Intro to Functions

Programming Structures

Gaston Sanchez

CC BY-NC-SA 4.0

STAT 33B, Fall 2025

About

In this part, we describe:

- the syntax for creating functions in R,
- the parts of an R function,
- various aspects about the arguments of functions

Functions

Writing Functions

- You can create functions with the function `function()`
- The arguments (inputs) go inside parenthesis, separated by commas.
- The code of the function is surrounded by braces (i.e. an *R expression*).

```r
# example
square <- function(x) {
  x^2
}
```

Writing Functions

Curly braces are optional if the body is a single expression.

```
square <- function(x) x^2
```

If the body of a function is a compound expression we have to use braces:

```
sum_sqr <- function(x, y) {
  xy_sum <- x + y
  xy_ssqr <- (xy_sum)^2
  list(sum = xy_sum,
       sumsqr = xy_ssqr)
}

sum_sqr(3, 5)
```

```
$sum
[1] 8
```

Writing Functions

Once defined, functions can be used in other function definitions:

```r
sum_of_squares <- function(x) {
  sum(square(x))
}

sum_of_squares(1:5)
```

```
[1] 55
```

Nested Functions

We can also define a function inside another function:

```r
sum_of_squares <- function(x) {
  square <- function(x) {
    x^2
  }
  sum(square(x))
}

sum_of_squares(1:5)
```

```
[1] 55
```

Anatomy of a function

Functions with a body consisting of a simple expression can be written with no braces (i.e. in one single line):

```r
square <- function(x) x^2

square(5)
```

```
[1] 25
```

Anatomy of a function

Conceptual structure of a function

```
function_name <- function(arg1, arg2, etc) {
  expression_1
  expression_2
  ...
  expression_n
}
```

Anatomy of a function

- Generally we will assign the function to a name
- A function takes one or more inputs (or none), known as *arguments*
- The expressions forming the operations comprise the **body** of the function
- A function made of a simple expression doesn't require braces
- Functions return a single value

Anatomy of a function

```r
square <- function(x) {
  x^2
}
```

- the function's name is `"square"`
- it has one argument `x`
- the function's body consists of one simple expression
- it returns the value `x^2`

Function Names

Different ways to name functions

- `square()`
- `squ_are()`
- `s.quare()`
- `Square()`
- `.square()`: a function that starts with a dot is a valid name, but the function will be a *hidden* function.

Function names

Invalid names

- `5square()`: cannot begin with a number
- `_square()`: cannot begin with an underscore
- `squ-are()`: cannot use hyphenated names

Output of a Function

Function Output

The value (i.e. output) of a function can be established in two ways:

- As the last evaluated simple expression (in the body)
- An explicitly **returned** value via `return()`

Recall that:

- The body of a function is an expression
- Remember that every expression has a value
- Hence every function has a value

Function Output

Every function has a value (i.e. output)

```r
cm2in <- function(x) {
  x * 0.3937    # processing and output
}
```

Recall that every expression has a value: the value of the last statement that is evaluated; in this case is `x * 0.3937`

Function Output

Many useRs prefer to explicitly use a `return()` statement

```
cm2in <- function(x) {
  y <- x * 0.3937   # processing
  return(y)         # output
}
```

Function Output

Many amateur useRs like to use a `print()` statement (but you should avoid it)

```
cm2in <- function(x) {
  y <- x * 0.3937    # processing
  print(y)           # output
}
```

Note: Using `print()` to specify the output of a function could work in most cases. But it largely depends on the object that is printed. Recall that `print()` is not a single function but a method—there are multiple flavors of `print()`. So to play safe, it's better to use `return()` than `print()`

`return()` versus `print()`

- The function `print()` is a **generic** method in R.
- This means that `print()` has a different behavior depending on its input
- Unless you want to print intermediate results while the function is being executed, there is no need to return the output via `print()`

Function Output

Keep in mind that depending on what's returned or what's the last evaluated expression, just calling a function might not print anything:

```r
cm2in <- function(x) {
  y <- x * 0.3937   # processing
}

cm2in(5)
```

Note: the code of the function works, and the function has a value. But in this case the value is an assignment command, NOT an (implicit) printing command.

Function Output

Here we call the function and assign it to an object.

```
cm2in <- function(x) {
  y <- x * 0.3937   # processing
}

z = cm2in(5)
z
```

```
[1] 1.9685
```

The last evaluated expression of the function has the same value as in the preceding slide. However, the very last command is a printing command which allows you to see the output.

