

Error checking, functions, and loops

Lecture 03

Dr. Colin Rundel

1

Error Checking

stop and stopifnot

Often we want to validate user input, function arguments, or other assumptions in our code - if our assumptions are not met then we often want to report (throw) an error and stop execution.

```
1 ok = FALSE
```

```
1 if (!ok)
2   stop("Things are not ok.")
```

Error in eval(expr, envir, enclos): Things are not ok.

```
1 stopifnot(ok)
```

Error: ok is not TRUE

```
1 if (!ok)
2   stop(
3     "Things are not ok.",
4     call. = FALSE
5   )
```

Error: Things are not ok.

```
1 stopifnot("Still not ok" = ok)
```

Error: Still not ok

Note - an error (like the one generated by `stop`) will prevent an RMarkdown or Quarto document from compiling unless `error = TRUE` or `#| error: true` is set for that code chunk.

3

Style choices

Do stuff:

```
1 if (condition_one) {  
2   ## Do stuff  
3 } else if (condition_two) {  
4   ## Do other stuff  
5 } else if (condition_error) {  
6   stop("Condition error occurred")  
7 }
```

Do stuff (better):

```
1 # Do stuff better  
2 if (condition_error) {  
3   stop("Condition error occurred")  
4 }  
5  
6 if (condition_one) {  
7   ## Do stuff  
8 } else if (condition_two) {  
9   ## Do other stuff  
10 }
```

Exercise 1

Write a set of conditional(s) that satisfies the following requirements,

- If x is greater than 3 and y is less than or equal to 3 then print “Hello world!”
- Otherwise if x is greater than 3 print “!dlrow olleH”
- If x is less than or equal to 3 then print “Something else ...”
- `stop()` execution if x is odd and y is even and report an error, don’t print any of the text strings above.

Test out your code by trying various values of x and y .

R Code

Start Over

Run Code

1

!

OJS Error

bf: Error in `mount(mountpoint, data_url)` : Can't download Emscripten filesystem image metadata.

05:00

5

Why errors?

R has a spectrum of output that can be provided to users,

- Printed output - i.e. `cat()`, `print()`
- Diagnostic messages - i.e. `message()`
- Warnings - i.e. `warning()`
- Errors - i.e. `stop()`, `stopifnot()`

Each of these provides outputs while also providing signals which can be interacted with programmatically (e.g. catching errors or treating warnings as errors).

Handling errors

```
1 flip = function() {  
2   if (runif(1) > 0.5)  
3     stop("Heads")  
4   else  
5     "Tails"  
6 }
```

```
1 flip()
```

```
[1] "Tails"
```

```
1 x = try(flip(), silent=TRUE)  
2 str(x)
```

```
chr "Tails"
```

```
1 flip()
```

```
Error in flip(): Heads
```

```
1 x = try(flip(), silent=TRUE)  
2 str(x)
```

```
'try-error' chr "Error in flip() : Heads\n"  
- attr(*, "condition")=List of 2  
..$ message: chr "Heads"  
..$ call    : language flip()  
..- attr(*, "class")= chr [1:3] "simpleError"  
"error" "condition"
```

The more powerful / flexible version of `try()` is `tryCatch()` but it is beyond the scope of this lecture

7

Functions

What is a function?

Functions are abstractions in programming languages that allow us to modularize our code into small “self contained” units.

In general the goals of writing functions is to,

- Simplify a complex process or task into smaller sub-steps
- Allow for the reuse of code without duplication
- Improve the readability of your code
- Improve the maintainability of your code

Functions as objects

Functions are 1st order objects in R and have a mode of `function`. They are assigned names like other objects using `=` or `<-`.

```
1 gcd = function(x1, y1, x2 = 0, y2 = 0) {  
2   R = 6371 # Earth mean radius in km  
3  
4   # distance in km  
5   acos(sin(y1)*sin(y2) + cos(y1)*cos(y2) * cos(x2-x1)) * R  
6 }
```

```
1 typeof(gcd)
```

```
[1] "closure"
```

```
1 mode(gcd)
```

```
[1] "function"
```

We use mode here because there are two kinds of functions in R, `closures` and primitive functions (with type `builtin`).

10

Function elements

In R functions are defined by *two* components: the arguments (`formals`) and the code (`body`).

```
1 str( formals(gcd) )
```

Dotted pair list of 4

```
$ x1: symbol  
$ y1: symbol  
$ x2: num 0  
$ y2: num 0
```

```
1 typeof( formals(gcd) )
```

```
[1] "pairlist"
```

```
1 body(gcd)
```

```
{  
  R = 6371  
  acos(sin(y1) * sin(y2) + cos(y1) *  
cos(y2) * cos(x2 - x1)) *  
  R  
}
```

```
1 typeof( body(gcd) )
```

```
[1] "language"
```

Note when using `body()` here the code we get back has had comments removed, if you want to access the full code you can use `attr(gcd, "srcref")`.

