R Basics*

Chapter 2

Anna Ziff

R Workflow for Economists

Contents

*Please contact anna.ziff@duke.edu if there are errors.

Basic Operations

Basic operations allow R to be used as a calculator and are building blocks for more complex calculations. Recall that any text after `#` is a comment and R does not evaluate it.

```
6 / (1 + 1) * 2.5 - 15
```

```
## [1] -7.5
```

```
10 %/% 3 # Integer division
```

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

```
10 %% 3 # Remainder
```

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

```
2^10 # Alternatively, 2 ** 10
```

```
## [1] 1024
```

```
27^(1 / 3)
```

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

R has two values for undefined operations: `NaN` (not a number) and `Inf` (infinity).

```
0 / 0
```

```
## [1] NaN
```

```
pi / 0
```

```
## [1] Inf
```

```
-pi / 0
```

```
## [1] -Inf
```

```
Inf - Inf
```

```
## [1] NaN
```

Practice Exercises 2.1

1. The formula to convert fahrenheit (f) to celsius (c) is $(f - 32)\frac{5}{9} = c$. Convert 60 degrees fahrenheit to celsius.
2. Is 7,865,695,837 divisible by 3?

Basic Functions and Assignment

Functions in R are comprised of a name and arguments. They generally, but do not have to, output a value.

```
abs(-3)
```

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

```
sqrt(4)
```

```
## [1] 2
```

```
factorial(10)
```

```
## [1] 3628800
```

```
exp(10)
```

```
## [1] 22026.47
```

```
cos(pi)
```

```
## [1] -1
```

```
log(1)
```

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

```
round(2.7)
```

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

```
ceiling(2.7)
```

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

```
floor(2.7)
```

```
## [1] 2
```

Functions can have more than one argument. The value of the argument is specified with `=`. Required arguments do not need to follow the `argument name = argument value` format, although they can.

```
log(3, base = 10)
```

```
## [1] 0.4771213
```

```
round(0.183104, digits = 3)
```

```
## [1] 0.183
```

Recall from chapter 0 that the documentation for functions is readily available from the command line. Online searches can help bolster this information with troubleshooting and examples.

```
?date # Alternatively help(date)
date()
```

```
## [1] "Tue Aug  2 14:56:20 2022"
```

Typing the function without `()` will show the function's code. This is useful if you want to see exactly what a function does. The actual substance of the function `factorial()` is `gamma(x + 1)`. The `<bytecode:...>` is where the function is stored and `<environment:...>` is the package that defines that function.

```
factorial
```

```
## function (x)
## gamma(x + 1)
## <bytecode: 0x12e8bbef0>
## <environment: namespace:base>
```

The symbol `<-` is called the assignment arrow. This is used to define the values of objects. The first line assigns the variable `x` to have value 1 (assignment). The second line displays the value of `x` (evaluation).

```
x <- 1
x
```

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

It used to be the case that `=` could not be used for assignment. This changed in 2001 and `=` can be used for assignment in most cases. However, it is most general and best practice to only use `<-` for assignment and `=`

to specify values for arguments of functions.

Another detail of the assignment arrow is that it can clarify the direction of assignment. That is, `<-` or `->`. Usually, it will make sense to only use the `<-` assignment arrow.

```
2 -> x
x
```

```
## [1] 2
```

Finally, you may rarely see the use of `«-` and `-»`. These uncommon assignment arrows are used to assign values to global variables. That is, those objects that retain their value regardless of what is happening in the code (loops, functions, etc.). You should practice using `<-` for assignment and `=` when specifying the values of arguments.

There are some basic requirements for variable names (Workflow chapter 3 has more details on stylistic conventions).

- Variable names are case-sensitive.
- Variable names may contain letters (`a-z`, `A-Z`), numbers (`0-9`), periods (`.`), and underscores (`_`).
- Variable names may not contain spaces, hyphens, start with a number, or be the same as functions, operations, and instructions unless the name is in quotation marks (not recommended).

Practice Exercises 2.2

1. Do all of these lines give the same result? Why or why not?

```
log(3, base = 10)
log(x = 3, base = 10)
log(base = 10, x = 3)
log(3, 10)
log(10, 3)
```

2. Functions can be nested, i.e., the outputs of functions can be the inputs of other functions. Calculate ln(3!).
3. Define a variable `x` to take the value 10. Then, try the following code exactly as it is written. Why do you get an error? Fix the code.

```
X^2
```

Types

Variables can take non-numeric values. You may have noticed above that we did not specify that we were defining numeric variables above. Unlike other programming languages, R recognizes the type automatically. **The core types are numeric (integer, double), complex, logical, character string, and binary.**

Numeric

There are two numeric types, integers and doubles (double precision floating point numbers). Integers can only contain integers, as the name suggests. The integer type requires less memory than the double type.

