R introduction and dplyr

Gregor Pirs, Jure Demsar and Erik Strumbelj

25/7/2019

R and Rstudio

R (https://www.r-project.org/) is a free open-source software for statistical computing. The basic interface to R is via console, which is quite rigid. RStudio (https://www.rstudio.com) provides us with a better user interface and additional functionalities (R notebooks, RMarkdown,...).

Usually the page in RStudio is separated into four parts. Upper left part is used for scripts. These are R files (or similar) which include our code and represent the main building blocks of our programs. Lower left part is the console, equivalent to the basic console interface of R. Upper right part is dedicated to the environment and history. Lower right part shows our workspace, plots, packages, and help.

To create a new script, go to File -> New File -> R Script. To run the code, highlight the desired part of the code and press Ctrl + Enter. Alternatively, you can run the code by clicking the run icon in the top-right corner of the script.

To specify the working directory, use `setwd()` function, where you provide the working directory in parentheses. For example to set the working directory to C:/Author you would call

```
setwd("C:/Author")
```

Note that R recognizes either a slash or double backslash in the path, but it does not recognize a single backslash, which is the default if you copy the path on a Slovenian computer.

```
setwd("C:/Author") # OK
setwd("C:\\Author") # OK
setwd("C:\Author") # Not OK.
```

Variables

Variables are the main data type of every program. In R, we define the values of variables with the syntax `<-`. We do not need to initialize the type of the variables, as R predicts it. We denote strings with `""`. Comments are written with `#`.

Let's create some variables.

```
n            <- 20
x            <- 2.7
m            <- n # m gets value 20
my_flag      <- TRUE
student_name <- "Luke"
student_name <- Luke # because there is no variable Luka, it returns an error
```

```
## Error in eval(expr, envir, enclos): object 'Luke' not found
```

By using the function `typeof()` we can check the type of a variable.

```
typeof(n)
```

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

```
typeof(student_name)
```

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

```
typeof(my_flag)
```

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

We can change the types of variables with as.type functions. The main types are **integer**, **double**, **character** (strings), and **logical**. Note that the type character is used for strings and we do not have a separate type for single characters.

```
typeof(as.integer(n))
```

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

```
typeof(as.character(n))
```

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

Another common type is date. We can convert a character string to a date with the `as.Date()` function. When using this function, we have to be careful to provide the correct format of the date.

```
some_date <- as.Date("2019-01-01", format = "%Y-%m-%d")
some_date
```

```
## [1] "2019-01-01"
```

To access the values of the variables, we use variable names.

```
n
```

```
## [1] 20
```

```
m
```

```
## [1] 20
```

```
my_flag
```

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

```
student_name
```

```
## [1] "Luke"
```

We can apply arithmetic operations on numerical variables.

```
n + x
```

```
## [1] 22.7
```

```
n - x
```

```
## [1] 17.3
```

```
diff <- n - x # variable diff gets the difference between n and x
diff
```

```
## [1] 17.3
```

```
n * x
```

```
## [1] 54
```

```
n / x
```

```
## [1] 7.407407
```

```
x^2
```

```
## [1] 7.29
```

```
sqrt(x)
```

```
## [1] 1.643168
```

```
n > 2 * n # logical is greater
```

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

```
n == n # equals
```

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

```
n == 2 * n
```

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

```
n != n # not equals
```

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

We can concatenate strings with functions `paste()` and `paste0()`. The difference between these functions is that the first one forces a space between inputs, while the second one does not.

```
paste(student_name, "is", n, "years old")
```

```
## [1] "Luke is 20 years old"
```

```
paste0(student_name, "is", n, "years old")
```

```
## [1] "Lukeis20years old"
```

```
L_username <- paste0(student_name, n)
```

Function `paste()` can get an additional parameter `sep`, which should be used between the inputs. If we want to find out more about a function, we put a question mark before the function's name in the console.

```
# ?paste
paste(student_name, "is", n, "years_old", sep = "_")
```

```
## [1] "Luke_is_20_years_old"
```

Basic data structures

Vector

Vectors are the most common data structure in R. They consist of several elements of the same type. We create them with the function `c()` (combine).

```
student_ages  <- c(20, 23, 21)
student_names <- c("Luke", "Jen", "Mike")
passed        <- c(TRUE, TRUE, FALSE)
```

To access individual elements of vectors we use square brackets with the sequential number of the elements we want. **The indexing in R starts with 1**, as opposed to 0 (C++, Java,...).

```
student_ages[2]
```

```
## [1] 23
```

```
student_names[2]
```

```
## [1] "Jen"
```

```
passed[2]
```

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

To get the length of the vector use `length()`.

```
length(student_names)
```

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

We can use element-wise arithmetic operations on vectors, and we can use the scalar product (`%*%`). Note that you have to be careful with vector lengths. For example, if we have an operation on two elements—in our case vectors—and they are not of the same length, the smaller one will start preiodically repeating itself, until it reaches the size of the larger one. In that case, R will provide us with a warning.

```
a <- c(1, 3, 5)
b <- c(2, 2, 1)
d <- c(6, 7)
a + b
```

```
## [1] 3 5 6
```

```
a * b
```

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

```
a + d # not the same length, d becomes (6, 7, 6)
```

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

```
## [1]  7 10 11
```

```
a + 2 * b
```

```
## [1] 5 7 7
```

```
a %*% b # scalar product
```