11

Return values

As with most other languages, functions are most often used to process inputs and return a value as output. There are two approaches to returning values from functions in R - *explicit* and *implicit* returns.

Explicit - using one or more `return` calls

```
1 f = function(x) {  
2   return(x * x)  
3 }  
4 f(2)
```

```
[1] 4
```

Implicit - return value of the last expression is returned.

```
1 g = function(x) {  
2   x * x  
3 }  
4 g(3)
```

```
[1] 9
```

Most expressions in R return a value even if this may not be obvious at the time

12

Invisible returns

Many functions in R make use of an invisible return value

```
1 f = function(x) {  
2   print(x)  
3 }
```

```
1 y = f(1)
```

```
[1] 1
```

```
1 y
```

```
[1] 1
```

```
1 g = function(x) {  
2   invisible(x)  
3 }
```

```
1 g(2)
```

```
1 z = g(2)
```

```
2 z
```

```
[1] 2
```

Returning multiple values

If we want a function to return more than one value we can group results using atomic vectors or lists.

```
1 f = function(x) {  
2   c(x, x^2, x^3)  
3 }  
4  
5 f(1:2)
```

```
[1] 1 2 1 4 1 8
```

```
1 g = function(x) {  
2   list(x, "hello")  
3 }  
4  
5 g(1:2)
```

```
[[1]]
```

```
[1] 1 2
```

```
[[2]]
```

```
[1] "hello"
```

More on `list()`s next time

14

Argument names

When defining a function we explicitly define names for the arguments, which become variables within the scope of the function.

When calling a function we can use these names to pass arguments in an alternative order.

```
1 f = function(x, y, z) {  
2   paste0("x=", x, " y=", y, " z=", z)  
3 }
```

```
1 f(1, 2, 3)
```

```
[1] "x=1 y=2 z=3"
```

```
1 f(z=1, x=2, y=3)
```

```
[1] "x=2 y=3 z=1"
```

```
1 f(1, 2, 3, 4)
```

Error in f(1, 2, 3, 4): unused argument
(4)

```
1 f(y=2, 1, 3)
```

```
[1] "x=1 y=2 z=3"
```

```
1 f(y=2, 1, x=3)
```

```
[1] "x=3 y=2 z=1"
```

```
1 f(1, 2, m=3)
```

Error in f(1, 2, m = 3): unused argument
(m = 3)

Argument defaults

It is also possible to give function arguments default values, so that they don't need to be provided every time the function is called.

```
1 f = function(x, y=1, z=1) {  
2   paste0("x=", x, " y=", y, " z=", z)  
3 }
```

```
1 f(3)
```

```
[1] "x=3 y=1 z=1"
```

```
1 f(x=3)
```

```
[1] "x=3 y=1 z=1"
```

```
1 f()
```

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

```
1 f(z=3, x=2)
```

```
[1] "x=2 y=1 z=3"
```

```
1 f(y=2, 2)
```

```
[1] "x=2 y=2 z=1"
```

This ability to freely mix the ordering of named and unnamed arguments is fairly *unique* to R

16

Scope

R has generous scoping rules, if it can't find a variable in the current scope (e.g. a function's body) it will look for it in the next higher scope, and so on until it runs out of environments or an object with that name is found.

```
1 y = 1
2
3 f = function(x) {
4   x + y
5 }
6
7 f(3)
```

[1] 4

```
1 y = 1
2
3 g = function(x) {
4   y = 2
5   x + y
6 }
7
8 g(3)
```

[1] 5

```
1 y
```

[1] 1

Scope persistence

Additionally, variables defined within a scope only persist for the duration of that scope, and do not overwrite variables at higher scope(s).

```
1 x = 1
2 y = 1
3 z = 1
4
5 f = function() {
6   y = 2
7   g = function() {
8     z = 3
9     return(x + y + z)
10  }
11  return(g())
12 }
```

```
1 f()
```

```
[1] 6
```

```
1 c(x,y,z)
```

```
[1] 1 1 1
```

R supports global assignment via `<-`, generally using global variables is considered bad practice and should be avoided. ¹⁸

Exercise 2 - scope

What is the output of the following code? Explain why.