R will automatically cast numeric objects as doubles.

```
v <- 10
typeof(v)
```

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

```
w <- 2.5
typeof(w)
```

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

You can change doubles to integers using the `as.integer()` function.

```
x <- as.integer(v)
typeof(x)
```

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

You can also add `L` to the end of integers to signal that R should cast these objects as integers.

```
y <- 2L
y
```

```
## [1] 2
```

```
typeof(y)
```

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

Just as the function `as.integer()` converts a double to an integer, `as.double()` converts an integer to a double.

```
z <- as.double(y)
z
```

```
## [1] 2
```

```
typeof(z)
```

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

Complex

Complex numbers have real and imaginary components.

```
x <- 1 + 2i
typeof(x)
```

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

```
Re(x) # Returns real component of x
```

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

```
Im(x) # Returns imaginary component of x
```

```
## [1] 2
```

Logical

The logical (or boolean) type takes two values: `TRUE` or `FALSE`. The abbreviations `T` and `F` can also be used, but it is best practice to use `TRUE` and `FALSE`. The logical type will result from a logical operation. Logical operators include: `>, >=, <, <=, ==, !=`.

```
a <- 10.5
b <- 3
b > a
```

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

```
a == b
```

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

```
is.numeric(a)
```

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

```
is.integer(b)
```

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

```
is.double(b)
```

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

```
is.complex(x)
```

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

```
f <- TRUE
is.logical(f)
```

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

The value `TRUE` corresponds to the numeric value 1 and the value `FALSE` corresponds to the numeric value 0. This makes it easy to check for the number of elements in a structure with more than one element (a vector of length 5 in this example) that are `TRUE`. More information on vectors is below.

```
a <- c(1, 4, 0, 12, 21)
b <- 1:5
sum(a > b)
```

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

```
a > b
```

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

Missing Values

A missing value is denoted `NA` (not available). Technically, it is a logical value rather than a separate data type.

```
x <- NA
is.na(x)
```

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

```
x + 3
```

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

The values `Inf` and `NaN` often gets confused with `NA`. Recall that `Inf` is infinity and `NaN` is "not a number" (the output of an undefined function).

```
y <- 0 / 0
is.nan(y)
```

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

```
is.finite(y)
```

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

```
is.infinite(y)
```

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

```
is.na(y)
```

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

Character Strings

Character strings are information between quotation marks. If the character string contains only numbers, it can be converted to a numeric type. Single and double quotations are interchangeable, but there is an official stylistic preference for double quotations (see `?Quotes`). Whichever you decide to use, make sure you are consistent!

```
hw <-"Hello World!"
typeof(hw)
```

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

```
is.character(hw)
```

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

```
x <- "10.3"
as.numeric(x)
```

```
## [1] 10.3
```

The function `paste()` creates and concatenates (combines) character strings in a multitude of situations.

```
paste(hw, 7)
```

```
## [1] "Hello World! 7"
```

```
paste("Number", 3.5)
```

```
## [1] "Number 3.5"
```

The argument `sep` specifies if there should be a character separating the inputs. Note the difference between these two lines.

```
paste("I", "live", "in", "Wonderland")
```

```
## [1] "I live in Wonderland"
```

```
paste("I", "live", "in", "Wonderland", sep = "! ")
```

```
## [1] "I! live! in! Wonderland"
```

The function `paste()` can also be applied to data structures with more than one element.

```
1:5
```

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

```
paste("Class", 1:5, sep = " #")
```

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

The function `paste0()` does the same as `paste()` but without spaces.

```
paste0("All", "one", "word")
```

```
## [1] "Alloneword"
```

The function `as.character()` converts objects to character strings. The function `toString()` produces a single character string from several objects.

```
as.character(c("June", 29, 2021))
```

```
## [1] "June" "29"   "2021"
```

```
toString(c("June", 29, 2021))
```

```
## [1] "June, 29, 2021"
```

Data are frequently in character string format. Character strings can also be useful to print directly to the console or a file.

```
print(hw)
```

```
## [1] "Hello World!"
```

```
noquote(hw) # Alternatively, print(hw, quote = FALSE)
```

```
## [1] Hello World!
```

The function `sprintf()` comes from C and allows for additional formatting. Strings are substituted wherever there is `%s`.

```
a <- "elephants"
sprintf("My favorite animals are %s.", a)
```

```
## [1] "My favorite animals are elephants."
```

```
b <- "turtles"
sprintf("I like %s and %s.", a, b)
```

```
## [1] "I like elephants and turtles."
```

The power of `sprintf()` is more evident when printing numbers. Integers are substituted with `%d`.

```
int <- 1
sprintf("This is class number %d.", int)
```

```
## [1] "This is class number 1."
```

```
sprintf("This is class: %2d.", int) # Add leading spaces
```

```
## [1] "This is class:  1."
```

```
sprintf("Class number %02d.", int) # Add leading zeroes (2 digits total)
```

```
## [1] "Class number 01."
```

Doubles are substituted with `%f`.

```
sprintf("%f", pi)
```

```
## [1] "3.141593"
```

```
sprintf("%.3f", pi) # 3 digits past the decimal
```

```
## [1] "3.142"
```

```
sprintf("%1.0f", pi) # 1 integer and no decimal digits
```

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

```
sprintf("%1.7f", pi) # 1 integer and 7 decimal digits
```

```
## [1] "3.1415927"
```

```
sprintf("%+f", pi) # Add a sign
```

```
## [1] "+3.141593"
```

```
sprintf("%-f", -pi)
```

```
## [1] "-3.141593"
```

```
sprintf("% f", pi) # Leading space
```

```
## [1] " 3.141593"
```

Binary

Binary (or raw) data is in hexadecimal format and is a more basic type of data. This data type does not generally arise for empirical work.

```
b <- as.raw(9)
b
```

```
## [1] 09
```

```
typeof(b)
```

```
## [1] "raw"
```

Practice Exercises 2.3

1. Define a variable with the value 82. Using this variable, define another variable that is $82^2 - 7$.
2. What do you think the following will output? Test it out.

```
paste("Me", "Myself", "&", "I")
```

3. Edit the code in question 3 to output all one word without spaces.
4. Edit the code in question 3 to output commas between each of the elements.

