STA 326 2.0 R Programming and Data Analysis

Thiyanga S Talagala

2020-02-07

Contents

Learning outcomes functions:

In this tutorial we learned what functions in R programming are, the basic syntax of functions in R programming, in-built functions and how to use them to make our work easier, the syntax of a user-defined function, and different types of user-defined functions. In the next session, we are going to learn how to read files in R programming.

Chapter 1

Introduction

1.1 R programming language

1.2 RStudio

RStudio is an integrated development environment (IDE) for R that provides an alternative interface to R that has several advantages over other default interfaces.

1.3 Installation

The first thing you need to do to get started with R is to install it on your computer. R works on pretty much every platform available, including the widely available Windows, Mac OS X, and Linux systems. If you want to watch a step-by-step tutorial on how to install R for Mac or Windows, you can watch these videos:

- Installation R on Windows
- Installing R on the Mac

Next you can install Rstudio. Remember, you must have R already installed before installing Rstudio. If you want to watch a step-by-step watch the vedio here.

1.4 Working with R scripts files

Rather than typing R commands into the Console. This allows for **reproducibility**, share scripts with someone else.

To create a new R script

File –> New File –> R Script

Commenting on R scripts

1.5 R packages

1.5.1 Installation

There is a large community of R users who contribute various packages that do useful things. Before you start using an R package, you must first install it into your environment. There are two ways to install a package

1.
2.

1.5.2 Load a package

one time, then load package

1.6 Important things to know about R

1. R is case-sensitive
2. R works with numerous data types. Some of the most basic types to get started are:

    i. **numeric**: decimal values like 8.5
    ii. **integers**: natural numbers like 8
    iii. **logical**: Boolean values (TRUE or FALSE)
    iv. **character**: strigs(text) like "statistics"

1.7 Objects

The entities R operates on are technically known as **objects**. There are two types of objects:

1. Data structures

2. Functions

1.8 Getting help

1.9 Variable assignment

1.10

1.11 Data permanency and removing objects

Chapter 2

Data structures in base R

There are five data types in R

1. Atomic vector
2. Matrix
3. Array
4. List
5. Data frame

2.1 Atomic vectors

- This is a 1-dimensional
- All elements of an atomic vector must be the same type, Hence it is a **homogeneous** type of object. Vectirs can hold numeric data, charactor data or logical data.

2.1.1 Creating vectors

Vectors can be created by using the function concatenation `c`

Syntax

```
vector_name <- c(element1, element2, element3)
```

Examples

```
first_vec <- c(10, 20, 50, 70)
second_vec <- c("Jan", "Feb", "March", "April")
third_vec <- c(TRUE, FALSE, TRUE, TRUE)
fourth_vec <- c(10L, 20L, 50L, 70L)
```

2.1.2 Types and tests with vectors

1. `typepf()` returns types of their elements

```
typeof(first_vec)
```

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

```
typeof(fourth_vec)
```

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

2. To check if it is a

- vector: `is.vector()`

```
is.vector(first_vec)
```

```
[1] TRUE
```

- charactor vector: `is.charactor()`

```
is.character(first_vec)
```

```
[1] FALSE
```

- double: `is.double()`

```
is.double(first_vec)
```

```
[1] TRUE
```

- integer: `is.integer()`

```
is.integer(first_vec)
```

```
[1] FALSE
```

- logical: `is.logical()`

```
is.logical(first_vec)
```

```
[1] FALSE
```

- atomic: `is.atomic()`

```
is.atomic(first_vec)
```

```
[1] TRUE
```

3. `length()` returns number of elements in a vector

```
length(first_vec)
```

```
[1] 4
```

```
length(fourth_vec)
```

```
[1] 4
```

2.1.3 Coercion

Vectors must be homogeneous. When you attempt to combine different types they will be coerced to the most flexible type so that every element in the vector is of the same type.