R Code Start Over Run Code

```
1 z = 1
2
3 f = function(x, y, z) {
4   z = x+y
5
6   g = function(m = x, n = y) {
7     m/z + n/z
8   }
9
10  z * g()
11 }
12
13 f(1, 2, x = 3)
```

 OJS Error

bf: Error in `mount(mountpoint, data_url)` : Can't download Emscripten filesystem image metadata.

03:00

19

Lazy evaluation

Another interesting / unique feature of R is that function arguments are lazily evaluated, which means they are only evaluated when needed.

```
1 f = function(x) {  
2   TRUE  
3 }
```

```
1 f(1)
```

```
[1] TRUE
```

```
1 f(stop("Error"))
```

```
[1] TRUE
```

```
1 g = function(x) {  
2   x  
3   TRUE  
4 }
```

```
1 g(1)
```

```
[1] TRUE
```

```
1 g(stop("Error"))
```

```
Error in g(stop("Error")): Error
```

More practical lazy evaluation

The previous example is not particularly useful, a more common use for this lazy evaluation is that this enables us define arguments as expressions of other arguments.

```
1 f = function(x, y=x+1, z=1) {  
2   x = x + z  
3   y  
4 }  
5  
6 f(x=1)
```

[1] 3

```
1 f(x=1, z=2)
```

[1] 4

Operators as functions

In R, operators are actually a special type of function - using backticks around the operator we can write them as functions.

```
1 `+`
```

```
function (e1, e2) .Primitive("+")
```

```
1 typeof(`+`)
```

```
[1] "builtin"
```

```
1 x = 4:1
```

```
2 x + 2
```

```
[1] 6 5 4 3
```

```
1 `+`(x, 2)
```

```
[1] 6 5 4 3
```

Getting Help

Prefixing any function name with a `?` will open the related help file for that function.

```
1 ?`+`  
2 ?sum
```

For functions not in the base package, you can generally see their implementation by entering the function name without parentheses (or using the `body` function).

```
1 lm
```

```
function (formula, data, subset, weights, na.action, method = "qr",  
  model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,  
  contrasts = NULL, offset, ...)  
{  
  ret.x <- x  
  ret.y <- y  
  cl <- match.call()  
  mf <- match.call(expand.dots = FALSE)  
  m <- match(c("formula", "data", "subset", "weights", "na.action",  
    "offset"), names(mf), 0L)  
  mf <- mf[c(1L, m)]  
  mf$drop.unused.levels <- TRUE  
  mf[[1L]] <- quote(stats::model.frame)  
  mf <- eval(mf, parent.frame())  
  if (method == "model.frame")  
    return(mf)
```

23

Less Helpful Examples

```
1 list
```

```
function (...) .Primitive("list")
```

```
1 `[`
```

```
.Primitive("[")
```

```
1 sum
```

```
function (..., na.rm = FALSE) .Primitive("sum")
```

```
1 `+`
```

```
function (e1, e2) .Primitive("+")
```

For the curious the [lookup](#) package will help you track down the source code of these functions.

24

Infix functions (operators)

We can define our own infix functions like `+` or `*`, the only requirement is that the function name must start and end with a `%`.

```
1 `%nand%` = function(x, y) {  
2   !(x & y)  
3 }
```

```
1 TRUE %nand% TRUE
```

```
[1] FALSE
```

```
1 TRUE %nand% FALSE
```

```
[1] TRUE
```

```
1 FALSE %nand% TRUE
```

```
[1] TRUE
```

```
1 FALSE %nand% FALSE
```

```
[1] TRUE
```

Replacement functions

We can also define functions that allow for ‘inplace’ modification like `attr` or `names`.

```
1 `last<--` = function(x, value) {  
2   x[length(x)] = value  
3   x  
4 }
```

```
1 x = 1:10
```

```
1 last(x) = 5L  
2 x
```

```
[1] 1 2 3 4 5 6 7 8 9 5
```

```
1 last(x) = NA  
2 x
```

```
[1] 1 2 3 4 5 6 7 8 9 NA
```

```
1 last(1)
```

Error in last(1): could not find function "last"

```
1 `modify<--` = function(x, pos, value) {  
2   x[pos] = value  
3   x  
4 }
```

```
1 x = 1:10
```

```
1 modify(x, 1) = 5L  
2 x
```

```
[1] 5 2 3 4 5 6 7 8 9 10
```

```
1 modify(x, 9:10) = 1L  
2 x
```

```
[1] 5 2 3 4 5 6 7 8 1 1
```

