Module 3

Understand: Understanding Data and Data Structures

Dr. Anil Dolgun

Last updated: 09 April, 2018

Overview

Summary

Importing data successfully doesn't mean that we have all the information about our data. Understanding data structures and variable types in the data set are also crucial for conducting data preprocessing. We shouldn't be performing any type of data preprocessing without understanding what we have in hand. In this module, I will provide the basics of variable types and data structures. You will learn to check the types of the variables, dimensions and structure of the data, and levels/values for the variables. We will also cover how to manipulate the format of the data (i.e., data type conversions). Finally, the difference between wide and long formatted data will be explained.

Learning Objectives

The learning objectives of this module are as follows:

- Understand R's basic data types (i.e., character, numeric, integer, factor, and logical).
- Understand R's basic data structures (i.e., vector, list, matrix, and data frame) and main differences between them.

- Learn to check attributes (i.e., name, dimension, class, levels etc.) of R objects.
- Learn how to convert between data types/structures.
- Understand the difference between wide vs. long formatted data.

Types of variables

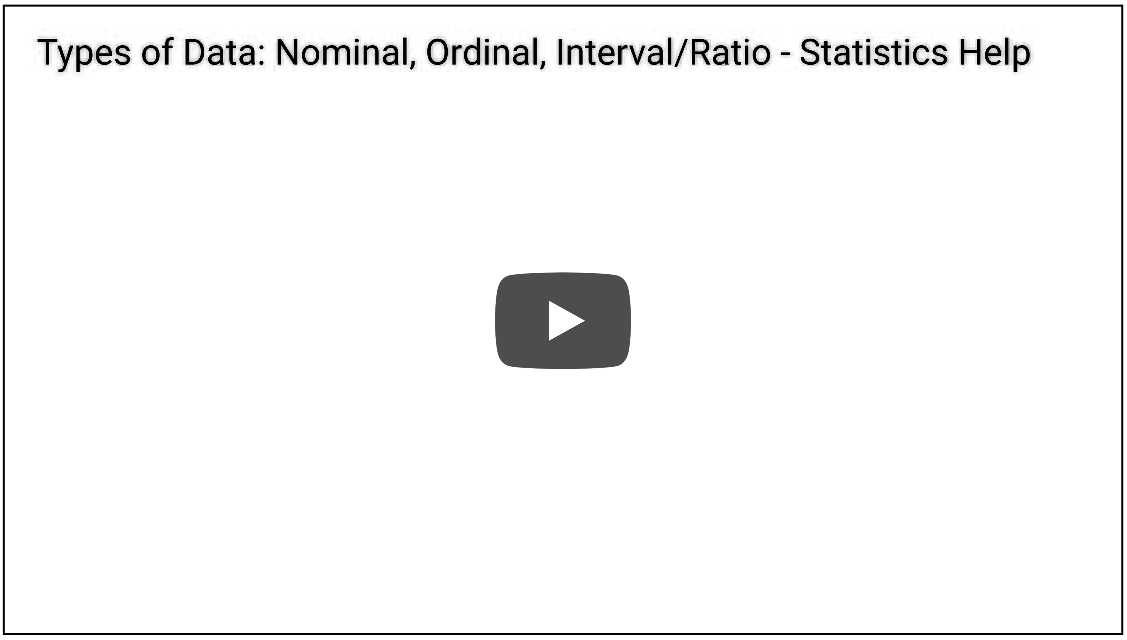
A data set is a collection of measurements or records which are often called as variables and there are two major types of variables that can be stored in a data set: qualitative and quantitative. The **qualitative variable** is often called as **categorical** and they have a non-numeric structure such as gender, hair colour, type of a disease, etc. The qualitative variable can be nominal or ordinal.

- **Nominal variable**: They have a scale in which the numbers or letters assigned to objects serve as labels for identification or classification. Examples of this variable include binary variables (e.g., yes/no, male/female) and multinomial variables (e.g. religious affiliation, eye colour, ethnicity, suburb).
- **Ordinal variable**: They have a scale that arranges objects or alternatives according to their ranking. Examples include the exam grades (i.e., HD, DI, Credit, Pass, Fail etc.) and the disease severity (i.e., severe, moderate, mild).

