

Introductory R Tutorial 2: Variables, Data, and Functions

Shane T. Mueller shanem@mtu.edu Michigan Technological University

2020-05-28

[Return to main site](#) | [Lesson 1](#) | [Lesson 2](#) | [Lesson 3](#) | [Lesson 4](#) | [Lesson 5](#)
[Download Lesson 2 files here](#)

Annotation with [hypothes.is](#)

You can share annotations, questions, and answers on any of these pages using [hypothes.is](#). Use this link to join the group

Goals

The goals of this session are to introduce you to data. This includes:

- Variables
- Data types
- Data structures
- Operations on data
- Functions of data

Variables

In many stats packages, you have a single table that structures all of your data. That table usually is not named; it is simply the data you are working from. Spreadsheets give you multiple tables (sheets), but R is much more flexible about storing data, and allows you to maintain many tables, and data in other formats as well. So to handle this, we need to be able to save data into an object we can refer to and use later. To do this, we can use the assignment operator `<-` to assign the value of an expression to a variable. Here, we assign the value 3.5 to a variable we name `x`.

```
x <- 3.5
```

Once you run this, you can click on the ‘Environment’ tab in the upper right and see that it lists `x`, with value 3.5. All the variables you create will be visible in this tab. Once defined, you can use `x` in another expression, and even assign that value to another variable like this:

```
y <- x/3+5  
print(x)
```

```
## [1] 3.5
```

```
print(y)
```

```
## [1] 6.166667
```

The value of `x` does not change when you do this, R simply looks up the value assigned to `x`, makes the calculation, and assigns that new value to `y`. You can see in the environment pane that `y` has a value of

6.166667. Note that if you want the value of *y*, R does not repeat the calculations used to make *y*—it only knows the value assigned to *y*.

Understanding Data types

Data are stored in a number of formats in R, and on computers in general. There are many special-purpose formats we will not consider in this workshop, but will cover just the basics.

Logical values

Sometimes, you want to know whether something is true or false. R has a basic data type called ‘logical’ which represent these boolean values.

A value of *true* is named in one of three ways:

```
a <- TRUE
b <- T
c <- 1
```

Similarly, a value of *false* is named in three ways:

```
aa <- FALSE
bb <- F
cc <- 0
```

T is equivalent to TRUE, which is actually equivalent to 1. Similarly, FALSE is F is 0. R is case sensitive, so *t* and *false* will not work. We can test equality with the `==` operator, and this returns another logical value.

```
a == b
```

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

```
a == c
```

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

```
a == aa
```

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

```
bb == cc
```

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

Notice that all of these are equivalent, except TRUE versus FALSE.

Numbers

You might want to represent someone’s weight or age. This would generally be done in a **numeric** format, which is a high-precision number that can represent values that are very large (billions and larger), or very small (billionths or smaller). Computers refer to this as ‘double-precision’, and so R will sometimes report this as a *double*. This general purpose number format might be all you need for most data. Any time you enter a number into the R console, it has a numeric format. Both *x* and *y* have this format right now. You can determine the type using either the `str()` function or the `typeof()` function.

```
str(x)
```

```
## num 3.5
```

```
typeof(x)
```

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

```
str(y)
```

```
## num 6.17
```

```
typeof(y)
```

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

`str()` tell us it is numeric with value 3.5. `typeof()` tells is it is double-precision, which is essentially the same thing.

Integers

Sometimes, you want a small number of categories represented by numbers. For example, you might have recorded data on five different days, and want to keep track of day. Because of the way computers represent double-precision numbers, an integer value might not be exactly that number; 45 might really be something like 44.99999999999999. This is usually not a problem, but if you were to somehow calculate day 45 from a formula, you might get another value close to 45, like 45.00000000000001. Since these two are not equal, even though they appear to be, you could get into trouble. R does a pretty good job of handling these situations, but there are times when you want the exact value represented. For example:

```
(.6+.3) == .9
```

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

```
print(.3 + .6 - .9)
```

```
## [1] -1.110223e-16
```

```
print(.3,digits=18)
```

```
## [1] 0.299999999999999989
```

Integer types take care of this. You will use integers frequently, even if you don't know it, because it works behind the scenes when appropriate. You can convert a numeric value to an integer using the `as.integer` function.

```
a <- 45
```

```
b <- as.integer(a)
```

```
typeof(a)
```

