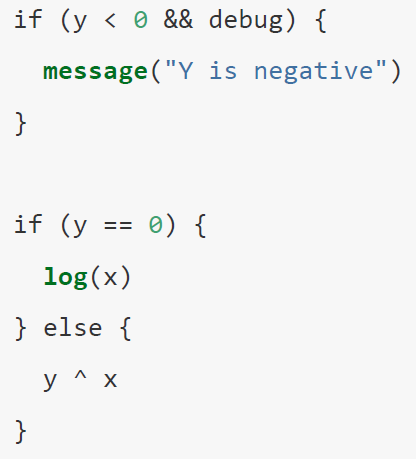
**Chapter 19 Functions:**

# Introduction to functions:

|  |  |
| --- | --- |
| **Functions** | * functions are part of **baseR** * a function is an **object you can call** * basically, it is a machine with internal logic that takes a group of inputs (parameters or arguments) and returns a value as output * In fact, in R environment, **everything we use is an object**, **everything we do is a function,** and, maybe to your surprise, **all functions are still objects** * although we might sometimes call it differently, any command that needs further processing from side of R is basically a function 🡪 even <- or + are functions, taking two arguments (old object, operation overwriting old object) and returning a value as output (new object) 🡪 although they are called binary operators, they are at their core functions * **when we do casual, interactive data analysis, at times, we won't have to write any function on our own since the built-in functions and those provided by thousands of packages are usually enough** |
| **When to use functions** | * however, if **one needs to repeat a logic or a process in data manipulation or analysis, those functions may not fully serve our purpose** bcs they are not designed to meet specific **needs of a task or format of specific dataset**   + **then one needs to create own functions targeting the specific set of demands** * functions allow you to **automate tasks and decrease duplications**   + **that is the main goal of functions** * i.e. if one needs to repeat one’s logic or a process in data manipulation or analysis, it might be necessary to create one’s own function * one rule of thumb w/ regards to functions: **you should consider writing a function whenever you’ve copied and pasted a block of code more than twice** * **advantages of functions:**    + as requirements change, code needs to be updated in only one place   + one elimantes the chance of making mistakes when copy and pasting (e.g. updating a variable in one place and not another) |
| **Syntax of a function** | * **functions have 3 basic components:**    + formal argument list   + a body   + and an environment   Input 🡪 ‚f(x)‘ – part and parameters – the placeholders  Basic syntax:  <name of function> <- function**(**<comma-separated list of arguments of the function>**)** **{**<function body**}**  code of the function   * Comma-separated list of arguments: * the arguments of the function should contain **anything that should be provided to the function manually, each time the latter is performed** * ***Argument types: an argument can be:***   1. a ***symbol (=name=variable)***       + ***i.e.* data**: **data arguments should always come first**      + ***or other parameters***      + **including parameters, allows one to variate the computation of our function not only in terms of the different data that is provided,  but also in terms of the parameters that refer to different functions used in our function**      + e.g.:      + when we use a log() function in our created function, we can include a base argument here, so that we later on can variate the base      + or when we use the mean() function in our created function, we can include an na.rm = argument, so that we can later on variate how NAs should be handled and thereby increase self-awareness of NAs in the data   2. or a ***default (i.e. ‘symbol = default’ construct): used to specify a default value for an argument***      + ***this value will be used if the function is called w/out overwriting the default***      + you specify a default value the same way when you call a default function **(e.g. na.rm = True)**      + **set default values with caution**: e.g. it makes sense **to default na.rm to False, so that missing values are not silently ignored**   3. or the ***special argument ‘…’; Source:*** <https://www.burns-stat.com/the-three-dots-construct-in-r/>      + ***three-dots allows:***      + *an arbitrary number and variety of (named and unnamed) arguments*        - many functions in R take an arbitrary number of inputs        - two prime cases are c() and list() functions        - both allow you to give them an arbitrary number and types of arguments      + *passing arguments on to other funtions*        - one example is the apply() function: function (X, MARGIN, FUN, ...)        - e.g.: apply(Mat, 2, mean, **na.rm=TRUE**)        - na.rm does not match any arguments of apply, ***so it is put into the three-dots***      + ***i.e.:*** when to use ellipsis construct:      + 1) when it makes sense to call a function with a variable number of arguments      + 2) when, within your function, you call other functions, and these functions can have a variable number of arguments      * + - here we only have to assign any kind of input variables for the function to compute the str\_c function     - using the ellipsis construct does come at a price: **any misspelled arguments will not raise an error**     - **note: when writing a function using the three-dots, one always has to pass it to some other function for it to be useful; e.g.:**      * + - here we want to allow the user to change the look of the plot * **note:** the type of inputs are not fixed prior to the calling: e.g. theoretically same function should work with dates as with scalars:  Basic example:        * Function body * the **function body contains a series of expressions**, considering the arguments provided before   + formally, the body is a parsed R statement   + it is ***usually a series of statements/expressions in braces*** (e.g. if-else statements)   + but ***it can be a single statement, a symbol or even a constant***   + in alignment with standard return rule, function returns the last value that it computed |
|  | * **a functions env:**    + **see environments in R\_Base\_BasicObjects Rmd for details**   + the combination of the code of the function and the bindings in its env, are called a ‘function closure’ (term from functional programming theory) |