Structures

The data types can be organized into data structures. **The basic structures are vectors, matrices, arrays, lists, data frames, and factors.**

Vectors

Vectors are sequences of data points of the same type. Even if you try to make a vector with different data types, the resulting vector will be coerced so that every component has the same type. R defaults to the more general type.

```
c(1000, 1, 2)
```

```
## [1] 1000    1    2
```

```
c(1000, TRUE, 2)
```

```
## [1] 1000    1    2
```

```r
c(1000, TRUE, "2")
```

```
## [1] "1000" "TRUE" "2"
```

```r
c("a", "b", "c")
```

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

```r
1:3 # Same as c(1, 2, 3) but stored as integer and less memory-intensive
```

```
## [1] 1 2 3
```

```r
3:1
```

```
## [1] 3 2 1
```

```r
seq(from = 1, to = 10, by = 0.5)
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
## [16]  8.5  9.0  9.5 10.0
```

```r
seq(from = 100, to = 90, by = -2)
```

```
## [1] 100  98  96  94  92  90
```

```r
seq(from = 1, to = 10, length = 10)
```

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

```r
rep(3, times = 10)
```

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

```r
rep(1:3, each = 2)
```

```
## [1] 1 1 2 2 3 3
```

Sometimes it is useful to add names to the elements of the vector, rather than just the numerical index.

```r
s <- c(1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3)
names(s) <- letters[1:13] # First 13 letters of the alphabet
s
```

```
## a b c d e f g h i j k l m
## 1 3 3 2 1 4 2 4 1 8 5 1 3
```

```r
length(s)
```

```
## [1] 13
```

```r
class(s)
```

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

```r
attributes(s) # We added an attribute
```

```
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
```

The names can also be added when making the vector initially.

```r
s <- c(a = 1, b = 3, c = 3, d = 2)
s
```

```
## a b c d
```

```
## 1 3 3 2
```

Another piece of metadata (attribute) available for vectors is `comment`.

```
comment(s) <- "Letter scores in Scrabble"
attributes(s)
```

```
## $names
## [1] "a" "b" "c" "d"
##
## $comment
## [1] "Letter scores in Scrabble"
```

Vectorization

Many operations and functions in R are designed to be particularly efficient with vectors. Instead of going through each element of the vector, you can apply operations and functions to the full vector.

```
x <- c(0, 1, 4)
y <- c(5, 9, 2)
x + y
```

```
## [1]  5 10  6
```

```
x * y
```

```
## [1] 0 9 8
```

```
factorial(x)
```

```
## [1]  1  1 24
```

```
log(y)
```

```
## [1] 1.6094379 2.1972246 0.6931472
```

Recycling

If you perform an operation on vectors of unequal lengths, R will recycle elements of the shorter vector. In this example, `x` has 3 elements and `y` has 6 elements. The first 3 elements of `y` are added to the elements of `x`, and the same for the second 3 elements `y`

```
x <- c(0, 1, 4)
y <- c(5, 9, 2, 2, 1, 0)
x + y
```

```
## [1]  5 10  6  2  2  4
```

Recycling often occurs silently, i.e., no warning message is displayed. An exception is when the longer vector is not a multiple of the shorter object.

```
x <- c(0, 1, 4)
y <- c(5, 9, 2, 2, 1)
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

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

While recycling may seem counter-intuitive, it is very useful within the framework of R. Notice that there are no scalars, only vectors with one element. Without recycling, dealing with very simple operations like the one below would be burdensome.

```
c <- 2
c
```

```
## [1] 2
```

```
length(c)
```

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

```
x
```

```
## [1] 0 1 4
```

```
x * c
```

```
## [1] 0 2 8
```

Manipulation

The function `c()` is also used to add on to existing vectors.

```
x <- 1:10
c(x, 90:100)
```

```
##  [1]   1   2   3   4   5   6   7   8   9  10  90  91  92  93  94  95  96  97  98
## [20]  99 100
```

Subsetting vectors involves accessing certain elements. It can be done with positive numbers, negative numbers, logical values, or names.

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

Subsetting can be done with positive numbers. This returns the elements at the specified positions.

```
x[2]
```

```
## B
## 2
```

```
x[15:17]
```

```
##  O  P  Q
## 15 16 17
```

```
x[c(24, 2, 11)]
```

```
##  X  B  K
## 24  2 11
```

```
x[c(2, 2, 7)]
```

```
## B B G
## 2 2 7
```

Subsetting can also be done with negative numbers. This returns elements except at the specified positions.

