Intro to Functions

R Programming Structures

Gaston Sanchez

R Coding Compendium

Donate

About

R comes with many functions and packages that let us perform a wide variety of tasks.

Sometimes, however, there's no function to do what we want to achieve. In these cases we need to create our own functions.

The content in this slides assume that you are familiar with the notion of **compound expressions** in R.

Function Basics

Centimeters to inches

R does not have a function to convert from centimeters to inches. But if you know the conversion factor (1 cm = 0.3937 in), you could do something like this:

```r
# cm to inches
cms <- 10
inches <- cms * 0.3937
inches
```

```
## [1] 3.937
```

Centimeters to inches

I don't know about you, but I can never remember the conversion factor. Wouldn't be nice to have a function `cm2in()` for converting centimeters to inches?

```r
# cm to inches
x <- 10
y <- cm2in(x)
y
```

```
## [1] 3.937
```

Centimeters to inches

Think of these lines of code as:

```
# cm to inches
x <- 10                 # input

y <- x * 0.3937         # processing
y                       # output
```

Centimeters to inches

Building a function—conceptually speaking—step by step:

Step 1: Wrap the **body** of the function within an R expression

```
# cm to inches
x <- 10              # input
{
  y <- x * 0.3937    # processing
  y                  # output
}
```

Centimeters to inches

Building a function—conceptually speaking—step by step:

Step 2: Specify inputs inside `function()`

```
# cm to inches
x <- 10                # input
function(x) {
  y <- x * 0.3937      # processing
  y                    # output
}
```

Centimeters to inches

Building a function—conceptually speaking—step by step:

Step 3: Assign the code of the function to a name

```
# cm to inches
x <- 10               # input
cm2in <- function(x) {
  y <- x * 0.3937     # processing
  y                   # output
}
```

Centimeters to inches

Building a function—conceptually speaking—step by step:

Step 4: Use it and test it

```r
# cm to inches
x <- 10                 # input
cm2in <- function(x) {
  y <- x * 0.3937      # processing
  y                    # output
}

cm2in(5)
```

```
## [1] 1.9685
```

Centimeters to inches

`cm2in()` works like any other function in R, and because it uses the productor operator `*`, it is also *vectorized*

```
cm2in(1:5)
```

```
## [1] 0.3937 0.7874 1.1811 1.5748 1.9685
```

Anatomy of a function

`function()` allows us to create a function. It has the following structure:

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

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
cm2in <- function(x) x * 0.3937

cm2in(10)
```

```
## [1] 3.937
```

Anatomy of a function

If the body of a function is a compund expression we 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
##
## $sumsqr
## [1] 64
```

Nested Functions

We can also define a function inside another function:

```r
getmax <- function(a) {
  maxpos <- function(u) {
    which.max(u)
  }
  list(position = maxpos(a),
       value = max(a))
}

getmax(c(2, -4, 6, 10, pi))
```

```
## $position
## [1] 4
##
## $value
## [1] 10
```

Anatomy of a function

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

Anatomy of a function

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

- the function's name is `cm2in`
- it has one *formal* argument `x`
- the function's body consists of two simple expressions
- it returns the value `y`

Function names

Different ways to name functions

- `cm2in()`
- `cm_in()`
- `cm.in()`
- `cmToIn()`
- `.cm2in()`: a function that starts with a dot is a valid name, but the function will be a *hidden* function.

Function names

Invalid names

- `2cmin()`: cannot begin with a number
- `_cm_in()`: cannot begin with an underscore
- `cm-in()`: cannot use hyphenated names

Output of a Function

Function Output

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

Function Output

The value of a function can be established in two ways:

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

Function Output

We can further simplify the function's body:

```
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

```r
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 ouput via `print()`

Function Output

Depending on what's returned or what's the last evaluated expression, just calling a function might not print anything:

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

cm2in(5)
```

Note: the code of the function works, and the function has a value. But the value is just not printed.

Function Output

Depending on what's returned or what's the last evaluated expression, just calling a function might not print anything:

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

z = cm2in(5)
z
```

```
## [1] 1.9685
```

Note: here we call the function and assign it to an object. The last evaluated expression has the same value as in the preceding slide.

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)
```

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!)

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

hey()
```

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

```r
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:

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

cm2in()
```

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

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)

```
normal_distrib(2)
```

```
## [1] 0.05399097
```

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

```
## [1] 0.05399097
```

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

```
## [1] 0.05399097
```