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

```
typeof(b)
```

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

```
a == b
```

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

Notice that the double-45 tests as equal to the integer-45, even though they are in different formats. If we add them or multiply them, the result inherits the form of the least restrictive part; a double. Adding, subtracting or multiplying integers results in an integer. Adding, subtracting, or multiplying an integer by a double is a double, and dividing any number by any other number results in a double.

```
typeof(b+b)
```

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

```
typeof(b/b)
```

```
## [1] "double"
typeof(b*as.integer(2))
```

```
## [1] "integer"
typeof(b+b)
```

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

Characters

Sometimes, data are text values; usually because they are categories, maybe specifying a condition or a written response. These are stored as a **character** type.

```
z <- "hello"
typeof(z)
```

```
## [1] "character"
str(z)
```

```
## chr "hello"
```

Factors

Sometimes we want our categorical variables to have a bit more structure. R uses what it calls factors for representing this. They are actually a kind of mix between characters and integers; they have names, but they also have an order, so your factors can have levels. It also restricts the levels so you know exactly what all the levels you are considering are.

```
a <- factor("A", levels=LETTERS)
print(a)
```

```
## [1] A
## Levels: A 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
str(a)
```

```
## Factor w/ 26 levels "A","B","C","D",...: 1
typeof(a)
```

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

You can see the value is now A, but it could take on any value of the alphabet. The levels also have an order. `str()` shows that this is a factor with value 1 (because A is the first level). `typeof()` actually shows that internally, it is an integer, which seems a bit strange. These are very handy for any categorical data forms, where you want to ensure that you know all the possible values it can take on. They are also useful for ordered factors, like day-of-week, month-of-year, or even at times likert-style responses (although these would normally be coded as integers or numeric).

Converting between data types

For every data type, there is a function that starts with `as.` that converts any other format to that data type in as sensible a way as it can. These include `as.numeric`, `as.character`, `as.integer`, `as.factor`, and `as.logical`. These will not always convert the way you want them converted, so you should understand how it works.

Exercise 1

Create data that is an integer, numeric, factor, logical, and character by converting from some other data type. Examine cases that should work as well as cases where automatic conversion might go awry, and determine what happens, including:

- Converting a text description of a number to a number
- Convert a double-precision number to an integer. Does it round up or down or what?
- Convert a text description of a non-number to a number.
- Convert a double to a factor.
- Convert a text description of a number to a factor.
- Convert this factor back to a number.
- Convert the factor to text then a number.

Summary

These are the main data types in R. You can probably represent any kind of data in these. There are some more specialty forms that can be accessed if you really need them, possibly to take advantage of special hardware or software libraries, but these are very advanced.

Understanding Data Structures

Data are usually plural—we need a way of combining multiple values together in order to do anything interesting. We will call a way of combining data together a *data structure*. There are many data structures available in R and R libraries, often specially created for particular analyses, and there are ways to create your own special data structures. We will cover just a few here.

Vectors/Arrays

A set of values that have the same type is called a vector or an array. All of the values we have used so far are actually vectors of length 1. This is why the [1] shows up when we look at them.

Creating a vector

You can create a vector using the `c()` function, like this:

```
a <-c(1,2,3,4,5,6,7,8,9,11)
```

```
a
```

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

```
typeof(a)
```

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

```
str(a)
```

```
## num [1:10] 1 2 3 4 5 6 7 8 9 11
```

We can see how when we print this out, it prepends with [1] to help us know that that is the first element. Looking at the type, these are stored as doubles. This makes more sense when we have a long vector. We can create a long sequence using either the `seq()` function or the `:` operator, or a repeated list using the `rep()` function

```
a <- seq(0,200,5)
```

```
b <- 50:120
```

```
d <- rep(100,50)
```

```
print("This is 'a':")
```

```
## [1] "This is 'a':"
```

```
a
```

```
## [1] 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115
```

```
## [30] 145 150 155 160 165 170 175 180 185 190 195 200
```

```
typeof(a)
```

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

```
print("This is 'b':")
```

```
## [1] "This is 'b':"
```

```
b
```

```
## [1] 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
```

```
## [30] 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102
```

```
## [59] 108 109 110 111 112 113 114 115 116 117 118 119 120
```

```
typeof(b)
```

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

```
print("This is 'd':")
```

```
## [1] "This is 'd':"
```

```
d
```

```
## [1] 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
```

```
## [30] 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
```

```
typeof(d)
```

