R functions

Clemens Schmid

Functions

What is a function?

Functions are code modules that perform a specific operation

- Self-contained
  - Take in specific input and return output
  - Code in the function body runs in an own scope
  - Black box: We don't have to understand a function to use it
- Reusable
  - Can be called over and over again in different contexts
  - Can be made more or less generic for more or less specific usecases
  - Can be made available to other developers (via R Packages)

Writing packages is mostly writing functions

Functions in R

Functions in R

R provides several thousand functions in the core packages (base, graphics, stats, methods, ...)

```
mean(c(1,2,3))
```

More functions are distributed through packages in a large ecosystem (CRAN, Github, ...)

```
c14bazAAR::get_c14data("IRDD")
```

Functions in R

Developers and user can easily define own functions

```
# function definition
myfunc <- function(x, y) {
  z <- x + y
  return(z)
}
# function application
myfunc(1,2)
```

Syntax

R uses the following syntax for function definition

```
myfunc <- function(x, y) {
  z <- x + y
  return(z)
}
```

- Function name: `myfunc`
- Input arguments: `x` and `y`
- Function body:

```
z <- x + y
return(z)
```

- Return value: `z`

Exercise

Write a function that

- takes a numeric vector without NA and
- returns the arithmetic mean of this input vector

Exercise

Write a function that

- takes a numeric vector without NA and
- returns the arithmetic mean of this input vector

Possible solution:

```
mymean <- function(x) {
  z <- sum(x) / length(x)
  return(z)
}
```

```
mymean(c(1,2,3,4)) # 2.5
```

Syntax

We can reduce the function definition:

Functions don't need an explicit `return` statement. The value of the last statement is automatically returned

```
myfunc <- function(x, y) { x + y }
```

For one-line function bodies we also don't need curly brackets

```
myfunc <- function(x, y) x + y
```

Syntax

Functions do not even need a name: Lambda functions

```
function(x, y) x + y
```

This is useful when we want to use a function only once for a very specific purpose (later: Higher Order functions)

```
myvec1 <- c(1,2,3)
myvec2 <- c(5,6,7)
Map(
  f = function(x, y) x + y,
  myvec1, myvec2
)
```

Syntax

R 4.1 introduced even shorter syntax for function definition

```
\(x, y) x + y
```

This makes for really elegant code

```
myvec1 <- c(1,2,3)
myvec2 <- c(5,6,7)
Map(
  f = \(x, y) x + y,
  myvec1, myvec2
)
```

But this syntax is very new and only works for the latest R versions

Input and Output

Each function has zero, one or multiple input arguments, but only exactly one output argument

Even a function without output returns `NULL`

```
myfunc <- function(x, y) { message(x) }
myfunc(1) # NULL
```

Usually the output of functions is immediately printed on the console. This can be prevented with `invisible` instead of `return`

```
myfunc <- function(x, y) {
  z <- x + y
  invisible(z)
}
```

Types

Every object in a programming environment has some type. But R functions have no way of being explicit about this: Dynamic type system

```
myfunc <- function(x, y) { x + y }
```

We can use `myfunc` with every input for which the + operator is defined, so all kinds of numeric data types

```
class(1.1) # "numeric"
myfunc(1.1, 2.4) # 3.5
```

```
class(1L) # "integer"
myfunc(1L, 2L) # 3
```

```
class(1i) # "complex"
myfunc(1i, 2i) # 0+3i
```

Types

But it fails for other datatypes

```
class("A") # "character"
myfunc("A", "B")
# Error in x + y :
# non-numeric argument to binary operator
```

Types

Advantages of dynamic typing

- less verbose function definition, because the types don't have to be named
- potentially very flexible functions (functions often automatically work for multiple input types)
- rapid prototyping

Disadvantages

- Nasty error messages for wrong input: No explicit input validation
- No type validation at "compile"-time: significant loss of robustness

