Session 10
Debugging 2

R for Social Data Science

Jeffrey Ziegler, PhD

Assistant Professor in Political Science & Data Science
Trinity College Dublin

Fall 2022

Road map for today

Last time:

- Software bugs
- Debugging

This time:

- Handling conditions
- Testing
- Defensive programming

Conditions

- Conditions are **events** that signal special situations during execution
- Some conditions can modify the control flow of a program (e.g. error)
- They can be *caught* and *handled* by your code
- You can also incorporate condition triggers into your code

Extra: Hadley Wickham - Conditions

CONDITIONS EXAMPLES

```r
42 + "ab" # Throws an error
```

```
Error in 42 + "ab": non-numeric argument to binary operator
```

```r
as.numeric(c("42","55.3","ab","7")) # Triggers a warning
```

```
Warning message in eval(expr, envir, enclos):
"NAs introduced by coercion"
```

```r
library("dplyr") # Shows a message
```

```
Attaching package: ‘dplyr’

The following objects are masked from ‘package:stats’:

filter, lag

The following objects are masked from ‘package:base’:

intersect, setdiff, setequal, union

Warning message:
package ‘dplyr’ was built under R version 4.1.2
```

Conditions examples

```r
stop("Error message")
```

```
Error: Error message
```

```r
warning("Warning message")
```

```
Warning message:
Warning message
```

```r
message("Message")
```

```
Message
```

Errors

- In R errors are signaled (or thrown) with ‘stop()‘
- By default, error message includes call
- Program execution stops once an error is raised

```r
if (c(TRUE, TRUE, FALSE)) {
  print("This used to work pre R-4.2.0")
}
```

```
[1] "This used to work pre R-4.2.0"
Warning message:
In if (c(TRUE, TRUE, FALSE)) { :
the condition has length > 1 and only the first element
will be used
```

Warnings

- Weaker versions of errors:
  - Something went wrong, but the program has been able to recover and continue
- Single call in result in multiple warnings (as opposed to a single error)
- Take note of the warnings resulting from base R operations
- Some of them might eventually become errors

```r
# Will become an error in future versions of R
c(TRUE, FALSE) && c(TRUE, TRUE)
```

```
Warning message in c(TRUE, FALSE) && c(TRUE, TRUE):
“’length(x) = 2 > 1’ in coercion to ’logical(1)’”
Warning message in c(TRUE, FALSE) && c(TRUE, TRUE):
“’length(x) = 2 > 1’ in coercion to ’logical(1)’”
[1] TRUE
```

MESSAGES

- Messages serve mostly informational purposes
- They tell the user:
  - that something was done
  - the details of how something was done
- Sometimes these actions are not anticipated
- Useful for functions with side-effects (accessing server, writing to disk, etc.)

```r
# Will become an error in future versions of R
c(TRUE, FALSE) && c(TRUE, TRUE)
```

```
New names:
• ‘‘ -> ‘...1‘
Rows: 3193 Columns: 20
- Column specification -
Delimiter: ","
dbl (20): ...1, x, year, congress, chalspend, incspend, difflog,
presvote, voteshare, inparty, incparty, seniority,
midterm, chalquality, south, population, urban, age65,
milpop, unemployed

Use ‘spec()‘ to retrieve the full column specification for this data.
Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

Handling conditions

- Every condition has default behaviour:
  - Errors terminate program execution
  - Warnings are captured and displayed in aggregate
  - Message are shown immediately
- But with condition **handlers** we can override the default behaviour

Ignoring conditions

- The simplest way of handling conditions is **ignoring** them
- Heavy-handed approach, given type of condition does not make further distinctions
- Bear in mind risks of ignoring information (especially, errors!)
- Functions for handling conditions depend on the type of condition:
  - ‘try()‘ for errors
  - ‘suppressWarnings()‘ for warnings
  - ‘suppressMessages()‘ for messages

```r
# suppressPackageStartupMessages() - a variant for package startup messages
# But suppressMessages() would also work
suppressPackageStartupMessages(library("dplyr"))
```

Ignoring errors

```r
f1 <- function(x) {
  log(x)
      10
}
f1("x")
```

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

