R Shiny: Bug hunting

Logging, error handling and post-mortem analysis

Feb 15, 2023 by J. A.

Source and author see: https://github.com/aryoda/R_shiny_post_mortem_analysis_training

What’s the problem?

Have you ever tried to

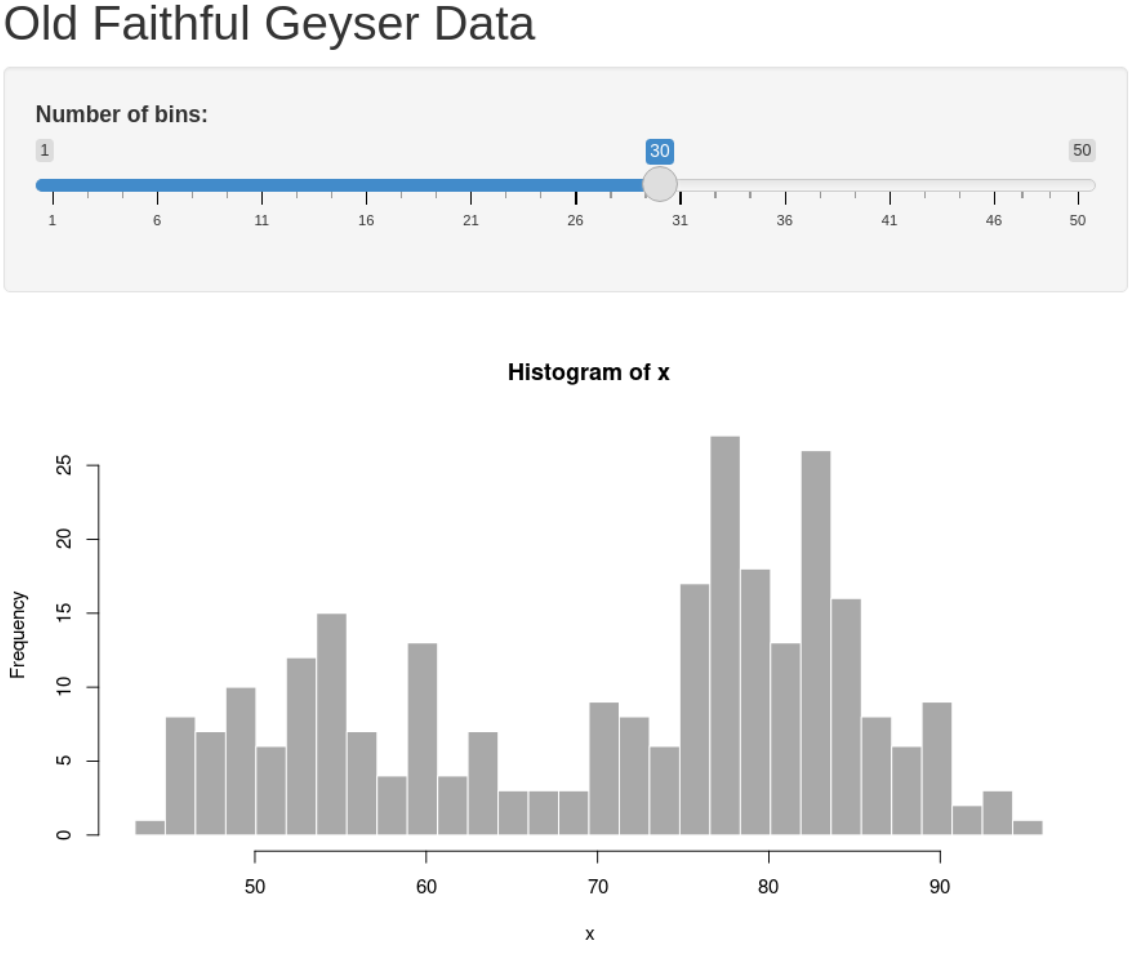
- find out why your shiny app **does not show correct results**?
- find the **reason of an error or warning** in your code?
- fix a bug that **happens only in your production system (shiny server)** but not on your DEV computer?
- fix a bug that does occur only randomly (= is **not reproducible**)?

