R 101 – 4. Operators

Stephan.Huber@hs-fresenius.de

Hochschule Fresenius

Describe the meaning of the # sign and the <- operator.

- The # sign is used in R-scripts to add comments. That is usefull that you and others can understand what the R code is about. Comments are not run as R code, so they will not influence your result.
- <- is an assignment operator that assigns a value to a name/variable. A variable allows you to store a value or an object in R and you can create that with the assignment operator. You can then later use it to access the value or the object that is stored within this variable.

Explain briefly ways to get help via the Console.

- ?: Search R documentation for a specific term.
- ?? Search R help files for a word or phrase.
- RSiteSearch(): Search search.r-project.org
- findFn(): Search search.r-project.org for functions (Hint: requires the "sos" library loaded!)
- help.start(): Access to html manuals and documentations implemented in R
- apropos(): Returns a character vector giving the names of objects in the search list matching (as a regular expression) what.
- find(): Returns where objects of a given name can be found.
- vignette(): View a specified package vignette, i.e., supporting material such as introductions.

Explain what the webpage RSeek.org can do for you.

It makes it easy to search for information about R while filtering out sites hat match "R" but don't contain the information you're looking for. It's just like a Google search, but restricts the search just to those sites known to contain information about R.

State the arithmetic operators (plus, minus, etc.) in R.

Operator	Description
+	addition
-	subtraction
	multiplication
/	division
^ or **	exponentiation

Name the logical operators (greater than, equal to, etc.) that can be used in R.

Operator	Description
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to

What does ! mean in R?

It's the logical operator for negation and it also means the opposite of a function.

```
(5>1)
```

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

```
!(5>1)
```

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

```
(1>5)
```

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

```
!(1>5)
```

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

What is the pipe operator, %>% in R?

The pipe operator, %>%, comes from the magrittr package which is also a part of the tidyverse package. The pipe is to help you write code in a way that is easier to read and understand. As R is a functional language, code often contains a lot of parenthesis, (and). Nesting those parentheses together is complex and you easily get lost. This makes your R code hard to read and understand. Here's where %>% comes in to the rescue! Consider the following chunk of code to explain the usage of the pipe:

```
# create some data `x`
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)
x
```

```
## [1] 0.109 0.359 0.630 0.996 0.515 0.142 0.017 0.829 0.907
```

```
# take the logarithm of `x`,
x2 <- log(x)
x2
```

```
## [1] -2.216407397 -1.024432890 -0.462035460 -0.004008021 -0.663588378
## [6] -1.951928221 -4.074541935 -0.187535124 -0.097612829
```

```
# compute the lagged and iterated differences (see `diff()`)
x3 <- diff(x2)
x3
```

```
## [1]  1.19197451  0.56239743  0.45802744 -0.65958036 -1.28833984 -2.12261371
## [7]  3.88700681  0.08992229
```

```
# compute the exponential function
x4 <- exp(x3)
x4
```

```
## [1]  3.2935780  1.7548747  1.5809524  0.5170683  0.2757282  0.1197183 48.7647059
## [8]  1.0940893
```

```
#  Make yourself familiar with the functions round() and round the result (1 digit)
x5 <- round(x4, 1)
x5
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

That is rather long and we actually don't need objects x2, x3, and x4. Well, then let us write that in a nested function:

```
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)

round(exp(diff(log(x))), 1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

This is short but you easily loose overview. The solution is the *pipe*:

```
# load one of these packages: `magrittr` or `tidyverse`
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
# Perform the same computations on `x` as above
x %>% log() %>%
    diff() %>%
    exp() %>%
    round(1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

You can read the %>% with "and then" because it takes the results of some function "and then" does something with that in the next. Another example can be found in this short clip: Using the pipe operator in R

Read out loud the following code:

```
iris %>%
  subset(Sepal.Length > 5) %>%
  aggregate(. ~ Species, ., mean)
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa     5.313636    3.713636     1.509091   0.2772727
## 2 versicolor     5.997872    2.804255     4.317021   1.3468085
## 3  virginica     6.622449    2.983673     5.573469   2.0326531
```

A solution may be the following: "you take the Iris data *and then* you subset the data *and then* you aggregate the data and show the mean".

What sort of *extract operators* exist in R and how can they be used?

The extract operator is used to retrieve data from objects in R. The operator may take four forms, including `[`, `[[`, `$`, and `@`. The fourth form, `@`, is called the slot operator, and is a more advanced topic so we won't discuss it here.

The first form, `[`, can be used to extract content from vector, lists, or data frames. Since vectors are one dimensional, i.e., they contain between 1 and N elements, we apply the extract operator to the vector as a single number or a list of numbers as follows.

