Chapter 8: Conditions

Maya Gans

R4DS Reading Group

Overview

- Signaling conditions
  - Error
  - Warning
  - Message
- Ignoring conditions
  - try
  - suppress Warning/Message
- Handling conditions
  - tryCatch
  - withCallingHandlers

```
brewing_materials <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2020/2020-03-
beer_taxed <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2020/2020-03-31/beer_
brewer_size <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2020/2020-03-31/brew
beer_states <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2020/2020-03-31/bee
```

Signaling Conditions

- Error
- Warning
- Message

Errors

> "An error message should start with a general statement of the problem then give a concise description of what went wrong. Consistent use of punctuation and formatting makes errors easier to parse.
>
> (This guide is currently almost entirely aspirational; most of the bad examples come from existing tidyverse code.)"

https://style.tidyverse.org/error-messages.html

stop

```
beer_mean_error <- function(x) {
  if (!is.numeric(x)) {
    stop("Need numeric column", call.=FALSE)
    mean(which(!is.na(x)[x]))
  } else{
    mean(which(!is.na(x)[x]))
  }
}
```

success

```
beer_mean_error(beer_states$barrels)
```

```
## [1] 806.4551
```

fail

```
beer_mean_error(beer_states$states)
```

```
Error: Need numeric column
   2. stop("Need numeric column", call. = FALSE)
   1. beer_mean(beer_states$state)
```

rlang::abort

```
beer_mean_abort <- function(x) {
  if (!is.numeric(x)) {
    abort(
      message = "Need numeric column",
      arg = x
      )
  } else{
    mean(which(!is.na(x)[x]))
  }
}
```

success

```
beer_mean_abort(beer_states$barrels)
```

```
## [1] 806.4551
```

fail

```
beer_mean_abort(beer_states$state)
```

```
Error: Need numeric column
   4. stop(fallback)
   3. signal_abort(cnd)
   2. abort(message = "Need numeric column", arg = x)
   1. beer_mean(beer_states$state)
```

abort + glue

```
beer_mean_abort_glue <- function(data, x) {

  column_name <- x

  msg <- glue::glue("Can't calculate mean, {column_name} is not numeric")

  if (!is.numeric(data[[x]])) {
    abort(
      message = msg,
      arg = column_name,
      data = data
    )
    mean(which(!is.na(data[[x]])[data[[x]]]))
  } else {
    mean(which(!is.na(data[[x]])[data[[x]]]))
  }
}
```

success

```
beer_mean_abort_glue(beer_states, "barrels")
```

```
## [1] 806.4551
```

fail

```
beer_mean_abort_glue(beer_states, "state")
```

```
Error: Can't calculate mean, "state" is not numeric
Run `rlang::last_error()` to see where the error occurred.
   4. stop(fallback)
   3. signal_abort(cnd)
   2. abort(message = msg, arg = column_name, data = data)
   1. beer_mean(beer_states, "state")
```

abort metadata

```
str(catch_cnd(beer_mean_abort(beer_states, "state")))
```

```
[1] "\"state\""
List of 5
$ message: 'glue' chr "Can't calculate mean, \"state\" is not numeric"
 $ trace  :List of 4
 ...
 $ parent : NULL
$ arg    : chr "\"state\""
$ data   : tibble [1,872 × 4] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
  ..$ state  : chr [1:1872] "AK" "AK" "AK" "AK" ...
  ..$ year   : num [1:1872] 2008 2009 2010 2011 2012 ...
  ..$ barrels: num [1:1872] 2068 2264 1929 2251 2312 ...
  ..$ type   : chr [1:1872] "On Premises" "On Premises" "On Premises" "On Premises" ...
  ..- attr(*, "spec")=
  .. .. cols(
  .. ..   state = col_character(),
  .. ..   year = col_double(),
  .. ..   barrels = col_double(),
  .. ..   type = col_character()
  .. .. )
 - attr(*, "class")= chr [1:3] "rlang_error" "error" "condition"
```