Ways to solve the bug hunting challenge...

Use the CRAN package tryCatchLog + a supported logging package and learn how to do

- logging
- error handling
  → *only required parts using "tryCatchLog" to enable logging and post-mortem dumps are covered here*
- debugging in RStudio
  → *out of scope here (belongs to the "basics")*
- post-mortem analysis (using dump files)

Let’s play with this “hello world” shiny app...

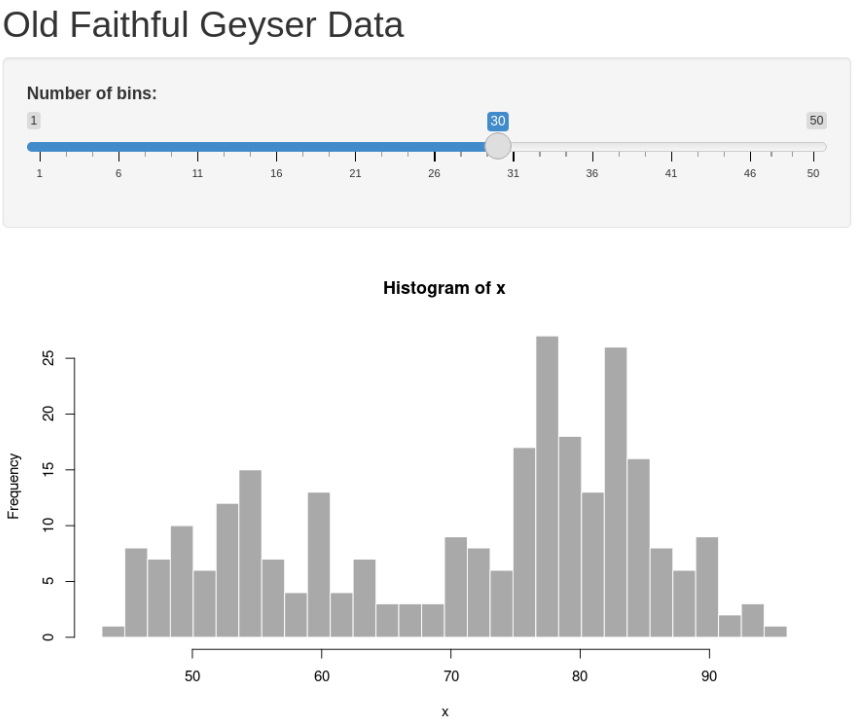


1) “Hello world” shiny app with code*

```
library(shiny)

ui <- fluidPage(

    titlePanel("Old Faithful Geyser Data"),

    sidebarLayout(
        sidebarPanel(
            sliderInput("bins",
                        "Number of bins:",
                        min = 1, max = 50, value = 30)
        ),

        mainPanel(plotOutput("distPlot"))
    )
)

server <- function(input, output) {

    output$distPlot <- renderPlot({

        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        hist(x, breaks = bins, col = 'darkgray', border = 'white')
    })
}

# Run the application
shinyApp(ui = ui, server = server)
```

