item2 = letters[1:5], item3 = c(T, F, T, T))  
str(l2)  
## List of 3  
## $ item1: int [1:3] 1 2 3  
## $ item2: chr [1:5] "a" "b" "c" "d" ...  
## $ item3: logi [1:4] TRUE FALSE TRUE TRUE  
attributes(l2)  
## $names  
## [1] "item1" "item2" "item3"

We can also add comments to lists. As previously mentioned, comments act as a note to the user without changing how the object behaves. With lists, we can add a general comment to the list using comment() and we can also add comments to specific list items with attr().

# adding a general comment to list l2 with comment()  
comment(l2) <- "This is a comment on a list"  
str(l2)  
## List of 3  
## $ item1: int [1:3] 1 2 3  
## $ item2: chr [1:5] "a" "b" "c" "d" ...  
## $ item3: logi [1:4] TRUE FALSE TRUE TRUE  
## - attr(\*, "comment")= chr "This is a comment on a list"  
attributes(l2)  
## $names  
## [1] "item1" "item2" "item3"  
##   
## $comment  
## [1] "This is a comment on a list"  
  
# adding a comment to a specific list item with attr()   
attr(l2, "item2") <- "Comment for item2"  
str(l2)  
## List of 3  
## $ item1: int [1:3] 1 2 3  
## $ item2: chr [1:5] "a" "b" "c" "d" ...  
## $ item3: logi [1:4] TRUE FALSE TRUE TRUE  
## - attr(\*, "comment")= chr "This is a comment on a list"  
## - attr(\*, "item2")= chr "Comment for item2"  
attributes(l2)  
## $names  
## [1] "item1" "item2" "item3"  
##   
## $comment  
## [1] "This is a comment on a list"  
##   
## $item2  
## [1] "Comment for item2"

## Subsetting

*"If list x is a train carrying objects, then x[[5]] is the object in car 5; x[4:6] is a train of cars 4-6"* - @RLangTip

To subset lists we can utilize the single bracket [ ], double brackets [[ ]], and dollar sign $ operators. Each approach provides a specific purpose and can be combined in different ways to achieve the following subsetting objectives:

* [Subset list and preserve output as a list](#list_subset_preserve)
* [Subset list and simplify output](#list_subset_simplify)
* [Subset list to get elements out of a list](#list_subset_out)
* [Subset list with a nested list](#list_subset_nested)

### Subset list and preserve output as a list

To extract one or more list items while **preserving**[[2]](#footnote-42) the output in list format use the [ ] operator:

# extract first list item  
l2[1]  
## $item1  
## [1] 1 2 3  
  
# same as above but using the item's name  
l2["item1"]  
## $item1  
## [1] 1 2 3  
  
# extract multiple list items  
l2[c(1,3)]  
## $item1  
## [1] 1 2 3  
##   
## $item3  
## [1] TRUE FALSE TRUE TRUE  
  
# same as above but using the items' names  
l2[c("item1", "item3")]  
## $item1  
## [1] 1 2 3  
##   
## $item3  
## [1] TRUE FALSE TRUE TRUE

### Subset list and simplify output

To extract one or more list items while **simplifying**[[3]](#footnote-45) the output use the [[ ]] or $ operator:

# extract first list item and simplify to a vector  
l2[[1]]  
## [1] 1 2 3  
  
# same as above but using the item's name  
l2[["item1"]]  
## [1] 1 2 3  
  
# same as above but using the `$` operator  
l2$item1  
## [1] 1 2 3

One thing that differentiates the [[ operator from the $ is that the [[ operator can be used with computed indices. The $ operator can only be used with literal names.

### Subset list to get elements out of a list

To extract individual elements out of a specific list item combine the [[ (or $) operator with the [ operator:

# extract third element from the second list item  
l2[[2]][3]  
## [1] "c"  
  
# same as above but using the item's name  
l2[["item2"]][3]  
## [1] "c"  
  
# same as above but using the `$` operator  
l2$item2[3]  
## [1] "c"

### Subset list with a nested list

If you have nested lists you can expand the ideas above to extract items and elements. We'll use the following list l3 which has a nested list in item 2.

l3 <- list(item1 = 1:3,   
 item2 = list(item2a = letters[1:5],   
 item3b = c(T, F, T, T)))  
str(l3)  
## List of 2  
## $ item1: int [1:3] 1 2 3  
## $ item2:List of 2  
## ..$ item2a: chr [1:5] "a" "b" "c" "d" ...  
## ..$ item3b: logi [1:4] TRUE FALSE TRUE TRUE

If the goal is to subset l3 to extract the nested list item item2a from item2, we can perform this multiple ways.