```
x[-1]
```

```
##  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
##  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

```
x[-(1:13)]
```

```
##  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
## 14 15 16 17 18 19 20 21 22 23 24 25 26
```

```
x[-c(2, 4, 6, 8, 13:26)]
```

```
##  A  C  E  G  I  J  K  L
##  1  3  5  7  9 10 11 12
```

Subsetting with logical values returns elements corresponding to `TRUE`.

```
x[c(TRUE, rep(FALSE, 25))]
```

```
## A
## 1
```

```
x[x < 13]
```

```
##  A  B  C  D  E  F  G  H  I  J  K  L
##  1  2  3  4  5  6  7  8  9 10 11 12
```

```
x[x > 22 | x < 10]
```

```
##  A  B  C  D  E  F  G  H  I  W  X  Y  Z
##  1  2  3  4  5  6  7  8  9 23 24 25 26
```

```
x[c(TRUE, FALSE)] # Remember recycling!
```

```
##  A  C  E  G  I  K  M  O  Q  S  U  W  Y
##  1  3  5  7  9 11 13 15 17 19 21 23 25
```

Finally, when there are names, vectors can be subset by names. This returns elements corresponding to the specified names.

```
x["A"]
```

```
## A
## 1
```

```
x[c("A", "Q", "M")]
```

```
##  A  Q  M
##  1 17 13
```

Practice Exercises 2.4

1. R has built-in functions to create vectors of random numbers from different distributions. Here is an example that creates a vector with 1,000 random draws from the uniform $[0, 1]$ distribution. Add this code to your script to test it.

```
uniform <- runif(1000, min = 0, max = 1)
hist(uniform)
```

2. Create two vectors, `x` and `y`, each comprised of 10 random draws from the uniform $[0, 1]$ distribution.
3. Subset `x` to only keep the first 3 realizations. Define a vector called `z` with this subset.
4. What will `y + z` output? Check your answer.
5. Subset `y` to only keep elements greater than 0.75.

Matrices and Arrays

Matrices are like vectors but with two dimensions. Just like for vectors, all elements of matrices and arrays should have the same data type. The columns need to have the same number of elements.

```
N <- matrix(1:12, nrow = 4, ncol = 3) # Automatically fills by column
N
```

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

```
str(N)
```

```
##  int [1:4, 1:3] 1 2 3 4 5 6 7 8 9 10 ...
```

```
attributes(N)
```

```
## $dim
## [1] 4 3
```

```
dim(N)
```

```
## [1] 4 3
```

Note that the matrix fills in by column. Specifying the `byrow` option will result in the matrix filling in by row.

```
O <- matrix(1:12, nrow = 4, ncol = 3, byrow = TRUE)
O
```

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

Matrices can contain other data types, as long as they are uniform. Here are examples with character strings, doubles, logical values, and missing values.

```
P <- matrix(letters, nrow = 2, ncol = 13)
P
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,] "a"  "c"  "e"  "g"  "i"  "k"  "m"  "o"  "q"  "s"   "u"   "w"   "y"
## [2,] "b"  "d"  "f"  "h"  "j"  "l"  "n"  "p"  "r"  "t"   "v"   "x"   "z"
```

```
Q <- matrix(runif(9, min = 0, max = 1), nrow = 3, ncol = 3)
Q
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.6870248 0.6735310 0.1639514
## [2,] 0.1661428 0.8071722 0.3382885
## [3,] 0.6210469 0.4806281 0.6471651
```

```
R <- matrix(c(TRUE, TRUE, FALSE, TRUE), nrow = 2, ncol = 2)
R
```

```
##      [,1]  [,2]
## [1,] TRUE FALSE
## [2,] TRUE  TRUE
```

```
S <- matrix(NA, nrow = 3, ncol = 4) # Take note of the recycling here!
S
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA
```

Matrices always have the `dim` attribute. It is possible to add other attributes, including row names, column names, and comments.

```
rownames(O) <- c("row1", "row2", "row3", "row4")
colnames(O) <- c("col1", "col2", "col3")
dimnames(O)
```

```
## [[1]]
## [1] "row1" "row2" "row3" "row4"
##
## [[2]]
## [1] "col1" "col2" "col3"
```

```
dimnames(O)[[1]] <- c("new1", "new2", "new3", "new4")
O
```

```
##      col1 col2 col3
## new1    1    2    3
## new2    4    5    6
## new3    7    8    9
## new4   10   11   12
```

```
comment(O) <- "Comment for matrix O."
attributes(O)
```

```
## $dim
## [1] 4 3
##
## $dimnames
## $dimnames[[1]]
## [1] "new1" "new2" "new3" "new4"
##
## $dimnames[[2]]
## [1] "col1" "col2" "col3"
##
##
## $comment
## [1] "Comment for matrix O."
```

When the number of elements of the data is smaller than the number of elements in the matrix, the data are recycled.

```
matrix(1:10, nrow = 2, ncol = 10, byrow = TRUE)
```

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

```
matrix(1:10, nrow = 2, ncol = 6, byrow = TRUE)
```

```
## Warning in matrix(1:10, nrow = 2, ncol = 6, byrow = TRUE): data length [10] is
```

```
## not a sub-multiple or multiple of the number of columns [6]
```

