McGill | School of Information Studies

Basic R Programming

WEEK 2

13 Sept 2022

Operations in R

\+ - addition

\- - subtraction

* - multiplication

/ - division

^ - exponentiation

```
5+3
```

```
[1] 8
```

```
7*3/2
```

```
[1] 10.5
```

```
2^3
```

```
[1] 8
```

```
(2-5)*8^2
```

```
[1] -192
```

Assignment Operator

One of the most important concepts in R is the assignment of names to objects. So far the objects we have encountered are simple numbers. To assign a name to a number, you use the *assignment operator* <- (no space between < and -) or = like this:

Examples:

```
x <- 3

some.complicated.name <- 7
```

```
x + some.complicated.name
```

```
[1] 10
```

Basic Structures

VECTORS

A *numeric* containing only whole numbers (like my_vector) can be called an *integer*, which is a subtype of numeric.

Examples:

```
c(8, 2, 4, 6, 2, 1)
```

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

You can store vectors in variables as well.

```
my_vector <- c(8, 2, 4, 6, 2, 1)
```

```
my_vector
```

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

Basic Structures

NUMERIC

A vector in R is a sequence of elements of the same data type. You build a vector with the c() function.

Examples:

```
a <- c(1, 2, 3, 4)
a*3
```

```
[1]  3  6  9 12
```

```
b <- c(2, 4, 6, 7)
a+b
```

```
[1]  3  6  9 11
```

```
v <- c(1.5, 3.234, 7, 0.12356)
v
```

```
long <- c(1, 2, 3, 4)
short <- c(1, 2)
long+short
```

```
[1] 2 4 4 6
```

Basic Structures

CHARACTER

R can not only deal with numbers, it can also deal with text. A piece of text is called a *string* and is written in a pair of double or single quotes. A vector containing strings as elements is called a *character vector*:

Examples:

```
v2 <- c("male", "female", "female", "male")
v2
```

```
[1] "male"   "female" "female" "male"
```

```
v3 <- c('blue', 'brown', 'yellow')
v3
```

```
[1] "blue"   "brown"  "yellow"
```

Basic Structures

LOGICAL

Another important type of vector is the *logical* vector, the elements of which are the so called *booleans* TRUE and FALSE, which can be shortened by T and F (cases matter, you have to use upper case letters in both versions.)

Examples:

```
c(TRUE, FALSE, TRUE)
```

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

```
c(F, T, T, T)
```

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

The most common *logical operators* we will use are the following:

- AND `&`
- OR `|`
- NOT `!`
- greater than `>`
- greater or equal `>=`
- less than `<`
- less or equal `<=`
- equal to `==` (yes, you need *two* equal signs)
- not equal to `!=`

Basic Structures

Examples:

```
3 < 1
```

```
[1] FALSE
```

```
5 > 2
```

```
[1] TRUE
```

```
5 == 5
```

```
[1] TRUE
```

```
5 != 5
```

```
[1] FALSE
```

```
TRUE & FALSE
```

```
[1] FALSE
```

```
TRUE | FALSE
```

```
[1] TRUE
```

```
!FALSE
```

```
[1] TRUE
```

Basic Structures

DATA FRAMES

If you want to do statistics, the most likely format your data will come in is some kind of table. In R, the basic form of a table is called a data.frame.

Examples:

```
View(iris)
```

Table 1.1: The first 10 rows of the iris data set.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa

Basic Structures

LISTS

While data.frames are useful to bundle together vectors of the same length, *lists* are used to combine more heterogeneous data.

Examples:

```
#create list
my.list <- list(my_vector, long, iris[1:10,])
#print list
my.list
```

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

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

[[3]]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa
3           4.7         3.2          1.3         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
5           5.0         3.6          1.4         0.2  setosa
6           5.4         3.9          1.7         0.4  setosa
7           4.6         3.4          1.4         0.3  setosa
8           5.0         3.4          1.5         0.2  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
```

Functions

In principle, you can imagine a function as a little machine that takes some input (usually some kind of data), processes that input in a certain way and gives back the result as output.

Examples:

```
mean(c(2, 4, 6))
```

```
[1] 4
```

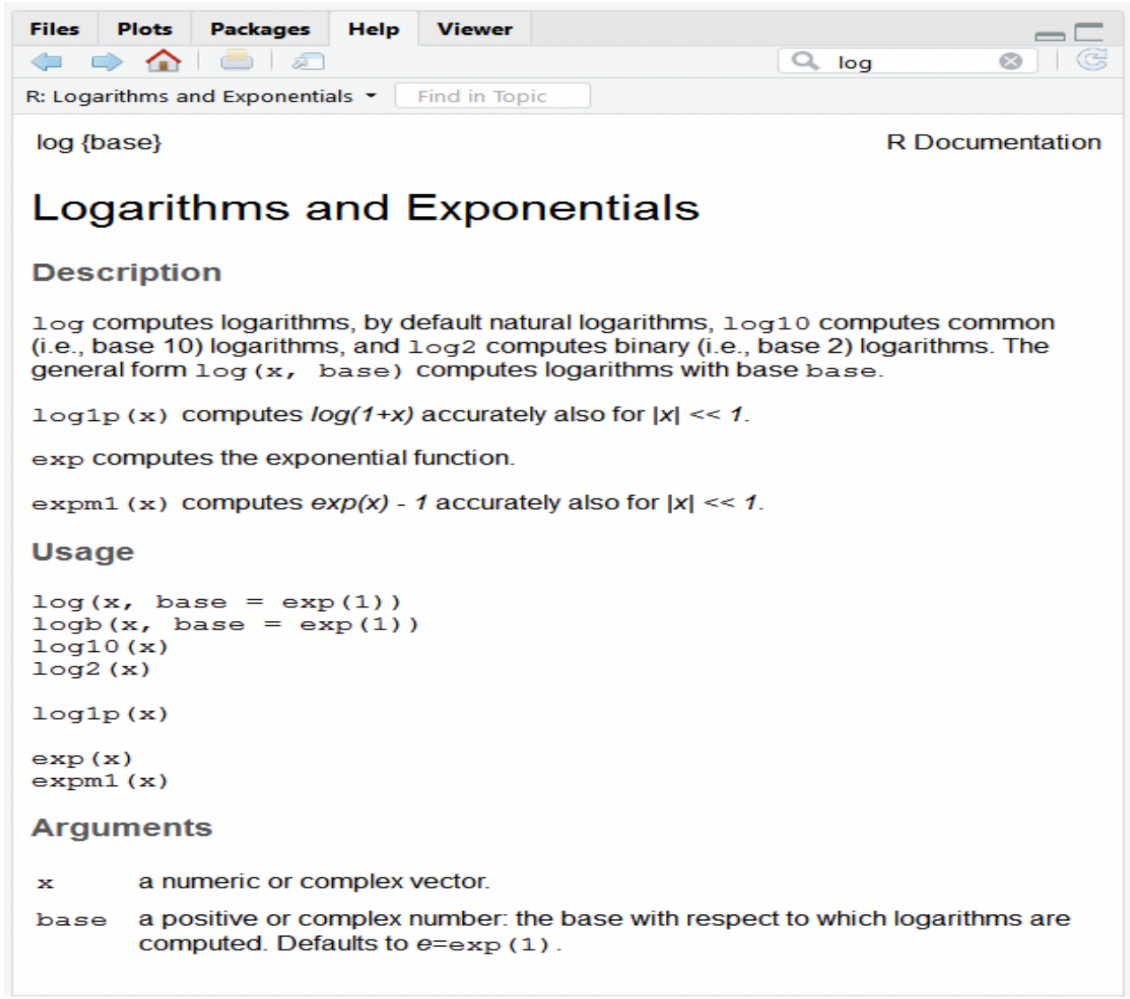
```
log(x = c(10, 20, 30), base = 10)
```

```
[1] 1.000000 1.301030 1.477121
```

How to Use a Function

Examples:

```
?log
```



Files | Plots | Packages | Help | Viewer

log

R: Logarithms and Exponentials | Find in Topic

log {base} R Documentation

Logarithms and Exponentials

Description

`log` computes logarithms, by default natural logarithms, `log10` computes common (i.e., base 10) logarithms, and `log2` computes binary (i.e., base 2) logarithms. The general form `log(x, base)` computes logarithms with base `base`.

`log1p(x)` computes *log(1+x)* accurately also for |x| << *1*.

`exp` computes the exponential function.

`expm1(x)` computes *exp(x) - 1* accurately also for |x| << *1*.

Usage

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

log1p(x)

exp(x)
expm1(x)
```