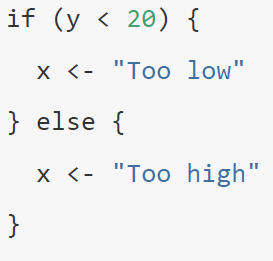
# Conventions

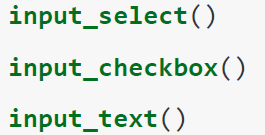
* **function and if statements should always be followed by {} and the contents should be inserted by two spaces**
  + **an opening bracket should never go on its own line**
  + **a closing bracket should always go on its own line, unless it is followed by else**
  + **always intend the code inside the curly brackets**
  + **note: the {} are not mandatory for the if statement**
    - it is ok to drop them, if one has a very short if-statement
    - otherwise {} provides a better overview

**w/out {}**



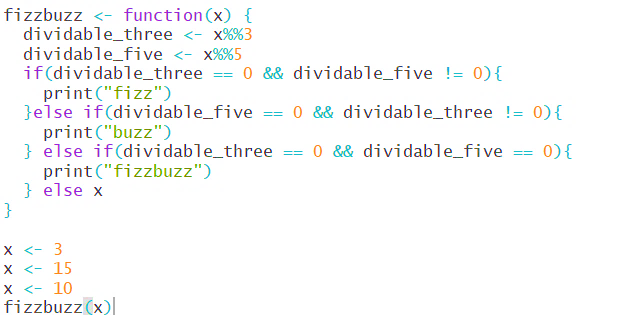
**w/ {}**

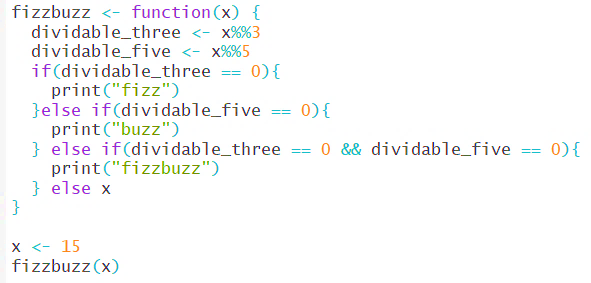


* **name the function appropriately:** so that you can already guess from the name, what it is that the function performs
  + generally, **function names should be verbs**; exceptions: function computes a very well known noun (e.g. mean, variance etc.)
    - common verbs to use: get, compute, calculate, determine (e.g. determine\_proportion)
  + as usual, separate words by lower case: collapse\_years
  + if one has a family of functions: use common **prefix to indicate that they are connected**
  + 
  + if possible, avoid overwriting already taken function names from base R
* **use comments (#) to explain the why(s) of the function**
  + explain why you chose this approach instead of another; what else did you try that didn’t work?
  + avoid commenting on the ‘what’ or the ‘how’: if you cannot understand clearly from the code what the function does, you probably have to rewrite it
  + e.g. add intermediate variables/functions with useful names

# Learnings function body

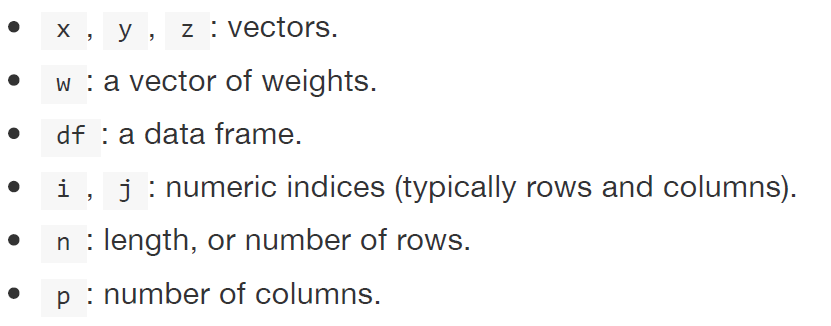
* es ist wichtig, dass die einzelnen if-statements disjunkt sind
  + ansonsten wird, R mehrere TRUE’s finden und dann einfach dasjenige if-statement anwenden, das als erstes TRUE ergibt
  + **z.B.:**