The second type of variable is called the **quantitative variable**. These variables are the numerical data that we can either measure or count. The quantitative variables can be either **discrete** or **continuous**.

- **Continuous quantitative variable:** They arise from a measurement process. Continuous variables are measured on a continuum or scale. They can have almost any numeric value and can be meaningfully subdivided into finer and finer increments, depending upon the precision of the measurement system. For example: time, temperature, wind speed may be considered as continuous quantitative variables.
- **Discrete quantitative variable:** They arise from a counting process. Examples include the number of text messages you sent this past week and the number of faults in a manufacturing process.

The following short video by Nicola Petty provides a great overview on the variable types. Note that, in some statistical sources, the "type of the data" and the "type of the variables" are used synonymously. In the following video, the term "**types of data**" are used to refer the "**types of variables**".



Data Structures in R

In the previous section, we defined the types of variables in a general sense. However, as R is a programming language it has own definitions of data types and structures. Technically, R classifies all the different types of data into four classes:

- **Logical**: This class consists of TRUE or FALSE (binary) values. A logical value is often created via comparison between variables.

```
x <- 10
y <- (x > 0)
y
```

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

We can use `class()` function to check the class of an object.

```
# check the class of y

class(y)
```

```
## [1] "logical"
```

- **Numeric** (integer or double): Quantitative values are called as numerics in R. It is the default computational data type. Numeric class can be integer or double. Integer types can be seen as discrete values (e.g., 2) whereas, double class will have floating point numbers (e.g., 2.16).

Here is an example of a double numeric variable:

```
# create a double-precision numeric variable

dbl_var <- c(4, 7.5, 14.5)

# check the class of dbl_var

class(dbl_var)
```

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

To check whether a numeric object is integer or double, you can also use `typeof()` .

```
# check the type of dbl_var object

typeof(dbl_var)
```

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

In order to create an integer variable, we must place an `L` directly after each number. Here is an example:

```
# create an integer (numeric) variable

int_var <- c(4L, 7L, 14L)

# check the class of int_var

class(int_var)
```

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

- **Character**: A character class is used to represent string values in R. The most basic way to generate a character object is to use quotation marks `" "` and assign a string/text to an object.

```
# create a character variable using " " and check its class

char_var <- c("debit", "credit", "Paypal")

class(char_var)
```

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

- **Factor**: Factor class is used to represent qualitative data in R. Factors can be ordered or unordered. Factors store the nominal values as a vector of integers in the range [$1 \ldots k$] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers.

Factor objects can be created with the `factor()` function:

```
# create a factor variable using factor()

fac_var1 <- factor( c("Male", "Female", "Male", "Male") )
fac_var1
```

```
## [1] Male   Female Male   Male
## Levels: Female Male
```

```
# check its class

class(fac_var1)
```

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

To see the levels of a factor object `levels()` function will be used:

```
# check the factor levels

levels(fac_var1)
```

```
## [1] "Female" "Male"
```

By default, the levels of the factors will be ordered alphabetically. Using the `levels()` argument, we can control the ordering of the levels while creating a factor:

```
# create a factor variable using factor() and order the levels using levels()
 argument

fac_var2 <- factor( c("Male", "Female", "Male", "Male"),
                    levels = c("Male", "Female") )
fac_var2
```

```
## [1] Male   Female Male   Male
## Levels: Male Female
```

```
# check its levels

levels(fac_var2)
```

```
## [1] "Male"   "Female"
```

We can also create ordinal factors in a specific order using the `ordered = TRUE` argument:

```
# create a ordered factor variable using factor() and order the levels using
 levels() argument

ordered_fac <-factor( c("DI", "HD", "PA", "NN", "CR", "DI", "HD", "PA"),
                       levels = c("NN", "PA", "CR", "DI", "HD"), ordered=TRUE
 )

ordered_fac
```

```
## [1] DI HD PA NN CR DI HD PA
## Levels: NN < PA < CR < DI < HD
```

The ordering will be reflected as `NN < PA < CR < DI < HD` in the output.

