Error handling and debugging in R

Ákos Bede-Fazekas

2022.10.20

Preparation

Please

- download the code (R_errors_ABF.r) and the presentation (R_errors_ABF.pdf) from **tinyurl.com/talentia-errors**
- and (if you want) open the code in RStudio (or in your preferred code editor)
- and run the code (with Ctrl+Enter) that is shown in the presentation

Also, please set the working directory by `setwd()` to point to the location of the downloaded files.

Main topics

- throwing errors, warnings, messages
- suppressing and handling errors, warnings, messages
- call stack
- type of errors, finding bugs
- interactive and non-interactive debugging (briefly)

Preface

Errors (and warnings) are frustrating and we hate them...
However, this is the way R can communicate with the user.
Communication is **good**.

Communication with the user

Ther are 3 types/levels of communication:

- message (simple information)
- warning (sign of a possible problem, the user can evaluate whether it is an issue or not)
- error (a problem that cannot be skipped)

Warnings and errors are called exceptions.

Communication with the user

Message

- rarely used
- e.g. when loading a package with `library()`
- can be ignored

Warning

- needs decision by the user
- thrown after the script is terminated

Error

- needs no decision, must be solved
- thrown inmediately
- running is stopped

Communication with the user

We (the script-writers) can communicate with the user (= the future-us) through R.

- message: `message()`
- warning: `warning()`
- error: `stop()`

Each of them can be called with or without parameter.

Communication with the user

```
message()
>
warning()
>  Warning:
stop()
>  Error in eval(expr, envir, enclos):
```

Communication with the user

```
message("This is a simple information. You can ignore
 this.")
>  This is a simple information. You can ignore this.
warning("Well, it is strange. You might want to fix this.")
>  Warning: Well, it is strange. You might want to fix this.
stop("Oh no. I must stop running here.")
> Error in eval(expr, envir, enclos): Oh no. I must stop
 running here.
```

Communication with the user

Typically we throw warnings/error in some specific cases (within `if` statements).

```
divide <- function(a, b) {
    if (b == 0) stop("You cannot divide by 0.")
    else if (b == 1) warning("There might be a typo
     here. There is no reason to divide by 1.")
    return(a / b)
}
```

Communication with the user

```
divide(5, 2)
>  [1] 2.5
divide(5, 1)
> Warning in divide(5, 1): There might be a typo here.
 There
>  is no reason to divide by 1.
>  [1] 5
divide(5, 0)
>  Error in divide(5, 0): You cannot divide by 0.
```

Scaling up warnings to errors

Warnings

- are collected and displayed after the script is terminated
- only 50 warnings are stored by default (can be increased by e.g. `options(nwarnings = 1000)`)
- only the last 50 warnings can be displayed by `warnings()`
- we might be interested in the first warning (which is hidden from us)

Scaling up warnings to errors

```
for (i in 1:60) {
    if (i == 60) warning("60 is an ugly number.")
    warning("I don't like for loops. Please use
     sapply() instead.")
}
```

```
> There were 50 or more warnings (use warnings() to see the
first 50)
```

Scaling up warnings to errors

```
length(last.warning)
>   [1] 50
```

But we know we have 60+1 warnings!!

Scaling up warnings to errors

Sometimes we should check (and solve) each warning one by one.
We can display only the first 50 warnings...

```
warnings()
>  Warning messages:
>  1: In eval(expr, envir, enclos) :
>    I don't like for loops. Please use sapply() instead.
>  2: In eval(expr, envir, enclos) :
>    I don't like for loops. Please use sapply() instead.
>  3: In eval(expr, envir, enclos) :
>    I don't like for loops. Please use sapply() instead.
>  4: In eval(expr, envir, enclos) :
>    I don't like for loops. Please use sapply() instead.
>  5: In eval(expr, envir, enclos) :
>    I don't like for loops. Please use sapply() instead.
>  6: In eval(expr, envir, enclos) :
>    I don't like for loops. Please use sapply() instead.
>  7: In eval(expr, envir, enclos) :
>    I don't like for loops. Please use sapply() instead.
>  8: In eval(expr, envir, enclos) :
```

Scaling up warnings to errors

Solution: we can turn every warnings to errors.
Hence, the script will stop when a warning (= error) is thrown.