Order from least to most flexible

`logical` –> `integer` –> `double` –> `charactor`

```
a <- c(3.1, 2L, 3, 4, "GPA")
typeof(a)
```

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

```
anew <- c(3.1, 2L, 3, 4)
typeof(anew)
```

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

2.1.4 Explicit coercion

Vectors can be explicitly coerced from one class to another using the `as.*` functions, if available. For example, `as.charactor`, `as.numeric`, `as.integer`, and `as.logical`.

```
vec1 <- c(TRUE, FALSE, TRUE, TRUE)
typeof(vec1)
```

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

```
vec2 <- as.integer(vec1)
typeof(vec2)
```

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

```
vec2
```

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

Question

Why the below output produce NAs?

```
x <- c("a", "b", "c")
as.numeric(x)
```

```
Warning: NAs introduced by coercion
```

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

2.1.5 Simplifying vector creation

1. colon `:` produce regular spaced ascending or descending sequences.

```r
a1 <- 10:16
a1
```

```
[1] 10 11 12 13 14 15 16
```

```r
b1 <- -0.5:8.5
b1
```

```
 [1] -0.5  0.5  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5
```

2. sequence `seq()`. There are three arguments we need to provide, i) initial value, ii) final value, and iii) increment

syntax

```r
seq(initial_value, final_value, increment)
```

example

3. repeats `rep()`

```r
rep(9, 5)
```

```
[1] 9 9 9 9 9
```

```r
rep(1:4, 2)
```

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

```r
rep(1:4, each=2) # each element is repeated twice
```

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

```r
rep(1:4, times=2) # whole sequence is repeated twice
```

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

```
rep(1:4, each=2, times=3)
```

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

```
rep(1:4, 1:4)
```

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

```
rep(1:4, c(4, 1, 4, 2))
```

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

2.1.6 Logical operations

2.1.7 Subsetting

There are situations where we want to select only some of the elements of a vector. Following codes show various ways to select part of a vector object.

```
data <- c(10, 20, 103, 124, 126)

data[1] # shows the first element
```

```
[1] 10
```

```
data[-1] # shows all except the first item
```

```
[1]  20 103 124 126
```

```
data[1:3] # shows first three elements
```

```
[1]  10  20 103
```

```
data[c(1, 3, 4)]
```

```
[1]  10 103 124
```

```r
data[data > 3]
```

```
[1]  10  20 103 124 126
```

```r
data[data<20|data>120]
```

```
[1]  10 124 126
```

Example: How do you replace the 3rd element in the data vector by 203?

```r
data[3] <- 203
data
```

```
[1]  10  20 203 124 126
```

2.1.8 Vector arithmetic

Vector operations are perfored element by element.

```r
c(10, 100, 100) + 2 # two is added to every element in the vector
```

```
[1]  12 102 102
```

Vector operations between two vectors

```r
v1 <- c(1, 2, 3)
v2 <- c(10, 100, 1000)
v1 + v2
```

```
[1]   11  102 1003
```

Add two vectors of unequal length

```r
longvec <- seq(10, 100, length=10)
shortvec <- c(1, 2, 3, 4, 5)

shortvec+longvec
```

```
 [1]  11  22  33  44  55  61  72  83  94 105
```

2.1.9 Missing values

Use `NA` to place a missing value in a vector.

```
z <- c(10, 101, 2, 3, NA)
is.na(z)
```

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

2.1.10 Factor

A factor is a vector that can contain only predefined values, and is used to store categorical data.