# preserve the output as a list  
l3[[2]][1]  
## $item2a  
## [1] "a" "b" "c" "d" "e"  
  
# same as above but simplify the output  
l3[[2]][[1]]  
## [1] "a" "b" "c" "d" "e"  
  
# same as above with names  
l3[["item2"]][["item2a"]]  
## [1] "a" "b" "c" "d" "e"  
  
# same as above with `$` operator  
l3$item2$item2a  
## [1] "a" "b" "c" "d" "e"  
  
# extract individual element from a nested list item  
l3[[2]][[1]][3]  
## [1] "c"

# Managing Matrices

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. In R, the elements that make up a matrix must be of a consistant mode (i.e. all elements must be numeric, or character, etc.). Therefore, a matrix can be thought of as an atomic vector with a dimension attribute. Furthermore, all rows of a matrix must be of same length. In this chapter I will illustrate how to [create matrices](#matrix_create), [add additional elements to pre-existing matrices](#matrix_add), [add attributes to matrices](#matrix_attribute), and [subset matrices](#matrix_subset).

## Creating

Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" corner and running down the columns. We can create a matrix using the matrix() function and specifying the values to fill in the matrix and the number of rows and columns to make the matrix.

# numeric matrix  
m1 <- matrix(1:6, nrow = 2, ncol = 3)  
m1  
## [,1] [,2] [,3]  
## [1,] 1 3 5  
## [2,] 2 4 6

The underlying structure of this matrix is simply an integer vector with an added 2x3 dimension attribute.

str(m1)  
## int [1:2, 1:3] 1 2 3 4 5 6  
attributes(m1)  
## $dim  
## [1] 2 3

Matrices can also contain character values. Whether a matrix contains data that are of numeric or character type, all the elements must be of the same class.

# a character matrix  
m2 <- matrix(letters[1:6], nrow = 2, ncol = 3)  
m2  
## [,1] [,2] [,3]  
## [1,] "a" "c" "e"   
## [2,] "b" "d" "f"  
  
# structure of m2 is simply character vector with 2x3 dimension  
str(m2)  
## chr [1:2, 1:3] "a" "b" "c" "d" "e" "f"  
attributes(m2)  
## $dim  
## [1] 2 3

Matrices can also be created using the column-bind cbind() and row-bind rbind() functions. However, keep in mind that the vectors that are being binded must be of equal length and mode.

v1 <- 1:4  
v2 <- 5:8  
  
cbind(v1, v2)  
## v1 v2  
## [1,] 1 5  
## [2,] 2 6  
## [3,] 3 7  
## [4,] 4 8  
  
rbind(v1, v2)  
## [,1] [,2] [,3] [,4]  
## v1 1 2 3 4  
## v2 5 6 7 8  
  
# bind several vectors together  
v3 <- 9:12  
  
cbind(v1, v2, v3)  
## v1 v2 v3  
## [1,] 1 5 9  
## [2,] 2 6 10  
## [3,] 3 7 11  
## [4,] 4 8 12

## Adding on to

We can leverage the cbind() and rbind() functions for adding onto matrices as well. Again, its important to keep in mind that the vectors that are being binded must be of equal length and mode to the pre-existing matrix.

m1 <- cbind(v1, v2)  
m1  
## v1 v2  
## [1,] 1 5  
## [2,] 2 6  
## [3,] 3 7  
## [4,] 4 8  
  
# add a new column  
cbind(m1, v3)  
## v1 v2 v3  
## [1,] 1 5 9  
## [2,] 2 6 10  
## [3,] 3 7 11  
## [4,] 4 8 12  
  
# or add a new row  
rbind(m1, c(4.1, 8.1))  
## v1 v2  
## [1,] 1.0 5.0  
## [2,] 2.0 6.0  
## [3,] 3.0 7.0  
## [4,] 4.0 8.0  
## [5,] 4.1 8.1

## Adding attributes

As previously mentioned, matrices by default will have a dimension attribute as illustrated in the following matrix m2.