- upscale: `options(warn = 2)`
- restore the default: `options(warn = 0)`

Scaling up warnings to errors

```
options(warn = 2)
for (i in 1:10) {
    warning("I found a possible problem.")
}
> Error in eval(expr, envir, enclos): (converted from
 warning) I found a possible problem.
print(i)
>  [1] 1
```

The `for` loop stopped at the 1st iteration.

Scaling up warnings to errors

```
options(warn = 0)
```

Restored to the default (warnings are only warnings).

Suppressing errors, warnings, messages

Sometimes we are not interested in R's communication attempts.
We can ignore messages and warnings.
But we can also suppress (= hide from the screen) them.

Suppressing errors, warnings, messages

Three main functions:

- `suppressMessages()`: suppress all texts thrown by `meassage()`
- `suppressPackageStartupMessages()`: suppress messages thrown when loading packages with `library()`
- `suppressWarnings()`: suppress all texts thrown by `warning()`

We can embrace the original function call.
The result of the call is re-returned by these suppressing functions.

Suppressing errors, warnings, messages

```
x <- divide(5, 1)
> Warning in divide(5, 1): There might be a typo here.
 There
>  is no reason to divide by 1.
print(x)
>  [1] 5
y <- suppressWarnings(divide(5, 1))
print(y)
>  [1] 5
```

Suppressing errors, warnings, messages

```
hello <- function(name) {
    name_uppercase <- toupper(name)
    message(paste0("Hello ", name_uppercase))
    return(name_uppercase)
}
```

Suppressing errors, warnings, messages

```
hello("Mary")
>  Hello MARY
>  [1] "MARY"
suppressMessages(hello("Joe"))
>  [1] "JOE"
```

Suppressing errors, warnings, messages

Why suppressing? E.g. to get visually pleasing output.

```
sum_to_n <- function(n) {
    s <- 0
    writeLines("Calculation is in progress", sep = "")
    for (i in 1:n) {
        Sys.sleep(0.5) # wait 0.5 sec
        s <- s + i
        message(paste0("i = ", as.character(i)))
        writeLines(".", sep = "")
    }
    writeLines(" done")
    return(s)
}
```

Suppressing errors, warnings, messages

```
sum_to_n(5)
>   Calculation is in progress
>   i = 1
>   .
>   i = 2
>   .
>   i = 3
>   .
>   i = 4
>   .
>   i = 5
>   . done
>   [1] 15
```

Suppressing errors, warnings, messages

```
suppressMessages(sum_to_n(5))
>  Calculation is in progress..... done
>  [1] 15
```

Suppressing errors, warnings, messages

```
# install.packages("sf")
library(sf)
>  Linking to GEOS 3.8.0, GDAL 3.0.4, PROJ 6.3.1
detach("package:sf", unload = TRUE) # unload package
suppressPackageStartupMessages(library(sf))
```

Suppressing errors, warnings, messages

`suppressMessages()` does not suppress all text information.

```
hello2 <- function(name) {
    name_uppercase <- toupper(name)
    text_to_display <- paste0("Hello ", name_uppercase)
    message(text_to_display)
    print(text_to_display)
    cat(text_to_display, "\n")
    writeLines(text_to_display)
    return(name_uppercase)
}
```

Suppressing errors, warnings, messages

```
result <- suppressMessages(hello2("Ann"))
>  [1] "Hello ANN"
>  Hello ANN
>  Hello ANN
print(result)
>  [1] "ANN"
```

Suppressing errors, warnings, messages

If we want to suppress almost all text information from the screen, we must `sink()` the text outputs from the default display device (= screen) to

- a permanent text file (we are interested in the output, but do not want to see on the screen),
- a temporary text file (we are not interested in the output), or
- nowhere (in Unix-like operating systems: `"/dev/null"`; we are not interested in the output).

I do not show here, please call `?sink` to get information.