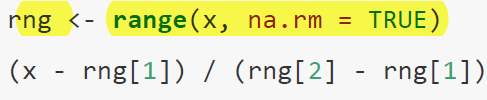
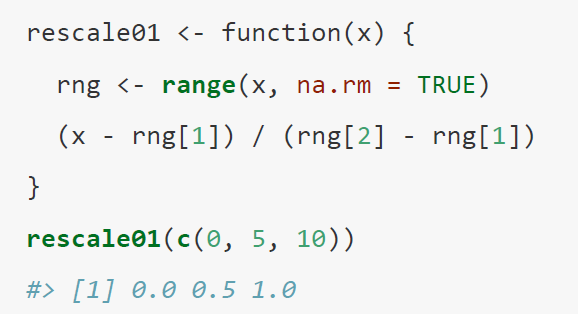


* + die Funktion ergibt „fizz“, wenn eine Zahl durch 3 teilbar ist; „buzz“, wenn sie durch 5 teilbar ist; „fizzbuzz“, wenn sie durch 3 und 5 teilbar ist und die Zahl selbst wenn keines der Bedingungen zutrifft
  + es ist maßgeblich, dass die ersten beiden if-statements jeweils noch das Argument enthalten, das NICHT eintreffen darf
    - denn wenn eine Zahl z.B. durch 3 und 5 teilbar ist, dann ist es jeweils eben auch durch 3 (d.h. erstes if-statement = T) und durch 5 (d.h. zweites if-statement = T) teilbar
    - R würde dann automatisch nur die Operation des ersten if-statements durchführen:
    - 
* if you are referring to a specific package in your function that you do not use on a daily basis, state it with <package>:: in your function

# Learnings function arguments

* **make use of assignment expressions**
* **1) input variables shall be named generally, e.g. x, y, z etc.**
  + this makes the input more clear and provides a better overview of how many input variables are considered in the function
  + also, changes do only have to be done at one point**\***
  + **commonly used names:**



* **2) pull out intermediate calculations into named variables as well:** 
  + e.g.: 
  + this provides a better overview of what the function actually performs
  + furthermore, it also decreases the # of changes**\*** that have to be made if function should get adjusted, and thus, decreases the risk of inconsistencies
  + with both, 1) and 2) the final function looks a lot more tidier:  
    
* **\*** this is part of the DRY principle (“do not repeat yourself”):
  + the more repetition you have in your code, the more places you need to remember to update when things change
  + this should be avoided

# Creating a function

|  |  |
| --- | --- |
| **Creating a function** | * the following steps are illustrated by the example of **creating a function to calculate the variance** (of course w/out using R’s provided function of var())  1. **create temporary (exemplary) input variables:**     * it is easier to make the function **after one has figured out how to make it work with a simple function**    * **it is harder to make a function and then try to make it work** 2. create temporary variables for intermediate calculations in the function body      1. create the function, **integrating the input variables and temp. var. of intermediate calculation**:      * + **note:**   + the last calculation must always be a real calculation and not an assignment; otherwise, the function does not have anything to perform  1. **test the function by supplying different input values afterwards**   input is provided using the following syntax:  > <name of function>(<values>)     * + eventually one wants to automatize those informal tests with different input   + that process is called unit testing: <http://r-pkgs.had.co.nz/tests.html>   note: **there will still be some kind of duplication, as one has to apply the function to multiple columns; this can be eliminated by *iterations (see Ch. 21)*** |
| **Set default values for a function** | * we can easily assign a value to one of the arguments and thereby establishing a default * **if the function gets called, we only have to specify the remaining arguments**:   default value    we only have to define x in this example   * note: a variable can also be defined as a vectors |
| **Calling a function** | * Once the function is defined, we can call the function just as we do in math   + we just need to provide e.g. values for the arguments: * **R process**:   + when we call the function, R will find out if there is a function named *add* in the environment   + Then, it will figure out that *add* refers to the function we just created and creates a local environment in which x takes 2 and y takes 3.   + The expression in the function body is then evaluated given the values of the arguments   + Finally, the function returns the value of that expression, 5 * note: as long as the function body, hence the mathematical expression, allows for it, the input can be anything :   + the type of inputs are not fixed prior to a function’s calling |
| **Generalizing a function** | * Developers often want a function to be general enough to adapt to a wide range of use cases   + thereby, we can use it for similar problems, w/out rewriting it completely * making a function more widely applicable is called ***generalization*** * we can generalize a function easily by using an **if-else if statement**:     sozusagen NB   1. **Define all functions arguments** needed for mathemacial expression + **a placeholder variable** (in this example *type*) 2. define the placeholder in the function body: if (*placeholder* == *specific value) {do specific calculation} else if*(*placeholder* == *specific value)(..)* 3. set {stop} argument  **🡪 it is important to end the loop!** 🡪 note: the text in columns is only for information purposes and does not serve any function in the if-else if statement   Again, once the function is set, we can call it providing respective function arguments:  Avoid default of R automatically proceeding by picking first matching argument of placeholder argument in if-else if statements:   * the last error message occurs because we have provided 2 entries for the “type” arguments, but only 1 value for the “type” argument in the if- else-if statements * in order to avoid that R automatically picks the first match it finds in the statement, as stated in the error message, we can restrict our if-else if statement further by **adding another if-stop statement:**        * command that only logical vectors of length one shall be taken into account 🡪 remember: with logical operators, comparisons yield TRUE or FALSE only * note: after a stop, we start w/ if statement again |