# basic matrix  
m2 <- matrix(1:12, nrow = 4, ncol = 3)  
m2  
## [,1] [,2] [,3]  
## [1,] 1 5 9  
## [2,] 2 6 10  
## [3,] 3 7 11  
## [4,] 4 8 12  
  
# the dimension attribute shows this matrix has 4 rows and 3 columns  
attributes(m2)  
## $dim  
## [1] 4 3

However, matrices can also have additional attributes such as row names, column names, and comments. Adding names can be done individually, meaning we can add row names or column names separately.

# add row names as an attribute  
rownames(m2) <- c("row1", "row2", "row3", "row4")  
m2  
## [,1] [,2] [,3]  
## row1 1 5 9  
## row2 2 6 10  
## row3 3 7 11  
## row4 4 8 12  
  
# attributes displayed will now show the dimension, list the row names  
# and will show the column names as NULL  
attributes(m2)  
## $dim  
## [1] 4 3  
##   
## $dimnames  
## $dimnames[[1]]  
## [1] "row1" "row2" "row3" "row4"  
##   
## $dimnames[[2]]  
## NULL  
  
# add column names  
colnames(m2) <- c("col1", "col2", "col3")  
m2  
## col1 col2 col3  
## row1 1 5 9  
## row2 2 6 10  
## row3 3 7 11  
## row4 4 8 12  
attributes(m2)  
## $dim  
## [1] 4 3  
##   
## $dimnames  
## $dimnames[[1]]  
## [1] "row1" "row2" "row3" "row4"  
##   
## $dimnames[[2]]  
## [1] "col1" "col2" "col3"

Another option is to use the dimnames() function. To add row names you assign the names to dimnames(m2)[[1]] and to add column names you assign the names to dimnames(m2)[[2]].

dimnames(m2)[[1]] <- c("row\_1", "row\_2", "row\_3", "row\_4")  
m2  
## col1 col2 col3  
## row\_1 1 5 9  
## row\_2 2 6 10  
## row\_3 3 7 11  
## row\_4 4 8 12  
  
# column names are contained in the second list item  
dimnames(m2)[[2]] <- c("col\_1", "col\_2", "col\_3")  
m2  
## col\_1 col\_2 col\_3  
## row\_1 1 5 9  
## row\_2 2 6 10  
## row\_3 3 7 11  
## row\_4 4 8 12

Lastly, similar to lists and vectors you can add a comment attribute to a list.

comment(m2) <- "adding a comment to a matrix"  
attributes(m2)  
## $dim  
## [1] 4 3  
##   
## $dimnames  
## $dimnames[[1]]  
## [1] "row\_1" "row\_2" "row\_3" "row\_4"  
##   
## $dimnames[[2]]  
## [1] "col\_1" "col\_2" "col\_3"  
##   
##   
## $comment  
## [1] "adding a comment to a matrix"

## Subsetting

To subset matrices we use the [ operator; however, since matrices have 2 dimensions we need to incorporate subsetting arguments for both row and column dimensions. A generic form of matrix subsetting looks like: matrix[rows, columns]. We can illustrate with matrix m2:

m2  
## col\_1 col\_2 col\_3  
## row\_1 1 5 9  
## row\_2 2 6 10  
## row\_3 3 7 11  
## row\_4 4 8 12

By using different values in the rows and columns argument of m2[rows, columns], we can subset m2 in multiple ways.

# subset for rows 1 and 2 but keep all columns  
m2[1:2, ]  
## col\_1 col\_2 col\_3  
## row\_1 1 5 9  
## row\_2 2 6 10  
  
# subset for columns 1 and 3 but keep all rows  
m2[ , c(1, 3)]  
## col\_1 col\_3  
## row\_1 1 9  
## row\_2 2 10  
## row\_3 3 11  
## row\_4 4 12  
  
# subset for both rows and columns  
m2[1:2, c(1, 3)]  
## col\_1 col\_3  
## row\_1 1 9  
## row\_2 2 10  
  
# use a vector to subset  
v <- c(1, 2, 4)  
m2[v, c(1, 3)]  
## col\_1 col\_3  
## row\_1 1 9  
## row\_2 2 10  
## row\_4 4 12  
  
# use names to subset  
m2[c("row\_1", "row\_3"), ]  
## col\_1 col\_2 col\_3  
## row\_1 1 5 9  
## row\_3 3 7 11

Note that subsetting matrices with the [ operator will simplify the results to the lowest possible dimension. To avoid this you can introduce the drop = FALSE argument:

# simplifying results in a named vector  
m2[, 2]  
## row\_1 row\_2 row\_3 row\_4   
## 5 6 7 8  
  
# preserving results in a 4x1 matrix  
m2[, 2, drop = FALSE]  
## col\_2  
## row\_1 5  
## row\_2 6  
## row\_3 7  
## row\_4 8

# Managing Data Frames

A data frame is the most common way of storing data in R and, generally, is the data structure most often used for data analyses. Under the hood, a data frame is a list of equal-length vectors. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows. As a result, data frames can store different classes of objects in each column (i.e. numeric, character, factor). In essence, the easiest way to think of a data frame is as an Excel worksheet that contains columns of different types of data but are all of equal length rows. In this chapter I will illustrate how to [create data frames](#dataframes_create), [add additional elements to pre-existing data frames](#dataframes_add), [add attributes to data frames](#dataframes_attribute), and [subset data frames](#dataframes_subset).

## Creating

Data frames are usually created by reading in a dataset using the read.table() or read.csv(); this will be covered in the [importing and scraping data chapters](#import_scrape_export). However, data frames can also be created explicitly with the data.frame() function or they can be coerced from other types of objects like lists. In this case I'll create a simple data frame df and assess its basic structure:

df <- data.frame(col1 = 1:3,   
 col2 = c("this", "is", "text"),   
 col3 = c(TRUE, FALSE, TRUE),   
 col4 = c(2.5, 4.2, pi))  
  
# assess the structure of a data frame  
str(df)  
## 'data.frame': 3 obs. of 4 variables:  
## $ col1: int 1 2 3  
## $ col2: Factor w/ 3 levels "is","text","this": 3 1 2  
## $ col3: logi TRUE FALSE TRUE  
## $ col4: num 2.5 4.2 3.14  
  
# number of rows  
nrow(df)  
## [1] 3  
  
# number of columns  
ncol(df)  
## [1] 4

Note how col2 in df was converted to a column of factors. This is because there is a default setting in data.frame() that converts character columns to factors. We can turn this off by setting the stringsAsFactors = FALSE argument:

df <- data.frame(col1 = 1:3,   
 col2 = c("this", "is", "text"),   
 col3 = c(TRUE, FALSE, TRUE),   
 col4 = c(2.5, 4.2, pi),   
 stringsAsFactors = FALSE)  
  
# note how col2 now is of a character class  
str(df)  
## 'data.frame': 3 obs. of 4 variables:  
## $ col1: int 1 2 3  
## $ col2: chr "this" "is" "text"  
## $ col3: logi TRUE FALSE TRUE  
## $ col4: num 2.5 4.2 3.14

We can also convert pre-existing structures to a data frame. The following illustrates how we can turn multiple vectors, a list, or a matrix into a data frame:

v1 <- 1:3  
v2 <-c("this", "is", "text")  
v3 <- c(TRUE, FALSE, TRUE)  
  
# convert same length vectors to a data frame using data.frame()  
data.frame(col1 = v1, col2 = v2, col3 = v3)  
## col1 col2 col3  
## 1 1 this TRUE  
## 2 2 is FALSE  
## 3 3 text TRUE  
  
# convert a list to a data frame using as.data.frame()  
l <- list(item1 = 1:3, item2 = c("this", "is", "text"), item3 = c(2.5, 4.2, 5.1))  
l  
## $item1  
## [1] 1 2 3  
##   
## $item2  
## [1] "this" "is" "text"  
##   
## $item3  
## [1] 2.5 4.2 5.1  
  
as.data.frame(l)  
## item1 item2 item3  
## 1 1 this 2.5  
## 2 2 is 4.2  
## 3 3 text 5.1  
  
# convert a matrix to a data frame using as.data.frame()  
m1 <- matrix(1:12, nrow = 4, ncol = 3)  
m1  
## [,1] [,2] [,3]  
## [1,] 1 5 9  
## [2,] 2 6 10  
## [3,] 3 7 11  
## [4,] 4 8 12  
  
as.data.frame(m1)  
## V1 V2 V3  
## 1 1 5 9  
## 2 2 6 10  
## 3 3 7 11  
## 4 4 8 12

## Adding on to

We can leverage the cbind() function for adding columns to a data frame. Note that one of the objects being combined must already be a data frame otherwise cbind() could produce a matrix.