input binding

output binding

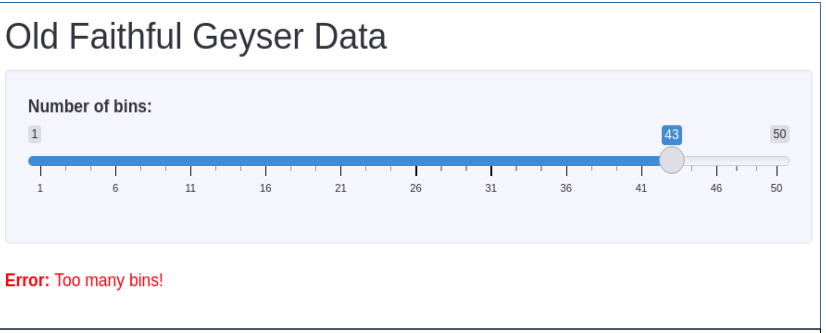


*) *Example taken from the shiny package via RStudio > New project*

2) Shiny app with unhandled warning and error

```
# File: app.R

[... unchanged code omitted...]

server <- function(input, output) {

    output$distPlot <- renderPlot({

        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # some fancy logic
        source("script_with_error.R", local = TRUE)

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', [...]

    })
}
```

```
# File: script_with_error.R

if (input$bins > 40)
  stop("Too many bins!")

print(log(-1))  # just provoke a warning for demo purposes
```




```
> runApp()
Listening on http://127.0.0.1:5067
Warning in log(-1) : NaNs produced
[1] NaN
Warning: Error in eval: Too many bins!
  173: stop
  168: renderPlot [app.R#40]
  166: func
  126: drawPlot
  112: <reactive:plotObj>
   96: drawReactive
   83: renderFunc
   82: output$distPlot
    1: runApp
```

Note: Errors are shown as “warning” in Shiny!

Observations regarding unhandled conditions*

Unhandled errors

1) stop the creation of the user interface (leaving an **incomplete UI**)
2) are catched and **reported as "warning"** (to avoid crashing the complete shiny app)

Unhandled conditions*

3) do **almost never show the real line of code** that produced the error
4) do not show the data (variables) that cause the unexpected condition

It is not easy to reproduce and analyze the problem to fix a bug!

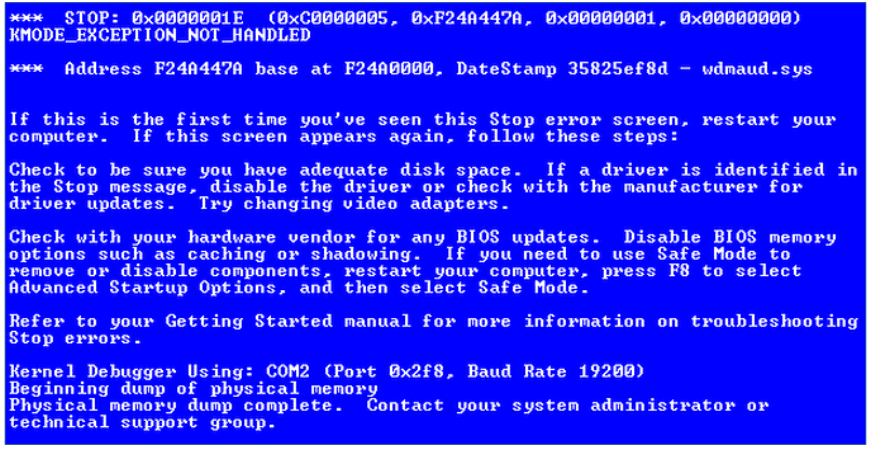
**) "condition" is the general R term for errors, warnings, messages and other "exceptions" that are derived from the R super class* `condition`

Approaches to hunt bugs

1) **Logging (“tracing”, “auditing”)**
   - ➔ **print** info about the current operation (code location + relevant data)

2) **Interactive debugging**
   - ➔ **halt** the code execution at chosen code locations
   - ➔ **inspect** the current state of variables
   - ➔ **execute** code (blocks) step-by-step

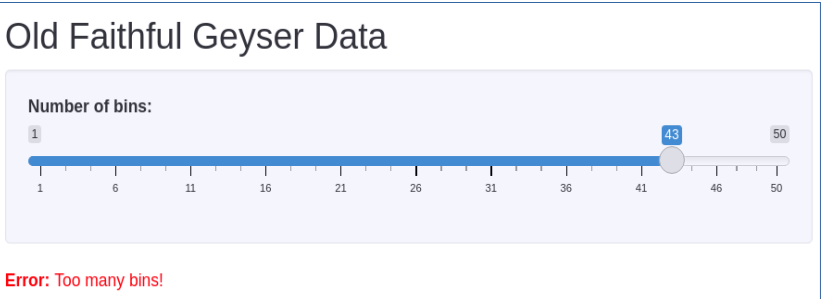
3) **Post-mortem analysis**
   - ➔ **create a memory dump** file of all objects incl. the call stack when an error occurs
   - ➔ inspect the objects along the call stack