As mentioned previously, a data set is a collection of measurements or records which can be in any class (i.e., logical, character, numeric, factor, etc.). Typically, data sets contain many variables of different length and type of values. In R, we can store data sets using vectors, lists, matrices and data frames. In R, vectors, lists, matrices, arrays and data frames are called **“Data Structures”**.

According to Wickham (2014), R’s base data structures can be organised by their dimensionality (i.e., one-dimension, two-dimension, or n-dimension) and whether they’re homogeneous (i.e., all contents/variables must be of the same type) or heterogeneous (i.e., the contents/variables can be of different types). Therefore, there are five data structures given in the following table (adapted from Advanced R, Wickham (2014).)

Dimension	Homogeneous	Heterogeneous
one-dimension	Atomic vector	List
two-dimension	Matrix	Data frame
n-dimension	Array	–

In this section, we won’t cover the multi-dimensional arrays, but we will go into the details of vectors, lists, matrices, and data frames.

Vectors

A vector is the basic structure in R, which consists of one-dimensional sequence of data elements of the same basic type (i.e., integer , double , logical, or character). Vectors are created by combining multiple elements into one dimensional array using the `c()` function. The one-dimensional examples illustrated previously are considered vectors.

```
# a double numeic vector

dbl_var <- c(4, 7.5, 14.5)

# an integer vector

int_var <- c(4L, 7L, 14L)

# a logical vector

log_var <- c(T, F, T, T)

# a character vector

char_var <- c("debit", "credit", "Paypal")
```

All elements of a vector must be the same type, if you attempt to combine different types of elements they will be coerced to the most flexible type possible. Here are some examples:

```
# vector of characters and numerics will be coerced to a character vector

ex1 <- c("a", "b", "c", 1, 2, 3)

# check the class of ex1

class(ex1)
```

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

```
# vector of numerics and logical will be coerced to a numeric vector

ex2 <- c(1, 2, 3, TRUE, FALSE)

# check the class of ex2

class(ex2)
```

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

```
# vector of logical and characters will be coerced to a character vector

ex3 <- c(TRUE, FALSE, "a", "b", "c")

# check the class of ex3

class(ex3)
```

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

In order to add additional elements to a vector we can use `c()` function.

```
# add two elements (4 and 6) to the ex2 vector

ex4 <- c(ex2, 4, 6)

ex4
```

```
## [1] 1 2 3 1 0 4 6
```

To subset a vector, we can use square brackets `[ ]` with positive/negative integers, logical values or names. Here are some examples:

```
# take the third element in ex4 vector

ex4[3]
```

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

```
# take the first three elements in ex4 vector

ex4[1:3]
```

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

```
# take the first, third, and fifth element

ex4[c(1,3,5)]
```

```
## [1] 1 3 0
```

```
# take all elements except first

ex4[-1]
```

```
## [1] 2 3 1 0 4 6
```

```
# take all elements less than 3

ex4[ ex4 < 3 ]
```

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

Lists

A list is an R structure that allows you to combine elements of different types and lengths. In order to create a list we can use the `list()` function.

```
# create a list using list() function

list1 <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.5, 4.2))

# check the class of list1

class(list1)
```

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

To see the detailed structure within an object we can use the structure function `str()`, which provides a compact display of the internal structure of an R object.

```
# check the structure of the list1 object

str(list1)
```

```
## List of 4
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.5 4.2
```

Note how each of the four list items above are of different classes (integer, character, logical, and numeric) and different lengths.