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

The functions `cbind()` (column bind) and `rbind()` (row bind) can be used to create matrices from vectors or to add on to matricies.

```
a <- c(2, 4, 6)
b <- c(1, 3, 5)
c <- c(0, 0, 1)
cbind(a, b)
```

```
##      a b
## [1,] 2 1
## [2,] 4 3
## [3,] 6 5
```

```
rbind(a, b)
```

```
##   [,1] [,2] [,3]
## a    2    4    6
## b    1    3    5
```

```
A <- cbind(a, b, c)
A
```

```
##      a b c
## [1,] 2 1 0
## [2,] 4 3 0
## [3,] 6 5 1
```

```
A <- cbind(A, c(1, 1, 1))
A
```

```
##      a b c
## [1,] 2 1 0 1
## [2,] 4 3 0 1
## [3,] 6 5 1 1
```

```
rbind(A, c(8, 7, 1, 1))
```

```
##      a b c
## [1,] 2 1 0 1
## [2,] 4 3 0 1
## [3,] 6 5 1 1
## [4,] 8 7 1 1
```

Brackets, with the general format `matrix[rows, columns]`, are used to subset matrices.

```
O
```

```
##      col1 col2 col3
## new1    1    2    3
## new2    4    5    6
## new3    7    8    9
## new4   10   11   12
```

```
O[1, 2]
```

```
## [1] 2
```

You can select multiple items by passing vectors of indices.

```
O[1:2, c(1, 3)]
```

```
##      col1 col3
## new1    1    3
## new2    4    6
```

Leaving one argument empty selects all the rows or columns.

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

```
##      col1 col3
## new1    1    3
## new2    4    6
## new3    7    9
## new4   10   12
```

```
O[1:2, ]
```

```
##      col1 col2 col3
## new1    1    2    3
## new2    4    5    6
```

If you want to preserve the attributes of the matrix, specify `drop = FALSE`.

```
O[, 2]
```

```
## new1 new2 new3 new4
##    2    5    8   11
```

```
O[, 2, drop = FALSE]
```

```
##      col2
## new1    2
## new2    5
## new3    8
## new4   11
```

Arrays are just like matrices but with more than two dimensions.

```
P <- array(1:12, dim = c(2, 2, 3))
P
```

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

```
## [1,]    9   11
## [2,]   10   12
```

```
dim(P)
```

```
## [1] 2 2 3
```

```
class(P)
```

```
## [1] "array"
```

```
Q <- array(1:12, dim = c(2, 2 ,2 ,2))
Q
```

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

Subsetting arrays works much the same way as subsetting matrices, except there are more than two dimensions.

```
P[, , 1]
```

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

```
P[1:2, 1, ]
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
```

Matrix Algebra

We define the matrices `A`, `B`, and `C` to demonstrate matrix operations and functions.

```
A <- matrix(1:4, nrow = 2)
A
```

```
##      [,1] [,2]
```

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

```
B <- matrix(5:8, nrow = 2)
B
```

```
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

```
C <- diag(2)
C
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Scalar addition, subtraction, multiplication, and division are straightforward.

```
A + 1
```

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

```
B - 0.5
```

```
##      [,1] [,2]
## [1,]  4.5  6.5
## [2,]  5.5  7.5
```

```
C * 3
```

```
##      [,1] [,2]
## [1,]    3    0
## [2,]    0    3
```

```
A / 4
```

```
##      [,1] [,2]
## [1,] 0.25 0.75
## [2,] 0.50 1.00
```

Element-wise addition, subtraction, multiplication, and division are the same but with two matrices rather than scalars.

```
A + B
```

```
##      [,1] [,2]
## [1,]    6   10
## [2,]    8   12
```

```
A - B
```

```
##      [,1] [,2]
## [1,]   -4   -4
## [2,]   -4   -4
```

```
A * B
```

```
##      [,1] [,2]
## [1,]    5   21
## [2,]   12   32
```

```
A / B
```

```
##           [,1]      [,2]
## [1,] 0.2000000 0.4285714
## [2,] 0.3333333 0.5000000
```

Below are the core functions to manipulate, characterize, and multiply matrices. There are many others, both in the basic R functions and in external packages.

```
t(B) # Transpose
```

```
##      [,1] [,2]
## [1,]    5    6
## [2,]    7    8
```

```
A %*% B # Matrix multiplication
```

```
##      [,1] [,2]
## [1,]   23   31
## [2,]   34   46
```

```
solve(B) # Inverse
```

```
##      [,1] [,2]
## [1,]   -4  3.5
## [2,]    3 -2.5
```

```
det(A) # Determinant
```

```
## [1] -2
```

```
sum(diag(A)) # Trace
```

```
## [1] 5
```

```
kronecker(A, B) # Kronecker product
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    5    7   15   21
## [2,]    6    8   18   24
## [3,]   10   14   20   28
## [4,]   12   16   24   32
```

```
eigen(B) # Spectral decomposition
```

```
## eigen() decomposition
## $values
## [1] 13.1520673 -0.1520673
##
## $vectors
##            [,1]       [,2]
## [1,] -0.6514625 -0.8053759
## [2,] -0.7586809  0.5927644
```

Practice Exercises 2.5

1. Create the following matrices.

$$X = \begin{bmatrix} 1 & 0 & 9 \\ 1 & 0 & 14 \\ 1 & 0 & 12 \\ 1 & 1 & 12 \\ 1 & 1 & 14 \\ 1 & 1 & 10 \end{bmatrix} \quad Y = \begin{bmatrix} 415 \\ 740 \\ 582 \\ 493 \\ 623 \\ 530 \end{bmatrix}$$