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

Selecting from a vector

Now we can see how the [1] help us identify which element of the vector has which value. You can access any element of a vector with the [] operator. Also, we can see that the seq() returns a double, but the : returns an integer, which is probably what we want when we use :

```
a[19]
```

```
## [1] 90
```

```
b[47]
```

```
## [1] 96
```

You can select more than one element of a vector, using the [] operator and a set of indexes. This is usually done as a range using :, or as another vector using c():

```
a[1:5] ##First five elements
```

```
## [1] 0 5 10 15 20
```

```
a[c(1,3,5,9,11)] ## first five odd elements
```

```
## [1] 0 10 20 40 50
```

Combining different data types in a vector

A vector can take on any data type, but only one. See what happens when we try to combine numbers and characters

```
x <- c("a", "b", 3, 4, 5)
x
```

```
## [1] "a" "b" "3" "4" "5"
```

A vector cannot combine characters and numbers, so it converts them all to characters. This can get you in trouble, because what looks like a number isn't. One place this can arise is if you read data in from an excel file, and have added additional labels at the bottom of your excel data. It will see a bunch of numbers, followed by one row of text, and not know how to handle it, so interpret it all as text or possibly a factor. So you should always be ready to check the data type when things go awry.

Matrix data type

Creating a matrix

A matrix is a 2-dimensional vector. There are some times you need to use matrices, especially for special plotting functions and analyses. But like a vector, a matrix has to be all of the same type. You can create a matrix by giving it the values you want in a vector and the rows/columns

```
mat <- matrix(1:80, nrow=10, ncol=8)
```

Now, if you look at the Environment pane in RStudio, **mat** should appear in a new category: Data. If you click on **mat**, it will open up a table in the editor pane that you can examine the data from. You cannot edit the data there, but you can look at it. Notice that it fills by columns, not rows.

Selecting from a matrix

Just like a vector, you can access an entire row of a matrix, or an individual cell, using the `[]` operator, but now you need to specify whether you want to select on the row or column. You do this with the two slots in `[,]`, which refer to the rows and columns, and you can pick out a row and column by their index (starting with 1).

```
mat <- matrix(1:49, 7)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1     8    15    22    29    36    43
## [2,]    2     9    16    23    30    37    44
## [3,]    3    10    17    24    31    38    45
## [4,]    4    11    18    25    32    39    46
## [5,]    5    12    19    26    33    40    47
## [6,]    6    13    20    27    34    41    48
## [7,]    7    14    21    28    35    42    49
```

```
mat[3,]
```

```
## [1]  3 10 17 24 31 38 45
```

```
mat[,5]
```

```
## [1] 29 30 31 32 33 34 35
```

```
mat[3,5]
```

```
## [1] 31
```

```
mat[1] ##This treats mat like a vector and picks out the first number only.
```

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

You can also pick out several columns, or a range of columns, using a sequence specified with the `:` operator. Note that we are putting a vector into a matrix to pick out a matrix

```
mat[1:2,] ##first two rows
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    8   15   22   29   36   43
## [2,]    2    9   16   23   30   37   44
```

```
mat[,3:5] ##columns 3 to 5
```

```
##      [,1] [,2] [,3]
## [1,]   15   22   29
## [2,]   16   23   30
## [3,]   17   24   31
## [4,]   18   25   32
## [5,]   19   26   33
## [6,]   20   27   34
## [7,]   21   28   35
```

```
mat[1:2,1] ##First two rows and first column.
```

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

Mixing data types in a matrix

Also, like a vector, we cannot mix data types, and it will convert them to the least restrictive type.

```
mat2 <- matrix(c("a","b","c",1,2,3),ncol=2)
mat2
```

```
##      [,1] [,2]
## [1,] "a"  "1"
## [2,] "b"  "2"
## [3,] "c"  "3"
```

If you look at `mat2` in the viewer window, it looks fine, but looking more carefully, the second column is not numbers; it is text that looks like numbers.

So, a matrix seems like it would be good for representing the kinds of data we often have. Maybe we have a survey with 100 respondents and 10 Likert-scale questions rated 1 to 5. But our data usually is not like that. We probably also have a demographic question or two which are not 1 to 5, or a participant code which might have letters in it and probably shouldn't be treated like a number even if it is a number. We cannot combine these two kinds of data in a matrix, and we need something else: a data frame.