Suppressing errors, warnings, messages

There is no way to suppress **all** texts. (Or at least I haven't found a solution yet.)
Some packages/functions push text to the screen from C or system calls.
I hate them!

Handling exceptions

Errors cannot be suppressed in the way described above.
But errors (and warnings) can be handled.
Handling error = catching the error and, if possible, resume the program running.

Handling exceptions

Errors can be handled by `try()` and `tryCatch()`.

- `try(expr, silent = FALSE)`
- `tryCatch(expr, error, warning, finally)`

I prefer `tryCatch()` (it is more flexible).

Handling exceptions - try()

`try(expr)` tries to evaluate the expression `expr`.

- if succeeds: returns the result of the evaluated expression
- if fails: returns the error (which is special object containing the type and the error message)

Typically, we call `try()` and then check its result (whether it is a child of class `try-error`):

```
result <- try(expr)
if (inherits(result, "try-error")) ...
```

Handling exceptions - try()

```
default_value <- 3
user_supplied_value <- "a"
result <- try(expr = log(user_supplied_value))
>  Error in log(user_supplied_value) :
>    non-numeric argument to mathematical function
```

Error is printed to the screen (because parameter `silent` of `try()` was set to the default, `FALSE`).
But the running of the program is not stopped.
The variable `result` can be a number or an error-object.

Handling exceptions - try()

```
if (inherits(result, "try-error")) {
    result <- log(default_value)
    warning(paste0("The value you provided (",
     as.character(user_supplied_value), ") was
     replaced by the default value (",
     as.character(default_value), ")."))
}
>  Warning: The value you provided (a) was replaced by the
>  default value (3).
```

Handling exceptions - try()

If we do not want to see the error message, set parameter `silent` to `TRUE`:

```
result <- try(expr = log("a"), silent = TRUE)
print(result)
> [1] "Error in log(\"a\") : non-numeric argument to
 mathematical function\n"
>  attr(,"class")
>  [1] "try-error"
>  attr(,"condition")
> <simpleError in log("a"): non-numeric argument to
 mathematical function>
```

Handling exceptions - try()

The expression can contain multiple statements.
Embrace them with {}.

```
result <- try(expr = {
    x <- "a"
    log(x)
}, silent = TRUE)
```

Handling exceptions - tryCatch()

`tryCatch()` is a bit more sophisticated than `try()`.
It has 4 main parameters:

- `expr`: the expression to be evaluated
- `error`: a one-parameter function that will be called if the expression throws an error
- `warning`: a one-parameter function that will be called if the expression throws a warning
- `finally`: some lines of code (i.e. an expression) to be run after `expr` and `error`/`warning` were evaluated

Parameters `error`, `warning` and `finally` are optional.

Handling exceptions - tryCatch()

Typical usage of `tryCatch()`:

```
tryCatch(
    expr = {...},
    error = function(error_caught) {...},
    finally = {...}
)
```

or:

```
tryCatch(
    expr = {...},
    error = function(error_caught) {...},
    warning = function(warning_caught) {...},
    finally = {...}
)
```

Handling exceptions - tryCatch()

```
value <- "five"
tryCatch(
    expr = {x <- 3 + value},
    error = function(error_caught) {
        x <- NA
        writeLines(paste0("A problem occured: ",
         as.character(error_caught)))
    },
    finally = {print(x)}
)
> A problem occured: Error in 3 + value: non-numeric
 argument to binary operator
>
>  [1] "a"
```

Handling exceptions - tryCatch()

```
value <- "five"
tryCatch(
    expr = {x <- 3 + value},
    error = function(error_caught) {
        x <- NA
        writeLines(paste0("A problem occured: ",
         as.character(error_caught)))
    },
    finally = {print(x)}
)
```

The parameter of the error-handling function contains the original error message.
It is not displayed unless we force it.

Handling exceptions - tryCatch()

```
value <- 5
tryCatch(
    expr = {x <- 3 + value},
    error = function(error_caught) {
        x <- NA
        writeLines(paste0("A problem occured: ",
         as.character(error_caught)))
    },
    finally = {print(x)}
)
>  [1] 8
```

The expression defined by parameter `finally` is always evaluated.