2. Calculate the OLS estimator: $(X'X)^{-1}(X'Y)$.

Lists

Lists are general as they can combine different data types. The elements of lists can even be other data structures (including other lists!), and do not need to be uniform length.

```
r <- list(1, "Hi", FALSE, 2.3, 19:23, matrix(1:4, nrow = 2))
r
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "Hi"
##
## [[3]]
## [1] FALSE
##
## [[4]]
## [1] 2.3
##
## [[5]]
## [1] 19 20 21 22 23
##
## [[6]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
r[[3]] # Access the third elements
```

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

```
length(r)
```

```
## [1] 6
```

```
class(r)
```

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

```
str(r)
```

```
## List of 6
##  $ : num 1
##  $ : chr "Hi"
##  $ : logi FALSE
```

```
##  $ : num 2.3
##  $ : int [1:5] 19 20 21 22 23
##  $ : int [1:2, 1:2] 1 2 3 4
```

If the order of the elements is not conceptually relevant, then it may make more sense to name the elements of lists.

```
s <- list(myinteger = 1,
          mycharacter = "Hi",
          mylogical = FALSE,
          mydouble = 2.3,
          myvector = 19:23,
          mymatrix = matrix(1:4, nrow = 2))
str(s)
```

```
## List of 6
##  $ myinteger  : num 1
##  $ mycharacter: chr "Hi"
##  $ mylogical  : logi FALSE
##  $ mydouble   : num 2.3
##  $ myvector   : int [1:5] 19 20 21 22 23
##  $ mymatrix   : int [1:2, 1:2] 1 2 3 4
```

The dollar sign, $ can be used to refer to a named element of a list.

```
s$myvector
```

```
## [1] 19 20 21 22 23
```

Names can also be added once a list is created.

```
t <- list(1:10, c("USA", "Mexico", "Canada"))
t
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
## [1] "USA"    "Mexico" "Canada"
```

```
attributes(t)
```

```
## NULL
```

```
names(t) <- c("numbers", "countries")
attributes(t)
```

```
## $names
## [1] "numbers"   "countries"
```

Comments can also be added, for the whole list or for certain elements

```
comment(t) <- "Example list"
attr(t, "countries") <- "North American Countries"
str(t)
```

```
## List of 2
##  $ numbers  : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ countries: chr [1:3] "USA" "Mexico" "Canada"
##  - attr(*, "comment")= chr "Example list"
##  - attr(*, "countries")= chr "North American Countries"
```

```
attributes(t)
```

```
## $names
## [1] "numbers"   "countries"
##
## $comment
## [1] "Example list"
##
## $countries
## [1] "North American Countries"
```

Manipulation

The function `list()` can be nested to append elements to lists. In this example, a new element is added to the first element (a list with 4 elements).

```
l1 <- list("R", 1:9, c(TRUE, FALSE), 1.5)
l1
```

```
## [[1]]
## [1] "R"
##
## [[2]]
## [1] 1 2 3 4 5 6 7 8 9
##
## [[3]]
## [1]  TRUE FALSE
##
## [[4]]
## [1] 1.5
```

```
l2 <- list(l1, letters[1:5])
str(l2)
```

```
## List of 2
##  $ :List of 4
##   ..$ : chr "R"
##   ..$ : int [1:9] 1 2 3 4 5 6 7 8 9
##   ..$ : logi [1:2] TRUE FALSE
##   ..$ : num 1.5
##  $ : chr [1:5] "a" "b" "c" "d" ...
```

To simply append a fifth element to the original list, use the function `append()`.

```
l3 <- append(l1, list(letters[1:5]))
str(l3)
```

```
## List of 5
##  $ : chr "R"
##  $ : int [1:9] 1 2 3 4 5 6 7 8 9
##  $ : logi [1:2] TRUE FALSE
##  $ : num 1.5
##  $ : chr [1:5] "a" "b" "c" "d" ...
```

Notice the difference without `list()` in the second argument.

```
l4 <- append(l1, letters[1:5])
str(l4)
```

```
## List of 9
##  $ : chr "R"
##  $ : int [1:9] 1 2 3 4 5 6 7 8 9
##  $ : logi [1:2] TRUE FALSE
##  $ : num 1.5
##  $ : chr "a"
##  $ : chr "b"
##  $ : chr "c"
##  $ : chr "d"
##  $ : chr "e"
```

Another way to add a new element is with the $ sign. This requires a name for the new element.

```
l3$greeting <- "Hello world!"
str(l3)
```

```
## List of 6
##  $         : chr "R"
##  $         : int [1:9] 1 2 3 4 5 6 7 8 9
##  $         : logi [1:2] TRUE FALSE
##  $         : num 1.5
##  $         : chr [1:5] "a" "b" "c" "d" ...
##  $ greeting: chr "Hello world!"
```

Subsetting of lists is done with `[]`, `[[]]`,and `$`. The exact approach depends on whether you want to preserve the output or simplify the output.