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

```
a > b # logical relations between elements
```

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

```
b == a
```

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

We often want to select only specific elements of a vector. There are several ways to do that—for example all of the calls below return the first two elements of vector `a`.

```
a[c(TRUE, TRUE, FALSE)] # selection based on logical vector
```

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

```
a[c(1,2)] # selection based on indexes
```

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

```r
a[a < 5] # selection based on logical condition
```

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

We can also use several conditions. If we want both conditions to hold, we use and (`&`), if only one has to hold we use if (`|`). Note that only here we use only a single symbol for each, as opposed to some other programming languages that use two.

```r
a[a > 2 & a < 4]
```

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

```r
a[a < 2 | a > 4]
```

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

Factor

Factors can be considered more as a variable type, than a data structure. But since they are based on vectors, we present them at this point.

Factors are used for coding categorical variables, which can only take a finite number of predetermined values. We can further divide categorical variables into nominal and ordinal. Nominal values don't have an ordering (for example car brand), while ordinal variables do (for example frequency—never, rarely, sometimes, often, always). Ordinal variables have an ordering but usually we can not assign values to them (for example sometimes is more than rarely, but we do not know how much more).

In R we create factors with function `factor()`. When creating factors, we can determine in advance, which values the factor can take with the argument `levels`. If we wish to add a non-existing level to a factor variable, R turns it into NA.

```r
car_brand <- factor(c("Audi", "BMW", "Mercedes", "BMW"), ordered = FALSE)
car_brand
```

```
## [1] Audi     BMW      Mercedes BMW
## Levels: Audi BMW Mercedes
```

```r
freq      <- factor(x       = NA,
                    levels  = c("never","rarely","sometimes","often","always"),
                    ordered = TRUE)
freq[1:3] <- c("rarely", "sometimes", "rarely")
freq
```

```
## [1] rarely    sometimes rarely
## Levels: never < rarely < sometimes < often < always
```

```r
freq[4]   <- "quite_often" # non-existing level, returns NA
```

```
## Warning in `[<-.factor`(`*tmp*`, 4, value = "quite_often"): invalid factor
## level, NA generated
```

```r
freq
```

```
## [1] rarely    sometimes rarely    <NA>
## Levels: never < rarely < sometimes < often < always
```

Matrix

Two-dimensional generalizations of vectors are matrices. We create them with the function `matrix()`, where we have to provide the values and either the number of rows or columns. Additionally, the argument `byrow`

= TRUE fills the matrix with provided elements by rows (default is by columns).

```
my_matrix <- matrix(c(1, 2, 1,
                      5, 4, 2),
                    nrow  = 2,
                    byrow = TRUE)
my_matrix
```

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

```
my_square_matrix <- matrix(c(1, 3,
                             2, 3),
                           nrow  = 2)
my_square_matrix
```

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

To access individual elements we use square brackets, where we divide the dimensions by a comma.

```
my_matrix[1,2] # first row, second column
```

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

```
my_matrix[2, ] # second row
```

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

```
my_matrix[ ,3] # third column
```

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

Some useful functions for matrices.

```
nrow(my_matrix) # number of matrix rows
```

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

```
ncol(my_matrix) # number of matrix columns
```

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

```
dim(my_matrix) # matrix dimension
```

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

```
t(my_matrix) # transpose
```

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

```
diag(my_matrix) # the diagonal of the matrix as vector
```

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

```
diag(1, nrow = 3) # creates a diagonal matrix
```

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

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

```
det(my_square_matrix) # matrix determinant
```

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

We can also use arithmetic operations on matrices. Note that we have to be careful with matrix dimensions. For matrix multiplication, we use %*%

```
my_matrix + 2 * my_matrix
```

```
##      [,1] [,2] [,3]
## [1,]    3    6    3
## [2,]   15   12    6
```

```
my_matrix * my_matrix # element-wise multiplication
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    1
## [2,]   25   16    4
```

```
my_matrix %*% t(my_matrix) # matrix multiplication
```

```
##      [,1] [,2]
## [1,]    6   15
## [2,]   15   45
```

```
my_square_matrix %*% my_matrix
```

```
##      [,1] [,2] [,3]
## [1,]   11   10    5
## [2,]   18   18    9
```

```
my_matrix %*% my_square_matrix # wrong dimensions
```

```
## Error in my_matrix %*% my_square_matrix: non-conformable arguments
```

We can transform a matrix into a vector.

```
my_vec <- as.vector(my_matrix)
my_vec
```

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

Array

Multi-dimensional generalizations of matrices are arrays.

```
my_array <- array(c(1, 2, 3, 4, 5, 6, 7, 8), dim = c(2, 2, 2))
my_array[1, 1, 1]
```

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

```
my_array[2, 2, 1]
```

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

```
my_array[1, , ]
```

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

```
## [2,]    3    7
```

```
dim(my_array)
```

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

Data frame

Data frames are the basic data structure used in R for data analysis. It has the form of a table, where columns represent individual variables, and rows represent observations. They differ from matrices, as the columns can be of different types. We access elements the same way as in matrices.