2.2 Matrix

Matrix is a 2-dimentional and a homogeneous data structure

Syntax to create a matrix

```
matrix_name <- matrix(vector_of_elements,
                      nrow=number_of_rows,
                      ncol=number_of_columns,
                      byrow=logical_value, # If byrow=TRUE, then the matrix is filled
                      dimnames=list(rnames, cnames)) # To assign row names and columns
```

Example

```
values <- c(10, 20, 30, 40)
matrix1 <- matrix(values, nrow=2) # Matrix filled by columns (default option)
matrix1
```

```
     [,1] [,2]
[1,]   10   30
[2,]   20   40
```

```
matrix2 <- matrix(values, nrow=2, byrow=TRUE) # Matrix filled by rows
matrix2
```

```
     [,1] [,2]
[1,]   10   20
[2,]   30   40
```

Naming matrix rows and columns

```
rnames <- c("R1", "R2")
cnames <- c("C1", "C2")
matrix_with_names <- matrix(values, nrow=2, dimnames=list(rnames, cnames))
matrix_with_names
```

```
   C1 C2
R1 10 30
R2 20 40
```

2.2.1 Matrix subscript

`matraix_name[i, ]` gives the ith row of a matrix

```
matrix1[1, ]
```

```
[1] 10 30
```

`matraix_name[, j]` gives the jth column of a matrix

```
matrix1[, 2]
```

```
[1] 30 40
```

`matraix_name[i, j]` gives the ith row and jth column element

```
matrix1[1, 2]
```

```
[1] 30
```

```
matrix1[1, c(1, 2)]
```

```
[1] 10 30
```

2.2.2 `cbind` and `rbind`

Matrices can be created by column-binding and row-binding with `cbind()` and `rbind()`

```
x <- 1:3
y <- c(10, 100, 1000)

cbind(x, y) # binds matrices horizontally
```

```
     x    y
[1,] 1   10
[2,] 2  100
[3,] 3 1000
```

```
rbind(x, y) #binds matrices vertically
```

```
  [,1] [,2] [,3]
x    1    2    3
y   10  100 1000
```

2.2.3 Matrix operations

2.3 Array

- 3 dimentional data structure
-

2.4 List

2.5 Data frame

- A dataframe is a rectangular arrangement of data with rows corresponding to observational units and columns corresponding to variables.
- A data frame is more general than a matrix in that different columns can contain different modes of data.
- It's similar to the datasets you'd typically see in SPSS and MINITAB.
- Data frames are the most common data structure you'll deal with in R.

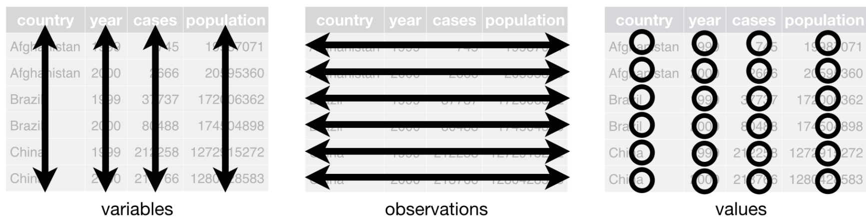


Figure 2.1: Figure 1: Components of a dataframe.

2.5.1 Creating a dataframe

Syntax

```
name_of_the_dataframe <- data.frame(
                          var1_name=vector of values of the first variable,
                          var2_names=vector of values of the second variable)
```

Example

```
corona <- data.frame(ID=c("C001", "C002", "C003", "C004"),
                     Location=c("Beijing", "Wuhan", "Shanghai", "Beijing"),
                     Test_Results=c(FALSE, TRUE, FALSE, FALSE))
corona
```

```
    ID Location Test_Results
1 C001  Beijing        FALSE
2 C002    Wuhan         TRUE
3 C003 Shanghai        FALSE
4 C004  Beijing        FALSE
```