Warnings

> Warnings occupy a somewhat challenging place between messages ("you should know about this") and errors ("you must fix this!")

```
beer_mean_warning <- function(data, x) {

  column_name <- deparse(substitute(x))

  if (is.Date(data[[x]])) {
    warning(glue::glue("Are you sure you wanna calculate the mean? {x} is of type date"))
    mean(data[[x]][which(!is.na(data[[x]]))])
  } else {
    mean(data[[x]][which(!is.na(data[[x]]))])
  }
}

beer_mean_warning(beer_states, "year")
```

```
## Warning in beer_mean_warning(beer_states, "year"): Are you sure you wanna
## calculate the mean? year is of type date
```

```
## [1] "2013-07-02"
```

Not shown: (1) mutating the year column to a date type, (2) creating an `is.Date` function to check for date types. See .Rmd for code

Messages

> Good messages are a balancing act: you want to provide just enough information so the user knows what's going on, but not so much that they're overwhelmed.

```
basic_summary_stats <- function(data, x, round_n = NULL, quiet = FALSE) {

  if (!is.numeric(data[[x]])) abort(glue::glue("Need numeric value to calculate stats and {x} is categorical"))
  if (is.null(round_n)) {

    if (isFALSE(quiet)) message("round_n argument null, rounding to 2 digits by default")

    data %>%
      summarise(
        missing = sum(is.na( data[[x]] )),
        mean = round(mean(which(!is.na(data[[x]])[data[[x]]])), 2)
       )
  } else {
    data %>%
      summarise(
        missing = sum(is.na( data[[x]] )),
        mean = round(mean(which(!is.na(data[[x]])[data[[x]]])), round_n)
       )
  }
}
```

```
basic_summary_stats(beer_states, "barrels")
```

```
## round_n argument null, rounding to 2 digits by default

## # A tibble: 1 x 2
##   missing  mean
##     <int> <dbl>
## 1      19  806.
```

```
basic_summary_stats(beer_states, "barrels", quiet = TRUE)
```

```
## # A tibble: 1 x 2
##   missing  mean
##     <int> <dbl>
## 1      19  806.
```

Ignoring Conditions

- try
- suppressMessage

try

```
beer_mean_try <- function(x) {
  try(beer_mean_abort(x), silent = TRUE)
}
```

success

```
beer_mean_try(beer_states$barrels)
```

```
## [1] 806.4551
```

failure

```
beer_mean_try(beer_states$state)
```

suppressMessages

```
testthat::test_that("beer mean function works", {

  # calculate mean using base and round to two digits
  base_mean <- round(mean(which(!is.na(beer_states$barrels)[beer_states$barrels])), 2)

  # use our function and suppress warning since we're not supplying a rounding argument
  suppressMessages(
    our_function <- basic_summary_stats(beer_states, "barrels") %>% pull(mean)
  )

  # test that they are equal
  testthat::expect_equal(base_mean, our_function)
  # test that our function will produce a message
  testthat::expect_message(basic_summary_stats(beer_states, "barrels") %>% pull(mean))
})
```

My thoughts behind this were that we don't want to clutter the output log when we run all our tests, but is this really best practice?

Handling Conditions

- tryCatch
- withCallingHandlers

The condition system splits the responsibilities into three parts–signaling:

1. a condition
2. handling it
3. restarting

tryCatch

```
beer_mean_tryCatch <- function(expr) {
  tryCatch(
    error = function(cnd) NA,
    {
      glue::glue(
        "Average Beer Barrels Produced: {round(expr,2)}"
      )
    },
    finally = { print("Thank God for Beer!") }
  )
}
```

success

```
beer_mean_tryCatch(beer_mean_abort(beer_states$barrels))
```

```
## [1] "Thank God for Beer!"

## Average Beer Barrels Produced: 806.46
```

failure

```
beer_mean_tryCatch(beer_mean_abort(beer_states$state))
```

```
## [1] "Thank God for Beer!"

## [1] NA
```

I find it interesting that the `finally` statement is printed before the code inside the tryCatch. Can anyone explain why?