Arguments

`x`	a numeric or complex vector.
`base`	a positive or complex number: the base with respect to which logarithms are computed. Defaults to *e*=`exp(1)`.

```
log(x = c(20,30,40)) #argument base can be omitted
```

```
[1] 2.995732 3.401197 3.688879
```

```
log(base=3) #argument x can not be omitted
```

```
Error in eval(expr, envir, enclos): Argument "x" fehlt (ohne Standardwert)
```

(Translates to `Error: Argument "x" is missing (without a default value)`) If you omit the names of the arguments in the function call, R will assume which object belongs to which argument:

```
log(c(10, 20, 30), 10)
```

```
[1] 1.000000 1.301030 1.477121
```

Packages

R and since the entirety of all R functions there are is way to big to install at once, most of the functions are bundeled into so called *packages*. A package is a bundle of functions you can download and install from the *Comprehensive R Archive Network (CRAN)* (https://cran.r-project.org/).

Examples:

```
install.packages("lubridate")
```

```
today()
```

```
Error in today(): konnte Funktion "today" nicht finden
```

(Translates to `Error in today(): Could not find function "today"`).

To activate the package, you use the function library(). This function activates the package for your current R session, so you have to do this once per session (a session starts when you open R/Rstudio and ends when you close the window).

```
library(lubridate)
today()
```

```
[1] "2021-05-25"
```

Loading and saving data into R

There are generally two ways of loading data into R: either your data is available in an R-format such as an .RData file, or your data comes in some non-R format.

Examples:

```
read.csv("NINDS.csv")
```

Or set a variable name

```
NINDS_csv <- read.csv("NINDS.csv")
```

```
install.packages("openxlsx") #only do this once
```

```
library(openxlsx)
NINDS_xlsx <- read.xlsx("NINDS.xlsx")
```

NOTE! Change read to write when saving it.

Data Manipulation

The tidyverse (Wickham et al. 2019) is an R package that contains a lot of useful functions to deal with these problems, so we'll start by installing and loading this package:

```
install.packages("tidyverse") #only do this once
```

```
library(tidyverse)
```

Examples:

```
d1 <- read_csv("data1.csv")
```

```
-- Column specification -----------
cols(
  weight = col_double(),
  age = col_double(),
  id = col_double()
)
```

```
d2 <- read_csv("data2.csv")
```

```
-- Column specification ----------
cols(
  height = col_double(),
  eyecolor = col_character(),
  id = col_double()
)
```

Data Manipulation

Examples:

```
d1$id <- as.character(d1$id)
d2$id <- as.character(d2$id)

#check type
class(d1$id)
```

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

```
class (d2$id)
```

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

```
class(d1)
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
d1
```

```
# A tibble: 5 x 3
  weight   age id
   <dbl> <dbl> <chr>
1     85    34 1
2     56    72 2
3     73    33 3
4     76    45 4
5     60    23 5
```

Select and Filter

The first two functions for data manipulation are select(), which allows you to keep only certain variables (i.e. columns) and filter(), which allows you to keep only certain rows.

Examples:

```
select(d1, c(id, age))
```

```
# A tibble: 5 x 2
  id      age
  <chr> <dbl>
1 1        34
2 2        72
3 3        33
4 4        45
5 5        23
```

```
select(d1, -c(weight, id))
```

```
# A tibble: 5 x 1
    age
  <dbl>
1    34
2    72
3    33
4    45
5    23
```

```
filter(d1,(age > 30) & (weight > 60) & (weight < 75) )
```

```
# A tibble: 1 x 3
  weight   age id
   <dbl> <dbl> <chr>
1     73    33 3
```

Join

Another useful operation is the the *join* which allows you to join two data sets by a common *key variable*.