To check if it is a datafrme

```
is.data.frame(corona)
```

```
[1] TRUE
```

To convert a matrix to a dataframe

```
mat <- matrix(10:81, ncol=4)
mat
```

```
      [,1] [,2] [,3] [,4]
 [1,]   10   28   46   64
 [2,]   11   29   47   65
 [3,]   12   30   48   66
 [4,]   13   31   49   67
 [5,]   14   32   50   68
 [6,]   15   33   51   69
 [7,]   16   34   52   70
 [8,]   17   35   53   71
 [9,]   18   36   54   72
[10,]   19   37   55   73
[11,]   20   38   56   74
[12,]   21   39   57   75
[13,]   22   40   58   76
[14,]   23   41   59   77
[15,]   24   42   60   78
[16,]   25   43   61   79
[17,]   26   44   62   80
[18,]   27   45   63   81
```

```
mat_df <- as.data.frame(mat)
mat_df
```

```
   V1 V2 V3 V4
1  10 28 46 64
2  11 29 47 65
3  12 30 48 66
4  13 31 49 67
5  14 32 50 68
6  15 33 51 69
7  16 34 52 70
8  17 35 53 71
9  18 36 54 72
10 19 37 55 73
11 20 38 56 74
12 21 39 57 75
13 22 40 58 76
14 23 41 59 77
15 24 42 60 78
16 25 43 61 79
17 26 44 62 80
18 27 45 63 81
```

2.5.2 Subsetting data frames

Select rows

```
head(mat_df) # default it shows 5 rows
```

```
  V1 V2 V3 V4
1 10 28 46 64
2 11 29 47 65
3 12 30 48 66
4 13 31 49 67
5 14 32 50 68
6 15 33 51 69
```

```
head(mat_df, 3) # To extract only the first three rows
```

```
  V1 V2 V3 V4
1 10 28 46 64
2 11 29 47 65
3 12 30 48 66
```

```
tail(mat_df)
```

```
   V1 V2 V3 V4
13 22 40 58 76
14 23 41 59 77
15 24 42 60 78
16 25 43 61 79
17 26 44 62 80
18 27 45 63 81
```

To select some specific rows

```
index <- c(1, 3, 7, 8)
mat_df[index, ]
```

```
  V1 V2 V3 V4
1 10 28 46 64
3 12 30 48 66
7 16 34 52 70
8 17 35 53 71
```

Select columns

1. Select column(s) by variable names

```
mat_df$V1 # Method 1
```

```
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

```
mat_df[, "V1"] # Method 2
```

```
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

2. Select column(s) by index

```
mat_df[, 2]
```

```
 [1] 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
```

2.5.3 Built in dataframes

Note: All objects in R have a class.

Chapter 3

Functions in R

A function is a block of organized and reusable code that is used to perform a specific task in a program. There are two types of functions in R:

1. In-built functions
2. User-defined functions

3.1 In-built functions

These functions in R programming are provided by R environment for direct execution, to make our work easier Some examples for the frequently used in-built functions are as follows.

```
mean(c(10, 20, 21, 78, 105))
```

```
[1] 46.8
```

3.2 User-defined functions

These functions in R programming language are dclared and defined by a user according to the requirements, to perform a specific task.

3.3 Main components of a function

All R functions have three main components: (Check this with Hadley's book)

1. **function name**: name of the function that is stored as an R object
2. **arguments:** are used to rovide specific inputs to a function while a function is invoked. A function can have zero, single, multiple or default arguments.
3. **function body:** contains the block of code that performs the specific task assigned to a function. **return value**

3.4 Passing arguments to a function

3.5 Some useful built-in functions in R

3.5.1 R can be used as a simple calculator.

Operator	Description
+	addition
-	substraction
*	multiplication
^	exponentiation (5^2 is 25)
%%	modulo-remainder of the division of the number to the left by the number on its right. (5%%3 is 2)

3.5.2 Some more maths functions

Operator	Description
abs(x)	absolute value of x
log(x, base=y)	logarithm of x with base y; if base is not specified, returns the natural logarithm
exp(x)	exponential of x
sqrt(x)	square root of x
factorial(x)	factorial of x

3.5.3 Basic statistic functions

Operator	Description
mean(x)	mean of x
median(x)	median of x
mode(x)	mode of x
var(x)	variance of x
scale(x)	z-score of x
quantile(x)	quantiles of x
summary(x)	summary of x: mean, minimum, maximum, etc.

