R-Ladies Advanced R Bookclub

Chapter 8: Conditions

Semiramis Castro

RLadies

- rladies.org
- Code of conduct

Let's keep in touch!

- @semiramis_cj

...and let's get started!!

We signal conditions as developers: "The state of things is..."

What would be the use cases?

- messages ✉

```
base::message("A message from the developer")
rlang::inform("This is a message from your developer")
```

- warnings ❗

```
base::warning("This is a warning!!")
rlang::warn("You might want to fix this")
```

- errors 💣

```
base::stop("An error occured!!!")
rlang::abort("You MUST fix this!")
```

- interrupt (only in interactive mode): Ctrl+C, Esc

We handle conditions as users: What is happening!?! How do I solve this?

- Just ignore the signaling

```
try() # For errors
suppressWarnings() # For warnings
suppressMessages() # For messages
```

- Do something about it

```
tryCatch() # For errors
withCallingHandlers() # For warnings and messages
rlang::catch_cnd() # For any condition
```

We ignore errors with try()

```
calculate_log_try <- function(x) {
  # We catch an error if it occurs
  try( log(x) )
  # But we continue with the execution as if nothing happened
  sum(1:5)

}
calculate_log_try("a")
```

```
## Error in log(x) : non-numeric argument to mathematical function

## [1] 15
```

We can ignore warnings or messages selectively

```
suppressWarnings({
  warning("Uhoh!")
  warning("Another warning")
  1
})
```

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

```
suppressMessages({
  message("Hello there")
  2
})
```

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

```
suppressWarnings({
  message("You can still see me because I am a message")
  3
})
```

```
## You can still see me because I am a message
```

catch_cnd()

The easiest way to see a condition object is to catch one from a signalled condition. That's the job of `rlang::catch_cnd()`

```
cnd <- catch_cnd(stop("An error"))
str(cnd)
```

```
## List of 2
##  $ message: chr "An error"
##  $ call   : language force(expr)
##  - attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Exiting handlers:

If we get an error, the downstream code will not be executed!! 💣

```
calculate_log_unprotected <- function(x, base=10){
  log(x)
  print("Finished with success!")
}
calculate_log_unprotected("10")
# Error in log(x) : non-numeric argument to mathematical function
```

We can use tryCatch() to continue the execution

```
tryCatch(
  message = function(any_error) "There was an error!",
  expr=
    {
    log("x") # What we try to do
    message("No errors found")
  }
)
# Error in log("x") : non-numeric argument to mathematical function
```

We can also provide a default value when there is an error

```
calculate_log_trycatch <- function(x, base) {
  tryCatch(
    error = function(any_error) NA, # NA will be our default value
    expr = {
    log(x, base) # What we want to do
    message("No errors found!")
    x + 1
    }
  )
}
```

```
calculate_log_trycatch(10, 10) # When nothing fails
```

```
## No errors found!
```

```
## [1] 11
```

```
# This code runs uninterrupted even if there is an error
calculate_log_trycatch("10", 10)
```

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

What if the execution must stop?

We can signal with `base::stop()` or with `rlang::abort()` 💣

```
calculate_log_verbose <- function(x, base = exp(1)) {
  # Check our inputs, stop the execution if they are not valid
  # But also tell the user where is the problem
  if (!is.numeric(x)) {
    abort(paste0(
      "`x` must be a numeric vector; not ", typeof(x), "."
    ))
  }
  if (!is.numeric(base)) {
    abort(paste0(
      "`base` must be a numeric vector; not ", typeof(base), "."
    ))
  }
  # We can run this if there are no errors
  log(x, base = base)
}
```

```
calculate_log_verbose(letters)
## Error: `x` must be a numeric vector; not character.
calculate_log_verbose(1:10, base = letters)
## Error: `base` must be a numeric vector; not character.
calculate_log_verbose(1:5, base = 10) # This code runs without problems
# [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700
```

Calling handlers

If what happened is not critical, and we want to continue with the flow of our script, we can use `withCallingHandlers()`

The messages are applied in the order we send them

```
withCallingHandlers(
  # We catch the condition and print to the console
  message = function(cnd) message("First message -from the top with base"),
  {
    # This code will be executed after catching the condition
    # After each message, the control will return to the top
    rlang::inform("Second message with rlang")
    rlang::warn("Ooops! A warning")
    message("Third message -with base")
  }
)
```

```
## First message -from the top with base

## Second message with rlang

## Warning: Ooops! A warning

## First message -from the top with base
## Third message -with base
```

The call stack tree gives us info about what was called and its order

We can explore the call stack tree with `traceback()` or with `lobstr::cst()`

```
f <- function() g()
g <- function() h()
h <- function() lobstr::cst()
f()
```