We can combine vectors into data frames with `data.frame()`. The function transforms variables of type character into factors by default. if we do not want that, we have to add an argument `stringsAsFactors = FALSE`. We can assign column names with the function `colnames()`.

```
student_data           <- data.frame(student_names, student_ages, passed,
                                     stringsAsFactors = FALSE)
colnames(student_data) <- c("Name", "Age", "Pass")
student_data
```

```
##   Name Age  Pass
## 1 Luke  20  TRUE
## 2  Jen  23  TRUE
## 3 Mike  21 FALSE
```

We can also assign column names directly, when creating a data frame.

```
student_data <- data.frame("Name" = student_names,
                           "Age"  = student_ages,
                           "Pass" = passed)
student_data
```

```
##   Name Age  Pass
## 1 Luke  20  TRUE
## 2  Jen  23  TRUE
## 3 Mike  21 FALSE
```

Similar to vectors, we can access the elements in data frames (and matrices) with logical calls. Here we need to be careful if we are selecting rows or columns. To access specific columns, we can also use the name of the column preceded by `$`.

```
student_data[ ,colnames(student_data) %in% c("Name", "Pass")]
```

```
##   Name  Pass
## 1 Luke  TRUE
## 2  Jen  TRUE
## 3 Mike FALSE
```

```
student_data[student_data$Pass == TRUE, ]
```

```
##   Name Age Pass
## 1 Luke  20 TRUE
## 2  Jen  23 TRUE
```

```
student_data$Pass
```

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

List

Lists are very useful data structure, especially when we are dealing with different data sets and data structures. We can imagine a list as a vector, where each element can be a different data structure. For example, a list can have a vector stored on index 1, a matrix on index 2, and a data frame on index 3. Moreover, a list can be an element of a list and so on.

```
first_list  <- list(student_ages, my_matrix, student_data)
second_list <- list(student_ages, my_matrix, student_data, first_list)
```

We access the elements of a list with double square brackets.

```
first_list[[1]]
```

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

```
second_list[[4]]
```

```
## [[1]]
## [1] 20 23 21
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    1    2    1
## [2,]    5    4    2
##
## [[3]]
##   Name Age  Pass
## 1 Luke  20  TRUE
## 2  Jen  23  TRUE
## 3 Mike  21 FALSE
```

```
second_list[[4]][[1]] # first element of the fourth element of second_list
```

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

We can also apply `length()` to get the number of elements in the list.

```
length(second_list)
```

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

To append to list, we use the call below.

```
second_list[[length(second_list) + 1]] <- "add_me"
second_list[[length(second_list)]] # check, what is on the last index
```

```
## [1] "add_me"
```

Additionally, we can name the elements of the list, and access them by name. For that we use the `names()` function.

```
names(first_list) <- c("Age", "Matrix", "Data")
first_list$Age
```

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

Packages

R is an open-source programming language and anyone can contribute to its development. Many packages exist that make our work in R easier. Additionally, some packages include different statistical models—some

of which are implemented in other languages for efficiency (for example C++). An open-source repository CRAN consists of most packages that you are going to need. To install a specific package, we use the function `install.packages()`, or we can use R-Studio's UI. Once a package is installed, we can load it into our workspace with `library()`. We will get to know several useful packages during this workshop.

```
install.packages("stats") # install package
library(stats) # load the package into workspace
```

Data import

We often encounter data in a csv (comma separated value) format. Different pacakges in R allow us to read data from csv, txt, xlsx, etc. formats. Here we will go through reading data from csv and xlsx formats.

To read csv data use `read.csv` from the package `utils`. Before we read the data, we need to check two things. First, what is the character that separates the columns and how the decimal places are denoted (comma or dot). Second, if the data have a header (Does the first row contain column names?). Function automatically returns a data frame. `read.csv()` assumes that comma is the separator and a decimal point. However, it allows the change of these default values by providing the corresponding arguments. It also assumes that we have a header by default. When saving your data in the CSV format, we recommend using a semi-colon as the separator, as comma is often used a) in text, b) as the decimal separator, or c) as thousands separator.

In our **data** folder, we have medical insurance data set acquired from Kaggle (https://www.kaggle.com/easonlai/sample-insurance-claim-prediction-dataset/). To show different reading functions, we saved the data set in three different formats—csv with a comma separator, csv with a semi-colon separator, and xlsx file. The file also contains a header. Function `head()` returns the first six rows of the data frame.

```
library(utils)
claim_data <- read.csv("./data/insurance01.csv")
head(claim_data)
```

```
##   age    sex    bmi children smoker    region   charges
## 1  19 female 27.900        0    yes southwest 16884.924
## 2  18   male 33.770        1     no southeast  1725.552
## 3  28   male 33.000        3     no southeast  4449.462
## 4  33   male 22.705        0     no northwest 21984.471
## 5  32   male 28.880        0     no northwest  3866.855
## 6  31 female 25.740        0     no southeast  3756.622
```

The dot in the string represents current working directory. We see that R automatically converted string variables (sex, smoker, region) to factors. In our case this is sensible. However, sometimes we want strings to remain strings. In those cases, change the argument `stringsAsFactors` to false.