Data frames

A data frame looks like a matrix, but it is really a set of linked vectors that all have to be the same length, but each can be a different format. This is akin to what most statistics packages have, but more structured than a spreadsheet. If you have a valid data frame, you can be sure that each row represents a single case of values associated with one observation, subject, case, or condition.

Creating a data frame.

You can create a data frame from a matrix, or from a set of vectors, using the `data.frame()` command. But usually, you create one by reading it in from a file.

Example:

```
frame<- data.frame(time = 1:5,
                   letters = LETTERS[1:10],
                   truth = c(F,F,F,T,T,T,F,F,T,T))
```

frame

```
##      time letters truth
## 1      1      A FALSE
## 2      2      B FALSE
## 3      3      C FALSE
## 4      4      D  TRUE
## 5      5      E  TRUE
## 6      1      F  TRUE
## 7      2      G FALSE
## 8      3      H FALSE
## 9      4      I  TRUE
## 10     5      J  TRUE
```

Click on the frame in the environment pane and see how it contains named columns of different types.

Selecting from a data frame.

If you want to access a specific column, you can refer to it with a `[]` operator, or by name using `$` operator. Notice that by referring to a column, you get a vector. By referring to a row, you get a 1-row data frame. By referring to the column by name, you get a data frame with one variable. Also, you can use the same range selection methods as with a matrix. Notice what happens when we give bad arguments for selection.

```
frame[,1]      ## Select one value/variable/column
```

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

```
frame$time     ## Select first variable by name
```

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

```
frame[1,]      ##select one row.
```

```
##      time letters truth
## 1      1      A FALSE
```

```
frame[-1,]     ##remove instead of select.
```

```
##      time letters truth
## 2      2      B FALSE
## 3      3      C FALSE
## 4      4      D  TRUE
## 5      5      E  TRUE
## 6      1      F  TRUE
## 7      2      G FALSE
## 8      3      H FALSE
## 9      4      I  TRUE
## 10     5      J  TRUE
```

```
frame["time"]  ##select by name
```

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

```
frame[1:5,]    ## select first five rows
```

```
##      time letters truth
## 1      1      A FALSE
## 2      2      B FALSE
## 3      3      C FALSE
## 4      4      D  TRUE
## 5      5      E  TRUE
```

```
frame[,1:2]    ##select first two columns
```

```
##      time letters
## 1      1      A
## 2      2      B
## 3      3      C
## 4      4      D
## 5      5      E
## 6      1      F
## 7      2      G
## 8      3      H
## 9      4      I
## 10     5      J
```

This will fail and break the markdown file. Copy into the console to see what happens

```
#frame[,1:10]    ##Select first ten columns
```

```
frame[1:12,]    ## select first 12 rows
```

```
##      time letters truth
## 1      1      A FALSE
## 2      2      B FALSE
## 3      3      C FALSE
## 4      4      D  TRUE
## 5      5      E  TRUE
## 6      1      F  TRUE
## 7      2      G FALSE
## 8      3      H FALSE
## 9      4      I  TRUE
## 10     5      J  TRUE
## NA     NA     <NA>   NA
## NA.1   NA     <NA>   NA
```

Reading a data frame from a file.

More commonly, you read a data frame in directly from a file. You will often read data in from an excel spreadsheet (and there is a wizard in the File|Import data set menu to support this). If you have a simple .csv file, you can read a data set in with `read.csv`. I have a file called `data.csv`, which we can load right now. Notice that it has three different types of data, and they are read in in their appropriate data form.

```
data <- read.csv("data.csv")
data
```

```
##   time number letter
## 1     1     1.5     A
## 2     2     1.3     C
## 3     3     2.1     F
## 4     4     3.1     B
## 5     5     1.2     A
```

```
data$time
```

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

```
data$number
```

```
## [1] 1.5 1.3 2.1 3.1 1.2
```

```
data$letter
```

```
## [1] A C F B A
## Levels: A B C F
```

```
str(data$time)
```

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

```
str(data$number)
```

```
## num [1:5] 1.5 1.3 2.1 3.1 1.2
```

```
str(data$letter)
```

```
## Factor w/ 4 levels "A","B","C","F": 1 3 4 2 1
```