```
x[ selection criteria here ]
```

The following code defines a vector and then extracts the last 3 elements from it using two techniques. The first technique directly references elements 13 through 15. The second approach uses the length of the vector to calculate the indexes of last three elements.

```
x <- 16:30 # define a vector
x[13:15] # extract last 3 elements
```

```
## [1] 28 29 30
```

```
x[(length(x)-2):length(x)] # extract last 3 elements
```

```
## [1] 28 29 30
```

When used with a list, `[` extracts one or more elements from the list. When used with a data frame, the extract operator can select rows, columns, or both rows and columns. Therefore, the extract opertor takes the following form: rows then a comma, then columns.

```
x[select criteria for rows , select criteria for columns]
```

The second and third forms of the extract operator, `[[` and `$` extract a single item from an object. It is used to refer to an element in a list or a column in a data frame. The easiest way to see how the various features of the extract operator work is to get through some examples.The following snippetsuse the `mtcars` data set from the `datasets` package.

```
library(datasets)
data(mtcars)

# Here, we set up a column name in a variable to illustrate use
# of various forms of the extract operator with a column name stored in
# another R object
theCol <- "cyl"

# approach 1: use [[ form of extract operator to extract a column
#             from the data frame as a vector
#             this works because a data frame is also a list
mtcars[[theCol]]
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
# approach 2: use variable name in column dimension of data frame
mtcars[,theCol]
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
# approach 3: use the $ form of extract operator. Note that since this
#             form accesses named elements from the list, you can't use
#             variable substitution (e.g. theCol) with this version of
```

```
    #             extract
    mtcars$cyl
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
    # this version fails because the `$` version of extract does not
    # work with variable substitution (i.e. a computed index)
    mtcars$theCol
```

```
## NULL
```

x$y is actually just a short form for x[[“y”]].

The difference of `[ ]` and `[[ ]]` is that `[[ ]]` is used to access a component in a list or matrix whereas `[ ]` is used to access a single element in a matrix or array.

```
object <- list(a = 5, b = 6)

object ['a']
```

```
## $a
## [1] 5
```

```
object [['a']]
```

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

What is the *Console* in RStudio good for?

You can also execute R code straight in the console. This is a good way to experiment with R code, as your submission is not checked for correctness.

Creating sequences

We just learned about the `c()` operator, which forms a vector from its arguments. If we're trying to build a vector containing a sequence of numbers, there are several useful functions at our disposal. These are the colon operator `:` and the sequence function `seq()`.

`:` Colon operator:

```
1:10 # Numbers 1 to 10
```

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

```
127:132 # Numbers 127 to 132
```

```
## [1] 127 128 129 130 131 132
```

`seq` function: `seq(from, to, by)`

```
seq(1,10,1) # Numbers 1 to 10
```

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

```
seq(1,10,2) # Odd numbers from 1 to 10
```

```
## [1] 1 3 5 7 9
```

```
seq(2,10,2) # Even numbers from 2 to 10
```

```
## [1]  2  4  6  8 10
```

(a) Use `:` to output the sequence of numbers from 3 to 12

```
3:12
```

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

(b) Use `seq()` to output the sequence of numbers from 3 to 30 in increments of 3

```
seq(3, 30, 3)
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

(c) Save the sequence from (a) as a variable `x`, and the sequence from (b) as a variable `y`. Output their product `x*y`

```
x <- 3:12
y <- seq(3, 30, 3)
x * y
```

```
##  [1]   9  24  45  72 105 144 189 240 297 360
```

Solve the following exercises in your own R-Script.

1. RStudio offers a lot of helpful so-called Cheat Sheets (see: https://rstudio.com/resources/cheatsheets/) Download and make yourself familiar with the following Base R Cheeat Sheet
2. Open RStudio, create a R-script, set your working directory, load the data package, calculate *3 + 4* in R and add a comment to it.
3. Further, calculate

- divide 697 by 41
- take the square root of 12
- take 3 to the power of 12

2. Create a vector called vec1 that contain the numbers 2 5 8 12 16.
3. Use `x:y` notation to make (select) a second vector called vec2 containing the numbers 5 to 9.
4. Subtract vec2 from vec1 and look at the result.
5. Use seq() to make a vector of 100 values starting at 2 and increasing by 3 each time. Name the new vector nseries. (Hint: The example of the *creating sequences* section may be helpful.)
6.