Along with a semi-colon as the separator, the second file has a decimal comma. Therefore

```
claim_data <- read.csv("./data/insurance02.csv", sep = ";", dec = ",")
```

Data is often saved as xlsx. To read data from xlsx, we use the `read.xlsx` function from the package **xlsx**. However, this function can be quite slow, so if you are dealing with large data frames, it might be better to save the excel file as a csv file and then read it as csv.

```
library(xlsx)
claim_data <- read.csv("./data/insurance03.xlsx")
```

If statement

We often want to execute code based on some condition. For that we use the `if-else` pair.

```r
x <- 5
if (x < 0) {
  print("x is smaller than 0")
} else if (x == 0) {
  print("x is 0")
} else {
  print("x is greater than 0")
}
```

```
## [1] "x is greater than 0"
```

Loops

The most useful loop in R is the for loop. In the for loop we have to define a new variable, which will represent the different iterations of the loop. Then we have to define the values over which that variable will iterate. Often, these are sequential numbers. For example, let us add first 10 natural numbers.

```r
my_sum <- 0
for (i in 1:10) { # 1:10 returns a vector of natural numbers between 1 and 10
  my_sum <- my_sum + i
}
my_sum
```

```
## [1] 55
```

The values in a for loop do not have to be sequential numbers.

```r
my_sum       <- 0
some_numbers <- c(2, 3.5, 6, 100)
for (i in some_numbers) {
  my_sum <- my_sum + i
}
my_sum
```

```
## [1] 111.5
```

For example, let us calculate the average charges per region on our data set.

```r
regions <- unique(claim_data$region) # returns unique values in region column
for (reg in regions) {
  tmp_data   <- claim_data[claim_data$region == reg, ]
  charges    <- tmp_data$charges
  print(paste0("Region: ", reg,
               ", average charges: ", mean(charges)))
}
```

```
## [1] "Region: southwest, average charges: 12346.9373772923"
## [1] "Region: southeast, average charges: 14735.4114376099"
## [1] "Region: northwest, average charges: 12417.5753739692"
## [1] "Region: northeast, average charges: 13406.3845163858"
```

Functions

Base R consists of several function intended for easier work with data, for example `length()`, `dim()`, `colnames()`,... We can extend the set of functions with packages. For example, package **stats** allows us to create statistical models with the use of a single function—for example the linear model `lm()`. Here we will

present some useful functions, more complex functions will follow in later chapters. Remember, if you want additional information about functions, we can call the name of the function in the console, where we add a question mark (for example `?length`).

```
1:10 # special function that returns a sequence of numbers
```

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

```
sum(1:10) # sum of first 10 natural numbers
```

```
## [1] 55
```

```
sum(c(3,5,6,3))
```

```
## [1] 17
```

```
rep(1, times = 5) # returns a vector of lenght 5, where all values are 1
```

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

```
rep(c(1,2), times = 5) # returns a vector of length 5 where 1 and 2 are periodically changing
```

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

```
seq(0, 2, by = 0.5) # vector from 0 to 2, by adding 0.5
```

```
## [1] 0.0 0.5 1.0 1.5 2.0
```

```
prod(1:10) # multiply first 10 numbers
```

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

```
round(5.24)
```

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

```
5^5 # square
```

```
## [1] 3125
```

```
sqrt(16) # square root
```

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

```
as.character(c(1,6,3)) # transforms a numerical vector to a character vector
```

```
## [1] "1" "6" "3"
```

We often want a summary of our data. We can get it with `summary()`. We can use it on vectors and on data frames. The returned values are dependent on the types of variables.

```
summary(student_ages)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   20.00   20.50   21.00   21.33   22.00   23.00
```

```
summary(student_names)
```

```
##    Length     Class      Mode
##         3 character character
```

```
summary(passed)
```

```
##    Mode   FALSE    TRUE
## logical       1       2
```

```r
summary(car_brand)
```

```
##     Audi      BMW Mercedes
##        1        2        1
```

```r
summary(freq)
```

```
##     never    rarely sometimes     often    always      NA's
##         0         2         1         0         0         1
```

```r
summary(student_data) # summary of the whole data frame
```

```
##    Name        Age           Pass
##  Jen :1   Min.   :20.00   Mode :logical
##  Luke:1   1st Qu.:20.50   FALSE:1
##  Mike:1   Median :21.00   TRUE :2
##           Mean   :21.33
##           3rd Qu.:22.00
##           Max.   :23.00
```

Writing functions

We can write our own functions with `function()`. In the brackets, we define the parameters the function gets, and in curly brackets we define what the function does. We use `return()` to return values.

```r
sum_first_n_elements <- function (n) {
  my_sum <- 0
  for (i in 1:n) {
    my_sum <- my_sum + i
  }
  return (my_sum)
}
sum_first_n_elements(10)
```

```
## [1] 55
```

If we want that the function returns several different data structures, we use a list. For example, let us look at a function which gets a matrix as input, and returns its transpose and determinant.

```r
get_transpose_and_det <- function (mat) {
  trans_mat <- t(mat)
  det_mat   <- det(mat)
  out       <- list("transposed"  = trans_mat,
                    "determinant" = det_mat)
  return (out)
}
mat_vals <- get_transpose_and_det(my_square_matrix)
mat_vals$transposed
```

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

```r
mat_vals$determinant
```

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

Other useful functions for data summarizing

There are several functions that are useful when working with data. We already mentioned the `summary()` function. Let's look at some other functions.