Handling exceptions - tryCatch()

Evaluation of `expr` is stopped when

- the first handled error or warning is caught, or
- an unhandled error occurs (if parameters `error` and `warning` are not set)
- at the end (if no errors occured or no warning was caught)

Handling exceptions - tryCatch()

Warning is not caught:

```
tryCatch(
    expr = {
        for (i in 3:0) {
            print(divide(5, i))
        }
    },
    error = function(error_caught)
     {writeLines(paste0("Error found: ",
     as.character(error_caught)))}
)
```

Handling exceptions - tryCatch()

```
>  [1] 1.666667
>  [1] 2.5
> Warning in divide(5, i): There might be a typo here.
 There
>  is no reason to divide by 1.
>  [1] 5
> Error found: Error in divide(5, i): You cannot divide by
 0.
```

Handling exceptions - tryCatch()

Warning is caught, so we do not reach the error:

```
tryCatch(
    expr = {
        for (i in 3:0) {
            print(divide(5, i))
        }
    },
    error = function(error_caught)
     {writeLines(paste0("Error found: ",
     as.character(error_caught)))},
    warning = function(warning_caught)
     {writeLines(paste0("Warning found: ",
     as.character(warning_caught)))}
)
```

Handling exceptions - tryCatch()

```
>  [1] 1.666667
>  [1] 2.5
> Warning found: simpleWarning in divide(5, i): There
 might be a typo here. There is no reason to divide by 1.
```

Handling exceptions - tryCatch()

An unhandled error occurs (parameter `error` was not set):

```
tryCatch(
    expr = {
        for (i in 3:0) {
            print(divide(5, i))
        }
    }
)
>  [1] 1.666667
>  [1] 2.5
> Warning in divide(5, i): There might be a typo here.
 There
>  is no reason to divide by 1.
>  [1] 5
>  Error in divide(5, i): You cannot divide by 0.
```

Handling exceptions - tryCatch()

If no errors occur:

```
tryCatch(
    expr = {
        for (i in 5:2) {
            print(divide(5, i))
        }
    }
)
>   [1] 1
>   [1] 1.25
>   [1] 1.666667
>   [1] 2.5
```

Handling exceptions - tryCatch()

If we need all the errors/warnings to be caught, we should place the `tryCatch()` at low level:

```
for (i in 3:0) {
    tryCatch(
        expr = {
            print(divide(5, i))
        },
        error = function(error_caught)
         {writeLines(paste0("Error found: ",
         as.character(error_caught)))},
        warning = function(warning_caught)
         {writeLines(paste0("Warning found: ",
         as.character(warning_caught)))}
    )
}
```

Handling exceptions - tryCatch()

```
>  [1] 1.666667
>  [1] 2.5
> Warning found: simpleWarning in divide(5, i): There
 might be a typo here. There is no reason to divide by 1.
>
> Error found: Error in divide(5, i): You cannot divide by
 0.
```

Handling exceptions - tryCatch()

`tryCatch()` returns (invisibly)

- the last evaluated statement within `expr` (if no errors occur)
- the returned value of the error/warning handling functin (if error/warning occured)

We rarely use the returned value.

Handling exceptions - tryCatch()

No error occurs; last evaluated expression (`x * 100`) is returned:

```
result <- tryCatch(
    expr = {
        x <- divide(5, 3)
        x * 100
    },
    error = function(error_caught) {
        writeLines(paste0("Error found: ",
         as.character(error_caught)))
        return(NA)
    }
)
print(result)
>  [1] 166.6667
```

Handling exceptions - tryCatch()

Error caught; error-handling function returns the result of `tryCatch()`:

```
result <- tryCatch(
    expr = {
        x <- divide(5, 0)
        x * 100
    },
    error = function(error_caught) {
        writeLines(paste0("Error found: ",
         as.character(error_caught)))
        return(NA)
    }
)
> Error found: Error in divide(5, 0): You cannot divide by
 0.
print(result)
>  [1] NA
```

Handling exceptions - tryCatch()

When should we use finally?