- Extract the values at positions 5,10,15 and 20 in the vector of values you just created.
- Extract the values at positions 10 to 30
- Hint: Both of these actions require making a selection in the vector using the [ ] notation. Inside the square brackets you put a vector of index positions, so the problem here is to create the vector of index positions.

Solution:

```
697 / 41
```

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

```
sqrt(12)
```

```
## [1] 3.464102
```

```r
3 ^ 12
```

```
## [1] 531441
```

```r
c(2,5,8,12,16) -> vec1
5:9 -> vec2
vec1 - vec2
```

```
## [1] -3 -1  1  4  7
```

```r
seq(from=2,by=3,length.out=100) -> number.series
number.series
```

```
##   [1]   2   5   8  11  14  17  20  23  26  29  32  35  38  41  44  47  50  53
##  [19]  56  59  62  65  68  71  74  77  80  83  86  89  92  95  98 101 104 107
##  [37] 110 113 116 119 122 125 128 131 134 137 140 143 146 149 152 155 158 161
##  [55] 164 167 170 173 176 179 182 185 188 191 194 197 200 203 206 209 212 215
##  [73] 218 221 224 227 230 233 236 239 242 245 248 251 254 257 260 263 266 269
##  [91] 272 275 278 281 284 287 290 293 296 299
```

```r
number.series[c(5,10,15,20)]
```

```
## [1] 14 29 44 59
```

```r
number.series[seq(from=5,to=20,by=5)]
```

```
## [1] 14 29 44 59
```

```r
number.series[10:30]
```

```
##  [1] 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77 80 83 86 89
```

Can you plot data?

Please explain your classmates how to create the following plots with R:

- Pie Charts
- Bar Charts
- Boxplots
- Histograms
- Line Graphs
- Scatterplots

Can you create data frames and merge them?

Please try to understand the following lines of code:

```r
data1 <- data.frame(id = 1:6,                                    # Create first example data frame
                    x1 = c(5, 1, 4, 9, 1, 2),
                    x2 = c("A", "Y", "G", "F", "G", "Y"))
data2 <- data.frame(id = 4:9,                                    # Create second example data frame
                    y1 = c(3, 3, 4, 1, 2, 9),
                    y2 = c("a", "x", "a", "x", "a", "x"))

merge(data1, data2, by = "id")                                   # Merge data frames by columns names
```

```
##   id x1 x2 y1 y2
## 1  4  9  F  3  a
```

```
## 2  5  1  G  3  x
## 3  6  2  Y  4  a
```

```
merge(data1, data2, by = "id", all.x = TRUE)                    # Keep all rows of x-data
```

```
##   id x1 x2 y1   y2
## 1  1  5  A NA <NA>
## 2  2  1  Y NA <NA>
## 3  3  4  G NA <NA>
## 4  4  9  F  3    a
## 5  5  1  G  3    x
## 6  6  2  Y  4    a
```

```
merge(data1, data2, by = "id", all.y = TRUE)                    # Keep all rows of y-data
```

```
##   id x1   x2 y1 y2
## 1  4  9    F  3  a
## 2  5  1    G  3  x
## 3  6  2    Y  4  a
## 4  7 NA <NA>  1  x
## 5  8 NA <NA>  2  a
## 6  9 NA <NA>  9  x
```

```
merge(data1, data2, by = "id", all.x = TRUE, all.y = TRUE)      # Keep all rows of both data frames
```

```
##   id x1   x2 y1   y2
## 1  1  5    A NA <NA>
## 2  2  1    Y NA <NA>
## 3  3  4    G NA <NA>
## 4  4  9    F  3    a
## 5  5  1    G  3    x
## 6  6  2    Y  4    a
## 7  7 NA <NA>  1    x
## 8  8 NA <NA>  2    a
## 9  9 NA <NA>  9    x
```

```
data3 <- data.frame(id = 5:6,                                   # Create third example data frame
                    z1 = c(3, 2),
                    z2 = c("K", "b"))

data12 <- merge(data1, data2, by = "id")                        # Merge data 1 & 2 and store
merge(data12, data3, by = "id")                                 # Merge multiple data frames
```

```
##   id x1 x2 y1 y2 z1 z2
## 1  5  1  G  3  x  3  K
## 2  6  2  Y  4  a  2  b
```

Can you calculate growth rates and build up data frames?