```
##     ▉
##  1. └─global::f()
##  2.   └─global::g()
##  3.     └─global::h()
##  4.       └─lobstr::cst()
```

And the call stack tree structure varies depending on the type of handler

Exiting handlers are called in the context of the call to tryCatch():

```
tryCatch(f(), message = function(cnd) lobstr::cst())
```

```
##     ▉
##  1. ├─base::tryCatch(f(), message = function(cnd) lobstr::cst())
##  2. │ └─base:::tryCatchList(expr, classes, parentenv, handlers)
##  3. │   └─base:::tryCatchOne(expr, names, parentenv, handlers[[1L]])
##  4. │     └─base:::doTryCatch(return(expr), name, parentenv, handler)
##  5. └─global::f()
##  6.   └─global::g()
##  7.     └─global::h()
##  8.       └─lobstr::cst()
```

Calling handlers are called in the context of the call that signaled the condition:

```
withCallingHandlers(f(), message = function(cnd) {
  lobstr::cst()
  cnd_muffle(cnd)
})
```

```
##     ▉
##  1. ├─base::withCallingHandlers(...)
##  2. └─global::f()
##  3.   └─global::g()
##  4.     └─global::h()
##  5.       └─lobstr::cst()
```

Custom conditions are useful for not relying on string matching to catch them! 1/2

We create our custom condition "abort_bad_argument"

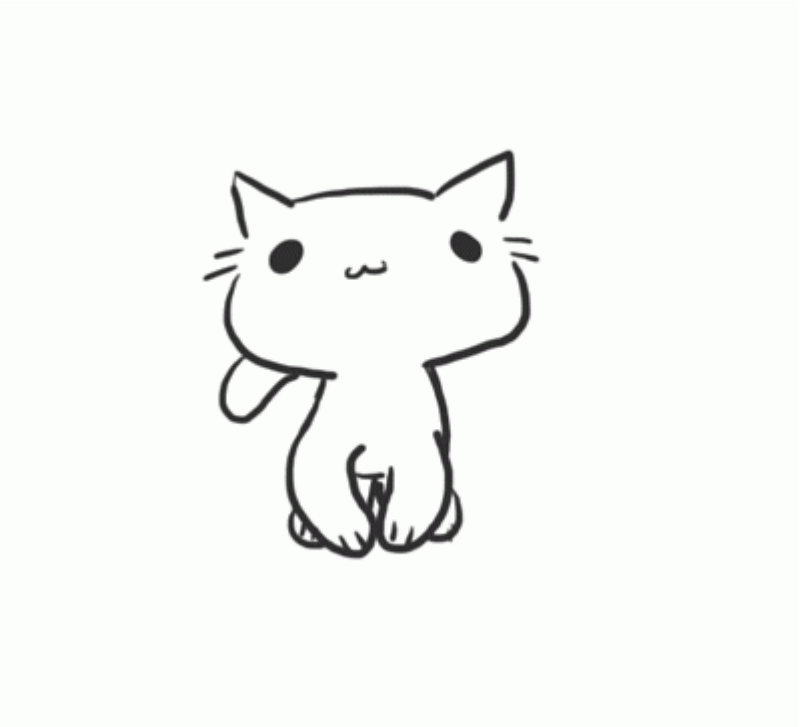
```
abort_bad_argument <- function(arg, must, not = NULL) {
  msg <- glue::glue("`{arg}` must {must}") # This text might change
  if (!is.null(not)) {
    not <- typeof(not)
    msg <- glue::glue("{msg}; not {not}.")
  }
  abort("error_bad_argument",
    message = msg,
    arg = arg,
    must = must,
    not = not
  )
}

log_custom_condition <- function(x, base = 10) {
  if (!is.numeric(x)) {
    # We are using our custom condition!
    abort_bad_argument("x", must = "be numeric", not = x)
  }
  if (!is.numeric(base)) {
    abort_bad_argument("base", must = "be numeric", not = base)
  }
```

Custom conditions are useful for not relying on string matching to catch them! 2/2

```
catch_cnd(
  log_custom_condition("10")
  )
```

```
## <error/error_bad_argument>
## `x` must be numeric; not character.
## Backtrace:
##   1. rmarkdown::render(...)
##  26. global::log_custom_condition("10")
##  27. global::abort_bad_argument("x", must = "be numeric", not = x)
```

Time for a break!



Quizz

1. What are the three most important types of condition?
2. What function do you use to ignore errors in block of code?
3. What's the main difference between `tryCatch()` and `withCallingHandlers()`?
4. Why might you want to create a custom error object?

Quizz - answers

What are the three most important types of condition?

- errors, warnings & messages

What function do you use to ignore errors in block of code?

- `try()` or `tryCatch()`

What's the main difference between `tryCatch()` and `withCallingHandlers()`?

- `tryCatch()` handles errors
- `withCallingHandlers()` is for warnings and messages

Why might you want to create a custom error object?

- To avoid comparison of error strings when we want to catch specific types of errors

Predict the results of evaluating the following code:

```
show_condition <- function(code) {
  tryCatch(
    # Errors, warnings and messages are catched from the start
    error = function(cnd) "error",
    warning = function(cnd) "warning",
    message = function(cnd) "message",
    # Our code is executed here
    {
    code
    NULL # The return value if nothing was signaled
    }
  )
}
```

```
show_condition(stop("!")) # case A)
show_condition(10) # case B)
show_condition(warning("?!")) # case C)
show_condition({ # case D)
  10
  message("?")
  warning("?!")
})
```

Answer:

- case A) will print "error"
- case B) will print "NULL"
- case C) will print "warning"
- case D will terminate when we arrive to the message. Remember: exiting handlers are called in the context of tryCatch()

```
show_condition({  # case D)
  10
  message("?")
  warning("?!")
})
```

```
## [1] "message"
```

Explain the results of running this code:

```
withCallingHandlers( # (1)
  message = function(cnd) message("b"),
  withCallingHandlers( # (2)
    message = function(cnd) message("a"),
    message("c")
  )
)
```

```
## b

## a

## b

## c
```

Answer:

```
withCallingHandlers( # (1)
  message = function(cnd) message("b"),
  withCallingHandlers( # (2)
    message = function(cnd) message("a"),
    message("c")
  )
)
```

```
## b

## a

## b

## c
```

- First, we enter into (1): the message is "b"
- then, we go to (2): the message is "a"
- we return to (1) because we didn't handle the message "b", so it bubbles up to the outer calling handler
- finally, we go to "c"

Compare the following two implementations of message2error(). What is the main advantage of withCallingHandlers() in this scenario? (Hint: look carefully at the traceback.)

```
message2error_withCallingHandlers <- function(code) {
  withCallingHandlers(code, message = function(e) stop(e))
}
```

```
message2error_tryCatch <- function(code) {
  tryCatch(code, message = function(e) stop(e))
}
```

Answer:

withCallingHandlers() returns more information and points us to the exact call in our code because it is called in the context of the call that signalled the condition, whereas exiting handlers are called in the context of tryCatch()

```
message2error_withCallingHandlers( {1;
  message("hidden error"); NULL} )
traceback()
# Error in message("hidden error") : h
# 9: stop(e) at <text>#2
# 8: (function (e)
#    stop(e))(list(message = "hidden e
# 7: signalCondition(cond)
# 6: doWithOneRestart(return(expr), re
# 5: withOneRestart(expr, restarts[[1L
# 4: withRestarts({
#        signalCondition(cond)
#        defaultHandler(cond)
#    }, muffleMessage = function() NUL
# 3: message("hidden error") at #1
# 2: withCallingHandlers(code, message
# 1: message2error_withCallingHandlers
#        1
#        message("hidden error")
#        NULL
#    })
```

```
message2error_tryCatch({1;
  message("hidden error"); NULL} )
traceback()
# Error in message("hidden error") : h
# 6: stop(e) at <text>#2
# 5: value[[3L]](cond)
# 4: tryCatchOne(expr, names, parenten
# 3: tryCatchList(expr, classes, paren
# 2: tryCatch(code, message = function
# 1: message2error_tryCatch({
#        1
#        message("hidden error")
#        NULL
#    })
```

Why is catching interrupts dangerous? Run this code to find out.

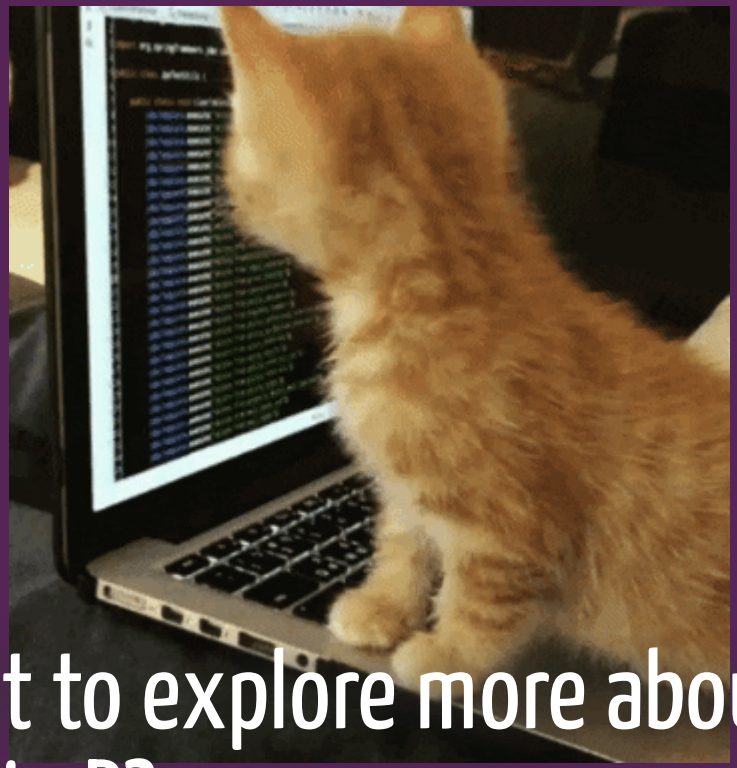
```
bottles_of_beer <- function(i = 99) {
  message(
    "There are ", i, " bottles of beer on the wall, ",
    i, " bottles of beer."
  )
  while(i > 0) {
    tryCatch(
      Sys.sleep(1),
      interrupt = function(err) {
        i <<- i - 1
        if (i > 0) {
          message(
            "Take one down, pass it around, ", i,
            " bottle", if (i > 1) "s", " of beer on the wall."
          )
        }
      }
    )
  }
  message(
    "No more bottles of beer on the wall, ",
    "no more bottles of beer."
  )
}
```

Answer:

If we run that code, we won't be able to stop it unless we kill the process from our terminal

