R Tutorial at the WZB

3 - R Basics II

Markus Konrad

November 08, 2018

Introductory note

For this session there are no slides, because I'm not at the WZB to hold a presentation. Instead I provide you with a full document for self-study. As always, there are some tasks at the end of the document which you should complete. I recommend that you not only read the document, but also try out some of the code examples and experiment with them.

Solution for tasks #2

The solutions for tasks #2 are now online on https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/ (https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/).

1 Functions

1.1 Making things happen with functions

> To understand computations in R, two slogans are helpful:
> 1. Everything that exists is an object.
> 2. Everything that happens is a function call.
>
> — John Chambers

With functions, we can *do* things with objects. We've used some of them before without noticing: `sum(...)`, `mean(...)`, `length(...)`, etc. but let's formally introduce you to the concept of functions in R in this section.

In order to do things with objects, we need to *pass* them to the function. We do so by assigning them as *arguments (also: parameters)*. Most of the time, the function will compute something (e.g. sum, mean, etc. of some values) and hence will produce a *return value*.

1.1.1 Anatomy of a function call

```
y <- some_func(x1, x2)
```

There are three components: 1. `some_func` : function name 2. `x1, x2` : function **arguments** (a.k.a. **parameters**) 3. **return value** (here, directly assigned to object `y`)

The only component that is obligatory, is the function name (this means, functions can have no arguments and/or return "nothing"). We can use (also: *call*) already defined functions (from base R or loaded packages) or define our own functions. We'll focus on the first in this session.

1.1.2 Function arguments

```
other_func(a, b = 2, c = FALSE)
```

Each argument can be either **obligatory or optional**. Optional arguments must have **default values** assigned to them with `=` . These are used when yo don't pass a value for this argument. You can either pass arguments in order or by argument name.

Example: R's function to compute a mean value is defined as:

```
mean(x, trim = 0, na.rm = FALSE)
```

Which arguments are obligatory, which optional?

- `x` is obligatory
- `trim` and `na.rm` have default values, hence they're optional

Remember:

- an assignment to an object is made with `<-`
- assigning a default value to a function argument is made with `=`

So don't do this: `mean(x, na.rm <- TRUE)` !

R's function to sort values of a vector `x` is defined as:

```
sort(x, decreasing = FALSE)
```

- parameter `decreasing` controls the sorting direction
- you can pass arguments either in order or by name, or mix both approaches:

Passing the arguments in order looks like this:

```
values <- c(10, 4, 12, 15, 3)
sort(values, TRUE)
```

```
## [1] 15 12 10  4  3
```

Explicitely naming the arguments is done like this:

```
sort(x = values, decreasing = TRUE)
```

```
## [1] 15 12 10  4  3
```

When you name the arguments, order does not matter:

```
sort(decreasing = TRUE, x = values)
```

```
## [1] 15 12 10  4  3
```

You can also pass the first argument(s) in order and additionally pass named arguments:

```
sort(values, decreasing = TRUE)
```

```
## [1] 15 12 10  4  3
```

1.2 Example of a function call

Example data: All ozone observations from built-in `airquality` dataset.

```
airquality$Ozone
## [1]  41  36  12  18  NA  28  23  19   8 ...
```

Passing the whole “Ozone” *vector*:

```
mean(airquality$Ozone)
## [1] NA
```

Here, `mean(...)` returns `NA` because all calculations that involve a `NA` value by default result in `NA` . So we should specify to *remove NAs* before calculation using the `na.rm` argument of `mean(...)` :

```
mean(airquality$Ozone, na.rm = TRUE)
## [1] 42.12931
```

Which is equivalent to (passing all arguments in order):

```
mean(airquality$Ozone, 0, TRUE)
## [1] 42.12931
```

1.3 Nesting function calls

You can directly send the result *(output)* of one function as input to the next function *(nesting functions)*.

Here we identify `NA` values which produces a logical vector, an then pass this vector to `sum()`, effectively counting the missing values:

```
sum(is.na(airquality$Ozone))
```

```
## [1] 37
```

Note the nested parantheses. For each opening paranthesis, there *must* be a closing counterpart. Nested functions are evaluated from inner to outer.

So `sum(is.na(airquality$Ozone))` is the same as:

```
ozone_NAs <- is.na(airquality$Ozone)
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE ...
```

```
sum(ozone_NAs)
```

```
## [1] 37
```

You can nest function calls as deep as you like:

```
mean(head(sort(airquality$Ozone), 10))
```

```
## [1] 6.7
```

- `sort(...)` orders the ozone values in increasing order
- `head(..., 10)` then selects the first 10 values (i.e. lowest 10 values from the ozone vector)
- `mean(...)` calculates the arithmetic mean of those values

2 Object types

As described in the previous section, each object in R has a *type*. We have already worked with vectors of different types (e.g. numerical, logical), with data frames (that consist of vectors of multiple types) and factors (the data type to store categorical data).