The `return()` command

`return()` can be useful when the output may be obtained in the middle of the function's body

```r
plus_minus <- function(x, y, add = TRUE) {
  if (add) {
    return(x + y)
  } else {
    return(x - y)
  }
}

plus_minus(2, 3, add = TRUE)
plus_minus(2, 3, add = FALSE)
```

Return statement

Likewise, to exit the function and return a result early, use `return()`:

```r
square <- function(x) {
  if (!is.numeric(x)) {
    return(NA)
  }
  x^2
}
square(6)
```

```
[1] 36
```

```r
square("hi")
```

```
[1] NA
```

Function Arguments

Function Arguments

Functions can have any number of arguments (even zero arguments)

```r
# function with 2 arguments
add <- function(x, y) x + y

# function with no arguments
hi <- function() print("Hi there!")

hi()
```

```
[1] "Hi there!"
```

Arguments

Arguments can have default values (highly recommended!)

```
hey <- function(x = "") {
  cat("Hey", x, "\nHow is it going?")
}

hey()
```

```
Hey
How is it going?
```

```
hey("Gaston")
```

```
Hey Gaston
How is it going?
```

Arguments with no default values

If you specify an argument with no default value, you must give it a value everytime you call the function, otherwise you'll get an error:

```r
cm2in <- function(x) {
  x * 0.3937
}

cm2in()
```

```
Error in cm2in(): argument "x" is missing, with no default
```

Arguments with no default values

Sometimes we don't want to give default values, but we also don't want to cause an error. We can use `missing()` to see if an argument is missing:

```r
abc <- function(a, b, c = 3) {
  if (missing(b)) {
    return((a * 2) + c)
  } else {
    return((a * b) + c)
  }
}

abc(1)
```

```
[1] 5
```

```r
abc(1, 4)
```

```
[1] 7
```

Arguments with no default values

You can also set an argument value to NULL if you don't want to specify a default value:

```r
abc <- function(a, b, c = 3, d = NULL) {
  if (is.null(d)) {
    return((a * b) + c)
  } else {
    return((a * b) + (c * d))
  }
}

abc(1, 2)
```

```
[1] 5
```

```r
abc(1, 2, 3, 4)
```

```
[1] 14
```

More about function arguments

Arguments of functions can be:

- positional

```r
# x and y are positional arguments
plus <- function(x, y) x + y
```

- named

```r
# x and y are named arguments
plus <- function(x = 1, y = 1) x + y
```

Argument Matching

```r
# normal distribution
normal_distrib <- function(x, mu = 0, sigma = 1) {
  constant <- 1 / (sigma * sqrt(2*pi))
  constant * exp(-((x - mu)^2) / (2 * sigma^2))
}

normal_distrib(2)
normal_distrib(2, sigma = 3, mu = 1)
normal_distrib(mu = 1, sigma = 3, 2)
normal_distrib(mu = 1, 2, sigma = 3)
```

Argument Matching

R is "smart" enough in doing pattern matching with arguments' names (not recommended though)

```r
normal_distrib(2)
```

```
[1] 0.05399097
```

```r
normal_distrib(2, m = 0, s = 1)
```

```
[1] 0.05399097
```

```r
normal_distrib(2, sig = 1, m = 0)
```

```
[1] 0.05399097
```

Lazy Evaluation

In R, function arguments are **lazily evaluated**: they're only evaluated if needed.

For example, this code doesn't cause any problems because x is never used:

```
toss <- function(x) {
  sample(c("heads", "tails"), size = 1)
}

toss()
```

The dots parameter

The **dots** parameter ... accepts any number of arguments, and it is often used to forward arguments to another function.

For example:

```r
# Mean function with tolerance:
mean_tol <_ function(x, tol, ...) {
  mean(x[x > tol], ...)
}

mean_tol(c(1, 3, 5, 0.01, 0.2, NA), 0.5)
mean_tol(c(1, 3, 5, 0.01, 0.2, NA), 0.5, na.rm = TRUE)
mean(c(1, 3, 5, 10))
```

The dots parameter

You can access elements of `...` with the `...elt()` function:

```
hey <- function(x, ...) {
  ...elt(2)
  x + ...elt(1)
}

hey(3, 5, message("hi"))
```

```
hi
```

```
[1] 8
```

This only evaluates the argument you accessed.

The dots parameter

You can convert `...` to a list with the `list()` function:

```r
hey <- function(...) list(...)

hey(hi = 1, 3, 4)
```

```
$hi
[1] 1

[[2]]
[1] 3

[[3]]
[1] 4
```

This evaluates all of the arguments.