df  
## col1 col2 col3 col4  
## 1 1 this TRUE 2.500000  
## 2 2 is FALSE 4.200000  
## 3 3 text TRUE 3.141593  
  
# add a new column  
v4 <- c("A", "B", "C")  
cbind(df, v4)  
## col1 col2 col3 col4 v4  
## 1 1 this TRUE 2.500000 A  
## 2 2 is FALSE 4.200000 B  
## 3 3 text TRUE 3.141593 C

We can also use the rbind() function to add data frame rows together. However, severe caution should be taken because this can cause changes in the classes of the columns. For instance, our data frame df currently consists of an integer, character, logical, and numeric variables.

df  
## col1 col2 col3 col4  
## 1 1 this TRUE 2.500000  
## 2 2 is FALSE 4.200000  
## 3 3 text TRUE 3.141593  
str(df)  
## 'data.frame': 3 obs. of 4 variables:  
## $ col1: int 1 2 3  
## $ col2: chr "this" "is" "text"  
## $ col3: logi TRUE FALSE TRUE  
## $ col4: num 2.5 4.2 3.14

If we attempt to add a row using rbind() and c() it converts all columns to a character class. This is because all elements in the vector created by c() must be of the same class so they are all coerced to the character class which coerces all the variables in the data frame to the character class.

df2 <- rbind(df, c(4, "R", F, 1.1))  
df2  
## col1 col2 col3 col4  
## 1 1 this TRUE 2.5  
## 2 2 is FALSE 4.2  
## 3 3 text TRUE 3.14159265358979  
## 4 4 R FALSE 1.1  
str(df2)  
## 'data.frame': 4 obs. of 4 variables:  
## $ col1: chr "1" "2" "3" "4"  
## $ col2: chr "this" "is" "text" "R"  
## $ col3: chr "TRUE" "FALSE" "TRUE" "FALSE"  
## $ col4: chr "2.5" "4.2" "3.14159265358979" "1.1"

To add rows appropriately, we need to convert the items being added to a data frame and make sure the columns are the same class as the original data frame.

adding\_df <- data.frame(col1 = 4, col2 = "R", col3 = FALSE, col4 = 1.1,   
 stringsAsFactors = FALSE)  
  
df3 <- rbind(df, adding\_df)  
df3  
## col1 col2 col3 col4  
## 1 1 this TRUE 2.500000  
## 2 2 is FALSE 4.200000  
## 3 3 text TRUE 3.141593  
## 4 4 R FALSE 1.100000  
str(df3)  
## 'data.frame': 4 obs. of 4 variables:  
## $ col1: num 1 2 3 4  
## $ col2: chr "this" "is" "text" "R"  
## $ col3: logi TRUE FALSE TRUE FALSE  
## $ col4: num 2.5 4.2 3.14 1.1

There are better ways to join data frames together than to use cbind() and rbind(). These are covered later on in the [transforming your data with dplyr](#dplyr) chapter.

## Adding attributes

Similar to matrices, data frames will have a dimension attribute. In addition, data frames can also have additional attributes such as row names, column names, and comments. We can illustrate with data frame df.

# basic matrix  
df  
## col1 col2 col3 col4  
## 1 1 this TRUE 2.500000  
## 2 2 is FALSE 4.200000  
## 3 3 text TRUE 3.141593  
dim(df)  
## [1] 3 4  
attributes(df)  
## $names  
## [1] "col1" "col2" "col3" "col4"  
##   
## $row.names  
## [1] 1 2 3  
##   
## $class  
## [1] "data.frame"

Currently df does not have row names but we can add them with rownames():

# add row names  
rownames(df) <- c("row1", "row2", "row3")  
df  
## col1 col2 col3 col4  
## row1 1 this TRUE 2.500000  
## row2 2 is FALSE 4.200000  
## row3 3 text TRUE 3.141593  
attributes(df)  
## $names  
## [1] "col1" "col2" "col3" "col4"  
##   
## $row.names  
## [1] "row1" "row2" "row3"  
##   
## $class  
## [1] "data.frame"

We can also also change the existing column names by using colnames() or names():

# add/change column names with colnames()  
colnames(df) <- c("col\_1", "col\_2", "col\_3", "col\_4")  
df  
## col\_1 col\_2 col\_3 col\_4  
## row1 1 this TRUE 2.500000  
## row2 2 is FALSE 4.200000  
## row3 3 text TRUE 3.141593  
attributes(df)  
## $names  
## [1] "col\_1" "col\_2" "col\_3" "col\_4"  
##   
## $row.names  
## [1] "row1" "row2" "row3"  
##   
## $class  
## [1] "data.frame"  
  