Knowing the object type you're dealing with or that you want to use for storing your data is extremely important. It defines what you can store and what you can *do* with the object, i.e. which functions and operators you can apply to the data.

2.1 Using the wrong object type

Imagine that you record postal codes and cities in two vectors like this:

```
postal_codes <- c(31231, 56080, 04668, 82341)
cities <- c('Acity', 'Othercity', 'Somecity', 'Avillage')
```

You want to combine them to form an address with postal code + city name.

```
paste(postal_codes, cities)
```

```
## [1] "31231 Acity"     "56080 Othercity" "4668 Somecity"   "82341 Avillage"
```

This works well, however, we introduced an error: The leading zero was removed in “4668 Somecity”! This happened because `postal_code` was defined as a numeric vector and leading zeros are always stripped in numbers.

→ using the wrong data type might cause loss of information

Using a character vector for the postal codes fixes the error:

```
postal_codes <- c('31231', '56080', '04668', '82341')
cities <- c('Acity', 'Othercity', 'Somecity', 'Avillage')
paste(postal_codes, cities)
```

```
## [1] "31231 Acity"     "56080 Othercity" "04668 Somecity"  "82341 Avillage"
```

Image you define whether a participant is a smoker in a character vector with options `'yes'` or `'no'`. You want to find out how many participants are smokers. When using character vectors you can’t use the `sum()` function (directly) for the counting:

```
smoker <- c('yes', 'yes', 'no', 'yes', 'no')
sum(smoker)
## Error in sum(smoker) : invalid 'type' (character) of argument
```

The above fails because `sum()` can’t convert the input vector to numbers which it can sum up.

→ using the wrong data type might limit you in which functions you can use or how the function behaves

However, using a logical vector for the binary outcome of being a smoker allows us to use `sum()` to count the occurrences of `TRUE`:

```
smoker <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
sum(smoker)
```

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

This is because a logical vector can be converted to `1`s and `0`s for `TRUE`s and `FALSE`s.

2.2 Finding out the object type

The function `class(...)` helps to find out the type of any object:

```
is_female <- c(FALSE, TRUE, TRUE, FALSE, TRUE)
class(is_female)
```

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

You can use it on any object:

```
class(airquality)
```

```
## [1] "data.frame"
```

```
class(sum)
```

```
## [1] "function"
```

```
class(airquality$Wind)
```

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

2.3 Compactly displaying complex data structures

The function `str` allows to show the structure of more complex object types such as factors, data frames or lists (to be introduced later):

```
color <- factor(c('red', 'red', 'green', 'red', 'blue'))
str(color)
```

```
##  Factor w/ 3 levels "blue","green",..: 3 3 2 3 1
```

For a data frame, it compactly displays the number of observations and variables and lists each variable with its type (abbreviated as “int”, “num”, etc.) and the first few values:

```
str(airquality)
```

```
## 'data.frame':    153 obs. of  6 variables:
##  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
##  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
##  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
##  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

3 Comparisons and logical expressions

Comparisons are fundamental during data preparation and analysis. **The result of a comparison is always a logical vector** with the *Boolean truth values* `TRUE` or `FALSE` . With **logical expressions** you can combine simple comparisons to complex expressions in different ways. Such expressions are important for example for:

- Filtering (e.g. filter for all participants that are smokers *and* younger than 30)
- Grouping (e.g. group by smokers / non-smokers)

- Counting (e.g. count the proportion of female non-smokers)

3.1 Simple comparisons

We'll use the following vectors for demonstration:

```
age <- c(20, 35, 19, 51, 20)
smoker <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
country <- factor(c('USA', 'GB', 'GB', 'DE', 'USA'))
```

R provides the standard set of comparison operators. Let's start with equality and inequation.

3.1.1 Equality (==) and inequation (!=)

The equality operator `==` (double "equals" signs) checks if both sides are equal:

```
age == 20
```

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

You will usually compare a vector of values with a single scalar value as in the above example. In this case, each value in the vector is compared to the scalar vector (`20` in the above example). Another common operation is that you compare two vectors of the same size, for example:

```
age == c(20, 35, 19, 51, 22)
```

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

Here, each vector element on the left side is directly compared with its companion element on the right side. Watch out when comparing vectors of different sizes. For this, the resul of vector recycling (see last session) apply.

A very common mistake for beginners is to accidently use only a single equals sign `=` instead of double equals signs `==` . **If you do this, you will accidently reassign a new value to the variable!**