But look what happens if we have a data file we have done some analysis in already, in excel or another spreadsheet:

```
data.bad <- read.csv("data-bad.csv")
data.bad
```

```
##   time number letter
## 1     1     1.5     A
## 2     2     1.3     C
## 3     3     2.1     F
## 4     4     3.1     B
## 5     5     1.2     A
## 6    NA
## 7    NA   Mean
## 8    NA   1.84
```

```
typeof(data.bad$number)
```

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

```
str(data.bad$number)
```

```
## Factor w/ 8 levels "", "1.2", "1.3", ...: 4 3 6 7 2 1 8 5
```

If we look closely, we see that the second column, which included the average, is now coded as a factor, which is interpreted as an integer! It adds several blank/missing rows to the data frame for the other variables as well. You should always inspect your data when you read it in because these things can occur. The easiest approach is to edit the .csv file so it only has data in it. You can also use the `skip` and `nrow` arguments to help filter out specific rows. Here, we can specify `nrow=5`:

```
data.bad <- read.csv("data-bad.csv", nrow=5)
data.bad
```

```
##   time number letter
## 1    1    1.5      A
## 2    2    1.3      C
## 3    3    2.1      F
## 4    4    3.1      B
## 5    5    1.2      A
```

```
typeof(data.bad$number)
```

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

```
str(data.bad$number)
```

```
## num [1:5] 1.5 1.3 2.1 3.1 1.2
```

Now, it reads number column as a double, and ignores the bad rows.

Conversion between data structures

You can convert between data structures using `as.vector()`, `as.matrix()`, and `as.data.frame()`. Recognize that this will also often convert the data type, because vectors and matrices need to be all of the same type. Furthermore, you can convert the type of a vector, but it sometimes has unintended consequences.

Exercise 2: some strange cases of conversions

Converting usually works well, but there are some strange cases you should be wary of. Try to predict what will happen in these cases, before seeing the results.

```
as.vector(matrix(1:10, ncol=2))
```

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

```
as.factor(c("3.1", "2.5", "119.5"))
```

```
## [1] 3.1 2.5 119.5
## Levels: 119.5 2.5 3.1
```

```
as.numeric(as.factor(c("3.1", "2.5", "119.5")))
```

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

```
as.numeric(as.character(as.factor(c("3.1", "2.5", "119.5"))))
```

```
## [1] 3.1 2.5 119.5
```

```
as.factor(c(10.1, 3.1, 5.3))
```

```
## [1] 10.1 3.1 5.3
## Levels: 3.1 5.3 10.1
```

```
as.data.frame(matrix(1:20,nrow=5))
```

```
##   V1 V2 V3 V4
## 1  1  6 11 16
## 2  2  7 12 17
## 3  3  8 13 18
## 4  4  9 14 19
## 5  5 10 15 20
```

```
as.matrix(data.frame(a=LETTERS[1:10],b=1:10))
```

```
##      a  b
## [1,] "A" " 1"
## [2,] "B" " 2"
## [3,] "C" " 3"
## [4,] "D" " 4"
## [5,] "E" " 5"
## [6,] "F" " 6"
## [7,] "G" " 7"
## [8,] "H" " 8"
## [9,] "I" " 9"
## [10,] "J" "10"
```

Operations on data

Overview

Many of the calculator operations we used on individual numbers work on data vectors and matrices as well. Suppose we have a data frame with two observations, maybe the measured heart rate of an individual at two different points in time. The code below creates this, but don't worry about what it is doing

```
hr1 <- round(runif(10)*50+50,1)
hr2 <- round(hr1 + runif(10)*20,1)
hr <- data.frame(hr1,hr2)
hr
```

```
##      hr1  hr2
## 1  74.6 79.7
## 2  89.0 91.5
## 3  94.2 98.8
## 4  60.4 72.4
## 5  65.4 69.6
## 6  66.5 75.8
## 7  59.9 72.8
## 8  61.8 81.0
## 9  63.7 77.2
## 10 79.6 88.5
```

Let's suppose we determined the hr2 was biased because of a bad sensor so it is 10 units too high. If we want to adjust it, we can use an R expression. The following gets the hr2 vector and subtracts 10 from each measurement. This does not change the hr data frame at all, it just returns the vector of numbers that have been changed.

```
hr$hr2 - 10
```

```
## [1] 69.7 81.5 88.8 62.4 59.6 65.8 62.8 71.0 67.2 78.5
```