- if the `expr` starts something
- that should be stopped/finalized
- regardless of whether we have caught an error or not

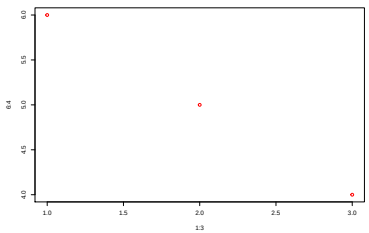
Examples:

- a (file) connection opened by `open()` for writing should be closed by `close()`
- a printing device opened by `png()`, `jpeg()` etc. should be closed by `dev.off()`
- a folder or file should be deleted by `unlink()` or `file.remove()`
- `sink()`ing of outputs to file/nowhere should be stopped by `sink(NULL)`
- original (graphical) options should be restored by `options()` or `par()`

Handling exceptions - tryCatch()

Let's see a typical example.
Original code:

```
png(file = "plot1.png", width = 400, height = 400)
plot(x = 1:3, y = 6:4, col = "red")
dev.off()
>  pdf
>    2
```



But what happens when an error occurs within `plot()`?
Script is stopped, and the device is not closed. (We cannot open the png file.)

Handling exceptions - tryCatch()

Embraced by `tryCatch()`, no error:

```
tryCatch(
    expr = {
        png(file = "plot1.png", width = 400, height = 400)
        plot(x = 1:3, y = 6:4, col = "red")
    },
    finally = {
        dev.off()
    }
)
```

Does the same, although it is a bit longer...

Handling exceptions - tryCatch()

Embraced by `tryCatch()`, error within `plot()`:

```
tryCatch(
    expr = {
        png(file = "plot2.png", width = 400, height = 400)
        plot(x = 1:3, y = 6:1, col = "red")
    },
    finally = {
        dev.off()
    }
)
> Error in xy.coords(x, y, xlabel, ylabel, log): 'x' and
 'y' lengths differ
```

Call stack

Function `f()` can call function `g()`, which can call function `h()`, and so on.
In R, high-level functions regularly call low-level functions through a sequence of intermediate-level functions.
"Call stack": the sequence of the functions calling each other.
(Stack is a special data type used in different programming languages.)

Call stack

```
high_level <- function() {
    intermediate_level_1()
}
intermediate_level_1 <- function() {
    intermediate_level_2()
}
intermediate_level_2 <- function() {
    low_level()
}
low_level <- function() {
    message("Hello.")
}
```

Call stack

If we call function `high_level()`, we'll get some message from function `low_level()`:

```
high_level()
>  Hello.
```

Call stack

R is known to throw error messages

- that are hard to understand
- IMHO, they are not hard to understand
- but they are not thrown in the optimal level.

Usually we are familiar with the high-level function

- we know it's name
- we know its parameterization
- we know its desired behavior etc.

And we usually do not know which intermediate and low-level functions are called in the background and how they work.

Call stack

But the errors usually occur at low level. . .
Hence we got an error message that is totally unexpected and ununderstandable.

Call stack

Let's see it in a toy example:

```
high_level <- function() {
    intermediate_level_1()
}
intermediate_level_1 <- function() {
    intermediate_level_2()
}
intermediate_level_2 <- function() {
    low_level()
}
low_level <- function() {
    stop("Some problems are found here.")
}
```

Call stack

```
high_level()
>  Error in low_level(): Some problems are found here.
```

We are familiar with function `high_level()`.
What is function `low_level()`?! We never called it! Why does it throw an error?

Call stack

Solution: we can print the call stack right after the error occurs by `traceback()`.

```
high_level()
traceback()
```

```
>  Error in low_level(): Some problems are found here.
>  5: high_level()
>  4: intermediate_level_1()
>  3: intermediate_level_2()
>  2: low_level()
>  1: stop("Some problems are found here.")
```

Call stack

Even basic functions like `print()` calls some low-level functions:

```
plot(x = 1:3, y = 6:1, col = "red")
traceback()
```

```
> Error in xy.coords(x, y, xlabel, ylabel, log): 'x' and
 'y' lengths differ
>  4: plot(x = 1:3, y = 6:1, col = "red")
>  3: plot.default(x = 1:3, y = 6:1, col = "red")
>  2: xy.coords(x, y, xlabel, ylabel, log)
>  1: stop("'x' and 'y' lengths differ")
```