In order to add on to lists we can use the `append()` function. Let's add a fifth element to the `list1` and store it as `list2`:

```
# add another list c("credit", "debit", "Paypal") on list1

list2 <- append(list1, list(c("credit", "debit", "Paypal")))

# check the structure of the list2 object

str(list2)
```

```
## List of 5
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.5 4.2
##  $ : chr [1:3] "credit" "debit" "Paypal"
```

R objects can also have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. 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. Let's check if `list2` has any attributes.

```
attributes(list2)
```

```
## NULL
```

We can add names to lists using `names()` function.

```
# add names to a pre-existing list

names(list2) <- c ("item1", "item2", "item3", "item4", "item5")

str(list2)
```

```
## List of 5
##  $ item1: int [1:3] 1 2 3
##  $ item2: chr "a"
##  $ item3: logi [1:3] TRUE FALSE TRUE
##  $ item4: num [1:2] 2.5 4.2
##  $ item5: chr [1:3] "credit" "debit" "Paypal"
```

Now, you can see that each element has a name and the names are displayed after a dollar `$` sign.
In order to subset lists, we can use dollar `$` sign or square brackets `[ ]`. Here are some examples:

```
# take the first list item in list2

list2[1]
```

```
## $item1
## [1] 1 2 3
```

```
# take the first list item in list2 using $

list2$item1
```

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

```
# take the third element out of fifth list item

list2$item5[3]
```

```
## [1] "Paypal"
```

```
# take multiple list items

list2[c(1,3)]
```

```
## $item1
## [1] 1 2 3
##
## $item3
## [1]  TRUE FALSE  TRUE
```

Matrices

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. In R, the elements of a matrix must be of same class (i.e. all elements must be numeric, or character, etc.) and all columns of a matrix must be of same length.

We can create a matrix using the `matrix()` function using `nrow` and `ncol` arguments.

```
# create a 2x3 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 can be seen using `str()` and `attributes()` functions as follows:

```
str(m1)
```

```
##  int [1:2, 1:3] 1 2 3 4 5 6
```

```
attributes(m1)
```

```
## $dim
## [1] 2 3
```

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

```
# create two vectors

v1 <- c( 1, 4, 5)
v2 <- c( 6, 8, 10)

# create a matrix using column-bind

m2 <- cbind(v1, v2)
m2
```

```
##      v1 v2
## [1,]  1  6
## [2,]  4  8
## [3,]  5 10
```

```
# create a matrix using row-bind

m3 <- rbind(v1, v2)
m3
```

```
##    [,1] [,2] [,3]
## v1    1    4    5
## v2    6    8   10
```

We can also use `cbind()` and `rbind()` functions to add onto matrices.

```
v3 <- c(9, 8, 7)

m4 <- rbind(m3, v3)
m4
```

```
##    [,1] [,2] [,3]
## v1    1    4    5
## v2    6    8   10
## v3    9    8    7
```

We can add names to the rows and columns of a matrix using `rownames` and `colnames`. Let's add row names as subject1, subject2, and subject3 and column names as var1, var2, and var3 for `m4`:

```
# add row names to m4

rownames(m4) <- c("subject1", "subject2", "subject3")

# add column names to m4

colnames(m4) <- c("var1", "var2", "var3")


# check attributes

attributes(m4)
```

```
## $dim
## [1] 3 3
##
## $dimnames
## $dimnames[[1]]
## [1] "subject1" "subject2" "subject3"
##
## $dimnames[[2]]
## [1] "var1" "var2" "var3"
```

In order to subset matrices we use the `[` operator. As matrices have two 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 it using matrix m4:

```
m4
```

```
##          var1 var2 var3
## subject1    1    4    5
## subject2    6    8   10
## subject3    9    8    7
```

```
# take the value in the first row and second column

m4[1,2]
```

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

```
# subset for rows 1 and 2 but keep all columns

m4[1:2, ]
```

```
##          var1 var2 var3
## subject1    1    4    5
## subject2    6    8   10
```

```
# subset for columns 1 and 3 but keep all rows

m4[ , c(1, 3)]
```

```
##          var1 var3
## subject1    1    5
## subject2    6   10
## subject3    9    7
```

```
# subset for both rows and columns

m4[1:2, c(1, 3)]
```

```
##          var1 var3
## subject1    1    5
## subject2    6   10
```

```
# use column names to subset

m4[ , "var1"]
```

```
## subject1 subject2 subject3
##        1        6        9
```

```
# use row names to subset

m4["subject1" , ]
```

```
## var1 var2 var3
##    1    4    5
```

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. A data frame is a list of equal-length vectors and they can store different classes of objects in each column (i.e., numeric, character, factor).

Data frames are usually created by importing/reading in a data set using the functions covered in Module 2. 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 the following example, we will create a simple data frame `df1` and assess its basic structure:

```
#  create a data frame using data.frame()

df1 <- data.frame (col1 = 1:3,
                   col2 = c ("credit", "debit", "Paypal"),
                   col3 = c (TRUE, FALSE, TRUE),
                   col4 = c (25.5, 44.2, 54.9))

# inspect its structure

str(df1)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ col1: int  1 2 3
##  $ col2: Factor w/ 3 levels "credit","debit",..: 1 2 3
##  $ col3: logi  TRUE FALSE TRUE
##  $ col4: num  25.5 44.2 54.9
```

In the example above, `col2` is 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:

```
# use stringsAsFactors = FALSE

df1 <- data.frame (col1 = 1:3,
                   col2 = c ("credit", "debit", "Paypal"),
                   col3 = c (TRUE, FALSE, TRUE),
                   col4 = c (25.5, 44.2, 54.9),
                   stringsAsFactors = FALSE)

# inspect its structure

str(df1)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ col1: int  1 2 3
##  $ col2: chr  "credit" "debit" "Paypal"
##  $ col3: logi  TRUE FALSE TRUE
##  $ col4: num  25.5 44.2 54.9
```

We can add columns (variables) and rows (items) on to a data frame using `cbind()` and `rbind()` functions. Here are some examples:

```
# create a new vector

v4 <- c("VIC", "NSW", "TAS")

# add a column (variable) to df1

df2 <- cbind(df1, v4)
```

Adding attributes to data frames is very similar to what we have done in matrices. We can use `rownames()` and `colnames()` functions to add the row and column names, respectively.

```
# add row names

rownames(df2) <- c("subj1", "subj2", "subj3")

# add column names
colnames(df2) <- c("number", "card_type", "fraud", "transaction", "state")

# check the structure and the attributes

str(df2)
```

```
## 'data.frame':    3 obs. of  5 variables:
##  $ number     : int  1 2 3
##  $ card_type  : chr  "credit" "debit" "Paypal"
##  $ fraud      : logi  TRUE FALSE TRUE
##  $ transaction: num  25.5 44.2 54.9
##  $ state      : Factor w/ 3 levels "NSW","TAS","VIC": 3 1 2
```

```
attributes(df2)
```

```
## $names
## [1] "number"      "card_type"   "fraud"       "transaction" "state"
##
## $row.names
## [1] "subj1" "subj2" "subj3"
##
## $class
## [1] "data.frame"
```

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

```
df2
```

```
##       number card_type fraud transaction state
## subj1      1    credit  TRUE        25.5   VIC
## subj2      2     debit FALSE        44.2   NSW
## subj3      3    Paypal  TRUE        54.9   TAS
```

```
# subset by row numbers, take second and third rows only

df2[2:3, ]
```

```
##       number card_type fraud transaction state
## subj2      2     debit FALSE        44.2   NSW
## subj3      3    Paypal  TRUE        54.9   TAS
```

```
# same as above but uses row names

df2[c("subj2", "subj3"),  ]
```

```
##       number card_type fraud transaction state
## subj2      2     debit FALSE        44.2   NSW
## subj3      3    Paypal  TRUE        54.9   TAS
```

```
# subset by column numbers, take first and forth columns only

df2[, c(1,4)]
```

```
##       number transaction
## subj1      1        25.5
## subj2      2        44.2
## subj3      3        54.9
```

```
# same as above but uses column names

df2[, c("number", "transaction")]
```

```
##       number transaction
## subj1      1        25.5
## subj2      2        44.2
## subj3      3        54.9
```

```
# subset by row and column numbers

df2[2:3, c(1, 4)]
```

```
##       number transaction
## subj2      2        44.2
## subj3      3        54.9
```

```
# same as above but uses row and column names

df2[c("subj2", "subj3"), c("number", "transaction")]
```

```
##       number transaction
## subj2      2        44.2
## subj3      3        54.9
```

