Error checking, functions, and loops

Lecture 03

Dr. Colin Rundel

Downloading package: evaluate

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.

```r
ok = FALSE
```

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

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

```r
if (!ok)
  stop(
    "Things are not ok.",
    call. = FALSE
  )
```

```
Error: Things are not ok.
```

```r
stopifnot(ok)
```

```
Error: ok is not TRUE
```

```r
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 rendering unless `error = TRUE` or `#| error: true` is set for that code chunk.

Downloading package: evaluate

Style choices

Do stuff:

```r
if (condition_one) {
  ## Do stuff
} else if (condition_two) {
  ## Do other stuff
} else if (condition_error) {
  stop("Condition error occured")
}
```

Do stuff (better):

```r
# Do stuff better
if (condition_error) {
  stop("Condition error occured")
}

if (condition_one) {
  ## Do stuff
} else if (condition_two) {
  ## Do other stuff
}
```

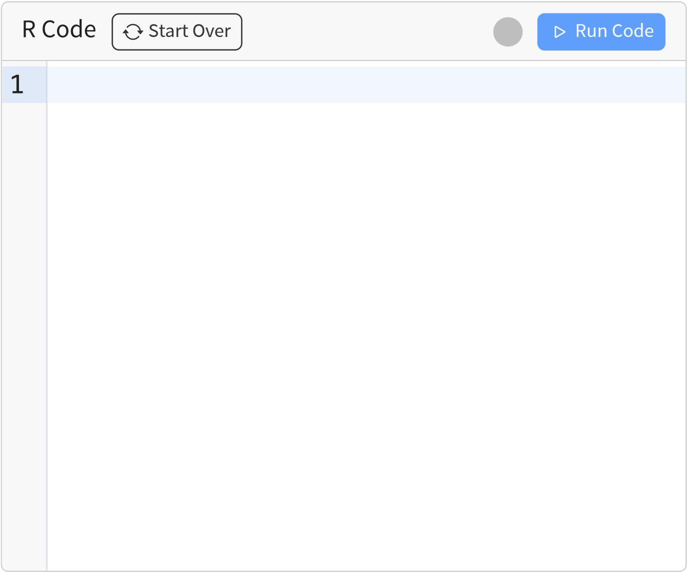
Downloading package: evaluate

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`.



OJS Error

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

05:00

Downloading package: evaluate

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

```r
flip = function() {
  if (runif(1) > 0.5)
    stop("Heads")
  else
    "Tails"
}
```

```r
flip()
```

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

```r
x = try(flip(), silent=TRUE)
str(x)
```

```
 chr "Tails"
```

```r
flip()
```

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

```r
x = try(flip(), silent=TRUE)
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

Downloading package: evaluate

Functions

Downloading package: evaluate

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 `<-`.

```r
gcd = function(x1, y1, x2 = 0, y2 = 0) {
  R = 6371 # Earth mean radius in km

  # distance in km
  acos(sin(y1)*sin(y2) + cos(y1)*cos(y2) * cos(x2-x1)) * R
}
```

```r
typeof(gcd)
```

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

```r
mode(gcd)
```

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

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

Function elements

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

```r
str( formals(gcd) )
```

```
Dotted pair list of 4
 $ x1: symbol
 $ y1: symbol
 $ x2: num 0
 $ y2: num 0
```

```r
body(gcd)
```

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

```r
typeof( formals(gcd) )
```

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

```r
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")`.

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

```r
f = function(x) {
  return(x * x)
}
f(2)
```

```
[1] 4
```

Implicit - return value of the last expression is returned.

```r
g = function(x) {
  x * x
}
g(3)
```

```
[1] 9
```

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

Downloading package: evaluate

Invisible returns

Many functions in R make use of an invisible return value

```r
f = function(x) {
  print(x)
}
```

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

```
[1] 1
```

```r
y
```

```
[1] 1
```

```r
g = function(x) {
  invisible(x)
}
```

```r
g(2)
```

```r
z = g(2)
z
```

```
[1] 2
```

Downloading package: evaluate

Returning multiple values

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

```r
f = function(x) {
  c(x, x^2, x^3)
}

f(1:2)
```

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

```r
g = function(x) {
  list(x, "hello")
}

g(1:2)
```

```
[[1]]
[1] 1 2

[[2]]
[1] "hello"
```

More on `list()`s next time

Downloading package: evaluate

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.

```r
f = function(x, y, z) {
  paste0("x=", x, " y=", y, " z=", z)
}
```

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

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

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

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

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

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

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

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

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

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

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

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

Downloading package: evaluate

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.

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

```r
f(3)
```

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

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

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

```r
f(x=3)
```

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

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

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

```r
f()
```

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

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

Downloading package: evaluate

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.