3.5.4 Probability distribution functions

- **d** prefix for the **distribution** function
- **p** prefix for the **cummulative probability**
- **q** prefix for the **quantile**
- **r** prefix for the **random** number generator

3.5.4.1 Illustration with Standard normal distribution

The general formula for the probability density function of the normal distribution with mean μ and variance σ is given by

$$f_X(x) = \frac{1}{\sigma\sqrt{(2\pi)}}e^{-(x-\mu)^2/2\sigma^2}$$

If we let the mean $\mu = 0$ and the standard deviation $\sigma = 1$, we get the probability density function for the standard normal distribution.

$$f_X(x) = \frac{1}{\sqrt{(2\pi)}}e^{-(x)^2/2}$$

```
dnorm(0)
```

```
[1] 0.3989423
```

```
pnorm(0)
```

```
[1] 0.5
```

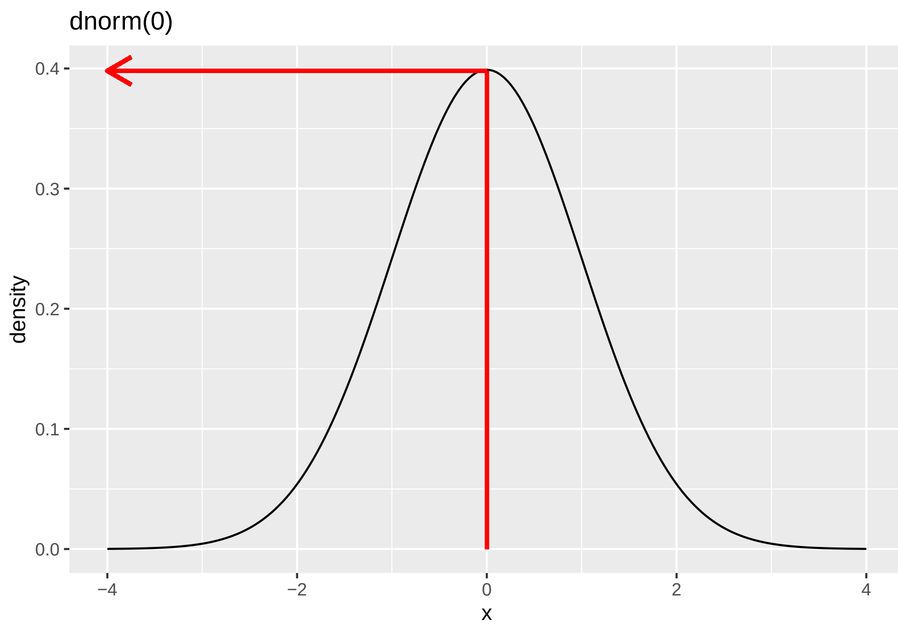


Figure 3.1: Standard normal probability density function: dnorm(0)