That's just the way R is designed

Namespaces

Functions we did not define ourself always come from some package

- Core R functions can just be accessed directly

```
mean()
```

- For all other packages we have to load the respective namespace to access their functions

```
library(ggplot2)
require(magrittr) # rarely used
```

- Or we contextualize the function explicitly with the `::` operator

```
dplyr::mutate()
c14bazAAR:::check_connection_to_url() # internal
```

Namespaces

When developing a package, this becomes less convenient:

All functions need to be explicitly called, except they come from the base package

```
stats::anova()
```

Alternatively we could use the `importFrom` statement in the `NAMESPACE` file

Exercise

Convert this function to a function that could exist in a package

```
library(palmerpenguins)

myfunc <- function() {
  bm <- penguins$body_mass_g
  bl <- penguins$bill_length_mm
  plot(bm, bl)
  pearson <- cor(bm, bl, use = "complete.obs")
  text(x = 5500, y = 35, labels = pearson)
}
```

Exercise

Convert this function to a function that could exist in a package

Possible solution:

```
myfunc <- function() {
  bm <- palmerpenguins::penguins$body_mass_g
  bl <- palmerpenguins::penguins$bill_length_mm
  graphics::plot(bm, bl)
  pearson <- stats::cor(bm, bl, use = "complete.obs")
  graphics::text(x = 5500, y = 35, labels = pearson)
}
```

Package code is often more verbose than script code

Tidy evaluation:
https://dplyr.tidyverse.org/articles/programming.html

Advanced topics

Infix operators

Infix operators are special binary functions, so functions with two arguments, that can be written in between the function arguments

```
3 + 5
```

R comes with a set of operators, some prefix, some infix, some postfix

`+, -, *, /, ^, &, |, :, ::, :::, $, =, <-, <<-, ==, <, <=, >, >=, !=, ~, &&, ||, !, ?, @, :=, (, {, [, [[`

R allows you to write infix operators as normal functions and to define own infix operators

Infix operators

Using an infix operator as a normal function

```
3 + 5 # 8
```

```
`+`(3, 5) # 8
```

3 is the LHS (Left-hand side) and 5 the RHS (Right-hand side) input of the + operator

Infix operators

Defining your own infix operator

```
`%horseplus%` <- function(x, y) {
  z <- x + y
  message("This horse likes bread.")
  return(z)
}
```

```
3 %horseplus% 5
# This horse likes bread.
# 8
```

All self-defined infix operators have to be fenced with %

https://stackoverflow.com/questions/24697248/is-it-possible-to-define-operator-without

Exercise

Define an infix operator `%na0plus%` that

- takes two numeric scalars (so individual numbers, not vectors)
- returns the sum of the input values, but replaces NA with 0

```
NA + 5 # NA
```

```
NA %na0plus% 5 # 5
```

Exercise

Define an infix operator `%na0plus%` that

- takes two numeric scalars (so individual numbers, not vectors)
- returns the sum of the input values, but replaces NA with 0

Possible solution:

```
`%na0plus%` <- function(x, y) {
  x <- `if`(is.na(x), 0, x)
  y <- `if`(is.na(y), 0, y)
  x + y
}
```

```
NA %na0plus% 5 # 5
```

https://codegolf.stackexchange.com/questions/4024/tips-for-golfing-in-r

Chaining functions together

The pipe %>% in the magrittr package is nothing but a clever infix operator

```
c(1,2,3) %>% mean()
```

It *pipes* the LHS *in* as the first argument of the function appearing on the RHS

That allows for sequences of functions ("tidyverse style")

```
mtcars %>%
  dplyr::group_by(cyl) %>%
  dplyr::summarise(mean_mpg = mean(mpg))
```

Default input values

R functions can have default values for all of their arguments. That is a great way to simplify complicated interfaces for normal usecases

```
myfunc <- function(x, y = 5) {
  z <- x + y
  return(z)
}
```

```
myfunc(1) # 6
myfunc(1, 2) # 3
```