Preserving the output means keeping the list format and attributes.

```
t
```

```
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $countries
## [1] "USA"    "Mexico" "Canada"
##
## attr(,"countries")
## [1] "North American Countries"
```

```
t[1]
```

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

```
t["numbers"]
```

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

```
t[1:2]
```

```
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $countries
## [1] "USA"    "Mexico" "Canada"
```

```
t[c("numbers", "countries")]
```

```
## $numbers
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $countries
## [1] "USA"    "Mexico" "Canada"
```

Simplifying the output extracts what is inside each element. Imagine that each element of a list is a box. The simplified output returns what is inside each box.

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

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

```
t[["numbers"]]
```

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

```
t$numbers
```

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

With a simplified output, you can also extract elements of the objects.

```
t[[1]][8]
```

```
## [1] 8
```

```
t[["countries"]][1:2]
```

```
## [1] "USA"    "Mexico"
```

```
t$countries[3]
```

```
## [1] "Canada"
```

Subsetting with a nested list follows the same idea as above. Continue to be mindful of persevering vs. simplifying output!

```
str(l2)
```

```
## List of 2
##  $ :List of 4
##   ..$ : chr "R"
##   ..$ : int [1:9] 1 2 3 4 5 6 7 8 9
##   ..$ : logi [1:2] TRUE FALSE
##   ..$ : num 1.5
##  $ : chr [1:5] "a" "b" "c" "d" ...
```

```
l2[[2]][3]
```

```
## [1] "c"
```

```
l2[[1]][1:5]
```

```
## [[1]]
## [1] "R"
##
## [[2]]
## [1] 1 2 3 4 5 6 7 8 9
##
## [[3]]
## [1]  TRUE FALSE
##
## [[4]]
```

```
## [1] 1.5
##
## [[5]]
## NULL
```

```
names(l2) <- c("item1", "item2")
names(l2[[1]]) <- paste0("subitem", 1:4)
str(l2)
```

```
## List of 2
##  $ item1:List of 4
##   ..$ subitem1: chr "R"
##   ..$ subitem2: int [1:9] 1 2 3 4 5 6 7 8 9
##   ..$ subitem3: logi [1:2] TRUE FALSE
##   ..$ subitem4: num 1.5
##  $ item2: chr [1:5] "a" "b" "c" "d" ...
```

```
l2[["item1"]][["subitem4"]]
```

```
## [1] 1.5
```

```
l2$item1$subitem2
```

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

```
l2$item1$subitem2[6:9]
```

```
## [1] 6 7 8 9
```

Practice Exercises 2.6

1. Create a list with the following elements. First, a character string with your name. Second, an integer with your age. Third, a logical vector whose elements are `TRUE` if you know how to ride/drive a motorcycle, bicycle, scooter, and car, and `FALSE` otherwise.
2. Access the second index of the third element of your list. That is, do you know how to ride a bicycle?

Factors

Factors organize character strings by extracting the possible values. This structure can be useful in datasets with character string variables. It has some efficiency advantages as well because R stores the vector as integers rather than character strings.

```
u <- factor(c("Macro", "Metrics", "Micro", "Micro", "Macro"))
u
```

```
## [1] Macro   Metrics Micro   Micro   Macro
## Levels: Macro Metrics Micro
```

```
length(u)
```

```
## [1] 5
```

```
class(u)
```

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

```
typeof(u)
```

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

```
levels(u)
```

```
## [1] "Macro"   "Metrics" "Micro"
```

```
summary(u)
```

```
##   Macro Metrics   Micro
##       2       1       2
```

The function `as.factor()` converts vectors of character strings or integers to factors.

```
a <- c("Red", "Blue", "Blue", "Red", "Red", "Blue")
as.factor(a)
```

```
## [1] Red  Blue Blue Red  Red  Blue
## Levels: Blue Red
```

```
b <- c(1:4, 4:1)
b
```

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

```
as.factor(b)
```

```
## [1] 1 2 3 4 4 3 2 1
## Levels: 1 2 3 4
```

Data Frames

We will discuss other data structures to handle the datasets we are interested in as empiricists. The most basic structure to represent individual × variable tables is the data frame. Each row corresponds to a unit and each column corresponds to a variable. The columns must have the same type across units, although different columns can have different types. The columns have names, corresponding to the variable names.

```
t <- data.frame(Salary = c(623, 515, 611, 729, 843),
                Dpt = c("IT", "Operations", "IT", "HR", "Finance"))
t
```

```
##   Salary        Dpt
## 1    623         IT
## 2    515 Operations
## 3    611         IT
## 4    729         HR
## 5    843    Finance
```

```
dim(t)
```

```
## [1] 5 2
```

```
class(t)
```

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

```
str(t)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ Salary: num  623 515 611 729 843
##  $ Dpt   : chr  "IT" "Operations" "IT" "HR" ...
```

Vectors can be combined into a data frame.

```r
a <- 1:3
b <- letters[1:3]
c <- LETTERS[1:3]
abc <- data.frame(var1 = a, var2 = b, var3 = c)
abc
```

```
##   var1 var2 var3
## 1    1    a    A
## 2    2    b    B
## 3    3    c    C
```

