

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

```
counter (index)  counter values  
for (i in c(2, 4, 7)) {  
  print(i)  
}  
print(i)
```

body: Action performed repeatedly, each time with a different value for *i*.

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

Useful function to generate a sequence is: `seq_along(x)` → if *x* is NULL, it returns `integer(0)` and no iteration is executed (in contrast to `1:length(x)`, which iterates over 1 and 0).

while.. Loop

while loops are based on a logical condition that is tested at the start of each iteration. (**repeat** loops have no condition and run until `break` is called.)

```
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. The condition must have length 1 (otherwise an error since R 4.2.0); for vectors use `ifelse()`.

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, 5)  
ifelse(x < 6,  
       "low", "high")
```

Vectorised version: returns a vector of the same length as the condition.

Functions

Function components

User-defined functions have 3 components (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 variables

```
The function name: user can call the function when typing roll_dice()  
The formal(s): the list of arguments which controls how the function is called.  
The default values: Optional values that R can use for the arguments if a user does not supply a value.  
roll_dice <- function(roll = 2) {  
  x <- sample(1:6, size = roll, replace = TRUE)  
  sum(x)  
}  
The last line of the code: The function will return the last line of the code (if return() is not explicitly called).  
The body: R will run this code whenever the function is called.
```

Function example

Example

```
std_error <- function(x) {
  val <- sd(x) / sqrt(length(x))
  return(val)
}
std_error(1:10)
[1] 0.9574271
```

Check

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

From loops to functional programming

```
set.seed(1)
df <- data.frame(
  x = rnorm(20),
  y = rnorm(20),
  z = rnorm(20)
)
```

Imagine you want to compute the mean of every column. You could do that with a for loop:

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
output
[1] 0.190523876 -0.006471519 0.138796773
```

If you want to do this iteration more frequently for many columns a function might be more handy:

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

Now imagine you want to do this for many other statistics, e.g. median, sd, you need to do a lot of copy and pasting!

```
col_median <- function(df) {... }
col_sd <- function(df) {... }
```

SOLUTION: Generalize your function and include the function for the statistic as an argument 'fun':

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, mean)
[1] 0.190523876 -0.006471519 0.138796773
col_summary(df, median)
[1] 0.35967550 -0.05496689 0.11438674
```

Functional programming

Useful base functions

Function	Description	Example
<code>do.call(what, args)</code>	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: <code>xlist <- split(iris, iris\$Species)</code> <code>do.call(rbind, xlist)</code>
<code>with(data, expr)</code>	Evaluate an R expression in an environment constructed from data.	<code>with(iris, list(summary(aov(Sepal.Length~Species)), summary(aov(Sepal.Width~Species))))</code>

The *apply* family

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

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

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

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

A tidyverse toolkit: purrr

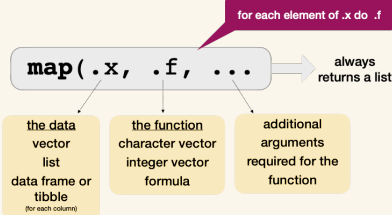


`library(purrr)` or `library(tidyverse)`

Apply functions with the *map* family of functions

Map functions operate similar to `lapply` & `co` but are more consistent and easier to learn.

The most basic function



Other output returned than list

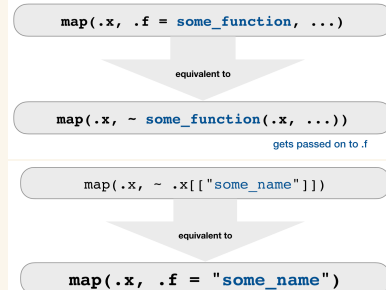
`map_lgl()` → logical vector
`map_int()` → integer vector
`map_dbl()` → double vector
`map_chr()` → character vector
`walk()` → triggers side effects, returns the input invisibly

Data frames: pipe the result of `map()` into `list_rbind()` (row-bind) or `list_cbind()` (column-bind).

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))
map(1:10, \(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

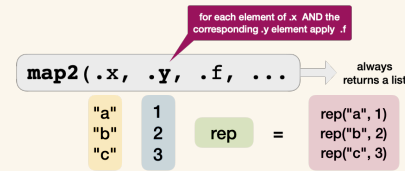
`map_if(.x, .p, .f, ..., .else)` Takes a predicate function *p* as input to determine which elements of *x* are transformed with *f*. Example: `map_if(iris, is.factor, as.character, .else=as.integer)`

`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), as.character)`

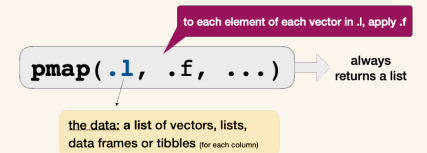
`map_depth(.x, .depth, .f, ...)` Apply *f* to a specific depth level of a nested vector.

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()`

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

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

Debugging and condition handling

Communicating conditions to users

<code>stop()</code>	Raise fatal error and force all execution to terminate (use when there is no way for a function to continue).
<code>warning()</code>	Display potential problems (use when some elements of a vectorized input are invalid).
<code>message()</code>	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

<code>try(expr, silent)</code>	Continue execution even when an error occurs. Useful if e.g. many models are fitted and some fail convergence. <i>expr</i> can be one or several function calls; if expression includes multiple lines wrap code with curly brackets: <code>try({expr})</code> . Suppress the message with <i>silent=TRUE</i> .
<code>tryCatch(expr, ...)</code>	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: <code>tryCatch(sqrt("a"), error=function(e) print("You can't take the square root of a character."))</code>
<code>suppressMessages(expr)</code>	Suppress messages returned by a function. Example: <code>suppressMessages(library(tidyverse))</code>
<code>suppressWarnings(expr)</code>	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:

```
> f("20")
Fehler: 'd' must be numeric
Show Traceback
Rerun with Debug
```

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:

Next | Previous | Continue | Stop

- **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()

Activates `browser()` internally but allows to enter environment of any of the calls in the call stack. It is typically activated by setting the following option 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 chapters [Conditions](#) and [Debugging](#) of the *Advanced R* book by Hadley Wickham.

x