Advanced programming in R

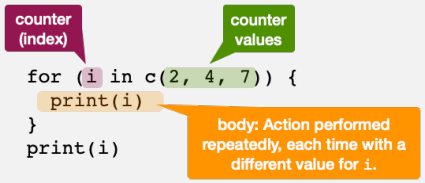
Iterations

for.. Loop

for loops execute the loop *body* for a prescribed number of times, as controlled by a counter or an index, incremented at each iteration cycle:

```
for (variable in sequence) {
    do something
}
```

Example



Outside the loop last state of `i` should be **7**.

Useful function to generate a sequence is: `seq_along(x)` → if *x* is NULL counter is NULL and no iteration is executed (in contrast to `1:length(x)`).

while.. Loop

while (or **repeat**) loops are based on the onset and verification of a logical condition. The condition is tested at the start (or the end) of the loop construct:

```
while (condition) {
    do something
}
```

Example

```
i <- 3
while (i < 10) {
     print(i)
     i <- i+ 1
}
```

Outside the loop last state of `i` should be **10**.

Conditions

if.. Statement

```
if (condition) {
  do something
}
```

Example

```
x <- sample(1:10,1)
if (x < 6) {
  print("true")
}
```

If `x` is greater or equal to 6, nothing happens with this statement.

if..else Statement

```
if (condition) {
  do this
} else {
  do that
}
```

Example

```
x <- sample(1:10,1)
if (x < 6) {
  print("true")
} else {
  x^2
}
```

ifelse.. Statement

```
ifelse(condition,
  do this, do that)
```

Example

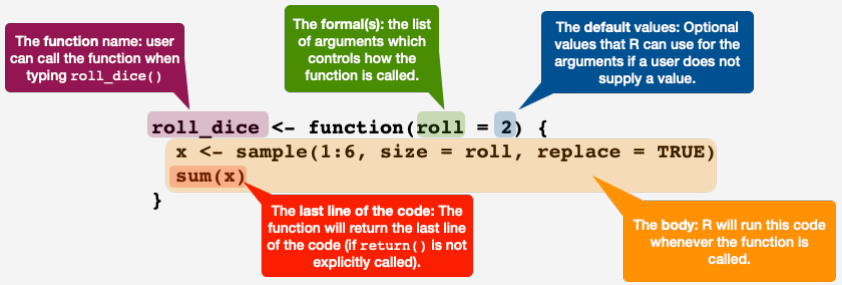
```
x <- sample(1:10,1)
ifelse(x < 6,
  TRUE, x^2)
```

Functions

Function components

User-defined functions have 3 components (if so-called *closures*):

1. its **formals** = the argument list
2. its **body** = the code inside the function
3. its **environment** = the 'map' of the location of the function's variable



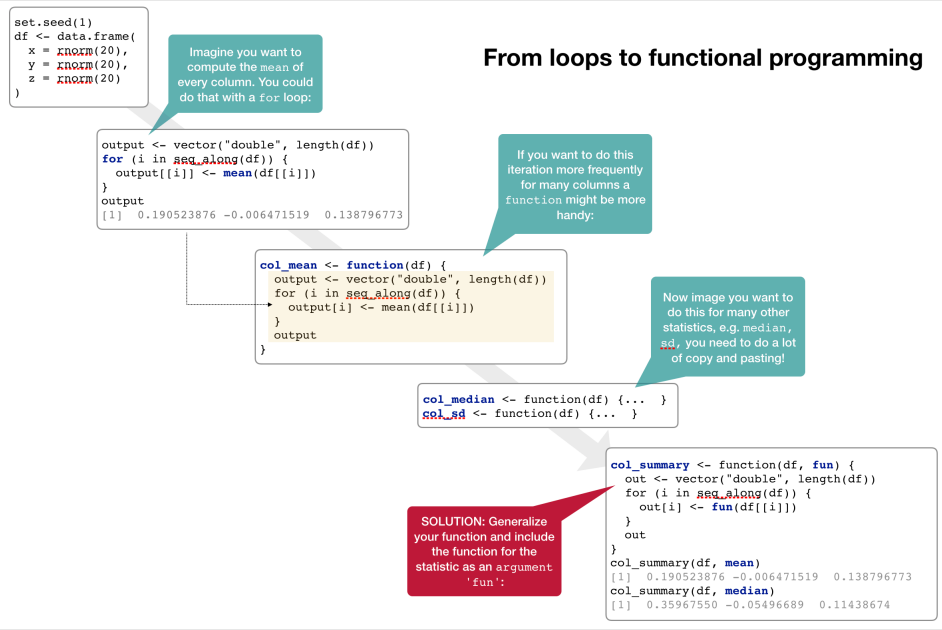
Function example

Example

```
sd_error <- function(x) {
  val <- var(x) / sqrt(length(x))
  return(val)
}
sd_error(1:10)
[1] 2.898755
```

Check

```
formals(sd_error)
$x
body(sd_error)
{
  val <- var(x)/sqrt(length(x))
  return(val)
}
environment(sd_error)
<environment: R_GlobalEnv>
```