Examples:

```
d1
```

```
# A tibble: 5 x 3
  weight   age id
   <dbl> <dbl> <chr>
1     85    34 1
2     56    72 2
3     73    33 3
4     76    45 4
5     60    23 5
```

```
d2
```

```
# A tibble: 5 x 3
  height eyecolor id
   <dbl> <chr>    <chr>
1    156 brown    2
2    164 blue     1
3    189 brown    4
4    178 green    3
5    169 blue     6
```

Inner join

The inner join only keeps cases (i.e. rows), that appear in both data sets. The function inner_join() takes two data.frames or tibbles and a string giving the name of the *key variable* that defines which rows belong together:

Examples:

```
inner_join(d1, d2, by="id")
```

```
# A tibble: 4 x 5
  weight   age id    height eyecolor
   <dbl> <dbl> <chr>  <dbl> <chr>
1     85    34 1        164 blue
2     56    72 2        156 brown
3     73    33 3        178 green
4     76    45 4        189 brown
```

Full join

The opposite of the inner join is the full join. full_join() takes the same arguments as inner_join() but returns all cases. If a case doesn't appear in the other data set, the missing information is indicated with NA:

Examples:

```
full_join(d1,d2,by="id")
```

```
# A tibble: 6 x 5
  weight   age id    height eyecolor
   <dbl> <dbl> <chr>  <dbl> <chr>
1     85    34 1        164 blue
2     56    72 2        156 brown
3     73    33 3        178 green
4     76    45 4        189 brown
5     60    23 5         NA <NA>
6     NA    NA 6        169 blue
```

Right and left join

The right and left join take one of the data sets fully and join only the rows from the other data set that fit this data set. That is, the right join takes the full right data set from the function call and attaches all fitting rows from the left data set, whereas the left join takes the full left data set from the function call and attaches all fitting rows from the right data set:

Examples:

```
left_join(d1,d2, by="id") #all cases from d1 are kept
```

```
# A tibble: 5 x 5
  weight   age id    height eyecolor
   <dbl> <dbl> <chr>  <dbl> <chr>
1     85    34 1        164 blue
2     56    72 2        156 brown
3     73    33 3        178 green
4     76    45 4        189 brown
5     60    23 5         NA <NA>
```

```
right_join(d1,d2, by="id") #all cases from d2 are kept
```

```
# A tibble: 5 x 5
  weight   age id    height eyecolor
   <dbl> <dbl> <chr>  <dbl> <chr>
1     85    34 1        164 blue
2     56    72 2        156 brown
3     73    33 3        178 green
4     76    45 4        189 brown
5     NA    NA 6        169 blue
```

Split-Apply-Combine Paradigm

One thing you'll want to do quite often in statistics is to compute a certain statistic not for your whole sample but individually for certain subgroups. The steps needed for this are the following:

- split your data in subsets according to some factor, e.g. eye color
- apply the statistic to each subset
- combine the results in a suitable way

Examples:

```
install.packages("plyr") #only do this once
```

```
library(plyr)# load plyr
library(dplyr) #load dplyr again after plyr
```

```
#create a summary with the summarise option
ddply(d,"eyecolor", summarise, meanBMI=mean(BMI, na.rm=T),
      medianHeight=median(height, na.rm=T))
```

	eyecolor	meanBMI	medianHeight
1	blue	31.60321	166.5
2	brown	22.14359	172.5
3	green	23.04002	178.0

```
#alternatively attach results to original data with the mutate option
ddply(d,"eyecolor", mutate, meanBMI=mean(BMI, na.rm=T),
      medianHeight=median(height, na.rm=T))
```

	weight	age	id	height	eyecolor	BMI	blueEyes	meanBMI	medianHeight
1	85	34	1	164	blue	31.60321	1	31.60321	166.5
2	NA	NA	6	169	blue	NA	1	31.60321	166.5
3	56	72	2	156	brown	23.01118	0	22.14359	172.5
4	76	45	4	189	brown	21.27600	0	22.14359	172.5
5	73	33	3	178	green	23.04002	0	23.04002	178.0

FIN

HANDS-ON EXERCISES

Sample Hands-on Exercise 2.1

Perform the following:

- Load the data1.csv and data2.csv datase
- print out each top 20 of the data frame
- Examine the representation of the factor in d1 and d2
- select Institution and Failure in d1
- unselect Institution and age in d2
- filter by Sex is Male, and 1st to 3rd Class
- right join d1 and d2 by PatientID
- Create new variable name as Sample by dividing age and failure
- Create a summary with the summarise option with meanAge and medianFailure
- Create a summary with the mutate option with meanAge and medianFailure

Sample Hands-on Exercise 2.2

Perform the following:

- Create a logical vector
- Negate this vector
- Compute the truth table for logical AND
- Explore arithmetic with logical and numeric
- Compute the intersection of {1,2,…,10} and {5,6,…,15}
- Create factor
- Examine the representation of the factor