```
# subset using $: take the column (variable) fraud

df2$fraud
```

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

```
# take the second element in the fraud column

df2$fraud[2]
```

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

Converting Data Types/Structures

Data type and structure conversions can be done easily using `as.` functions. Essentially, `as.` functions will convert the object to a given type (whenever possible) and `is.` functions will test for the given data type and return a logical value (`TRUE` or `FALSE`).

`as.` Functions	Changes type to	`is.` Functions	Checks if the type is
`as.numeric()`	numeric	`is.numeric()`	numeric
`as.integer()`	integer	`is.integer()`	integer
`as.double()`	double	`is.double()`	double
`as.character()`	character	`is.character()`	character
`as.factor()`	factor	`is.factor()`	factor
`as.logical()`	logical	`is.logical()`	logical
`as.vector()`	vector	`is.vector()`	vector
`as.list()`	list	`is.list()`	list

`as.` **Functions**	**Changes type to**	`is.` **Functions**	**Checks if the type is**
`as.matrix()`	matrix	`is.matrix()`	matrix
`as.data.frame()`	data frame	`is.data.frame()`	data frame

Here are some examples on data type conversions:

```
# create a numeric vector called num_vec

num_vec <- as.vector(8:17)

# check if it's a vector

is.vector(num_vec)
```

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

```
# convert num_vec into a character

char_vec <-as.character(num_vec)

# check if it's a character

is.character(char_vec)
```

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

```
# create a logical vector

log_vec <- c(FALSE, FALSE, TRUE)

# convert log_vec into a numeric vector

num_vec2 <- as.numeric(log_vec)

# check if it's a numeric vector

is.numeric(num_vec2)
```

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

The `as.` functions are also useful to initialise data types. The following example illustrates how you can initialise data using vectors and turn multiple vectors into a data frame:

```
# create different types of vectors

col1 <- 1:3
col2 <- c ("credit", "debit", "Paypal")
col3 <- c (TRUE, FALSE, TRUE)
col4 <- c (25.5, 44.2, 54.9)

# use cbind to combine vectors by columns

colvec <- cbind(col1, col2, col3, col4)

# check its class

class(colvec)
```

```
## [1] "matrix"
```

```
# convert matrix to a data frame

df <- as.data.frame(colvec, stringsAsFactors = FALSE)

df
```

```
##   col1   col2  col3 col4
## 1    1 credit  TRUE 25.5
## 2    2  debit FALSE 44.2
## 3    3 Paypal  TRUE 54.9
```

Long vs. wide format data

A single data set can be rearranged in many different ways. One of the ways is called "**long format (a.k.a long layout)**". In this layout, the data set is arranged in such a way that a single subject's information is stored in multiple rows.

In the **wide format (a.k.a wide layout)**, a single subject's information is stored in multiple columns. The main difference between a wide layout and a long layout is that the wide layout contains all the measured information in different columns.

An illustration of the same data set stored in wide vs. long format is given below:

Wide format

Country	2011	2012	2013
FR	7000	6900	7000
DE	5800	6000	6200
US	15000	14000	13000

=

Long format

Country	Year	n
FR	2011	7000
DE	2011	5800
US	2011	15000
FR	2012	6900
DE	2012	6000
US	2012	14000
FR	2013	7000
DE	2013	6200
US	2013	13000

Fig1. The same data set presented in wide vs. long format

In Module 4, we will see how we can convert a long format to a wide one and vice versa using R.

Additional Resources and Further Reading

Data Wrangling with R by Boehmke (2016) is a comprehensive source for all data types and structures in R. This book is also one the recommended texts in our course. It is available through RMIT Library (http://www1.rmit.edu.au/library).

Base R cheatsheet on http://github.com/rstudio/cheatsheets/raw/master/base-r.pdf (http://github.com/rstudio/cheatsheets/raw/master/base-r.pdf) is useful for remembering commonly used functions and arguments for data types and structures in R.

References

Boehmke, Bradley C. 2016. *Data Wrangling with R*. Springer.

Wickham, Hadley. 2014. *Advanced R*. CRC Press.