To generate random numbers we can use a variety of random number generators. Which we select depends on the data that we wish to generate. Usually, we want to be able to replicate our analysis exactly, therefore we recommend the use of a seed—this will generate the same random numbers everytime you call the function. There is a function for that in R called `set.seed()`.

```
set.seed(0)
norm_dat  <- rnorm(1000, 5, 6) # generate 1000 samples from the normal
                               # distribution with mean 5 and standard deviation 6
count_dat <- rpois(2000, 8) # generate 2000 samples from the Poisson
                            # distribution with mean 8
unif_dat  <- runif(1000, -2, 5) # generate 1000 samples from the uniform
                                # distribution form -2 to 5
```

In data science, we often work with statistics, so let's look at some functions which provide us with meaningful information about our data.

```
mean(norm_dat)
```

```
## [1] 4.905023
```

```
var(norm_dat) # variance
```

```
## [1] 35.85649
```

```
sd(norm_dat) # standard deviation
```

```
## [1] 5.988029
```

```
max(norm_dat)
```

```
## [1] 24.59849
```

```
min(norm_dat)
```

```
## [1] -14.41831
```

```
quantile(norm_dat) # calculates 5 quantiles of the data
```

```
##          0%         25%         50%         75%        100%
## -14.4183144   0.7492647   4.6467753   9.1258324  24.5984871
```

We often want to standardize the data, before doing analysis. We can do that manually, or we can use R's `scale()` function.

```
st_dat <- scale(norm_dat)
mean(st_dat)
```

```
## [1] -1.257609e-17
```

```
var(st_dat)
```

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

Debugging

For the debugging in R we will use the `browser()` function. It stops the execution of the code and you can access the variables in the environment at the moment that browser was called.

For browser commands see ?browser or type help when browser is active.

Data wrangling with dplyr

Dplyr is a package for easier data manipulation. It is a part of a collection of packages called **tidyverse**, which consist of several R packages intended for data science. Dplyr is especially useful for data frame manipulation.

The main format of working with data in tidyverse is a **tibble**. This data structure is very smilar to base R's data frame, however it is designed for easier work with other packages in tidyverse and also provides a different print output. Let's look at it on our insurance data set.

```
library(dplyr)
```

```
claim_data <- read.csv("./data/insurance01.csv")
head(claim_data)
```

```
##   age    sex    bmi children smoker    region   charges
## 1  19 female 27.900        0    yes southwest 16884.924
## 2  18   male 33.770        1     no southeast  1725.552
## 3  28   male 33.000        3     no southeast  4449.462
## 4  33   male 22.705        0     no northwest 21984.471
## 5  32   male 28.880        0     no northwest  3866.855
## 6  31 female 25.740        0     no southeast  3756.622
```

```
claim_data <- as_tibble(claim_data)
claim_data
```

```
## # A tibble: 1,338 x 7
##      age sex      bmi children smoker region    charges
##    <int> <fct>  <dbl>    <int> <fct>  <fct>       <dbl>
##  1    19 female  27.9        0 yes    southwest  16885.
##  2    18 male    33.8        1 no     southeast   1726.
##  3    28 male    33          3 no     southeast   4449.
##  4    33 male    22.7        0 no     northwest  21984.
##  5    32 male    28.9        0 no     northwest   3867.
##  6    31 female  25.7        0 no     southeast   3757.
##  7    46 female  33.4        1 no     southeast   8241.
##  8    37 female  27.7        3 no     northwest   7282.
##  9    37 male    29.8        2 no     northeast   6406.
## 10    60 female  25.8        0 no     northwest  28923.
## # ... with 1,328 more rows
```

A tibble only shows the first 10 rows of the data set for clarity. Additionally, it only prints as many columns as fit into a page, and lists other columns below. If we wish to see all of the tibble, we can use the function `View()`. Under the variable names, a tibble shows the type of the variables.

Now that we have our starting data set, we can begin manipulating it. This usually consists of selecting specific rows and columns, and adding statistics derived from variables in the data frame. Below we describe five functions which will enable us dynamic data set manipulation.

Filter

The function `filter()` allows us to select rows, based on values of the variables. As input it gets a tibble and the conditions and it outputs a new tibble that consists only of desired rows.

```
filter(claim_data, region == "southwest")
```

```
## # A tibble: 325 x 7
##      age sex      bmi children smoker region    charges
##    <int> <fct>  <dbl>    <int> <fct>  <fct>       <dbl>
##  1    19 female  27.9        0 yes    southwest  16885.
##  2    23 male    34.4        0 no     southwest   1827.
##  3    19 male    24.6        1 no     southwest   1837.
##  4    56 male    40.3        0 no     southwest  10602.
##  5    30 male    35.3        0 yes    southwest  36837.
##  6    30 female  32.4        1 no     southwest   4150.
##  7    31 male    36.3        2 yes    southwest  38711
##  8    22 male    35.6        0 yes    southwest  35586.
##  9    19 female  28.6        5 no     southwest   4688.
## 10    28 male    36.4        1 yes    southwest  51195.
## # ... with 315 more rows
```

```
filter(claim_data, region == "southwest", age >= 30)
```

```
## # A tibble: 226 x 7
##      age sex      bmi children smoker region    charges
##    <int> <fct>  <dbl>    <int> <fct>  <fct>       <dbl>
##  1    56 male    40.3        0 no     southwest  10602.
##  2    30 male    35.3        0 yes    southwest  36837.
##  3    30 female  32.4        1 no     southwest   4150.
##  4    31 male    36.3        2 yes    southwest  38711
##  5    60 male    39.9        0 yes    southwest  48173.
##  6    55 male    37.3        0 no     southwest  20630.
##  7    48 male    28          1 yes    southwest  23568.
##  8    61 female  39.1        2 no     southwest  14235.
##  9    53 female  28.1        3 no     southwest  11742.
## 10    44 male    27.4        2 no     southwest   7727.
## # ... with 216 more rows
```