```r
y = 1

f = function(x) {
  x + y
}

f(3)
```

```
[1] 4
```

```r
y = 1

g = function(x) {
  y = 2
  x + y
}

g(3)
```

```
[1] 5
```

```r
y
```

```
[1] 1
```

Downloading package: evaluate

Scope persistance

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

```r
x = 1
y = 1
z = 1

f = function() {
    y = 2
    g = function() {
      z = 3
      return(x + y + z)
    }
    return(g())
}
```

```r
f()
```

```
[1] 6
```

```r
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.

Downloading package: evaluate

Exercise 2 - scope

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

R Code   Start Over   Run Code

```r
z = 1

f = function(x, y, z) {
  z = x+y

  g = function(m = x, n = y) {
    m/z + n/z
  }

  z * g()
}

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

OJS Error

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

03:00

Downloading package: evaluate

Lazy evaluation

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

```r
f = function(x) {
  TRUE
}
```

```r
f(1)
```

```
[1] TRUE
```

```r
f(stop("Error"))
```

```
[1] TRUE
```

```r
g = function(x) {
  x
  TRUE
}
```

```r
g(1)
```

```
[1] TRUE
```

```r
g(stop("Error"))
```

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

Downloading package: evaluate

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.

```r
f = function(x, y=x+1, z=1) {
  x = x + z
  y
}

f(x=1)
```

```
[1] 3
```

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

```
[1] 4
```

Downloading package: evaluate

Operators as functions

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

```r
`+`
```

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

```r
typeof(`+`)
```

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

```r
x = 4:1
x + 2
```

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

```r
`+`(x, 2)
```

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

Downloading package: evaluate

Getting Help

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

```
?`+`
?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).

```
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)
```

Downloading package: evaluate

Less Helpful Examples

```r
list
```

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

```r
`[`
```

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

```r
sum
```

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

```r
`+`
```

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

For the curious the lookup package will help you track down the source code of these functions.

Downloading package: evaluate

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 `%`.

```r
`%nand%` = function(x, y) {
  !(x & y)
}
```

```r
TRUE %nand% TRUE
```

```
[1] FALSE
```

```r
TRUE %nand% FALSE
```

```
[1] TRUE
```

```r
FALSE %nand% TRUE
```

```
[1] TRUE
```

```r
FALSE %nand% FALSE
```

```
[1] TRUE
```

Downloading package: evaluate

Replacement functions

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

```r
`last<-` = function(x, value) {
  x[length(x)] = value
  x
}
```

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

```r
last(x) = 5L
x
```

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

```r
last(x) = NA
x
```

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

```r
last(1)
```

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

```r
`modify<-` = function(x, pos, value) {
  x[pos] = value
  x
}
```

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

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

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

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

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

Downloading package: evaluate

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.

```r
is_even = function(x) {
  res = c()

  for(val in x) {
    res = c(res, val %% 2 == 0)
  }

  res
}
```

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

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

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

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

Downloading package: evaluate

`while` loops

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

```r
make_seq = function(from = 1, to = 1, by = 1) {
  res = c(from)
  cur = from

  while(cur+by <= to) {
    cur = cur + by
    res = c(res, cur)
  }

  res
}
```

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

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

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

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

Downloading package: evaluate

repeat loops

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

```r
make_seq2 = function(from = 1, to = 1, by = 1) {
  res = c(from)
  cur = from

  repeat {
    cur = cur + by
    if (cur > to)
      break
    res = c(res, cur)
  }

  res
}
```

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

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

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

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

Downloading package: evaluate

Special keywords - `break` and `next`

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

- `break` - ends the current **loop**

```r
f = function(x) {
  res = c()
  for(i in x) {
    if (i %% 2 == 0)
      break
    res = c(res, i)
  }
  res
}
```

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

```
[1] 1
```

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

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

- `next` - ends the current **iteration**

```r
g = function(x) {
  res = c()
  for(i in x) {
    if (i %% 2 == 0)
      next
    res = c(res,i)
  }
  res
}
```

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

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

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

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

Downloading package: evaluate

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.

```r
4:7
```

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

```r
length(4:7)
```

```
[1] 4
```

```r
seq(4,7)
```

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

```r
seq_along(4:7)
```

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

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

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

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

```
[1] 4 6
```

Downloading package: evaluate

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`.

```r
f = function(x) {
  for(i in 1:length(x)) {
    print(i)
  }
}

f(2:1)
```

```
[1] 1
[1] 2
```

```r
f(2)
```

```
[1] 1
```

```r
f(integer())
```

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

```r
g = function(x) {
  for(i in seq_along(x)) {
    print(i)
  }
}

g(2:1)
```

```
[1] 1
[1] 2
```

```r
g(2)
```

```
[1] 1
```

```r
g(integer())
```

Downloading package: evaluate

What was the problem?

```r
length(integer())
```

```
[1] 0
```

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

```
[1] 1 0
```

```r
seq_along(integer())
```

```
integer(0)
```

Downloading package: evaluate

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

```r
primes = c( 2,  3,  5,  7, 11, 13, 17, 19, 23,
           29, 31, 37, 41, 43, 47, 53, 59, 61,
           67, 71, 73, 79, 83, 89, 97)

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

Downloading package: evaluate