tryCatch

We can use tryCatch within the for loop to catch errors without breaking the loop

```
for (indx in 1:ncol(beer_states)) {
    tryCatch(
        expr = {
            basic_summary_stats(beer_states, names(beer_states[indx]))
            message("Iteration ", indx, " successful.")
        },
        error = function(e){
            message("* Caught an error on itertion ", indx)
            print(e)
        }
    )
}
```

```
  Caught an error on itertion 1
<error/rlang_error>
Need numeric value to calculate stats and state is categorical
Backtrace:
 1. base::tryCatch(...)
 5. global::basic_summary_stats(beer_states, names(beer_states[indx]))
  Caught an error on itertion 2
<error/rlang_error>
Need numeric value to calculate stats and year is categorical
Backtrace:
 1. base::tryCatch(...)
 5. global::basic_summary_stats(beer_states, names(beer_states[indx]))
round_n argument null, rounding to 2 digits by default
  Iteration 3 successful.
  Caught an error on itertion 4
<error/rlang_error>
Need numeric value to calculate stats and type is categorical
Backtrace:
 1. base::tryCatch(...)
 5. global::basic_summary_stats(beer_states, names(beer_states[indx]))
```

withCallingHandlers

> R's error handling system lets you separate the code that actually recovers from an error from the code that decides how to recover. Thus, you can put recovery code in low-level functions without committing to actually using any particular recovery strategy, leaving that decision to code in high-level functions.


```
expensive_function <- function(x,
                               # warning print the warning and send us to browser
                               warning = function(w) { print(paste('warning:', w ));  browser() },
                               # error print the error and send us to browser
                               error=function(e) { print(paste('e:',e )); browser()} ) {

  print(paste("big expensive step we don't want to repeat for x:",x))

  z <- x  # the "expensive operation"

  # second function on z that isn't expensive but could potentially error
  repeat
  # put code in here that actually handles the errors
  withRestarts(
    withRestarts(
      withCallingHandlers(
        {
          print(paste("attempt cheap operation for z:",z))
          return(log(z))
        },
        warning = warning,
        error = error
      ),
      # action restart will take
      # if the function fails this will take us to the browser
      # and findRestart will find this name
      force_positive = function() {z <<- -z}
    ),
    # or we can use this is the browser
    set_to_one = function() {z <<- 1}
  )
}
```

http://www.win-vector.com/blog/2012/10/error-handling-in-r/?utm_source=rss&utm_medium=rss&utm_campaign=error-handling-in-r

Using browser

Success

```
expensive_function(2)
```

```
[1] "big expensive step we don't want to repeat for x: 2"
[1] "attempt cheap operation for z: 2"
[1] 0.6931472
```

Fail numeric

```
expensive_function(-2)
```

```
[1] "big expensive step we don't want to repeat for x: -2"
[1] "attempt cheap operation for z: -2"
[1] "warning: simpleWarning in log(z): NaNs produced\n"
Called from: (function(w) { print(paste('warning:', w ));
    browser() })(list(
    message = "NaNs produced", call = log(z)))
```

```
Browse[1]> invokeRestart("force_positive")
```

```
[1] "attempt cheap operation for z: 2"
[1] 0.6931472
```

Fail character

```
expensive_function('a')
```

```
expensive_function('a')
[1] "big expensive step we don't want to repeat for x: a"
[1] "attempt cheap operation for z: a"
[1] "e: Error in log(z): non-numeric
  argument to mathematical function\n"
  Called from: h(simpleError(msg, call))
```

```
Browse[1]> invokeRestart("set_to_one")
```

```
[1] "attempt cheap operation for z: 1"
[1] 0
```

Without `browser`

Get the handler functions to invoke the restart (rather than print error as in example above)

```
force_positive <- function(w) {invokeRestart("force_positive")}
set_to_one <- function(e) {invokeRestart("set_to_one")}

auto_expensive_function = function(x) {
  expensive_function(x, warning=force_positive, error=set_to_one)
}
```

```
auto_expensive_function(2)
```

```
## [1] "big expensive step we don't want to repeat for x: 2"
## [1] "attempt cheap operation for z: 2"

## [1] 0.6931472
```

```
auto_expensive_function(-2)
```

```
## [1] "big expensive step we don't want to repeat for x: -2"
## [1] "attempt cheap operation for z: -2"
## [1] "attempt cheap operation for z: 2"

## [1] 0.6931472
```

```
auto_expensive_function('a')
```

```
## [1] "big expensive step we don't want to repeat for x: a"
## [1] "attempt cheap operation for z: a"
## [1] "attempt cheap operation for z: 1"

## [1] 0
```