Functional programming

Useful base functions

Function	Description	Example
`do.call(what, args)`	Execute a function call from a name or a function and a list of arguments to be passed to it.	Bind data frames in list by rows: `xlist <- split(iris,iris$Species)` `do.call(rbind,xlist))`
`with(data, expr)`	Evaluate an R expression in an environment constructed from data.	`with(iris, list(` `summary(aov(Sepal.Length~Species)),` `summary(aov(Sepal.Width~Species))))`

The *apply* family

Apply a function to margins of 2- or -3-dimensional objects

Function	Description	Example
`apply(X, MARGIN, FUN,..)`	*MARGIN* takes a vector of subscripts which the function will be applied over. If *X* is a data frame/matrix: 1 = applied over rows, 2 = applied over columns and c(1,2) = both. .. is a placeholder for further function argument. Returns a vector, array or list.	`apply(iris[ ,1:4], 2, mean, na.rm=TRUE)`

Apply a function to each element of a vector or list *X*:

Function	Description	Example
`lapply(X, FUN,..)`	Returns a list of the same length as *X*.	`lapply(as.list(iris),class)`
`sapply(X, FUN,.., simplify)`	Wrapper for `lapply()` but returns a vector (if `simplify=TRUE`) → problematic if function returns different number of values!	`sapply(as.list(iris),class)`
`vapply(X, FUN, FUN.VALUE)`	Similar to `sapply()` but output format has to be specified (with `FUN.VALUE`).	`vapply(as.list(iris),class, FUN.VALUE = character(1))`
`tapply(X,INDEX, FUN,..)`	Apply function to a group of values in *X* given by a unique combination of a list of one or more factors specified in *INDEX*.	`tapply(iris$Sepal.Length, iris$Species, mean)`
`replicate(n, expr)`	Wrapper for `sapply()` for the repeated evaluation of an expression	`hist(replicate(20, mean(rnorm(5))))`
`mapply(FUN,..., MoreArgs)`	A multivariate version of `sapply()`: Apply function to first elements of each `...` argument, then to second elements, etc.	`mapply(rep, 1:4, 4:1)`

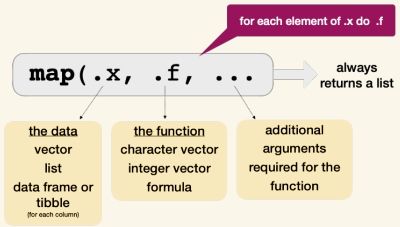
A tidyverse toolkit: purrr

`load(purrr)` or `load(tidyverse)`

Apply functions with the *map* family of functions

Map functions operate similar to lapply & co but can be faster (written in C++) and are more consistent and easier to learn.

The most basic function



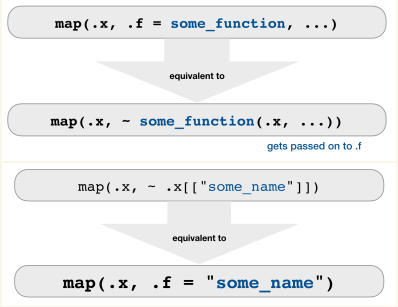
Other output returned than list

`map_lgl()` → logical vector
`map_int()` → integer vector
`map_dbl()` → double vector
`map_chr()` → character vector
`map_dfc()` → data frame (column-bind)
`map_dfr()` → data frame (row-bind)
`walk()` → triggers side effects, returns the input invisibly

Examples

```
# Different specifications of .f
map(.x=1:10, .f=rnorm, n=10)
map(.x=1:10, ~rnorm(10, .x))
# Anonymous function
map(1:10,.f=function(x) rnorm(10,x))
# .x here a data frame
map(iris[ ,1:4], mean, na.rm=TRUE)
Combine 2 map functions
map(1:10, rnorm, n=10) |>
  map_dbl( ~mean)
```

Ways of specifying *.f*



Apply function conditionally to elements

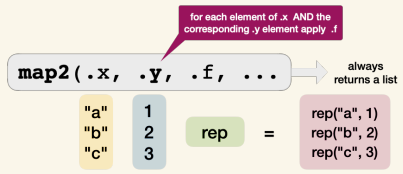
`map_if(.x.,p,.f, ..,.else)` Takes a predicate function *.p* as input to determine which elements of *.x* are transformed with *.f*. Example: `map_if(iris,is.factor,as.character,.else=as.integer)`

`map_at(.x,.at,.f,..)` Takes a vector of names or positions *.at* to specify which elements of *.x* are transformed with *.f*. Example: `iris |> map_at(c(4, 5), is.numeric)`

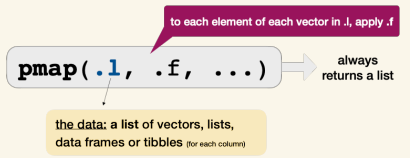
`map_depth(.x,.depth, .f,..)` Apply *.f* to a specific depth level of a nested vector.