Conditions

Conditions

There are three main functions for generating warnings and errors:

- **`message()`**: to print an informative message
- **`warning()`**: to raise a warning message (without stopping execution)
- **`stop()`**: to stop execution raising an error

Stop Execution

Use stop() to stop execution of a function (raising an error)

```r
meansd <- function(x, na.rm = FALSE) {
  if (!is.numeric(x)) {
    stop("input must be numeric")
  }
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

Stop Execution

```r
# ok
meansd(c(4, 5, 3, 1, 2))
```

```
    mean       sd
3.000000 1.581139
```

```r
# this causes an error
meansd(c('a', 'b', 'c'))
```

```
Error in meansd(c("a", "b", "c")): input must be numeric
```

Warning Messages

A warning() is useful when we don't want to stop the execution, but we still want to show potential problems

```r
meansd <- function(x, na.rm = FALSE) {
  if (!is.numeric(x)) {
    warning("non-numeric input coerced to numeric")
    x <- as.numeric(x)
  }
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

Warning Message

```r
# ok
meansd(c(4, 5, 3, 1, 2))
```

```
    mean       sd
3.000000 1.581139
```

```r
# this causes a warning
meansd(c(TRUE, FALSE, TRUE, FALSE))
```

```
Warning in meansd(c(TRUE, FALSE, TRUE, FALSE)):
non-numeric input coerced to numeric
```

```
     mean        sd
0.5000000 0.5773503
```

Generic Messages

Use `message()` to display a generic message that is not an error or warning.

```r
meansd <- function(x, na.rm = FALSE) {
  if (!is.numeric(x)) {
    message("non-numeric input detected")
    return(NA)
  }
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

Generic Message

```r
# no message
meansd(c(4, 5, 3, 1, 2))
```

```
    mean       sd
3.000000 1.581139
```

```r
# message
meansd(c(TRUE, FALSE, TRUE, FALSE))
```

```
non-numeric input detected
```

```
[1] NA
```

Documenting Functions

Documenting Functions

Documenting a function involves adding descriptions for what the purpose of a function is, the inputs it accepts, and the output it produces.

- Description: what the function does
- Input(s): what are the inputs or arguments
- Output: what is the output (returned value)

Documenting Functions

There are several approaches for writing documentation of a function.

I will show you various examples for documenting a function.

In particular, I will show you how to use what are called **roxygen comments** to achieve this task.

While not used by most useRs, roxygen comments are great when you want to take your code and make a package out of it.

Documenting Functions

One option to document a function is by simply adding a short description of what the arguments should be like. In this case, the description is outside the function

```r
# function for adding two numbers
# x: number
# y: number
add <- function(x, y) {
  x + y
}
```

Documenting Functions

In this case, the description is between `<-` and `function()`

```
add <-
  # function for adding two numbers
  # x: number
  # y: number
  function(x, y) {
  x + y
}
```

Documenting Functions

In this case, the description is inside the function

```r
add <- function(x, y) {
  # function for adding two numbers
  # x: number
  # y: number
  x + y
}
```

Documenting Functions

In this case, the description is inside the function

```r
# description of arguments
compound_interest <- function(principal = 1, rate = 0.01,
                              periods = 1, time = 1)
{
  # principal = Principal Amount
  # rate = Annual Nominal Interest Rate as a decimal
  # time = Time Involved in years
  # periods = number of compounding periods per unit time
  principal * (1 + rate/periods)^(time * periods)
}
```

Roxygen Comments

One interesting option to document functions is by using **roxygen comments**

```r
#' @title Standardize
#' @description Transforms values in standard units
#' @param x numeric vector
#' @param na.rm whether to remove missing values
#' @return standardized values
#' @examples
#'   standardize(runif(10))
standardize <- function(x, na.rm = FALSE) {
  z <- (x - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm)
  return(z)
}
```

Roxygen Comments

Roxygen comments are R comments formed by the hash symbol immediately followed by an apostrophe: `#'`

You specify the label of a field with `@` and a keyword: e.g. `@title`

The syntax highlighting of RStudio recognizes this type of comments and labels

Typical roxygen fields

label	meaning	description
`@title`	title	name of your function
`@description`	description	what the function does
`@param`	parameter	describe `input` parameter
`@return`	output	what is the returned value
`@example`	example	one or more usage examples

Time permitting, at the end of the semester we'll see how using Roxygen comments simplifies the creation of an R package.

General Recommendations

On Writing Functions

Before you write a function:

- Write down the goal. What should the function do?
  - Draw a picture if it helps clarify the goal
- Check whether the function already exists.
  - Check base R, packages, code you've written, or google it
- Write down the inputs and outputs
- Write code to handle a simple case

On Writing Functions

- Baby steps: always start simple with toy-values or small data sets
- Work first on what will be the body of the function
- Check out each step of the way (don't worry yet about efficiency or elegance or cleverness)
- Don't try to do much at once
- Create the function (i.e. encapsulate the body) once everything works
- Don't write long functions: write short / small functions (preferably less than 10 lines of code)
- Personally, I think more than 20 lines of code are too many.

On Writing Functions

- Include documentation; we suggest using Roxygen comments.
- Optional: after you have a function that works, then you may worry about “elegance”, “efficiency”, “cleverness”, etc.
- Often, it’s better to have an “ugly/inefficient” function that does the work, rather than wasting a lot of time, effort, and energy to get a “smart” function.
- The more you practice, the easier will be to create functions.
- As you get more experience, making more clever and elegant functions will be less difficult, and worth your time.