The possible attributes of data frames are row names, column names, and comments.

```r
attributes(abc)
```

```
## $names
## [1] "var1" "var2" "var3"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3
```

```r
rownames(abc) <- c("first", "second", "third")
names(abc) <- c("numbers", "lower", "upper") # Alternatively, colnames()
comment(abc) <- "This is a very small dataframe."
attributes(abc)
```

```
## $names
## [1] "numbers" "lower"   "upper"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] "first"  "second" "third"
##
## $comment
## [1] "This is a very small dataframe."
```

The function `as.data.frame()` converts lists and matrices to data frames.

```r
O
```

```
##      col1 col2 col3
## new1    1    2    3
## new2    4    5    6
## new3    7    8    9
## new4   10   11   12
```

```r
Odf <- as.data.frame(O)
str(Odf)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ col1: int  1 4 7 10
##  $ col2: int  2 5 8 11
##  $ col3: int  3 6 9 12
```

Columns can be added to data frames using `cbind()` as long as the number of elements of the vector(s) is the same as the number of rows of the data frame. Rows can be added with `rbind()`, but this is not advised unless you are confident that the data types are the same in the original data frame and the new row(s). If not, the columns will silently change type.

```
t
```

```
##   Salary        Dpt
## 1    623         IT
## 2    515 Operations
## 3    611         IT
## 4    729         HR
## 5    843    Finance
```

```
new <- c(1, 1, 1, 0, 0)
t <- cbind(t, new)
t
```

```
##   Salary        Dpt new
## 1    623         IT   1
## 2    515 Operations   1
## 3    611         IT   1
## 4    729         HR   0
## 5    843    Finance   0
```

```
t <- rbind(t, c(313, "Marketing", 1))
t
```

```
##   Salary        Dpt new
## 1    623         IT   1
## 2    515 Operations   1
## 3    611         IT   1
## 4    729         HR   0
## 5    843    Finance   0
## 6    313  Marketing   1
```

The more robust way to add rows is by first formatting the new rows as a data frame and then combining the two data frames.

```
newdf <- data.frame(Salary = 701, Dpt = "Finance", new = 0)
rbind(t, newdf)
```

```
##   Salary        Dpt new
## 1    623         IT   1
## 2    515 Operations   1
## 3    611         IT   1
## 4    729         HR   0
## 5    843    Finance   0
## 6    313  Marketing   1
## 7    701    Finance   0
```

Subsetting data frames is very similar to subsetting matrices and arrays. Importantly, you should consider if you need preserving or simplifying output. Subsetting can be done by row numbers, row names, column numbers, and column names.

```
t[1:2, ]
```

```
##   Salary        Dpt new
## 1    623         IT   1
```

```
## 2    515 Operations   1
```

```
t[c("2", "3"), ]
```

```
##   Salary        Dpt new
## 2    515 Operations   1
## 3    611         IT   1
```

```
t[c("Salary", "new")] # List-type subsetting
```

```
##   Salary new
## 1    623   1
## 2    515   1
## 3    611   1
## 4    729   0
## 5    843   0
## 6    313   1
```

```
t[ , c("Salary", "new")] # Matrix-type subsetting
```

```
##   Salary new
## 1    623   1
## 2    515   1
## 3    611   1
## 4    729   0
## 5    843   0
## 6    313   1
```

```
t[3:5, 1]
```

```
## [1] "611" "729" "843"
```

```
t[ , 2] # Simplifying
```

```
## [1] "IT"         "Operations" "IT"         "HR"         "Finance"
## [6] "Marketing"
```

```
t[ , 2, drop = FALSE] # Preserving
```

```
##          Dpt
## 1         IT
## 2 Operations
## 3         IT
## 4         HR
## 5    Finance
## 6  Marketing
```

Practice Exercises 2.7

1. Create a data frame from the following list.

```
scores <- list(scrabble = c(1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3,
                            1, 1, 3, 10, 1, 1, 1, 1, 4, 4, 8, 4, 10),
               sequential = 1:26,
               name = letters)
```

2. Subset the letters with Scrabble scores larger than 3. Do this once with simplifying and once with preserving output.
3. Add a variable called `reverse` that assigns a value of 26 to A and 1 to Z.

Other Structures

Dates and time series are two other basic data structures. See chapters 3.2.2.6 and 3.2.2.7 of Lafaye de Micheaux, Drouilhet, and Liquet (2013) and chapter 8 of Boehmke (2016) for more information.

There are many other types of structures. We will learn more about structures that are particularly helpful for data analysis. We will learn how to use the basic types and structures in conditional statements, loops, and functions.

Further Reading

The above information comes from chapters 3.5-3.6, 4, 5.1, 7.1, 9-13 of Boehmke (2016) and chapters 3.1-3.2, 10.2 of Lafaye de Micheaux, Drouilhet, and Liquet (2013).

References

Boehmke, Bradley C. 2016. *Data Wrangling with R*. Use R! Springer. https://link-springer-com.proxy.lib.duke.edu/content/pdf/10.1007%2F978-3-319-45599-0.pdf.

Lafaye de Micheaux, Pierre, Rémy Drouilhet, and Benoit Liquet. 2013. *The R Software : Fundamentals of Programming and Statistical Analysis*. Statistics and Computing, 1431-8784 ; 40. New York, NY: Springer. https://find.library.duke.edu/catalog/DUKE006082111.