Call stack

```
wrong_function <- function() {
    df <- data.frame(matrix(data = 1:10, nrow = 5, ncol = 2))
    return(lapply(X = colnames(df), FUN =
     function(column_name) {
        sapply(X = df[[column_name]], FUN = function(value) {
            value + "a" # error will be thrown here
        })
    }))
}
```

Call stack

```
wrong_function()
traceback()
```

```
> Error in value + "a": non-numeric argument to binary
 operator
>  6: wrong_function()
>  5: lapply(X = colnames(df), FUN = function(column_name) {
> sapply(X = df[[column_name]], FUN = function(value) {
>             value + "a"
>         })
>     })
>  4: FUN(X[[i]], ...)
>  3: sapply(X = df[[column_name]], FUN = function(value) {
>         value + "a"
>     })
>  2: lapply(X = X, FUN = FUN, ...)
>  1: FUN(X[[i]], ...)
```

Call stack

Hint: if you are going to create your own R function or package that you want to share with others, please:

- write the function/package in a foolproof way
- check every possible problems regarding the input parameters (length, type, `NA`s etc.)
- and `stop()` the function at high level with informative error message (including solution for the error)
- or at least provide some `warning()`s if the inputs might cause error at lower level
- never cause a situation that the user must call `traceback()` to understand the reason your function is not working!

Be aware of Murphy's low: Anything that can go wrong will go wrong. Future users will surely call your function with ill-shaped/wrong-type/erroneous inputs!

Call stack

Tracing back the propagation of the error through the call stack with `traceback()` is extremely useful when

- we have a script/function containing several lines of code
- and we de not know which line causes the error
- since the error message does not give information about the location

When a code is called from a script file (.r), then `traceback()` provide the line number.

Call stack

Let's see an example from a package available on CRAN.
Package `dismo` is used for species distribution modeling.
It has some high-level functions for modeling that call the low-level function `.roc()` for calculation a goodness-of-fit measure of the model.
The source code is available here: rdrr.io/cran/dismo/src/R/gbm.utils.R

Call stack

Let's import the function by `source()`ing its source file:

```
source("gbm.utils.r")
```

Function `.roc()` works quite well for datasets (100 $<$ size $<$ 100,000) typically used for distribution modeling:

```
size <- 1000
.roc(obsdat = sample(x = 0:1, size = size, replace =
 TRUE), preddat = runif(n = size, min = 0, max = 1))
>  [1] 0.4761
```

Call stack

But the package was not tested for really large datasets, when a bug occurs:

```
size <- 1000000
.roc(obsdat = sample(x = 0:1, size = size, replace =
 TRUE), preddat = runif(n = size, min = 0, max = 1))
>  Warning in n.x * n.y: NAs produced by integer overflow

>  Warning in n.x * n.y: NAs produced by integer overflow
>  [1] NA
```

NA value is produced with some warnings.

Call stack

Let's turn warnings to errors and call `traceback()` to locate the bug.

```
options(warn = 2)
size <- 1000000
.roc(obsdat = sample(x = 0:1, size = size, replace =
 TRUE), preddat = runif(n = size, min = 0, max = 1))
traceback()
```

```
> Error in n.x * n.y: (converted from warning) NAs
 produced by integer overflow
> 5: .roc(obsdat = sample(x = 0:1, size = size, replace =
 TRUE), preddat = runif(n = size,
>         min = 0, max = 1))
> 4: .signalSimpleWarning("NAs produced by integer
 overflow", base::quote(n.x *
>         n.y))
>  3: withRestarts({
> .Internal(.signalCondition(simpleWarning(msg, call),
 msg,
>             call))
```

Call stack

```
2: .signalSimpleWarning("NAs produced by integer overflow",
base::quote(n.x * n.y)) at gbm.utils.r#26
```

The bug is located, it is somewhere in (or before) line 26 in gbm.utils.r.
So we can start interactive debugging (later...).
(INFO: an `as.numeric()` was missing from the function that could convert integer to floating-point number...)
Now we restore the default warn option:

```
options(warn = 0)
```

Call stack

Functions alternative to `traceback()` are:

- `rlang::last_error()`
- `rlang::last_trace()`

Call stack

When I encounter an unexpected error, my first step is used to be calling `traceback()`.