Loops

for loops

There are the most common type of loop in R - given a vector it iterates through the elements and evaluate the code expression for each value.

```
1 is_even = function(x) {  
2   res = c()  
3  
4   for(val in x) {  
5     res = c(res, val %% 2 == 0)  
6   }  
7  
8   res  
9 }
```

```
1 is_even(1:10)
```

```
[1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
```

```
1 is_even(seq(1,5,2))
```

```
[1] FALSE FALSE FALSE
```

while loops

This loop repeats evaluation of the code expression until the condition is **not** met (i.e. evaluates to **FALSE**)

```
1 make_seq = function(from = 1, to = 1, by = 1) {  
2   res = c(from)  
3   cur = from  
4  
5   while(cur+by <= to) {  
6     cur = cur + by  
7     res = c(res, cur)  
8   }  
9  
10  res  
11 }
```

```
1 make_seq(1, 6)
```

```
[1] 1 2 3 4 5 6
```

```
1 make_seq(1, 6, 2)
```

```
[1] 1 3 5
```

repeat loops

Equivalent to a `while(TRUE){}` loop, it repeats until a `break` statement

```
1 make_seq2 = function(from = 1, to = 1, by = 1) {  
2   res = c(from)  
3   cur = from  
4  
5   repeat {  
6     cur = cur + by  
7     if (cur > to)  
8       break  
9     res = c(res, cur)  
10  }  
11  
12  res  
13 }
```

```
1 make_seq2(1, 6)
```

```
[1] 1 2 3 4 5 6
```

```
1 make_seq2(1, 6, 2)
```

```
[1] 1 3 5
```

30

Special keywords - **break** and **next**

These are special actions that only work *inside* of a loop

- **break** - ends the current **loop** (inner-most)
- **next** - ends the current **iteration**

```
1 f = function(x) {  
2   res = c()  
3   for(i in x) {  
4     if (i %% 2 == 0)  
5       break  
6     res = c(res, i)  
7   }  
8   res  
9 }
```

```
1 f(1:10)
```

```
[1] 1
```

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

```
[1] 1 1 1
```

```
1 g = function(x) {  
2   res = c()  
3   for(i in x) {  
4     if (i %% 2 == 0)  
5       next  
6     res = c(res,i)  
7   }  
8   res  
9 }
```

```
1 g(1:10)
```

```
[1] 1 3 5 7 9
```

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

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


Some helpful functions

Often we want to use a loop across the indexes of an object and not the elements themselves. There are several useful functions to help you do this: `:`, `length`, `seq`, `seq_along`, `seq_len`, etc.

```
1 4:7
```

```
[1] 4 5 6 7
```

```
1 length(4:7)
```

```
[1] 4
```

```
1 seq(4,7)
```

```
[1] 4 5 6 7
```

```
1 seq_along(4:7)
```

```
[1] 1 2 3 4
```

```
1 seq_len(length(4:7))
```

```
[1] 1 2 3 4
```

```
1 seq(4,7,by=2)
```

```
[1] 4 6
```

Avoid using `1:length(x)`

A common loop construction you'll see in a lot of R code is using `1:length(x)` to generate a vector of index values for the vector `x`.

```
1 f = function(x) {  
2   for(i in 1:length(x)) {  
3     print(i)  
4   }  
5 }  
6  
7 f(2:1)
```

```
[1] 1
```

```
[1] 2
```

```
1 f(2)
```

```
[1] 1
```

```
1 f(integer())
```

```
[1] 1
```

```
[1] 0
```

```
1 g = function(x) {  
2   for(i in seq_along(x)) {  
3     print(i)  
4   }  
5 }  
6  
7 g(2:1)
```

```
[1] 1
```

```
[1] 2
```

```
1 g(2)
```

```
[1] 1
```

```
1 g(integer())
```

What was the problem?

```
1 length(integer())
```

```
[1] 0
```

```
1 1:length(integer())
```

```
[1] 1 0
```

```
1 seq_along(integer())
```

```
integer(0)
```


Exercise 3

To the right is a vector containing all prime numbers between 2 and 100 and a separate vector `x` containing some values we would like to check for primality.

Write the R code necessary to print only the values of `x` that are *not* prime (without using subsetting or the `%in%` operator).

Your code will need to use *nested* loops to iterate through the vector of `primes` and `x`.

```
R Code ↺ Start Over ▶ Run Code
1 primes = c( 2, 3, 5, 7, 11, 13, 17, 19, 23,
2           29, 31, 37, 41, 43, 47, 53, 59, 61,
3           67, 71, 73, 79, 83, 89, 97)
4
5 x = c(3,4,12,19,23,51,61,63,78)
```

 OJS Error
bf: Error in `mount(mountpoint, data_url)` : Can't download Emscripten filesystem image metadata.

05:00

35