```
[redacted].~$ Rscript beer.R
There are 99 bottles of beer on the wall, 99 bottles of beer.
^CTake one down, pass it around, 98 bottles of beer on the wall.
^CTake one down, pass it around, 97 bottles of beer on the wall.
```

```
1  [|                                                    0.7%] Tasks: 158, 612 thr, 125 kthr; 1 running
2  [                                                     0.0%] Load average: 0.63 1.81 1.88
3  [                                                     0.0%] Uptime: 03:42:12
4  [||                                                   1.3%]
Mem[||||||||||||||||||||||||||||||||||||||||||||||||  2.34G/7.68G]
Swp[                                                     0K/7.88G]
Send signal:   PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 0 Cancel    16231 root       20   0     0     0     0 I  0.0  0.0  0:00.14 kworker/u8:2
 1 SIGHUP    16225 mcastro    20   0 30076  4708  3384 R  1.3  0.1  0:03.53 htop
 2 SIGINT    16215 mcastro    20   0 25704  5352  3440 S  0.0  0.1  0:00.05 bash
 3 SIGQUIT   16203 mcastro    20   0  161M 59228 11276 S  0.0  0.7  0:00.26 /usr/lib/R/bin/exec/R --slave --no-restore -
 4 SIGILL    16192 root       20   0     0     0     0 I  0.0  0.0  0:00.00 kworker/0:3
 5 SIGTRAP   16175 mcastro    20   0 25704  5744  3724 S  0.0  0.1  0:00.08 bash
 6 SIGABRT   16169 mcastro    20   0  714M 41332 30160 S  0.0  0.5  0:00.00 /usr/lib/gnome-terminal/gnome-terminal-serve
 6 SIGIOT    16168 mcastro    20   0  714M 41332 30160 S  0.7  0.5  0:00.47 /usr/lib/gnome-terminal/gnome-terminal-serve
 7 SIGBUS    16167 mcastro    20   0  714M 41332 30160 S  0.0  0.5  0:00.00 /usr/lib/gnome-terminal/gnome-terminal-serve
 8 SIGFPE    16166 mcastro    20   0  714M 41332 30160 S  0.7  0.5  0:06.80 /usr/lib/gnome-terminal/gnome-terminal-serve
 9 SIGKILL   15968 root       20   0     0     0     0 I  0.0  0.0  0:00.03 kworker/3:1
10 SIGUSR1   15967 root       20   0     0     0     0 I  0.0  0.0  0:00.00 kworker/2:0
11 SIGSEGV   15812 root       20   0     0     0     0 I  0.0  0.0  0:00.45 kworker/u8:1
12 SIGUSR2   15743 root       20   0     0     0     0 I  0.0  0.0  0:00.08 kworker/0:0
13 SIGPIPE   15497 root       20   0     0     0     0 I  0.0  0.0  0:00.19 kworker/2:1
14 SIGALRM   15495 root       20   0     0     0     0 I  0.0  0.0  0:00.04 kworker/1:2
15 SIGTERM   15479 root       20   0     0     0     0 I  0.0  0.0  0:00.00 kworker/3:2
```



Do you want to explore more about debugging in R?

- Check 🔗 Jenny Bryan's talk: "Object of type closure is not subsettable"

Don't miss any upcoming meet-ups!

- This RLadies Advanced R Bookclub
- Hadley Wickham's Advanced R Book

Slides created with the R package **xaringan**.