We can add it back to `hr` as a new data variable like this:

```
hr$new.hr2 <- hr$hr2-10  
hr
```

```
##      hr1  hr2 new.hr2  
## 1  74.6 79.7   69.7  
## 2  89.0 91.5   81.5  
## 3  94.2 98.8   88.8  
## 4  60.4 72.4   62.4  
## 5  65.4 69.6   59.6  
## 6  66.5 75.8   65.8  
## 7  59.9 72.8   62.8  
## 8  61.8 81.0   71.0  
## 9  63.7 77.2   67.2  
## 10 79.6 88.5   78.5
```

We can see we have a new data variable, recoded from `hr2`. Suppose we want to find the average of `hr1` and `new.hr2`. We can write another expression like this, adding the new values back into the data frame:

```
hr$hrmean <- (hr$hr1 + hr$new.hr2)/2  
hr
```

```
##      hr1  hr2 new.hr2 hrmean  
## 1  74.6 79.7   69.7  72.15  
## 2  89.0 91.5   81.5  85.25  
## 3  94.2 98.8   88.8  91.50  
## 4  60.4 72.4   62.4  61.40  
## 5  65.4 69.6   59.6  62.50  
## 6  66.5 75.8   65.8  66.15  
## 7  59.9 72.8   62.8  61.35  
## 8  61.8 81.0   71.0  66.40  
## 9  63.7 77.2   67.2  65.45  
## 10 79.6 88.5   78.5  79.05
```

We added another new variable that is the average of our two measures. Notice that the two vectors are the same length, and so they get added together item-by-item. Then, each one gets divided by 2.

Exercise 3: Applying operations

These data represent heart rate in beats per minute. Make a new variable which calculates the R-R interval—the time between heart beats, in milliseconds, from the mean HR data.

Exercise 4: Selection and data frame management

- Create a new data frame using just `hr1`, `new.hr2`, and `hrmean`
- Add a variable indicating the sample number 1..10
- Select only the odd rows of this new data frame.

Applying functions to data

We previously saw how a function can be run with different arguments. A statistic is a function applied to data. For something like “mean”, the function involves adding all the values up and dividing by the number of values. Many functions take a data vector as their argument, and return values computed by them.

Let's consider three functions: `mean()`, `sd()`, and `range()`. We will apply them each to the `hr1` data vector we already defined:

```
mean(hr1)
```

```
## [1] 71.51
```

```
sd(hr1)
```

```
## [1] 12.34535
```

```
range(hr1)
```

```
## [1] 59.9 94.2
```

We can see that the first two produced an intuitive result. The third one actually returned a vector of two values—the minimum and maximum values in our data. We can access them just like we access any other data vector.

However, you need to understand the kind of data you are applying the function to in order to get the right results. Let's say our `hr` data was stored in a matrix, so we have vector, matrix, and data frames. We can do it like this

```
hr.matrix <- as.matrix(hr)
hr.matrix
```

```
##      hr1  hr2 new.hr2 hrmean
## [1,] 74.6 79.7   69.7  72.15
## [2,] 89.0 91.5   81.5  85.25
## [3,] 94.2 98.8   88.8  91.50
## [4,] 60.4 72.4   62.4  61.40
## [5,] 65.4 69.6   59.6  62.50
## [6,] 66.5 75.8   65.8  66.15
## [7,] 59.9 72.8   62.8  61.35
## [8,] 61.8 81.0   71.0  66.40
## [9,] 63.7 77.2   67.2  65.45
## [10,] 79.6 88.5   78.5  79.05
```

```
mean(hr.matrix)
```

```
## [1] 73.5225
```

```
mean(hr)
```

```
## Warning in mean.default(hr): argument is not numeric or logical: returning NA
```

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

Notice that `mean(hr)` fails, but `mean(hr.matrix)` succeeds even though these are the exact same numbers. But also notice that neither of these are probably what you are really interested in. You normally want to apply the mean to just one column or variable, like this:

```
mean(hr$hr1)
```

```
## [1] 71.51
```

```
mean(hr.matrix[,1])
```

```
## [1] 71.51
```

These give the mean of the first column—a value we might care about.