Default input values

Default arguments can even be used in the definition of other default arguments

```
calibrate <- function (
  x, choices = c("calrange"), sigma = 2,
  calCurves = rep("intcal20", nrow(x))
) { ... }
```

The ellipsis

The ellipsis `...` is a very special function argument, that can collect an arbitrary amount of unspecified arguments

```
myfunc <- function(...) {
  ell_args <- list(...)
  z <- Reduce(`+`, ell_args, init = 0)
  return(z)
}
```

```
myfunc(1, 2) # 3
myfunc(x = 1, y = 2) # 3
myfunc(x = 1, y = 2, z = 5) # 8
myfunc(1,2,3,4,5,6,7,8,9,10,11,12) # 78
```

```
myfunc(1, 2, NA) # NA
```

The ellipsis

The ellipsis can also be combined with normal arguments

```
myfunc <- function(..., na.rm = T) {
  ell_args <- list(...)
  if (!na.rm) {
    z <- Reduce(`+`, ell_args, init = 0)
  } else {
    z <- Reduce(`%na0plus%`, ell_args, init = 0)
  }
  z <- return(z)
}
```

```
myfunc(1, 2, NA) # 3
```

That is equivalent to `base::sum()` (but very inefficient)

Higher-order functions

A higher-order function is a function that does one of the following:

- takes one or more functions as arguments
- returns a function as its result

R supports this, so functions are "first class citizens" in R

Why would one want to do this?

- make a function interface more powerful
- Mapping, Folding, Moving windows...

Higher-order functions

Functions as an input argument

```
myfunc <- function(vec, f) {
  z <- f(vec)
  return(z)
}
```

```
myfunc(c(1, 2, 3), mean) # 2
myfunc(c(1, 2, 3), sum) # 6
```

Higher-order functions

A function as a function's input and output

```
times_two <- function(x) { x * 2 }
```

```
do_it_twice <- function(f) {
  function(x) { f(f(x)) }
}
```

```
times_two(5) # 10
do_it_twice(times_two)(5) # 20
do_it_twice(do_it_twice(times_two))(5) # 80
```

Practical concerns

How to write functions

1. Define the purpose of your functions
   - What operation should be performed?
2. Think about the function interface
   - What goes into the function (input)?
   - What should the function return (output)?
3. Implement the function
   - Which algorithm is capable to perform the desired operation?

Functions in scripts

General advice for using functions in scripts

- Identify repeating code patterns in your script
- If you do something **three or more** times, then it is worth putting it into a function
- Cover small differences between patterns with function arguments
- Function length: One function should only do one thing, but complete atomization decreases readability

Scripts and Packages

A script usually covers one workflow, but in a package all code lives in functions, so workflows live in sequences of functions

```
read_data("path/to/file") %>%
  mypackage::manipulate_data_A() %>%
  another_package::manipulate_data_B() %>%
  mypackage::manipulate_data_C() %>%
  mypackage::plot_data_D()
```

Not only your functions, of course

Communication with the user

Package functions have (!) to communicate with the user (beyond the documentation)

- What is going on?
- Why did something fail?

Interface options to improve user feedback

- Check conditions and use `message()`, `warning()`, `stop()` to inform the user
- Write clear, helpful messages with advice how to solve an issue
- Show progress updates and progress bars for long operations
- Catch and handle errors that might occur in complex code (`try`, `?conditions`)

Input argument validation

As R is a dynamically typed language, a better user experience can be ensured with explicit input argument validation

```
myfunc <- function(x, y) {
  checkmate::assert_numeric(x, len = 1)
  checkmate::assert_numeric(y, len = 1)
  z <- x + y
  return(z)
}
```

Different packages simplify this, e.g. the `checkmate` package

```
myfunc(x = 5, y = "cookies")
# Assertion on 'y' failed: Must be of type 'numeric',
# not 'character'.
```

Final exercise

Final exercise

https://github.com/sslarch/caa2021_Rpackage_workshop/blob/main/exercises/exercise_functions.R