```
age = 20
## no logical comparison, instead age is now set to 20
```

This is because a single `=` can also be used as assignment operator instead of `<-` .

The opposite operation, inequation, can be applied using the `!=` operator, which will yield `TRUE` everywhere where the left and right side value does not match:

```
age != 20
```

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

Similar operations can be done with character string or factor variables:

```
country == 'GB'
```

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

```
country != 'GB'
```

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

3.1.2 Greater-than/greater-equal (>, >=) and less-than/less-equal (<, <=)

Greater-than `>` or greater-equal `>=` comparisons can be applied to numeric or ordered categorical variables:

```
age > 19
```

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

```
age >= 19
```

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

Lesser-than `>` or lesser-equal `>=` work similarly:

```
age < 50
```

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

```
age <= 20
```

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

3.1.3 Negation (!)

A negation turns all `TRUE` values into `FALSE` and vice versa. You can negate any Boolean expression by prepending it with an exclaimation mark (`!`):

```
smoker
```

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

```
!smoker
```

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

3.1.4 Gotchas

It depends on the data type, which operator can be used. For example, it doesn't make sense to use a greater-than or less-than comparison with a factor (categorical variable) unless it is ordered. Trying to use it will result in a warning and a vector of `NA` values:

```
country > 'GB'
```

```
## Warning in Ops.factor(country, "GB"): '>' not meaningful for factors
```

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

```
## '>' not meaningful for factors
## [1] NA NA NA NA NA
```

Senseless comparisons not always cause errors or warnings. For example, you *can* negate the `age`:

```
!age
```

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

This is because the negation operator would implicitely coerce the numeric vector to a logical vector, where each value that is `0` will become `FALSE` and all other values will become `TRUE`. Hence in the following example, only the third value (`0`) becomes `FALSE`, all others become `TRUE` and in the end the whole vector is negated:

```
!c(5, 2, 0, 10, 1)
```

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

Checking for equality or inequation with numeric values that are not integers (they are called **floating point** values) can cause some suprise. For example, the following certainly holds true: $\frac{1}{49} \cdot 49 = 1$.

However, not so in R:

```
1/49 * 49 == 1
```

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

This is not some incapacity of R (all programming languages expose this problem to some degree). Computers use finite precision arithmetic (you can't store an infinite number of digits in memory) and hence rounding occurs during computation. So when checking for equality with floating point values, **always use a function that incorporates a rounding tolerance** like `near()` from the package `dplyr`:

```
library(dplyr)
near(1/49 * 49, 1)
```

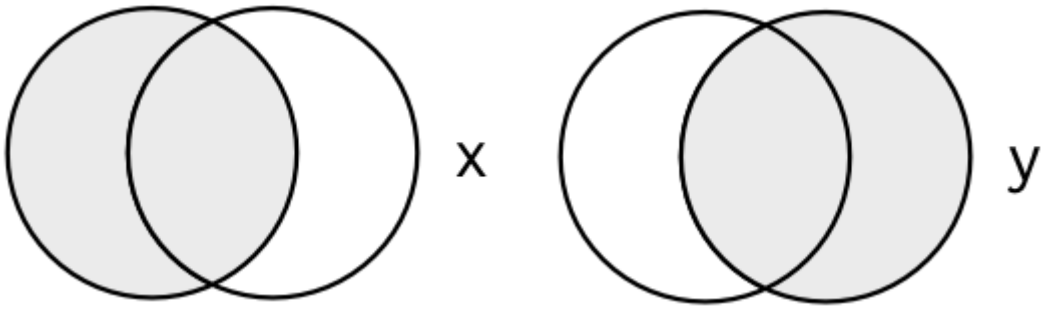
```
## [1] TRUE
```

3.2 Combining comparisons with logical operators

With **logical expressions** you can combine simple comparisons to form complex expressions with several criteria.

3.2.1 Logical *AND* (`&`)

You can think of two logical expressions \(x\) and \(y\) as two sets which we can combine with logical operators (also: *Boolean operators*).



Two sets: x, y.

Source: R for Data Science, G. Grolemund, H. Wickham

For example, one expression might be “age is less than 30” and another “country is GB” which yields the following logical vectors:

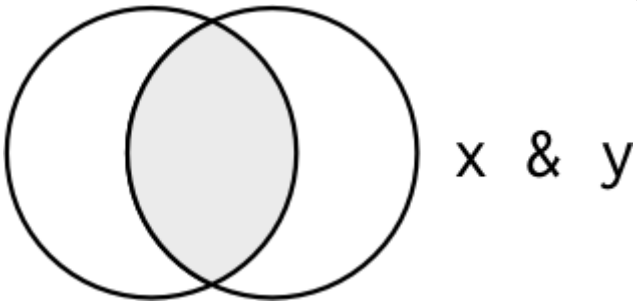
```
age < 30          # this is logical vector x
```

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