The conditions in filter use and—all conditions have to be satisfied. If we want to use or, we have to divide them with a pipe |.

```
filter(claim_data, region == "southwest" | region == "northwest")
```

```
## # A tibble: 650 x 7
##      age sex      bmi children smoker region    charges
##    <int> <fct>  <dbl>    <int> <fct>  <fct>       <dbl>
##  1    19 female  27.9        0 yes    southwest  16885.
##  2    33 male    22.7        0 no     northwest  21984.
##  3    32 male    28.9        0 no     northwest   3867.
##  4    37 female  27.7        3 no     northwest   7282.
##  5    60 female  25.8        0 no     northwest  28923.
##  6    23 male    34.4        0 no     southwest   1827.
##  7    19 male    24.6        1 no     southwest   1837.
##  8    56 male    40.3        0 no     southwest  10602.
##  9    30 male    35.3        0 yes    southwest  36837.
## 10    30 female  32.4        1 no     southwest   4150.
```

```
## # ... with 640 more rows
```

Or, the same can be achieved by using the operator %in%.

```
filter(claim_data, region %in% c("southwest", "northwest"))
```

```
## # A tibble: 650 x 7
##      age sex      bmi children smoker region    charges
##    <int> <fct>  <dbl>    <int> <fct>  <fct>       <dbl>
##  1    19 female  27.9        0 yes    southwest  16885.
##  2    33 male    22.7        0 no     northwest  21984.
##  3    32 male    28.9        0 no     northwest   3867.
##  4    37 female  27.7        3 no     northwest   7282.
##  5    60 female  25.8        0 no     northwest  28923.
##  6    23 male    34.4        0 no     southwest   1827.
##  7    19 male    24.6        1 no     southwest   1837.
##  8    56 male    40.3        0 no     southwest  10602.
##  9    30 male    35.3        0 yes    southwest  36837.
## 10    30 female  32.4        1 no     southwest   4150.
## # ... with 640 more rows
```

For example, let's say we are interested in doing further analysis on people older than 18, who live in the south. We can construct a new tibble, where we filter out the unnecessary rows.

```
claim_df <- filter(claim_data, region %in% c("southwest", "southeast"),
                   age >= 30)
```

Arrange

To arrange data we use dplyr's function `arrange()`, which gets a tibble and the variables on which to arrange. If we want a descending arrangement, we have to use function `desc()`.

```
arrange(claim_df, age)
```

```
## # A tibble: 475 x 7
##      age sex      bmi children smoker region    charges
##    <int> <fct>  <dbl>    <int> <fct>  <fct>       <dbl>
##  1    30 male    35.3        0 yes    southwest  36837.
##  2    30 female  32.4        1 no     southwest   4150.
##  3    30 male    35.5        0 yes    southeast  36950.
##  4    30 female  30.9        3 no     southwest   5326.
##  5    30 female  33.3        1 no     southeast   4151.
##  6    30 female  27.7        0 no     southwest   3554.
##  7    30 female  28.4        1 yes    southeast  19522.
##  8    30 female  43.1        2 no     southeast   4754.
##  9    30 male    37.8        2 yes    southwest  39241.
## 10    30 male    31.4        1 no     southwest   3659.
## # ... with 465 more rows
```

```
arrange(claim_df, age, desc(charges))
```

```
## # A tibble: 475 x 7
##      age sex      bmi children smoker region    charges
##    <int> <fct>  <dbl>    <int> <fct>  <fct>       <dbl>
##  1    30 female  39.0        3 yes    southeast  40932.
##  2    30 male    37.8        2 yes    southwest  39241.
##  3    30 male    35.5        0 yes    southeast  36950.
```

```
##  4    30 male    35.3        0 yes    southwest  36837.
##  5    30 female  28.4        1 yes    southeast  19522.
##  6    30 male    38.8        1 no     southeast  18963.
##  7    30 male    24.4        3 yes    southwest  18259.
##  8    30 female  30.9        3 no     southwest   5326.
##  9    30 male    31.6        3 no     southeast   4838.
## 10    30 female  43.1        2 no     southeast   4754.
## # ... with 465 more rows
```

Select

In our current data set we have a relatively small number of columns, so working with our tibble is not too complicated. However, we often encounter data sets with large numbers of columns. In such situations, we might want to select a subset of columns. For that we have the function `select`.