Exercise 5

In the following data set, we observed 15 subjects and measured three things about them (a, b, and c). Find the mean of each variable, and then find the mean of all of the data values in the data frame. The first column is a letter associated with a participant, and so should not be included in any means.

```
set.seed(100)
data1 <- data.frame(subject=LETTERS[1:15],a=round(runif(15),3),b=1:15,c=round(rnorm(15),3))
data1
```

```
##   subject      a  b      c
## 1      A 0.308  1  0.437
## 2      B 0.258  2 -0.365
## 3      C 0.552  3  0.497
## 4      D 0.056  4  0.556
## 5      E 0.469  5  0.671
## 6      F 0.484  6 -0.949
## 7      G 0.812  7  1.185
## 8      H 0.370  8 -0.590
## 9      I 0.547  9  1.465
## 10     J 0.170 10  1.687
## 11     K 0.625 11  1.224
## 12     L 0.882 12  0.330
## 13     M 0.280 13 -1.125
## 14     N 0.398 14  1.104
## 15     O 0.763 15  0.944
```

Exercise 6

Read in the following data, and again find the mean of each variable, and the mean of all of the variables you are able to.

```
data2 <- read.csv("data2.csv")
data2
```

```
##   time  number letter
## 1     1 3.282430      A
## 2     2 5.498959      C
## 3     3 4.214320      F
## 4     4 5.629371      B
## 5     5 4.679285      A
## 6     6 3.571785      D
## 7     7 3.230984      E
## 8     8 4.761748      G
## 9     9 5.306423      A
## 10    10 3.635993      B
## 11    11 5.642984      B
## 12    12 5.968484      C
## 13    13 4.148730      D
## 14    14 3.953843      E
## 15    15 4.086184      F
## 16    16 4.151948      A
## 17    17 3.248225      C
## 18    18 4.001791      B
## 19    19 5.771295      G
## 20    20 3.104298      B
```


Summary

In this lesson, you should have learned about different data types and data structures. You should understand how sometimes one type or structure can look like another, but you can always tell them apart. You should understand why we have different types and structures, and be able to know the right format you should use in R. You should also know about functions, and how to apply them to data, especially for cases when a function requires a specific data type or structure.

Exercise Solutions

Exercise 1

Create data that is an integer, numeric, factor, logical, and character from some other data type. Examine both cases that should work and cases where automatic conversion might go awry, and determine what happens, including:

```
##Converting a text description of a number to a number
##    Convert a double-precision number to an integer. Does it round up or down or what?
as.integer(33.1)
```

```
## [1] 33
```

```
as.integer(33.9)
```

```
## [1] 33
```

```
##    Convert a text description of a non-number to a number.
as.numeric("322.5")
```

```
## [1] 322.5
```

```
##    Convert a double to a factor.
as.factor(33.1311)
```

```
## [1] 33.1311
## Levels: 33.1311
```

```
##    Convert a text description of a numbers to a factor.
as.factor("33.1")
```

```
## [1] 33.1
## Levels: 33.1
```

```
##    Convert this factor back to a number.
as.numeric(as.factor("33.1"))
```

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

```
##    Convert the factor to text then a number
as.numeric(as.character(as.factor("33.1")))
```

```
## [1] 33.1
```

Exercise 2: Some strange cases of conversions

Converting usually works well, but there are some strange cases you should be wary of. Try to predict what will happen in these cases, before seeing the results.

```
as.vector(matrix(1:10,ncol=2)) # this works OK: numbers 1:10
```

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

```

as.factor(c("3.1","2.5","119.5")) ##factor levels look like numbers but are not. They are not in numeri

## [1] 3.1    2.5    119.5
## Levels: 119.5 2.5 3.1

as.numeric(as.factor(c("3.1","2.5","119.5"))) ##reconverts these to integers, in lexicographical (alpha

## [1] 3 2 1

as.numeric(as.character(as.factor(c("3.1","2.5","119.5")))) ##success!

## [1]    3.1    2.5 119.5

as.factor(c(10.1,3.1,5.3)) ##Levels of the factor are in numeric order

## [1] 10.1 3.1  5.3
## Levels: 3.1 5.3 10.1

as.data.frame(matrix(1:20,nrow=5)) ##Converts columns to named variables V1..V4

##      V1 V2 V3 V4
## 1     1  1  6 11 16
## 2     2  2  7 12 17
## 3     3  3  8 13 18
## 4     4  4  9 14 19
## 5     5  5 10 15 20

as.matrix(data.frame(a=LETTERS[1:10],b=1:10)) ##Converts numbers to characters

##      a  b
## [1,] "A" " 1"
## [2,] "B" " 2"
## [3,] "C" " 3"
## [4,] "D" " 4"
## [5,] "E" " 5"
## [6,] "F" " 6"
## [7,] "G" " 7"
## [8,] "H" " 8"
## [9,] "I" " 9"
## [10,] "J" "10"

```

Exercise 3.