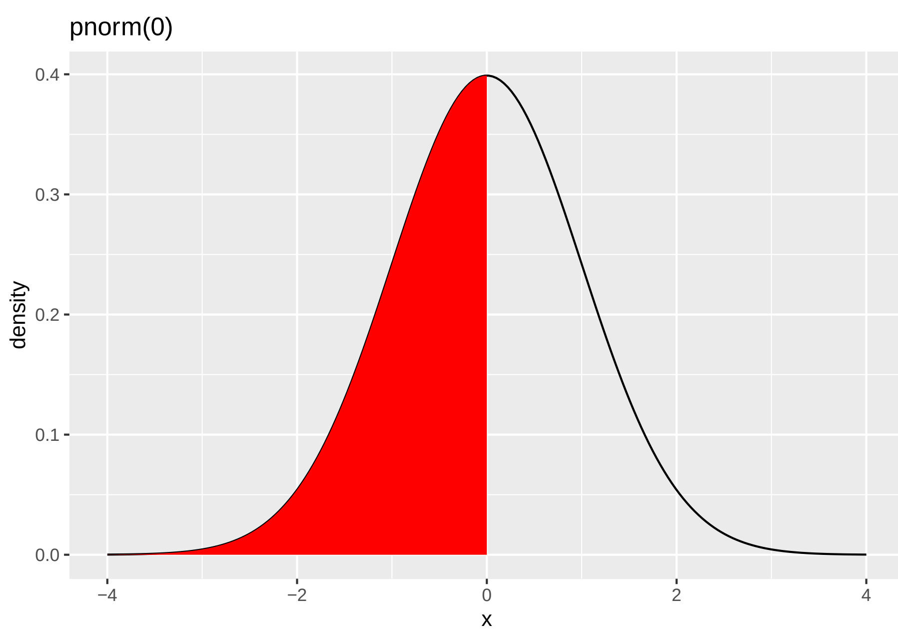


Figure 3.2: Standard normal probability density function: dnorm(0)

```
qnorm(0.5)
```

```
[1] 0
```