|  |  |
| --- | --- |
| **Conditions: if-statement** | Syntax: if(cond) **{**expr**}**   * cond: length 1 logical vector that is not NA   + Conditions of length greater than one are currently accepted with a warning, but only the first element is used   + **conditions always have to evaluate to a single TRUE or FALSE** * expr: expression in the formal sense (math.: Ausdruck)   Exp.:     * if-statement above, can be read as follows:   + if z is bigger 10, then return sequence of consecutive integers from 1 to 10   + else, return sequence from 1 to 9 |
| **if statement with logical operations**  **II, && any(), all()**  **identical(), dplyr::near()** | **use || (or) or && (and)**   * one can use **II** and **&&** to **combine multiple logical expressions** in the if-statement   + **if one wants to combine multiple conditions w/ logical operator, one CANNOT use the same expressions as in filter(), e.g. | or &**   + remember that these are vectorized logical operators hence they will return TRUE or FALSE for each vector element   + whereas II and && only assess the first element of an object     - as soon as || sees the first TRUE it returns TRUE, w/out computing anything else     - as soon as && sees the first FALSE it returns FALSE, w/out computing anything else   + if-statement will produce error, if it yields vector output; only the first element of that output-vector would be used   **testing for equality with ==:**  **== and any() or all()**   * be careful though: == is also vectorized   + we can use the **logical functions any() and all()** in conjunction with == to collapse logical vector into one Boolean variable   + > any():     - the value returned is TRUE **if at least one** of the values in x is TRUE,     - and FALSE if all of the values in x are FALSE   + > all():     - value returned is TRUE if **all of the values** in x are TRUE (including if there are no values),     - and FALSE if at least one of the values in x is FALSE   + note: any() and all() take na.rm = T argument   e.g.:  **any()**      🡪 returns “num>2!” if any number of the vector is > 2  **all()**    **== vs. identical()**   * >identical is not vectorized, i.e. it always returns either a single TRUE or FALSE   + identical tests whether **every** element of an object fulfills the condition       **== vs. dplyr::near()** |
| **multiple if- conditions**  **and switch() function** | * one can build a sequence of multiple if-else statements:   if … else if …. else if … (…) … else   * general syntax of the if-statement:      * the else-statements deal with the situations where the preceding expressions do not hold TRUE * **note: it is important to figure out the correct order for each condition and be careful about the dependencies between each branch:**   + **R tests for each condition in the order provided**   + **that means, if one condition should dominate/precede others, it has to be stated first**      * note: each computational command (x+y, or stop) has to be closed with }   **if-function and switch()-function:**   * if you end up with a very long chain of if-else statements, you should consider rewriting * one useful technique is the switch() function * switch () allows to evaluate an expression based on a position or name   switch(EXPR, ...)   |  |  | | --- | --- | | EXPR | an **expression evaluating to a number or a character string**. | | ... | the list of alternatives. If it is intended that EXPR has a character-string value these will be named, perhaps except for one alternative to be used as a ‘default’ value. |  * if EXPR is not a character, it is coerced to integer; then function **evaluates** the corresponding element in (…) based on position * if EXPR is a character, the function **evaluates** the corresponding element in (…) based on the given name; so, for example switch("cc", a = 1, cc =, cd =, d = 2) evaluates to 2   + note: switch always returns the first non-missing argument value for the first name it matches   + if there is more than one match the first matching element is returned   + if input is provided to switch function, that is not an argument in the switch function, output will be NULL:   e.g. switch function:    **if-function and cat function:** |