Mapping over 2 or more arguments: *map2()* and *pmap()*

2 arguments



>2 arguments



- To get a different output than a list use e.g. `map2_lgl()`,`map2_dbl()`, `map2_chr()`,`walk2()`, or `pmap_lgl()`,`pmap_chr()` etc.
- Other mapping functions
  - `lmap()`,`imap()`, `invoke_map()`,

Some useful functions for working with lists

The following examples are applied to this dummy list:

x <- list(a=list("x", 1:5), b=NULL, c=NA, d=list(), e=list(NULL, 10:1))

Function	Description	Example
Filter list		
`keep(.x,.p,..)`	Select top-level elements that pass a logical test.	`keep(x, ~length(.x) >1)`
`discard(.x,.p,..)`	Select top-level elements that do NOT pass a logical test.	`discard(x, ~length(.x) >1)`
`compact(.x)`	Drop elements top-level that are NULL or have length zero	`compact(x)`
Summarise list		
`every(.x,.p,..)`	Do all element pass a test?	`every(x, is.list)`
`some(.x,.p,..)`	Do some elements pass a test?	`some(x, is.list)`
`has_element(.x,.y)`	Does a list contain an element?	`has_element(x, list())`
Reshape list		
`flatten(.x)`	Remove a level of indexes from a list. Similar to `unlist()` but here only ONE level of hierarchy removed. Returns a list, otherwise use `flatten_dbl()`, `flatten_dfc()`, etc.	`flatten(x)`
`transpose(.l)`	Turns a list-of-lists "inside-out", e.g. turns a pair of lists into a list of pairs.	`keep(x, ~length(.x)>1) |> transpose()`
Work with lists		
`set_names(x, nm)`	Set the names of a vector or list directly or with a function (imported from *rlang* package).	`set_names(x, toupper)`

For more functions see documentation: purrr.tidyverse.org.

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Debugging and condition handling

Communicating conditions to users

`stop()`	Raise **fatal error** and force all execution to terminate (use when there is no way for a function to continue).
`warning()`	Display **potential problems** (use when some elements of a vectorized input are invalid).
`message()`	Give **informative output** that can easily be suppressed by the user (e.g. the steps of a program execution or which values were chosen for missing arguments).

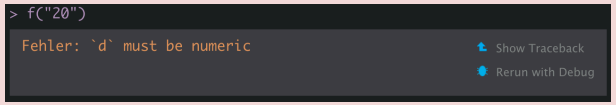
Handling of errors, warnings and messages

`try(expr, silent)`	Continue execution even when an error occurs. Useful if e.g. many models are fitted and some fail convergence. *expr* can be one or several function calls; if expression includes multiple lines wrap code with curly brackets: `try({expr})` Suppress the message with *silent=TRUE*
`tryCatch(expr,..)`	Can deal with all conditions and lets you specify handler functions (= named functions that are called with the condition as input) that control what happens when a condition is signaled. Example: `tryCatch(sqrt("a"), error=function(e) print("You can't take the square root of a character."))`
`?SuppressMessages(expr)`	Suppress messages returned by a function. Example: `suppressMessages(library(tidyverse))`
`?suppressWarnings(expr)`	Suppress warnings returned by a function.

See also `?condition` for more options.

Debugging methods

RStudio offers various tools for debugging, the default being the *Error handler* setting, which can be changed under *Debug > On Error*. If an error occurs you will see these two options on the right side in the console:



Locate errors

If an error occurs click on the icon **Show Traceback** in the console. This shows the sequence of calls (known as *call stack*) that lead to the error. Useful when an error occurs with an unidentifiable error message. If you're not using RStudio, you can use `traceback()` to get the same information. To permanently set this option when an error occurs type: `options(error = traceback)`

Interactive debugger

Sometimes `traceback()` is not sufficient and the debug mode has to be started:

Browsing on error

RStudio's **Rerun with Debug** tool opens an interactive debug session in which the command that created the error is rerun, pausing execution where the error occurred (you will see `Browse[1]>` in the console).
You will see:

- a tab in the Editor with the corresponding code
- objects in the current environment in the Environment pane
- the call stack in the Traceback pane

In this mode you can run regular R code as well as a few special commands:

- `n`: execute next step in function
- `s`: step into (new function)
- `f`: finish execution of loop or function
- `c`: leave interactive debugging and continue regular execution
- `Q`: stops debugging, terminates function
- `Enter` . repeats previous command, to turn off `options(browserNLdisabled=TRUE)`
- `where`: prints stack trace of active calls

Browsing arbitrary code

browser()
This function allows to manually switch into the interactive debug mode. Insert `browser()` where you want to pause and re-run the function call (remove later if solved!):

```
foo <- function(x) {
   browser()
   log(x)
}
foo("5")
```

RStudio's **breakpoints** tool
Breakpoints behave similar to `browser()` and are set by clicking to the left of the line number in the code.

recover()
Acitvates `browser()` internally but allows to enter environment of any of the calls in the call stack. `recover()` cannot be called directly, instead set the following options once: `options(error = recover)`
To return to default handling: `options(error = NULL)`

debug()
This function inserts automatically the `browser()` statement in the FIRST line of the specified function. `undebug()` removes it. Alternatively, use `debugonce()`.

For more information see the Advanced R book by Hadley Wickham.