```
country == 'GB'   # this is logical vector y
```

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

If we want that both criteria apply, we need to combine them with a *logical AND*.



Logical expression x AND y.

Source: R for Data Science, G. Grolemund, H. Wickham

In R, a logical AND is expressed with `&`. This will produce `TRUE` only where **both** logical vectors are `TRUE`. When you look at the above output, where is this the case?

Only for the third elements both \(x\) and \(y\) are `TRUE`, hence only there the *AND*-combined expression yields `TRUE`:

```
age < 30 & country == 'GB'
```

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

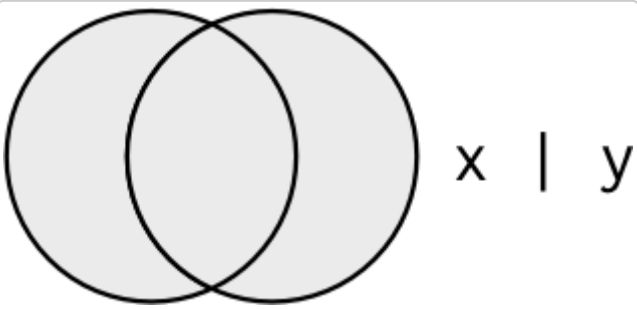
Quite often you will need to do comparisons with a **range** of values like \(age \in [20, 30)\). Unfortunately, you can't express this in R with `20 <= age < 30`. You need to construct an AND-combined logical expression:

```
age >= 20 & age < 30
```

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

3.2.2 Logical *OR* (|) / the `%in%` operator

Another important operator is *logical OR* which can be expressed with `|` in R (this is the "vertical bar" symbol which on Windows/Linux is usually located next to the shift key and can be typed via `ALT-GR + <`).



Logical x OR y.

Source: R for Data Science, G. Grolemund, H. Wickham

A logical OR yields `TRUE` whenever \(x\) or \(y\) or both are `TRUE`.

Let's consider both logical vectors again:

```
age < 30          # this is logical vector x
```

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

```
country == 'GB'   # this is logical vector y
```

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

Considering the definition of logical OR above, which elements will the expression \((age < 30) \text{ OR } (country = \text{GB})\) become `TRUE` ?

We see that only the fourth elements in both logical vectors are `FALSE` , all others have at least one `TRUE` on one side. This means that only the fourth element in the result vector is `FALSE` , all others `TRUE` :

```
age < 30 | country == 'GB'
```

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

But what does the above expression actually mean in natural language? If we consider the above data as a data set of participants and we wanted to select some participants by specifying some criteria, these criteria would state *“select those either younger than 30 or coming from GB (or both)”.* Only one of the five rows in the data set would be excluded: An individual from “DE” aged 51 (fourth element). For all others, *at least* one of the stated criteria holds true.

It is very common to check a single variable for multiple values by using `OR` expressions, e.g. to form a criterion like “individuals from GB *or* DE”. However, you cannot code this like you would write it in English. The following is wrong:

```
country == 'GB' | 'DE'
## Error: operations are possible only for numeric, logical or complex types
```

This is because R would try to evaluate the expression `'GB' | 'DE'` first like standing alone, which doesn’t make any sense. Instead, you must repeat the variable for a second comparison:

```
country == 'GB' | country == 'DE'
```

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

There is another operator which makes the above much shorter to write, especially if you have a longer list of values on the right side of each comparison: `x %in% y` . For each element in `x` , it checks whether it exists in a set `y` . So to check which elements of `country` occur in the set \(\{\text{GB}, \text{DE}\}\) we can write:

```
country %in% c('GB', 'DE')
```

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

3.2.3 Further notes on logical expressions

Of course you can always use the negation operator `!` in your logical expressions. Here we identify non-smokers younger than 30:

```
age < 30 & !smoker
```

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

You can also negate a whole complex expression, but don't forget to put parantheses around it:

```
!(age < 30 & !smoker)
```

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

This will negate *"non-smokers younger than 30"*, which evaluates to *"smoker **or** at least 30 years old"*.

Parentheses are important for more complex expressions as they control the order of evaluation. For example, you might want to find out all participants that are younger than 21 *OR* older than 35 *AND* smoke. If you simply write it like this, you might not get what you want:

```
age < 21 | age > 35 & smoker
```

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

You probably want that the age range is combined via *AND* with the `smoker` indicator which requires to wrap parantheses around the comparisons involving `age`:

```
(age < 21 | age > 35) & smoker
```

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

The takeaway here is that whenever you have more than two criteria in your logical expression, you should think about the order in which they should be evaluated and set parantheses accordingly.

Finally let's recall that you can use functions like `sum()` and `mean()` on any logical vector, so also on any output of a logical expression. For example, we can count the number of smokers aged between 18 and 25 using `sum()`:

```
sum((age >= 18 & age <= 25) & smoker)
```

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

With this, we can also count occurrences in character string or factor variables:

```
smoker <- c('yes', 'yes', 'no', 'yes', 'no')
# as shown, sum(smoker) will fail, but this works:
sum(smoker == 'yes')
```

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

4 Sequences

This will be a last short section about how to generate sequences of integers. They are mostly used for subsetting (next session) but they are also often used in comparisons.

Sequences of integers can be generated with the “colon” shortcut (`A:B`):

The following creates a sequence of integers from 1 to 10:

```
1:10
```

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

The following creates a sequence of integers from -1 to 1:

```
-1:1
```

```
## [1] -1  0  1
```

You can also generate a sequence from a bigger value to a smaller one:

```
3:-3
```

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

More advanced sequence generation can be achieved with the function `seq(...)` . It also supports a `by` argument to specify a step size. Here, for example, we generate numbers for 0 to 2 with a step size of 0.25:

```
seq(0, 2, by = 0.25)
```

```
## [1] 0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00
```

When you’re dealing with integer variables, you can use a sequence to specify a range of values for comparisons. The following example yields `TRUE` for all ages between 18 and 25:

```
age %in% 18:25
```

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

Be warned that you should *only* use this for integer variables! This is because only integers are part of the sequence, hence a floating point value like `20.5` is not part of the sequence:

```
20.5 %in% 18:25
```

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

5 Tasks

1. Consider the command `mean(head(sort(airquality$Ozone), 10))`. Describe in one sentence what the effect of this command is.
2. Use the same command as above, but reverse the ordering. How do you do that and what's the effect of it? HINT: Remember that `sort()` has an optional parameter `decreasing` to control the ordering direction.
3. Calculate the standard deviation of the variable `Solar.R` in the built-in data set `airquality`. Use the function `sd()` for this and set it to ignore `NA` values in the input vector.
4. `rnorm()` is a function to generate random numbers from a normal distribution. Find out which parameters it has using R's built-in help function. Now generate 100 random numbers from a normal distribution with mean 30 and standard deviation 2. Calculate the mean from these numbers. How much differs the mean of your generated numbers from the mean 30?
5. Use the 100 random numbers generated in the previous task and count how many of them are greater than or equal 30.
6. Load the package `MASS` and its data set `cats` as in the previous session. Answer the following questions (Hint: Create a logical expression and use `sum()` to count the occurrences of `TRUE` values, e.g. `sum(cats$Sex == 'F')`):
    1. How many female, how many male cats are there in the data set? Store the results in two objects `n_female` and `n_male`.
    2. How many female cats have a body weight of at least 2.5kg? What's the ratio of these in the group of all female cats?
    3. How many male cats have a body weight of at least 2.5kg? What's the ratio of these in the group of all male cats?
    4. How many female cats have a body weight of at least 2.5kg *or* a heart weight of 10g and more? What's the ratio of these in the group of all female cats?
    5. How many male cats have a body weight of at least 2.5kg *or* a heart weight of 10g and more? What's the ratio of these in the group of all male cats?
7. Complete **lesson 8 ("Logic")** of SWIRL Course **"R Programming"**. (See the notes in session 2 tasks about installing SWIRL if you have not done that yet.)
8. What's wrong with the following lines of code:

Example 1:

```
sum(airquality$Month = 5)
## Error
```

Example 2:

```
smoker <- c(TRUE, NA, FALSE, TRUE, FALSE)
sum(smoker, na.rm <- TRUE)
## Error
```

Example 3:

```
age <- c(20, NA, 19, 51, 20)
mean(age, na.rm == TRUE)
## Error
```

Example 4:

```
country <- factor(c('USA', 'GB', 'GB', 'DE', 'USA'))
country_is_usa <- (country = 'USA')
country_is_usa
```

```
## [1] "USA"
```

Example 5:

```
age <- c(20, NA, 19, 51, 20)
median(age, rm.na = TRUE)
```

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