3) Shiny app with logging and unhandled conditions

```
# File: app.R

[... unchanged code omitted...]

server <- function(input, output) {
    library(futile.logger)
    output$distPlot <- renderPlot({
        flog.info("Begin of renderPlot (bins=%i)", input$bins)
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # some fancy logic
        source("script_with_error.R", local = TRUE)

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', [...]
        flog.info("End of renderPlot")
    })
}
```

```
# File: script_with_error.R

if (input$bins > 40)
  stop("Too many bins!")

print(log(-1))  # just provoke a warning for demo purposes
```




```
> shiny::runApp()
INFO [2023-02-14 01:02:54] Begin of renderPlot
                           (bins=30)
Warning in log(-1) : NaNs produced
[1] NaN
INFO [2023-02-14 01:02:54] End of renderPlot
INFO [2023-02-14 01:04:05] Begin of renderPlot
                           (bins=43)
Warning: Error in eval: Too many bins!
  173: stop
  168: renderPlot [app.R#60]
  126: drawPlot
  112: <reactive:plotObj>
   96: drawReactive
   83: renderFunc
   82: output$distPlot
    1: shiny::runApp
```

Little detour: Which logging framework shall I use?

CRAN download statistics (of the last 30 days as of Feb 14, 2023) to indicate the **popularity***:

```
         package  N downloads
[1:         plogr 30     90222]  → header-only C++ logging library (not for “plain” R)
 2: futile.logger 30     72081
 3:        logger 30     22173
 4:           lgr 30     12827
 5:       logging 30     12707
 6:       debugme 30      8654
 7:         log4r 30      8025
 8:          logr 30      1434
 9:        loggit 30       693
```

The upcoming tryCatchLog release will support all major logging frameworks (not only futile.logger):
https://github.com/aryoda/tryCatchLog/blob/feature/71_indiv_settings_per_condition_type/NEWS.md

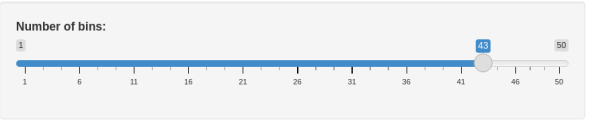
**) see: https://github.com/aryoda/tryCatchLog/issues/42*

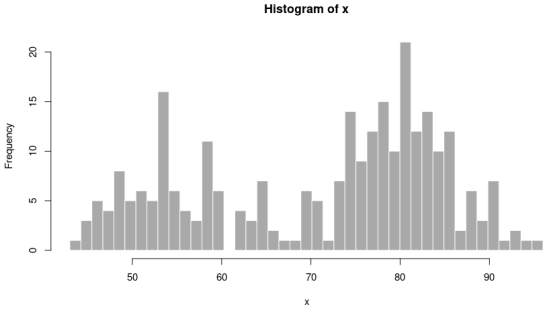
4) Shiny app with handled conditions

```
# File: app.R

[... unchanged code omitted...]

server <- function(input, output) {
    library(futile.logger)
    output$distPlot <- renderPlot({
        flog.info("Begin of renderPlot (bins=%i)", input$bins)
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # some fancy logic
        source("script_with_error_and_try.R", local = TRUE)

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', [...]
        flog.info("End of renderPlot")
    [...]
```

```
# File: script_with_error_and_try.R
try({
      if (input$bins > 40) {
        stop("Too many bins!")
      }
      print(log(-1))  # just provoke a warning for demo purposes
    })
```