> We can use the condition system to allow a low-level function to detect a problem and signal an error, to allow mid-level code to provide several possible ways of recovering from such an error, and to allow code at the highest level of the application to define a policy for choosing which recovery strategy to use.

- `simple_mean`
- `mean_count`
- `mean_or_count`

http://adv-r.had.co.nz/beyond-exception-handling.html

Low level function

```
simple_mean <- function(x) {
  #detecting of a problem
  if(!is.numeric(x)){ #if x is not numeric
    #create condition
    rlang::abort(
      "categorical_column",  #class of condition
      message = "Not sure what to do with categorical column", #message to signal error
      x = x  #metadata
    )
  }
  cat("Returning from simple mean()\n")
  return(mean(x[which(!is.na(x))]))
}
```

success

```
simple_mean(beer_states$barrels)
```

```
Returning from simple mean()
[1] 2286370
```

fail

```
simple_mean(beer_states$state)
```

```
Error: Not sure what to do with categorical column
```

Medium - work around error

```
mean_count <- function(y){

  as_count <- withRestarts( #establish restart
    simple_mean(y) ,
    #create code that recovers from errors in restart categorical_column_restart
    #restart name describes its action
    categorical_column_restart = function(z) {
      plyr::count(z)
    }
    #choosing this restart later in condition handler, will invoke it automatically
    #here you can define various restarts for other recoveries for
    #condition handler to choose from
  )
  cat("Returning from mean_count()\n")
  return(as_count)
}
```

success

```
mean_count(beer_states$barrels)
```

```
## Returning from simple mean()
## Returning from mean_count()

## [1] 2286370
```

fail

```
mean_count(beer_states$state)
```

```
Error: Not sure what to do with categorical column
```

High level function

```
mean_or_count <- function(z){
  as_mean_or_count <- withCallingHandlers(
    #call mean or count function
    mean_count(z),

    # if error occurs function that invokes restart
    error = function(err){

      # if the error is a 'categorical column error'
      if (inherits(err, "categorical_column")) {
        #if object err's class atrribute inherits from class of condition "categorical_column"
        #invoke the restarts called categorical_column_restart
        invokeRestart("categorical_column_restart",
                      #finds restart and invoke it with parameter
                      err$x #argument to pass to restart
        )
      } else {
        #otherwise re-raise the error
        stop(err)
      }
    }
  )
  cat("Returning from mean_or_count()\n")
  return(as_mean_or_count)
}
```

success

```
mean_or_count(beer_states$barrels)
```

```
## Returning from simple mean()
## Returning from mean_count()
## Returning from mean_or_count()

## [1] 2286370
```

fail

```
head(mean_or_count(beer_states$state))
```

```
## Returning from mean_count()
## Returning from mean_or_count()

##    x freq
## 1 AK   36
## 2 AL   36
## 3 AR   36
## 4 AZ   36
## 5 CA   36
## 6 CO   36
```

Other [base] Condition Functions R Documentation	
Function	Definition
signalCondition	Implements the mechanism of searching for an applicable condition handler and invoking its handler function
simpleCondition	default condition class
simpleError	default error class
simpleWarning	default warning class
simpleMessage	default message class
errorCondition	TODO
warningCondition	TODO
conditionCall	return sthe message of a condition
conditionMessage	returns the call of a condition
withRestarts	establishes recovery protocols
computeRestarts	returns a list of all restarts
findRestart	returns the most recently established restart of the specified name
invokeRestart	a way to specify how to handle errors and warnings
invokeRestartInteractively	TODO
isRestart	TODO: check if object is a restart function
restartDescription	TODO
restartFormals	TODO
suspendInterrupts	TODO
allowInterrupts	TODO