Do the following:

- Load the `sunspot.year` data which is part of R's datasets package `datasets`,
- generate a vector that contains the years 1700 to 1988,
- combine the two vectors into a data frame using `data.frame()`,
- calculate the growth rate of yearly sunspot using `growth.rate()` which is part of the `tis` package,
- add the growth variable to the data frame.

```r
library("datasets")
data("sunspot.year")
year <- 1700:1988
sunspot.frame <- data.frame(year, sunspot.year)

install.packages("tis")
```

```
## Installing package into '/home/sthu/R/x86_64-pc-linux-gnu-library/4.0'
## (as 'lib' is unspecified)
```

```r
library("tis")
```

```
##
## Attaching package: 'tis'
```

```
## The following object is masked from 'package:dplyr':
##
##     between
```

```r
growth <- growth.rate(sunspot.frame$sunspot.year, lag=1, simple = T)
year <- 1701:1988
sunspot.frame2 <- data.frame(year, growth)
merge(sunspot.frame, sunspot.frame2, by = "year", all.x = TRUE, all.y = TRUE)
```

Can you load R Built-in R Data set?

Study this webpage R Built-in R Data set and show how can you load R's *Motor Trend Car Road Tests* dataset?

Can you inspect data sets?

Here is a non-exhaustive list of functions to get a sense of the content/structure of data.

- All data structures - content display:
  - **`str()`:** compact display of data contents (env.)
  - **`class()`:** data type (e.g. character, numeric, etc.) of vectors and data structure of dataframes, matrices, and lists.
  - **`summary()`:** detailed display, including descriptive statistics, frequencies
  - **`head()`:** will print the beginning entries for the variable
  - **`tail()`:** will print the end entries for the variable
- Vector and factor variables:
  - **`length()`:** returns the number of elements in the vector or factor
- Dataframe and matrix variables:
  - **`dim()`:** returns dimensions of the dataset
  - **`nrow()`:** returns the number of rows in the dataset
  - **`ncol()`:** returns the number of columns in the dataset
  - **`rownames()`:** returns the row names in the dataset
  - **`colnames()`:** returns the column names in the dataset

Load the mtcars data set and play around with the functions above.

Do you know the cars data?

The `cars` data comes with the default installation of R. To see the first few columns of the data, just type `head(cars)`.

```
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

We'll do a bad thing here and use the `attach()` command, which will allow us to access the `speed` and `dist` columns of `cars` as though they were vectors in our workspace. The `attach()` function has the side effect of altering the search path and this can easily lead to the wrong object of a particular name being found. People do often forget to detach databases. Thus, it is better to use `$`.

```
attach(cars) # Using this command is poor style.  We will avoid it in the future.
speed
```

```
##  [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14 15 15
## [26] 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24 24 24 24 25
```

```
dist
```

```
##  [1]   2  10   4  22  16  10  18  26  34  17  28  14  20  24  28  26  34  34  46
## [20]  26  36  60  80  20  26  54  32  40  32  40  50  42  56  76  84  36  46  68
## [39]  32  48  52  56  64  66  54  70  92  93 120  85
```

(a) Calculate the average and standard deviation of speed and distance.

```
mean(speed)
```

```
## [1] 15.4
```

```
sd(speed)
```

```
## [1] 5.287644
```

```
mean(dist)
```

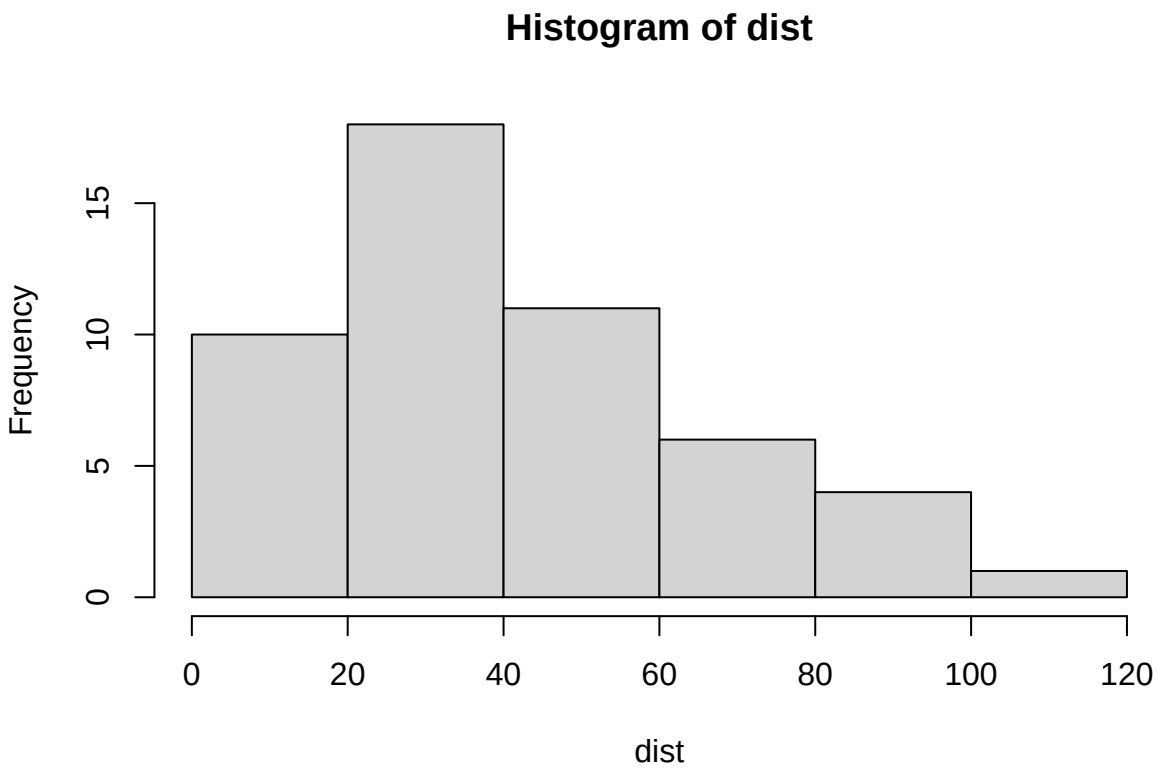
```
## [1] 42.98
```