Old Faithful Geyser Data



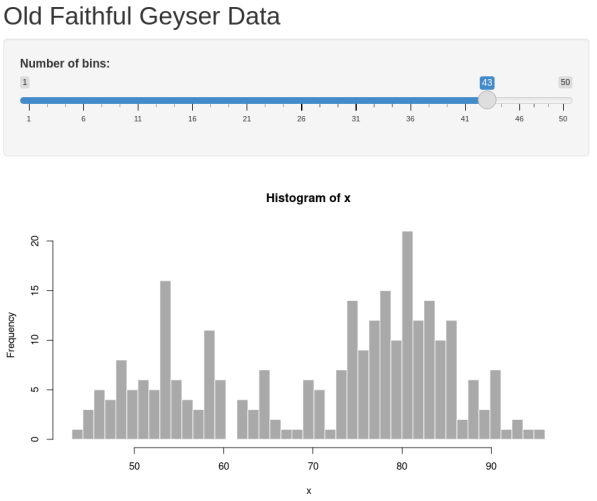



```
> shiny::runApp()
INFO [2023-02-14 01:02:54] Begin of renderPlot
                           (bins=30)
Warning in log(-1) : NaNs produced
[1] NaN
INFO [2023-02-14 01:02:54] End of renderPlot
INFO [2023-02-14 01:02:54] Begin of renderPlot
                           (bins=43)
Error in try({ : Too many bins!
INFO [2023-02-14 01:02:54] End of renderPlot
```

5) Shiny app with tryCatch handler to print call stack

```
# File: app.R

[... unchanged code omitted...]

server <- function(input, output) {
    library(futile.logger)
    output$distPlot <- renderPlot({
        flog.info("Begin of renderPlot (bins=%i)", input$bins)
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)
    [...]
```

```
# File: script_with_error_and_tryCatch_and_callstack.R
tryCatch(
    {
      if (input$bins > 40) {
        stop("Too many bins!")
      }
      print(log(-1))  # just provoke a warning for demo purposes
    } , error = function(e) {
        stack.trace <-
paste(as.character(limitedLabels(tail(sys.calls(), 100))),
collapse = "\n")
        cat("Error occured: ", e$message, "\n")  # error message
        cat(stack.trace, "\n")
    }
)
```

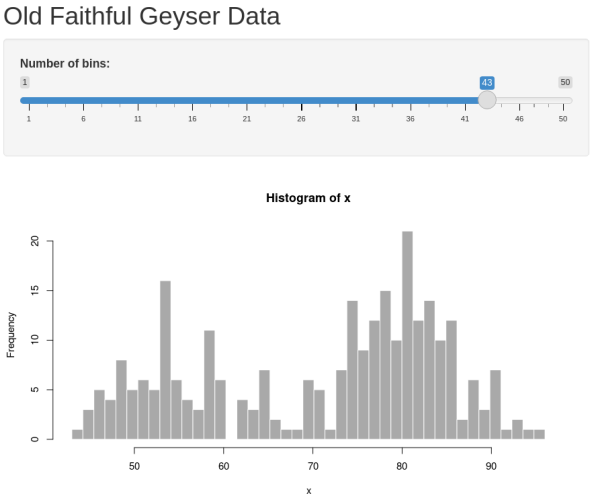



```
> shiny::runApp()
...
Error occured:  Too many bins!
domain$wrapSync(expr)
withCallingHandlers(expr, error = doCaptureStack)
...
app.R#60: source("script_with_error_and_tryCatch...
...
eval(ei, envir)
script_with_error_and_tryCatch_and_callstack.R#1:
tryCatch({ if (input$bins > 40) {
```

6) Shiny app with tryCatchLog handler

```
# File: app.R

[... unchanged code omitted...]

server <- function(input, output) {
    library(futile.logger)
    output$distPlot <- renderPlot({
        flog.info("Begin of renderPlot (bins=%i)", input$bins)
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)
    [...]
```

```
# File: script_with_error_and_tryLog.R
library(tryCatchLog)

options(keep.source = TRUE)
options(tryCatchLog.include.full.call.stack = FALSE)

tryLog(
  {
    if (input$bins > 40) {
      stop("Too many bins!")
    }
    print(log(-1))  # just provoke a warning for demo purposes
  }
)
```




```
> shiny::runApp()
...
ERROR [2023-02-14 02:22:24] Too many bins!
Compact call stack:
  1 shiny::runApp()
  2 app.R#57: source("script_with_error_and_tryLog.R"...
  3 script_with_error_and_tryLog.R#9: tryLog({
  4 tryLog.R#57: tryCatchLog(expr = expr, ...
  5 tryCatchLog.R#476: tryCatch(...
  6 tryCatchLog.R#476: withCallingHandlers(expr,...
  7 script_with_error_and_tryLog.R#12: stop("Too many
bins!")
```