# add/change column names with names()  
names(df) <- c("col.1", "col.2", "col.3", "col.4")  
df  
## col.1 col.2 col.3 col.4  
## row1 1 this TRUE 2.500000  
## row2 2 is FALSE 4.200000  
## row3 3 text TRUE 3.141593  
attributes(df)  
## $names  
## [1] "col.1" "col.2" "col.3" "col.4"  
##   
## $row.names  
## [1] "row1" "row2" "row3"  
##   
## $class  
## [1] "data.frame"

Lastly, just like vectors, lists, and matrices, we can add a comment to a data frame without affecting how it operates.

# adding a comment attribute  
comment(df) <- "adding a comment to a data frame"  
attributes(df)  
## $names  
## [1] "col.1" "col.2" "col.3" "col.4"  
##   
## $row.names  
## [1] "row1" "row2" "row3"  
##   
## $class  
## [1] "data.frame"  
##   
## $comment  
## [1] "adding a comment to a data frame"

## Subsetting

Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists and will return the selected columns with all rows; if you subset with two vectors, they behave like matrices and can be subset by row and column:

df  
## col.1 col.2 col.3 col.4  
## row1 1 this TRUE 2.500000  
## row2 2 is FALSE 4.200000  
## row3 3 text TRUE 3.141593  
  
# subsetting by row numbers  
df[2:3, ]  
## col.1 col.2 col.3 col.4  
## row2 2 is FALSE 4.200000  
## row3 3 text TRUE 3.141593  
  
# subsetting by row names  
df[c("row2", "row3"), ]  
## col.1 col.2 col.3 col.4  
## row2 2 is FALSE 4.200000  
## row3 3 text TRUE 3.141593  
  
# subsetting columns like a list  
df[c("col.2", "col.4")]  
## col.2 col.4  
## row1 this 2.500000  
## row2 is 4.200000  
## row3 text 3.141593  
  
# subsetting columns like a matrix  
df[ , c("col.2", "col.4")]  
## col.2 col.4  
## row1 this 2.500000  
## row2 is 4.200000  
## row3 text 3.141593  
  
# subset for both rows and columns  
df[1:2, c(1, 3)]  
## col.1 col.3  
## row1 1 TRUE  
## row2 2 FALSE  
  
# use a vector to subset  
v <- c(1, 2, 4)  
df[ , v]  
## col.1 col.2 col.4  
## row1 1 this 2.500000  
## row2 2 is 4.200000  
## row3 3 text 3.141593

Note that subsetting data frames with the [ operator will simplify the results to the lowest possible dimension. To avoid this you can introduce the drop = FALSE argument:

# simplifying results in a named vector  
df[, 2]  
## [1] "this" "is" "text"  
  
# preserving results in a 3x1 data frame  
df[, 2, drop = FALSE]  
## col.2  
## row1 this  
## row2 is  
## row3 text

# Dealing with Missing Values

A common task in data analysis is dealing with missing values. In R, missing values are often represented by NA or some other value that represents missing values (i.e. 99). We can easily work with missing values and in this chapter I illustrate how to [test for](#na_test), [recode](#na_recode), and [exclude](#na_exclude) missing values in your data.

## Testing for missing values

To identify missing values use is.na() which returns a logical vector with TRUE in the element locations that contain missing values represented by NA. is.na() will work on vectors, lists, matrices, and data frames.

# vector with missing data  
x <- c(1:4, NA, 6:7, NA)  
x  
## [1] 1 2 3 4 NA 6 7 NA  
  
is.na(x)  
## [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE  
  
# data frame with missing data  
df <- data.frame(col1 = c(1:3, NA),  
 col2 = c("this", NA,"is", "text"),   
 col3 = c(TRUE, FALSE, TRUE, TRUE),   
 col4 = c(2.5, 4.2, 3.2, NA),  
 stringsAsFactors = FALSE)  
  
# identify NAs in full data frame  
is.na(df)  
## col1 col2 col3 col4  
## [1,] FALSE FALSE FALSE FALSE  
## [2,] FALSE TRUE FALSE FALSE  
## [3,] FALSE FALSE FALSE FALSE  
## [4,] TRUE FALSE FALSE TRUE  
  
