Stat 243

Writing Functions

Gaston Sanchez

Functions

Motivation

- R comes with many functions (and packages) that let us perform a wide variety of tasks.
- Most of the things we do in R is via calling some function
- Sometimes, however, there's no function to do what we want to achieve.
- Now we want to write functions ourselves
- Idea: avoid repetitive coding (errors will creep in)

Anatomy of a function

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

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

Anatomy of a function

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

Function example

A function that squares its argument

```
square <- function(x) {
  x * x
}
```

- the function name is `"square"`
- it has one argument: `x`
- the function body consists of one simple expression
- it returns the value `x * x`

Function example

It works like any other function in R:

```
square(10)
```

```
## [1] 100
```

In this case, `square()` is also vectorized

```
square(1:5)
```

```
## [1]  1  4  9 16 25
```

Why is `square()` vectorized?

Function example

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

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

sum_of_squares(1:5)
```

```
## [1] 55
```

Function example

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

```
square <- function(x) x * x

square(10)
```

```
## [1] 100
```

However, we recommend you to always write functions using braces

Nested Functions

We can also define a function inside another function:

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

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

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

Naming Functions

Different ways to name functions

- `squareroot()`
- `SquareRoot()`
- `squareRoot()`
- `square.root()`
- `square_root()`

Function Names

Invalid names

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

In addition, avoid using an already existing name, e.g. `sqrt()`

Function Output

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()`

The `return()` command

Sometimes the `return()` command is included to explicitly indicate the output of a function:

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

add(2, 3)
```

```
## [1] 5
```

The `return()` command

If no `return()` is present, then R returns the last evaluated expression:

```
# output with return()
add <- function(x, y) {
  x + y
}

add(2, 3)
```

```
## [1] 5
```

Function Output

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

```
# nothing is printed
add <- function(x, y) {
  z <- x + y
}

add(2, 3)
```

Function Output

Here we call the function and assign it to an object. The last evaluated expression has the same value in both cases:

```
# nothing is printed
add <- function(x, y) {
  z <- x + y
}

a1 <- add(2, 3)
a1
```

```
## [1] 5
```

The return()

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

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

Function Writing

General Strategy for Writing Functions

- Always start simple with test toy-values
- Get what will be the body of the function working first
- Check out each step of the way
- Don't try and do too much at once
- Create (encapsulate body) the function once everything works

Variance Function Example

The sample variance is given by the following formula:

$$var(x) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})$$

Variance Function example

```
# start simple
x <- 1:10

# get working code
sum((x - mean(x)) ^ 2) / (length(x) - 1)
```

```
## [1] 9.166667
```

```
# test it: compare it to var()
var(1:10)
```

```
## [1] 9.166667
```

Variance Function example

```r
# encapsulate your code
variance <- function(x) {
  sum((x - mean(x)) ^ 2) / (length(x) - 1)
}

# check that it works
variance(x)
```

```
## [1] 9.166667
```

Variance Function example

```
# consider less simple cases
variance(runif(10))
```

```
## [1] 0.05926749
```

```
variance(c(1:9, NA))
```

```
## [1] NA
```

```
variance(rep(0, 10))
```

```
## [1] 0
```

Variance Function example

```
# adapt it gradually
variance <- function(x, na.rm = FALSE) {
  if (na.rm) {
    x <- x[!is.na(x)]
  }
  sum((x - mean(x)) ^ 2) / (length(x) - 1)
}

# check that it works
variance(c(1:9, NA), na.rm = TRUE)
```

```
## [1] 7.5
```

Naming Functions

- Choose meaningful names of functions
- Prefereably a verb
- Think about the users (who will use your functions)
- Think about extreme cases

Names of functions

Avoid this:

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

This is better

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

Function Arguments

Function Arguments

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

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

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

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

Arguments can have default values

```
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 defaults

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:

```
sqr <- function(x) {
  x^2
}

sqr()
```

```
## Error in sqr(): argument "x" is missing, with no defaul
```

Arguments with no default values

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

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

Arguments with no default values

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

```
abcd <- function(a, b = 2, c = 3, d = NULL) {
  if (is.null(d)) {
    result <- a * b + c
  } else {
    result <- a * b + c * d
  }
  result
}
```

More Arguments

```r
# arguments with and without default values
myplot <- function(x, y, col = "#3488ff", pch = 19) {
  plot(x, y, col = col, pch = pch)
}

myplot(1:5, 1:5)
```

- `x` and `y` have no default values
- `col` and `pch` have default values (but they can be changed)

Positional and Named Arguments

```
omg <- function(pos1, pos2, name1 = 1, name2 = 2) {
  (pos1 + name1) * (pos2 + name2)
}
```

- pos1 positional argument
- pos2 positional argument
- name1 named argument
- name2 named argument

Arguments

- Arguments with default values are known as **named** arguments
- Arguments with no default values are referred to as **positional** arguments

Argument Matching

Arguments can be matched positionally or by name

```
values <- seq(-2, 1, length.out = 20)