Call stack

We can ask R that it automatically calls `traceback()` when an error occurs by `options(error)`.

```
options(error = traceback)
plot(x = 1:3, y = 6:1, col = "red")
```

```
> Error in xy.coords(x, y, xlabel, ylabel, log): 'x' and
 'y' lengths differ
>  4: plot(x = 1:3, y = 6:1, col = "red")
>  3: plot.default(x = 1:3, y = 6:1, col = "red")
>  2: xy.coords(x, y, xlabel, ylabel, log)
>  1: stop("'x' and 'y' lengths differ")
```

An error occured, hence, `traceback()` is called.

Call stack

To restore the default way errors are handled:

```
options(error = NULL)
plot(x = 1:3, y = 6:1, col = "red")
```

```
> Error in xy.coords(x, y, xlabel, ylabel, log): 'x' and
 'y' lengths differ
```

Call stack

The call stack is not always useful. It is uninformative in the case of

- multisession/multicore (parallel) processing
- some tidyverse functions, like `ggplot()`
- etc.

Type of errors

There are two main types of errors in R code:

- syntax errors: result from invalid code statements that R doesn't understand
- semantic errors: result from valid code that successfully executes but produces unintended outcomes

Fortunately, syntax errors will always result in error messages

- along with the line number where (near) the error occurs
- hence, we have chance to detect syntax errors

Detecting, locating and debugging semantic errors are more challenging.

Type of errors

Common syntax errors:

- forgotten comma or other symbol
- unmatched parentheses or a bracket opened without closing it
  - *unexpected symbol in* error
- misspelled character by mistake, case-sensitivity issue
  - *object not found* error
- package is not loaded
  - *could not find function* error
- *subscript out of bounds* error
- longer or shorter object than desired
  - *replacement ha XX rows, data has YY* error
  - *number of items to replace is not a multiple of replacement length* warning

Type of errors

```
variable <- 5
print(varaible)
>  Error in print(varaible): object 'varaible' not found
```

```
if (variable = 5) print(5)
>  Error: <text>:1:14: unexpected '='
>  1: if (variable =
>                   ^
```

Type of errors

```
if (variable == 5) {
    print(5)}
}
>  Error: <text>:3:1: unexpected '}'
>  2:          print(5)}
>  3: }
>     ^
```

Syntax errors are easy to locate.
However, searching for missing parentheses might be challenging.

Debugging

Debugging is about finding a bug and solve the problem.
Finding a bug might be really frustrating and time-consuming. . .
Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true. — Norm Matloff

Debugging

There are some common approaches.

1. searching for the error message
   - in google.com or stackoverflow.com
   - remove all variable names and values from the error message
   - there are two packages for automated searching: `errorist`, `searcher`
2. improving reproducibility
   - finding the bug is an iterative process
   - it needs several runs
   - decrease the data size, decrease the duration of each iteration
   - reproducible example (reprex)
     - ★ we can share the problem with others
     - ★ and ask for help
     - ★ in forums (stackoverflow) and mailing lists (r-sig-ecology, r-sig-geo, ÖBI statistics group)
     - ★ or contact the package maintainer or raise an issue in github
     - ★ see package `reprex`

Debugging

3. locating the bug and understanding it
   - narrow the location of the bug
     - ★ both horizontally: remove all unnecessary lines of code
     - ★ and vertically: locate the level of the bug (`traceback()`)
   - also locate the cases when the bug occurs
     - ★ changing inputs (and environment, package version, R version, OS)
     - ★ ∼research: set hypotheses, test them and systematically narrow the circle
4. solving the problem
   - if we found the bug, it is usually obvious how to fix it
   - after fixing the bug, carefully check the results!
     - ★ fixing one bug can generate or bring up new bugs in other parts of the script
   - use previously created small tests to study the desired behavior of the script
     - ★ called: unit testing
     - ★ see packages `testthat` and `usethis`