These data represent heart rate in beats per minute. Make a new variable which calculates the R-R interval—the time between heart beats, in milliseconds, from the mean HR data.

```

## find minutes per heart beat:
rr.min <- 1/hr$hrmean

##calculate it in milliseconds
rr.sec <- rr.min * 60*1000
hr$rrms <- round(rr.sec)
hr$rrms

```

```
## [1] 832 704 656 977 960 907 978 904 917 759
```

These make sense—with BPMs around 60, we see individual times per heart beat at around a second, or 1000 ms.

Exercise 4: Selection and data frame management

- Create a new data frame using just `hr1`, `new.hr2`, and `hrmean`
- Add a variable indicating the sample number 1..10
- Select only the odd rows of this new data frame.

```
newdat <- data.frame(hr1=hr$hr1,hr2 = hr$new.hr2,mean=hr$hrmean)
newdat$sample <- 1:10
newdat2 <- newdat[(1:5)*2-1,] ##this will be 1,3,5,7,9. Be sure to put sequence in parens for clarity
newdat
```

```
##      hr1  hr2  mean sample
## 1  74.6 69.7 72.15      1
## 2  89.0 81.5 85.25      2
## 3  94.2 88.8 91.50      3
## 4  60.4 62.4 61.40      4
## 5  65.4 59.6 62.50      5
## 6  66.5 65.8 66.15      6
## 7  59.9 62.8 61.35      7
## 8  61.8 71.0 66.40      8
## 9  63.7 67.2 65.45      9
## 10 79.6 78.5 79.05     10
```

```
newdat2
```

```
##      hr1  hr2  mean sample
## 1  74.6 69.7 72.15      1
## 3  94.2 88.8 91.50      3
## 5  65.4 59.6 62.50      5
## 7  59.9 62.8 61.35      7
## 9  63.7 67.2 65.45      9
```

Exercise 5

In the following data set, we observed 15 subjects and measured three things about them (variables `a`, `b`, and `c`). Find the mean of each variable, and then find the mean of all of the data values in the data frame. The first column is a letter associated with a participant, and so should not be included in any means.

```
set.seed(100)
data1 <- data.frame(subject=LETTERS[1:15],a=round(runif(15),3),
                    b=1:15,
                    c=round(rnorm(15),3))
```

```
mean(data1) ##This won't work
```

```
## Warning in mean.default(data1): argument is not numeric or logical: returning NA
```

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

```
mean(data1$subject) ##This fails
```

```
## Warning in mean.default(data1$subject): argument is not numeric or logical: returning NA
```

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

```
mean(data1$a)
```

```
## [1] 0.4649333
```

```

mean(data1$b)

## [1] 8
mean(data1$c)

## [1] 0.4714
##We can put together the three means to form a grand mean:
mean(c(mean(data1$a), mean(data1$b), mean(data1$c)) )

## [1] 2.978778

```

Exercise 6

Read in the following data, and again find the mean of each variable, and the mean of all of the variables you are able to.

```

data2 <- read.csv("data2.csv")
data2

```

```

##      time  number letter
## 1      1 3.282430      A
## 2      2 5.498959      C
## 3      3 4.214320      F
## 4      4 5.629371      B
## 5      5 4.679285      A
## 6      6 3.571785      D
## 7      7 3.230984      E
## 8      8 4.761748      G
## 9      9 5.306423      A
## 10     10 3.635993      B
## 11     11 5.642984      B
## 12     12 5.968484      C
## 13     13 4.148730      D
## 14     14 3.953843      E
## 15     15 4.086184      F
## 16     16 4.151948      A
## 17     17 3.248225      C
## 18     18 4.001791      B
## 19     19 5.771295      G
## 20     20 3.104298      B

```

We can again see that only the first two variables provide numbers we can take the mean of:

```

mean(data2$time)

## [1] 10.5
mean(data2$number)

## [1] 4.394454
mean(data2$letter) ##this should fail!

## Warning in mean.default(data2$letter): argument is not numeric or logical: returning NA
## [1] NA

```