To select certain columns, input the names into select.

```
select(claim_df, age, sex)
```

```
## # A tibble: 475 x 2
##      age sex
##    <int> <fct>
##  1    31 female
##  2    46 female
##  3    62 female
##  4    56 female
##  5    56 male
##  6    30 male
##  7    30 female
##  8    59 female
##  9    31 male
## 10    60 male
## # ... with 465 more rows
```

We can also select all columns between two columns with a colon. Using a minus sign will select all columns except the ones in the expression.

```
select(claim_df, bmi:region)
```

```
## # A tibble: 475 x 4
##      bmi children smoker region
##    <dbl>    <int> <fct>  <fct>
##  1  25.7        0 no     southeast
##  2  33.4        1 no     southeast
##  3  26.3        0 yes    southeast
##  4  39.8        0 no     southeast
##  5  40.3        0 no     southwest
##  6  35.3        0 yes    southwest
##  7  32.4        1 no     southwest
##  8  27.7        3 no     southeast
##  9  36.3        2 yes    southwest
## 10  39.9        0 yes    southwest
## # ... with 465 more rows
```

```
select(claim_df, -(bmi:region))
```

```
## # A tibble: 475 x 3
```

```
##      age sex    charges
##    <int> <fct>    <dbl>
##  1    31 female   3757.
##  2    46 female   8241.
##  3    62 female  27809.
##  4    56 female  11091.
##  5    56 male    10602.
##  6    30 male    36837.
##  7    30 female   4150.
##  8    59 female  14001.
##  9    31 male    38711
## 10    60 male    48173.
## # ... with 465 more rows
```

There are several utility functions that let us select columns based on their names, for example `ends_with`, `starts_with`, or `contains`.

```
select(claim_df, starts_with("c"))
```

```
## # A tibble: 475 x 2
##    children charges
##       <int>   <dbl>
##  1        0   3757.
##  2        1   8241.
##  3        0  27809.
##  4        0  11091.
##  5        0  10602.
##  6        0  36837.
##  7        1   4150.
##  8        3  14001.
##  9        2  38711
## 10        0  48173.
## # ... with 465 more rows
```

Mutate

To create new variables in the data frame, dependent on the existing variables, we can use the `mutate()` function. For example, let's create a new variable, which will consist of charges per insured person.

```
claim_df <- mutate(claim_df, charges_per_person = charges / (children + 1))
claim_df
```

```
## # A tibble: 475 x 8
##      age sex      bmi children smoker region    charges charges_per_person
##    <int> <fct>  <dbl>    <int> <fct>  <fct>       <dbl>              <dbl>
##  1    31 female  25.7        0 no     southeast   3757.              3757.
##  2    46 female  33.4        1 no     southeast   8241.              4120.
##  3    62 female  26.3        0 yes    southeast  27809.             27809.
##  4    56 female  39.8        0 no     southeast  11091.             11091.
##  5    56 male    40.3        0 no     southwest  10602.             10602.
##  6    30 male    35.3        0 yes    southwest  36837.             36837.
##  7    30 female  32.4        1 no     southwest   4150.              2075.
##  8    59 female  27.7        3 no     southeast  14001.              3500.
##  9    31 male    36.3        2 yes    southwest  38711              12904.
## 10    60 male    39.9        0 yes    southwest  48173.             48173.
## # ... with 465 more rows
```

We can also use own functions when creating new variables. For example, let us create a new variable, which will classify the insured according to the standard BMI categories.

```
classify_bmi <- function (bmi) {
  bmi_classes <- rep("underweight", times = length(bmi))
  bmi_classes[bmi >= 18.5 & bmi < 25] <- "normal"
  bmi_classes[bmi >= 25]              <- "overweight"
  bmi_classes <- factor(bmi_classes, levels = c("underweight",
                                                "normal",
                                                "overweight"),
                        ordered = TRUE)
  return(bmi_classes)
}
claim_df <- mutate(claim_df, bmi_class = classify_bmi(bmi))
claim_df
```

```
## # A tibble: 475 x 9
##      age sex     bmi children smoker region charges charges_per_per~
##    <int> <fct> <dbl>    <int> <fct>  <fct>    <dbl>            <dbl>
##  1    31 fema~  25.7        0 no     south~   3757.            3757.
##  2    46 fema~  33.4        1 no     south~   8241.            4120.
##  3    62 fema~  26.3        0 yes    south~  27809.           27809.
##  4    56 fema~  39.8        0 no     south~  11091.           11091.
##  5    56 male   40.3        0 no     south~  10602.           10602.
##  6    30 male   35.3        0 yes    south~  36837.           36837.
##  7    30 fema~  32.4        1 no     south~   4150.            2075.
##  8    59 fema~  27.7        3 no     south~  14001.            3500.
##  9    31 male   36.3        2 yes    south~  38711            12904.
## 10    60 male   39.9        0 yes    south~  48173.           48173.
## # ... with 465 more rows, and 1 more variable: bmi_class <ord>
```

The tibble is too wide to show all variables. Let us use select to check the values of our new variable.

```
select(claim_df, bmi, bmi_class)
```

```
## # A tibble: 475 x 2
##      bmi bmi_class
##    <dbl> <ord>
##  1  25.7 overweight
##  2  33.4 overweight
##  3  26.3 overweight
##  4  39.8 overweight
##  5  40.3 overweight
##  6  35.3 overweight
##  7  32.4 overweight
##  8  27.7 overweight
##  9  36.3 overweight
## 10  39.9 overweight
## # ... with 465 more rows
```

Summarise

The `summarise` function aggregates the data according to some condition. Conditions are provided with the function `group_by`, if they are not, the data are aggregated over the whole tibble.

```
summarise(claim_df, mean_age = mean(age), mean_charges = mean(charges))
```

```
## # A tibble: 1 x 2
##   mean_age mean_charges
##      <dbl>        <dbl>
## 1     46.7       15341.
```

To get something more meaningful, we first need to group the data. For example let us look at the mean charges, dependent on whether the insured is a smoker and his BMI class.

```
g_data <- group_by(claim_df, smoker, bmi_class)
summarise(g_data, mean_charges = mean(charges))
```

```
## # A tibble: 5 x 3
## # Groups:   smoker [2]
##   smoker bmi_class   mean_charges
##   <fct>  <ord>              <dbl>
## 1 no     normal            10454.
## 2 no     overweight         9931.
## 3 yes    underweight       19023.
## 4 yes    normal            20420.
## 5 yes    overweight        38326.
```

The pipe

To arrive at the above results we made several changes to the original data set. However, we can use the pipe %>% to do all these calls sequentially, without creating an additional data set, or changing the original.