Debugging

Some common practices for locating the bug:

- insert `print()` calls in the code
  - to see which lines/clocks/if-else branches/loop iterations are evaluated
  - called "print debugging"
- insert `str()`/`class()`/`length()`
  - to check the type and shape of the variables
- insert `if () stop()` statements
  - to check conditions and immediately stop running if violated
  - alternatively, you can use function `stopifnot()`

Debugging

A typical example:

```
m <- matrix(data = 1:4, nrow = 2, ncol = 2)
if (class(m) == "matrix") m <- as.data.frame(m)
>  Warning in if (class(m) == "matrix") m <-
>  as.data.frame(m): the condition has length > 1 and only
>  the first element will be used
```

We expect that this code is OK, but it results a bug (a warning).

Debugging

Let's study variable `m`.

```
m <- matrix(data = 1:4, nrow = 2, ncol = 2)
print(m)
>      [,1] [,2]
>  [1,]    1    3
>  [2,]    2    4
```

It is as expected.

Debugging

"the condition has length > 1": let's study the condition.

```
print(class(m) == "matrix")
>  [1]  TRUE FALSE
length(class(m) == "matrix")
>  [1] 2
```

R is right, it is more than one logical value. Why?

Debugging

```
length(class(m))
>  [1] 2
print(class(m))
>  [1] "matrix" "array"
```

That's the reason: matrices are arrays as well.

After locating and understanding the bug, the solution is straightforward:

```
m <- matrix(data = 1:4, nrow = 2, ncol = 2)
if ("matrix" %in% class(m)) m <- as.data.frame(m)
```

Debugging

Usually these approches are good enough.
If not, there are some tools

- in R for interactive debugging
- in RStudio (or other IDEs) for interactive debugging
- in R for non-interactive debugging (aka post-mortem analysis)

I wanted to talk about debugging, but it needs one more Talentia session, sorry...

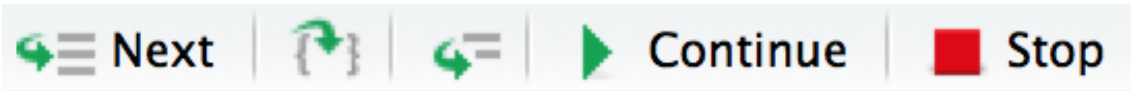
Debugging

So here I give you a very brief summary.
You can use

- `debug()`, `debugonce()` and `setBreakpoint()` for stepping through the execution of a function, line by line. At any point, you can print out values of variables or produce a graph of the results within the function;
- `browser()` for pausing the execution of a function at a certain point (aka breakpoint) and start debugging there;
- `trace()` for inserting bits of code into a function (rarely used, mainly for debugging code that we don't have the source for);
- `recover()` for navigating on the call stack and call the `browser()`'s environment in a certain level
- `options(error = recover)` for automatically start debugging at any level in the call stack when an error occurs

Debugging

Interactive debuggers accept some special commands (n, c, s, f, Q, where) for navigation.
Or you can use the debugger toolbar in RStudio.



Debugging

Debugging is most challenging when you can't run code interactively because

- it's run automatically (possibly on another computer), or
- the error doesn't occur when you run the same code interactively.

In this case, you can use

- `dump.frames()` for saving a dump file called `last.dump.rda` in the working directory
- that later, in an interactive session can be loaded by `load("last.dump.rda")`
- and `debugger()` to jump into the loaded environment and debug it interactively
- using an interface similar to `recover()`

Alternatively, you can use the slow and primitive "print debugging" approach.

Further reading

Chapter "Debugging" in Hadley Wickham: Advanced R: adv-r.hadley.nz/debugging.html

Also see package `tryCatchLog` and its functions `tryLog()` and `tryCatchLog()` for

- advanced error handling,
- configurable logging of call stack and
- post-mortem analysis via dump files

cran.r-project.org/web/packages/tryCatchLog/vignettes/tryCatchLog-intro.html

The end

Thank you for your kind attention.