# equivalent calls
mean(values)
mean(x = values)
mean(x = values, na.rm = FALSE)
mean(na.rm = FALSE, x = values)
mean(na.rm = FALSE, values)
```

Partial Matching

Named arguments can also be partially matched:

```
# equivalent calls
seq(from = 1, to = 2, length.out = 5)
seq(from = 1, to = 2, length = 5)
seq(from = 1, to = 2, len = 5)
```

`length.out` is partially matched with `length` and `len`

Arguments

```r
mean(c(NA, 1:9), na.rm = TRUE)

# saving typing
mean(c(NA, 1:9), na.rm = T)

# saving typing but dangerous
mean(c(NA, 1:9), na = T)
```

Arguments

```r
# Generally you don't need to name all arguments
mean(x = c(NA, 1:9), na.rm = TRUE)

# unusual orders best avoided
mean(na.rm = TRUE, x = c(NA, 1:9))
mean(na = T, c(NA, 1:9))
```

Arguments

```
# Don't need to supply defaults
mean(x = c(NA, 1:9), na.rm = FALSE)

# Need to remember too much about mean()
mean(x = c(NA, 1:9), , TRUE)

# Don't abbreviate too much
mean(c(NA, 1:9), n = T)
```

Arguments

```
f <- function(a = 1, abcd = 1, abdd = 1) {
  print(a)
  print(abcd)
  print(abdd)
}

# what will happen?
f(a = 5)
f(ab = 5)
```

```
## Error in f(ab = 5): argument 1 matches multiple formal a
```

```
f(abc = 5)
```

Names of arguments

Give meaningful names to arguments:

```
# Avoid this
area_rect <- function(x, y) {
  x * y
}
```

This is better

```
area_rect <- function(length, width) {
  length * width
}
```

Names of arguments

Even better: give default values (whenever possible)

```
area_rect <- function(length = 1, width = 1) {
  length * width
}
```

Meaningful names to arguments

Avoid this:

```
# what does this function do?
ci <- function(p, r, n, ti) { p * (1 + r/p)^(ti * p)
}
```

This is better:

```
compound_interest <-
function(principal, rate, periods, time) {
  principal * (1 + rate/periods)^(time * periods)
}
```

Messages

Messages

There are two main functions for generating warnings and errors:

- `stop()`
- `warning()`
- There's also the `stopifnot()` function

Stop Execution

Use `stop()` to stop the execution (this will raise an error)

```
meansd <- function(x, na.rm = FALSE) {
  if (!is.numeric(x)) {
    stop("x is not numeric")
  }
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

Warning Messages

Use `warning()` to show a warning message

```
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))
}
```

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

Function `stopifnot()`

`stopifnot()` ensures the truth of expressions:

```
meansd <- function(x, na.rm = FALSE) {
  stopifnot(is.numeric(x))
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}

meansd('hello')
```

```
## Error: is.numeric(x) is not TRUE
```

Documenting Functions

Documenting Functions

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

Documenting Functions

Documentation outside the function

```
# Description: calculates the area of a rectangle
# Inputs
#   length: numeric value
#   width: numeric value
# Output
#   area value
area_rect <- function(length = 1, width = 1) {
  length * width
}
```

Documenting Functions

Documentation inside the function's body

```
area_rect <- function(length = 1, width = 1) {
  # Description: calculates the area of a rectangle
  # Inputs
  #   length: numeric value
  #   width: numeric value
  # Output
  #   area value

  length * width
}
```

Roxygen comments

Documentation with roxygen documents (good for packaging purposes)

```
#' @title Area of Rectangle
#' @description Calculates the area of a rectangle
#' @param length numeric value
#' @param width numeric value
#' @return area (i.e. product of length and width)
#' @examples
#'   area_rect()
#'   area_rect(length = 5, width = 2)
#'   area_rect(width = 2, length = 5)
area_rect <- function(length = 1, width = 1) {
  length * width
}
```

Good Principles

- Don't write long functions
- Rewrite long functions by converting collections of related expressions into separate functions
- A function often corresponds to a verb of a particular step or task in a sequence of tasks
- Functions form the building blocks for larger tasks

Good Principles

- Write functions so that they can be reused in different settings.
- When writing a function, think about different scenarios and contexts in which it might be used
- Can you generalize it?
- Avoid hard coding values that the user might want to provide. Make them default values of new parameters.
- Make the actions of the function as few as possible, or allow the user to turn off some via logical parameters

Good Principles

- Separate small functions:
- are easier to reason about and manage
- clearly identify what they do
- are easier to test and verify they are correct
- are more likely to be reusable as they each do less and so you can pick the functions that do specific tasks

Good Principles

- Make functions parameterizable
- Allow the user to specify values htat might be computed in the function
- This facilitates testing and avoiding recomputing the same thing in different calls
- Can specify different value when testing
- Use a default value to do those computations that would be in the body of the function

Good Principles

- Always test the functions you've written
- Even better: let somebody else test them for you