Let us demonstrate how to get the same result as above with use of the pipe.

```
claim_df %>%
  filter(age >= 18, region %in% c("southwest", "southeast")) %>%
  mutate(bmi_class = classify_bmi(bmi)) %>%
  group_by(smoker, bmi_class) %>%
  summarise(mean_charges = mean(charges))
```

```
## # A tibble: 5 x 3
## # Groups:   smoker [2]
##   smoker bmi_class   mean_charges
##   <fct>  <ord>              <dbl>
## 1 no     normal            10454.
## 2 no     overweight         9931.
## 3 yes    underweight       19023.
## 4 yes    normal            20420.
## 5 yes    overweight        38326.
```

To count the number of cases in each group, use `count()`.

```
claim_df %>%
  filter(age >= 18, region %in% c("southwest", "southeast")) %>%
  mutate(bmi_class = classify_bmi(bmi)) %>%
  group_by(smoker, bmi_class) %>%
  count()
```

```
## # A tibble: 5 x 3
## # Groups:   smoker, bmi_class [5]
##   smoker bmi_class       n
##   <fct>  <ord>       <int>
```

```
## 1 no     normal         35
## 2 no     overweight    340
## 3 yes    underweight     1
## 4 yes    normal         15
## 5 yes    overweight     84
```

Long and wide data formats

Usually we encounter data in a wide format. A wide format of data is a format where each row represents an object, some columns represent identifiers of this object, and several columns contain measurements associated with this object. On the other hand, in a long format each row represents a measurement. In other words, the columns that contain object identifiers remain unchanged, but we get a new row for each of the measured values. The long format is usually easier to process, while the wide format is easier to comprehend. Also several R functions (for example `ggplot`) require a long data format.

The functions for conversion between the formats in **tidyr** are `gather` (wide to long) and `spread` (long to wide). Let us look how to use them on a stock market data (acquired from the R package **datasets**). Here we have the daily closing prices of four major European stock indices between the years 1991 and 1998. Each row represents an object – the day of the closing prices. Then we have four measurements (prices). This data frame is therefore in a wide format. Let us convert it to a long format, and then back to wide, to see how to use `gather` and `spread`.

```
library(tidyr)
stock_df <- datasets::EuStockMarkets
stock_df <- as_tibble(data.frame(X = as.matrix(stock_df), time=time(stock_df)))
stock_df
```

```
## # A tibble: 1,860 x 5
##    X.DAX X.SMI X.CAC X.FTSE  time
##    <dbl> <dbl> <dbl>  <dbl> <dbl>
##  1 1629. 1678. 1773.  2444. 1991.
##  2 1614. 1688. 1750.  2460. 1992.
##  3 1607. 1679. 1718   2448. 1992.
##  4 1621. 1684. 1708.  2470. 1992.
##  5 1618. 1687. 1723.  2485. 1992.
##  6 1611. 1672. 1714.  2467. 1992.
##  7 1631. 1683. 1734.  2488. 1992.
##  8 1640. 1704. 1757.  2508. 1992.
##  9 1635. 1698. 1754   2510. 1992.
## 10 1646. 1716. 1754.  2497. 1992.
## # ... with 1,850 more rows
```

```
df_long <- gather(stock_df, key = "stock", value = "price", -time)
df_long
```

```
## # A tibble: 7,440 x 3
##     time stock price
##    <dbl> <chr> <dbl>
##  1 1991. X.DAX 1629.
##  2 1992. X.DAX 1614.
##  3 1992. X.DAX 1607.
##  4 1992. X.DAX 1621.
##  5 1992. X.DAX 1618.
##  6 1992. X.DAX 1611.
##  7 1992. X.DAX 1631.
```

```
##  8 1992. X.DAX 1640.
##  9 1992. X.DAX 1635.
## 10 1992. X.DAX 1646.
## # ... with 7,430 more rows
```

```
df_wide <- spread(df_long, key = "stock", value = "price")
df_wide
```

```
## # A tibble: 1,860 x 5
##     time X.CAC X.DAX X.FTSE X.SMI
##    <dbl> <dbl> <dbl>  <dbl> <dbl>
##  1 1991. 1773. 1629.  2444. 1678.
##  2 1992. 1750. 1614.  2460. 1688.
##  3 1992. 1718  1607.  2448. 1679.
##  4 1992. 1708. 1621.  2470. 1684.
##  5 1992. 1723. 1618.  2485. 1687.
##  6 1992. 1714. 1611.  2467. 1672.
##  7 1992. 1734. 1631.  2488. 1683.
##  8 1992. 1757. 1640.  2508. 1704.
##  9 1992. 1754  1635.  2510. 1698.
## 10 1992. 1754. 1646.  2497. 1716.
## # ... with 1,850 more rows
```