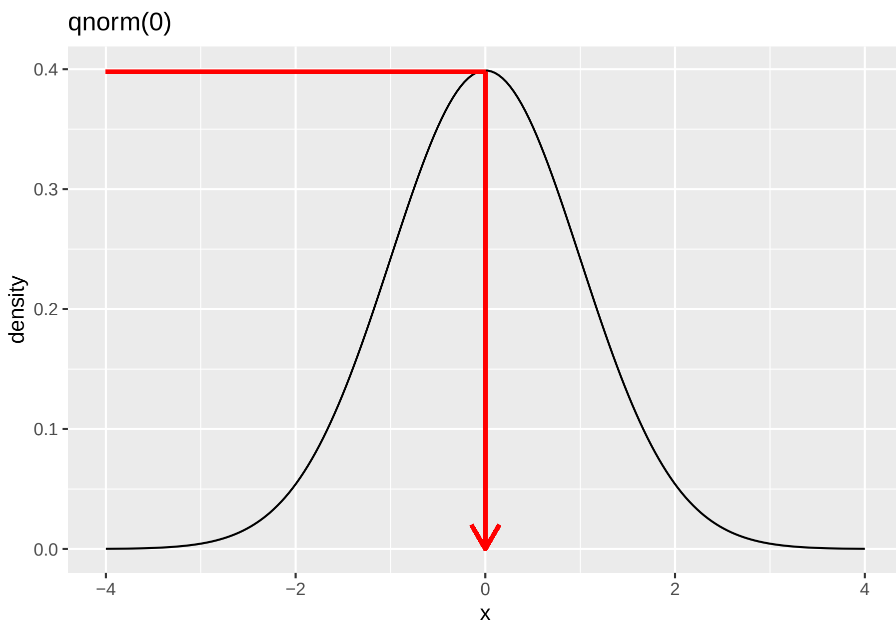


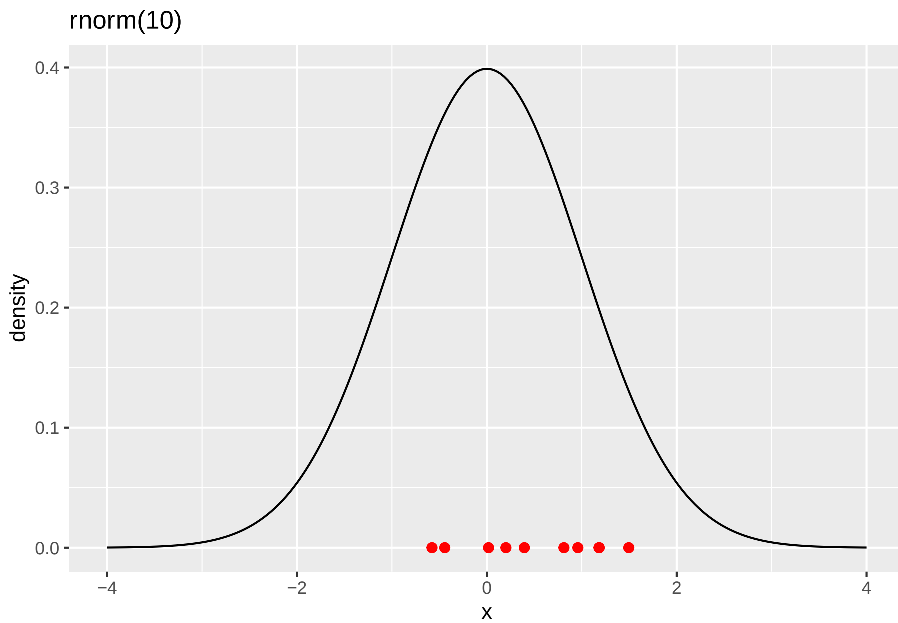
Figure 3.3: Standard normal probability density function: dnorm(0)

```
set.seed(262020)
random_numbers <- rnorm(10)
random_numbers
```

```
 [1]  0.20078181  0.95873346  1.18369056  1.49513750  1.18109222 -0.57789570
 [7]  0.01790671  0.81185245  0.39488199 -0.44337927
```

```
sort(random_numbers) ## sort the numbers then it is easy to map with the graph
```

```
 [1] -0.57789570 -0.44337927  0.01790671  0.20078181  0.39488199  0.81185245
 [7]  0.95873346  1.18109222  1.18369056  1.49513750
```



3.5.5 Reproducibility of scientific results

```r
rnorm(10) # first attempt
```

```
 [1]  1.4701904 -0.2375662  0.1765985 -0.5257483 -1.3674764 -1.4422500
 [7]  0.7576607  0.6475122 -1.1543034  0.9066248
```

```r
rnorm(10) # second attempt
```

```
 [1] -1.7603264 -0.3402939 -1.0335807  1.0645014 -0.3874459  0.5975271
 [7] -2.1535707  0.6602928  1.1581404  0.6133446
```

As you can see above you will get different results

```r
set.seed(1)
rnorm(10) # First attempt with set.seed
```

```
 [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
 [7]  0.4874291  0.7383247  0.5757814 -0.3053884
```

```r
set.seed(1)
rnorm(10) # Second attempt with set.seed
```

```
 [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
 [7]  0.4874291  0.7383247  0.5757814 -0.3053884
```

Chapter 4

Writing functions

4.1 When should we write functions?

- do many repetitive task

4.2 Glogal variables vs local variables

4.3 Control structures

- for loops

4.4 lapply, apply..

Chapter 5

Data analysis with tidyverse

Some *significant* applications are demonstrated in this chapter.

5.1 Tidy data

Two key principles:

1. Put each dataset in a dataframe
2. Put each variable in a column

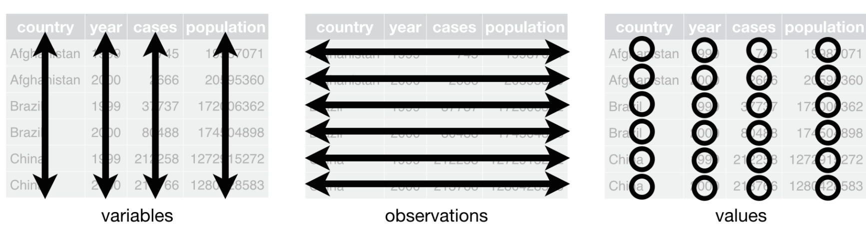


Figure 5.1: Figure 1: Components of a dataframe.

Vedio: https://www.youtube.com/watch?v=K-ss_ag2k9E

5.1.1 Convert from messy data to tidy data

"Tidy dataset are all alike; every messy dataset is messy in its own way." _ Hadley Wickham

Chapter 6

Data wrangliing

Chapter 7

Data visualisation

Chapter 8

Modelling

8.1 Simulation-based Inference