```r
f2 <- function(x) {
  try(log(x))
      10
}
f2("y")
```

```
Error in log(x) : non-numeric argument to mathematical function
[1] 10
```

Condition handlers

- More advanced approach to dealing with conditions is providing handlers
- They allow to override or supplement the default behaviour
- In particular, two function can:
  - ‘tryCatch()‘ define *exiting* handlers
  - ‘withCallingHandlers()‘ define *calling* handlers

```
tryCatch(
error = function(cnd) {
# code to run when error is thrown
},
code_to_run_while_handlers_are_active
)
```

```
withCallingHandlers(
warning = function(cnd) {
# code to run when warning is signalled
},
message = function(cnd) {
# code to run when message is signalled
},
code_to_run_while_handlers_are_active
)
```

Exiting handlers

- The handlers set up by ‘tryCatch()‘ are called exiting handlers
- After the condition is signalled, control flow passes to the handler
- It never returns to the original code, effectively meaning that the code exits

```r
f3 <- function(x) {
  tryCatch(
        error = function(e) NA,
    log(x)
      )
}
f3("x")
```

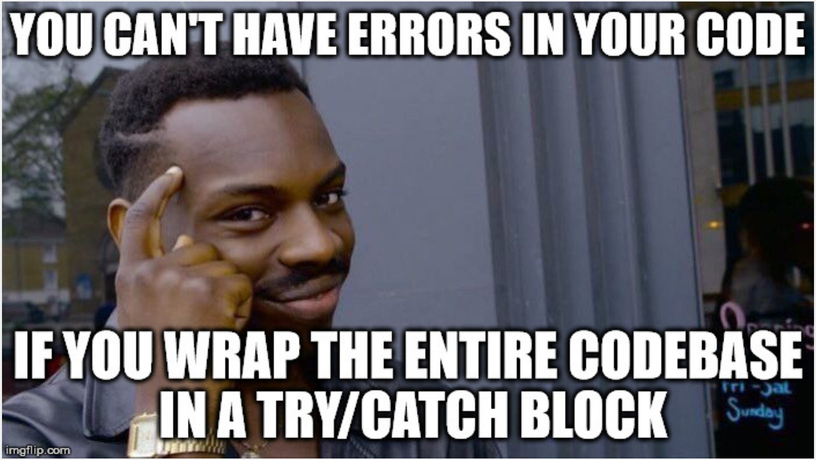
```
[1] NA
```

Calling handlers

- With calling handlers code execution continues normally once the handler returns
- A more natural pairing with the non-error conditions

```
# Infinite loop, analogous to while (TRUE)
repeat {
      num <- readline("Please, enter a number:")
      if (num != "") {
        withCallingHandlers(
      warning = function(cnd) {
            print("This is not a number. Please, try again.")
          },
      num <- as.numeric(num)
        )
  } else {
        print("No input provided. Please, try again.")
      }
  if (!is.na(num)) {
        print(pasteo("Your input ’", as.character(num), "’ is recorded"))
        break
  }
}
```

Discretion in condition handling


YOU CAN'T HAVE ERRORS IN YOUR CODE
IF YOU WRAP THE ENTIRE CODEBASE IN A TRY/CATCH BLOCK
imgflip.com

Expectations

- When designing a function you built in certain expectations about:
    - Acceptable inputs
    - Conditions triggered
    - Returned object
- Checking inputs at beginning helps fail fast

```r
calculate_median <- function(a) {
  if (!is.numeric(a)) {
      stop("Vector must be numeric")
    }
  a <- sort(a)
    n <- length(a)
  m <- (n + 1) %/% 2
  if (n %% 2 == 1) {
    med <- a[m]
    } else {
    med <- mean(a[m:(m+1)])
    }
  return(med)
}
```

Checking expectations

- What if we want to check whether our function’s behaviour matches our expectations?
- One option would be to use ‘==‘ (or ‘!=‘)
- However, for numerical values it can be problematic:

```
v3 <- c(7.22, 1.54, 3.47, 2.75)
calculate_median(v3)
```

```
[1] 3.11
```

```
v3 <- c(7.22, 1.54, 3.47, 2.75)
calculate_median(v3)
```

```
[1] FALSE
```

```
v3 <- c(7.22, 1.54, 3.47, 2.75)
calculate_median(v3)
```

```
[1] TRUE
```

Checking expectations

- A better way to compare values where a single ‘TRUE‘ or ‘FALSE‘ is expected is to use special functions:
  - ‘all.equal()‘ - approximately equal
  - ‘identical()‘ - exactly identical (incl. type)
  - ‘isTRUE()‘ - whether value is ‘TRUE‘
  - ‘isFALSE()‘ - whether value is ‘FALSE‘

```r
all.equal(length(calculate_median(v3)), 1)
```

```
[1] TRUE
```

```r
# Note that the output of length is of type integer
identical(length(calculate_median(v3)), 1)
identical(length(calculate_median(v3)), 1L)
```

```
[1] FALSE
```

```
[1] TRUE
```

Formalising expectations checks: Testing

- Process of running a program on pre-determined cases to ascertain that its functionality is consistent with expectations
- Test cases consist of different assertions (of equality, boolean values, etc.)
- Fully-featured unit testing framework in R is provided in ‘testthat‘ library

Extra: Hadley Wickham - Testing

Testing examples

```r
library("testthat")
calculate_median <- function(a) {
  if (!is.numeric(a)) {
        stop("Vector must be numeric")
      }
  a <- sort(a)
      n <- length(a)
  m <- (n + 1) %/% 2
  if (n %% 2 == 1) {
    med <- a[m]
      } else {
    med <- mean(a[m:(m+1)])
      }
  return(med)
}
```

Testing examples

```r
testthat::test_that("The length of result is 1", {
      testthat::expect_equal(
    length(calculate_median(c(0, 1, 2, 2))),
        1L
  )
      testthat::expect_equal(
    length(calculate_median(c(1, 2, 3))),
        1L
  )
      testthat::expect_equal(
    length(calculate_median(c(7.22, 1.54, 3.47, 2.75))),
        1L
  )
})
```

```
Test passed
```

Testing examples

```r
testthat::test_that("Error on non-numeric input", {
      testthat::expect_error(
    calculate_median(c("a", "bc", "xyz"))
      )
  testthat::expect_error(
        calculate_median(c(TRUE, FALSE, FALSE))
  )
      testthat::expect_error(
    calculate_median(c("0", "1", "2", "2"))
  )
})
```

```
Test passed
```

Testing examples

```r
1  testthat::test_that("The result is numeric", {
2        testthat::expect_true(
3      is.numeric(calculate_median(c(0, 1, 1, 2)))
4        )
5    testthat::expect_true(
6          is.numeric(calculate_median(c(1, 2, 3)))
7      )
8        testthat::expect_true(
9      is.numeric(calculate_median(c("a", "bc", "xyz")))
10       )
11 })
```

```
- Error (Line 8): The result is numeric -
Error in ‘calculate_median(c("a", "bc", "xyz"))‘: Vector must be numeric
Backtrace:
1. testthat::expect_true(...)
4. global calculate_median(c("a", "bc", "xyz"))

Error in reporter$stop_if_needed() : Test failed
```

Defensive programming

- Design your program to facilitate earlier failures, testing and debugging
- Make code fail fast and in well-defined manner
- Split up different componenets into functions or modules
- Be strict about accepted inputs, use assertions or conditional statements to check them
- Document assumptions and acceptable inputs using docstrings
- Document non-trivial, potentially problematic and complex parts of code

Tutorial - Debugging a function

- Look at the problematic ‘calculate_sd‘ function
- Run R debugger and step through it
- While inside function print out values of deviations and result of stand_dev
- Fix bug(s)

Tutorial - Testing

- Create tests for `pearson()` and `calculate_median()` functions that test:
  - Whether the sign of a calculated pearson correlation is correct
  - Whether median calculated on an array with even number of elements has an absolute difference of no more than 0.0001 from correct answer

Overview

This week:

- Software bugs
- Debugging
- Handling conditions
- Testing
- Defensive programming

Next week:

- Data wrangling