| using **vectorized ifelse or if\_else** | > ifelse(test, yes, no)   * test: **in contrast to if-statement, any object that can be coerced to a logical mode, i.e. could also be vector** 🡪 if we want to work w/ vectors >1, use ifelse instead of if * yes: values returned if test yields TRUE * no: values returned if test yield FALSE * **output: object with the same shape (e.g. length) as test**   + if yes or no are too short, their elements are recycled. yes will be evaluated if and only if *any* element of test is true, and analogously for no * Missing values in test give missing values in the result. * note: missing values in test, yield missing values in output   **Example:**    **all elements of the ifelse statement can also be vectors; yet those have to be of same length and type!**  **e.g.:**     * **ifelse is generally equivalent to if () {} else statement**   > if\_else(condition, true, false, missing = NULL)   * is more strict than base ifelse, as it checks if true and false are the same type |
| cut() function as alternative to long if statement | **another handy function to avoid a very long series of if-else if statements: cut()**   * advantage of cut() is that it works on vectors, whereas if only works on a single value * useful for when continuous variables shall be split into different intervals and divided accordingly   General syntax:  > cut(<**numeric vector**>, <breaks>, labels = <names for the diff. intervals>, right = TRUE)   * numeric vector gets **coerced to factors when applying cut() function** * breaks: arguments defines the intervals into which numeric vector shall be cut into   + can be numeric vector with respective cut points [Intervallsgrenzen]   + or can be a single number determining the number of intervals to create, w/out determining the cut point as such * right:   + if right = TRUE: Intervall ist zur rechten Seite „offen“, d.h. rechte Intervallgrenze zählt noch zum Intervall dazu 🡪 [x,y)   + if right = TRUE: Intervall ist zur linken Seite offen, d.h. linke Intervallgrenze zählt noch zum Intervall dazu 🡪 (x,y]      * e.g.:   + with right = T, 0 is labeled as freezing (Intervall [-Inf, 0))   with right = F, 0 is labeled as cold (Intervall (-Inf, 0] aber (0, 10]) |
| Make function more robust with **stop argument** | * it is good practice to check whether important preconditions with respect to the function’s arguments are met and output a stop-message if not * stop argument can be inserted as typical **if-statement**:      * alternatively, stopifnot can be used   > stopifnot(…)   * …: any number of logical expressions * if not all expressions are TRUE, stop is called, producing an error message providing the first element that is not TRUE     note, that in contrast to if-statement above, one asserts what should be true rather than checking for what might be wrong   * this function stops, if   + the na.rm argument provided is not logical and of length 1   + the two vectors provided are not of the same length |
| explicit **return statements** | * **the value returned by a function is usually the last statement it evaluates** * however, one can choose to return an output earlier by using >return () * > return should be used **only** in the following situation:   + we want to avoid that R runs a long and maybe complicated function,  whereas if certain conditions are simply not given, it could have stopped way earlier with a simpler solution     - example: **if inputs are empty, no computation possible/necessary:**   + you have an if-else statement with one complex and one simple block: |
| Writing **pipeable functions** | * **“pipeable function” means to create a function that returns a value which can be later used in the pipe**   + **thus, one has to consider the class of the returned value**   + **w/ dplyr and tidyr the object type should be a data frame** * there are 2 basic types of pipeable functions:   + **transformations (return(df)):** an **object** is **passed** **to the function’s first argument** and a **modified object is returned**   + **side-effects (invisible(df)):** passed object is not transformed, but **an action is performed on the object**, like drawing a plot or saving a file     - side-effect functions should **invisibly** return the first argument; then they are not printed, but can still be used in the pipeline   for example, the following function **prints the # of missing values** and **outputs the df,** but keeps it invisible with > invisible ()     * thereby, the whole function remains pipeable **as the final output remains a df, which, however, remains invisible due to invisible function**   e.g. we could use it in following pipe:     * here we purposely want to create NAs and use the function above (show\_missings) to output how many we have exactly created |