# identify NAs in specific data frame column  
is.na(df$col4)  
## [1] FALSE FALSE FALSE TRUE

To identify the location or the number of NAs we can leverage the which() and sum() functions:

# identify location of NAs in vector  
which(is.na(x))  
## [1] 5 8  
  
# identify count of NAs in data frame  
sum(is.na(df))  
## [1] 3

## Recoding missing values

To recode missing values; or recode specific indicators that represent missing values, we can use normal subsetting and assignment operations. For example, we can recode missing values in vector x with the mean values in x by first subsetting the vector to identify NAs and then assign these elements a value. Similarly, if missing values are represented by another value (i.e. 99) we can simply subset the data for the elements that contain that value and then assign a desired value to those elements.

# recode missing values with the mean  
x[is.na(x)] <- mean(x, na.rm = TRUE)  
round(x, 2)  
## [1] 1.00 2.00 3.00 4.00 3.83 6.00 7.00 3.83  
  
# data frame that codes missing values as 99  
df <- data.frame(col1 = c(1:3, 99), col2 = c(2.5, 4.2, 99, 3.2))  
  
# change 99s to NAs  
df[df == 99] <- NA  
df  
## col1 col2  
## 1 1 2.5  
## 2 2 4.2  
## 3 3 NA  
## 4 NA 3.2

## Excluding missing values

We can exclude missing values in a couple different ways. First, if we want to exclude missing values from mathematical operations use the na.rm = TRUE argument. If you do not exclude these values most functions will return an NA.

# A vector with missing values  
x <- c(1:4, NA, 6:7, NA)  
  
# including NA values will produce an NA output  
mean(x)  
## [1] NA  
  
# excluding NA values will calculate the mathematical  
# operation for all non-missing values  
mean(x, na.rm = TRUE)  
## [1] 3.833333

We may also desire to subset our data to obtain complete observations, those observations (rows) in our data that contain no missing data. We can do this a few different ways.

# data frame with missing values  
df <- data.frame(col1 = c(1:3, NA),  
 col2 = c("this", NA,"is", "text"),   
 col3 = c(TRUE, FALSE, TRUE, TRUE),   
 col4 = c(2.5, 4.2, 3.2, NA),  
 stringsAsFactors = FALSE)  
df  
## col1 col2 col3 col4  
## 1 1 this TRUE 2.5  
## 2 2 <NA> FALSE 4.2  
## 3 3 is TRUE 3.2  
## 4 NA text TRUE NA

First, to find complete cases we can leverage the complete.cases() function which returns a logical vector identifying rows which are complete cases. So in the following case rows 1 and 3 are complete cases. We can use this information to subset our data frame which will return the rows which complete.cases() found to be TRUE.

complete.cases(df)  
## [1] TRUE FALSE TRUE FALSE  
  
# subset with complete.cases to get complete cases  
df[complete.cases(df), ]  
## col1 col2 col3 col4  
## 1 1 this TRUE 2.5  
## 3 3 is TRUE 3.2  
  
# or subset with `!` operator to get incomplete cases  
df[!complete.cases(df), ]  
## col1 col2 col3 col4  
## 2 2 <NA> FALSE 4.2  
## 4 NA text TRUE NA

An shorthand alternative is to simply use na.omit() to omit all rows containing missing values.

# or use na.omit() to get same as above  
na.omit(df)  
## col1 col2 col3 col4  
## 1 1 this TRUE 2.5  
## 3 3 is TRUE 3.2

1. There are two additional vector types which I will not discuss - complex and raw. [↑](#footnote-ref-26)
2. Its important to understand the difference between simplifying and preserving subsetting. **Simplifying** subsets returns the simplest possible data structure that can represent the output. **Preserving** subsets keeps the structure of the output the same as the input. See Hadley Wickham's section on [Simplifying vs. Preserving Subsetting](http://adv-r.had.co.nz/Subsetting.html#subsetting-operators) to learn more. [↑](#footnote-ref-42)
3. Its important to understand the difference between simplifying and preserving subsetting. **Simplifying** subsets returns the simplest possible data structure that can represent the output. **Preserving** subsets keeps the structure of the output the same as the input. See Hadley Wickham's section on [Simplifying vs. Preserving Subsetting](http://adv-r.had.co.nz/Subsetting.html#subsetting-operators) to learn more. [↑](#footnote-ref-45)