```
sd(dist)
```

```
## [1] 25.76938
```

(b) Make a a histogram of stopping distance using the `hist` function.

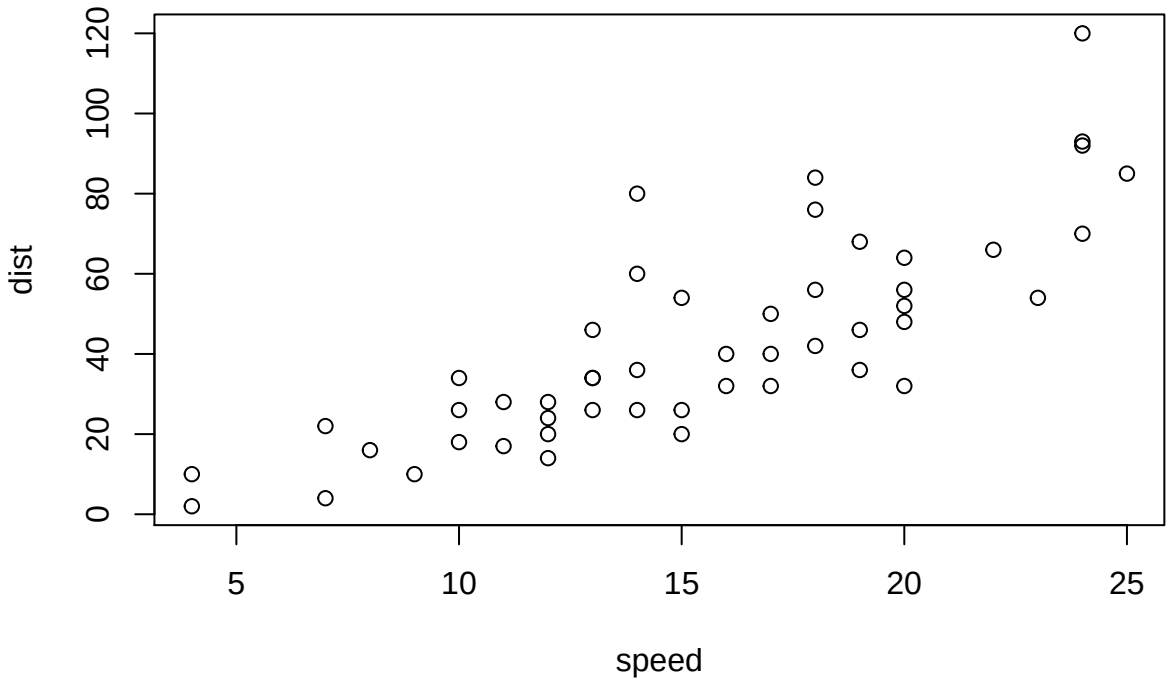
```
hist(dist) # Histogram of stopping distance
```



The `plot(x,y,...)` function plots a vector `y` against a vector `x`. You can type `?plot` into the Console to learn more about the basic plot function.

(c) Use the `plot(x,y)` function to create a scatterplot of dist against speed.

```
plot(speed, dist)
```


dist
0
20
40
60
80
100
120
5
10
15
20
25
speed