7) Intro into dumps and post-mortem debugging

```
# File: dump_frames_intro.R
library(tryCatchLog)

x.global <- 99

f <- function(x) {
  value <- x + 1
  ff(value)
}

ff <- function(y) {
  fff(y + 2)
}

fff <- function(z) {
  print(paste("z =", z))
 if(z > 4) stop("z is too big\n")
}

tryLog(f(2), write.error.dump.file = T, [...])
```

```
# File: post_mortem_analysis.R

# --- Reset workspace before debugging
# See `?dump.frames` and `?debugger` for details

load("dump.rda")  # use an already prepared dump
debugger()
```

```
> debugger()
Message:  z is too big
Available environments had calls:
1: source("dump_frames_intro.R")
2: withVisible(eval(ei, envir))
3: eval(ei, envir)
4: eval(ei, envir)
5: dump_frames_intro.R#25: tryLog(f(2), ...
6: tryLog.R#57: tryCatchLog(expr = expr, ...
7: tryCatchLog.R#476: tryCatch(withCallingHandlers(expr, ...
8: tryCatchList(expr, classes, parentenv, handlers)
9: tryCatchOne(expr, names, parentenv, handlers[[1]])
10: doTryCatch(return(expr), name, parentenv, handler)
11: tryCatchLog.R#476: withCallingHandlers(expr, ...
12: tryLog.R#57: f(2)
13: dump_frames_intro.R#9: ff(value)
14: dump_frames_intro.R#13: fff(y + 2)
15: dump_frames_intro.R#18: stop("z is too big\n")
16: .handleSimpleError(function (c)
{
    write.to.log <- TRUE
    log.as.severity <- NA
    config.check.result <- is.
17: h(simpleError(msg, call))

Enter an environment number, or 0 to exit
Selection:
```

Post-mortem debugging: Inspect variables

```
# File: dump_frames_intro.R
library(tryCatchLog)

x.global <- 99

f <- function(x) {
  value <- x + 1
  ff(value)
}

ff <- function(y) {
  fff(y + 2)
}

fff <- function(z) {
  print(paste("z =", z))
 if(z > 4) stop("z is too big\n")
}

tryLog(f(2), write.error.dump.file = T, [...])
```

```
# File: post_mortem_analysis.R

# --- Reset workspace before debugging
# See `?dump.frames` and `?debugger` for details

load("dump.rda")  # use an already prepared dump
debugger()
```

```
Enter an environment number, or 0 to exit
Selection: 4
Browsing in the environment with call:
   eval(ei, envir)
Called from: debugger.look(ind)
Browse[1]> ls()
[1] "f"  "ff"  "fff"  "last.dump" "x.global"
Browse[1]> x.global
[1] 99
Browse[1]> x.global <- 22
Browse[1]> x.global
[1] 22
Browse[1]>
Available environments had calls:
...
4: eval(ei, envir)
5: dump_frames_intro.R#25: tryLog(f(2), ...
...
13: dump_frames_intro.R#9: ff(value)
14: dump_frames_intro.R#13: fff(y + 2)
15: dump_frames_intro.R#18: stop("z is too big\n")
...

Enter an environment number, or 0 to exit
Selection:
```

use the parent call number (!) to look into variables visible at the call code location

list directly visible variables

show the value of a variable

change the value of a variable

press ENTER to leave the browser (“debugger”) and go back to the call selection

Post-mortem debugging: Hunt down the bug

```
# File: dump_frames_intro.R
library(tryCatchLog)

x.global <- 99

f <- function(x) {
  value <- x + 1
  ff(value)
}

ff <- function(y) {
  fff(y + 2)
}

fff <- function(z) {
  print(paste("z =", z))
 if(z > 4) stop("z is too big\n")
}

tryLog(f(2), write.error.dump.file = T, [...])
```

```
# File: post_mortem_analysis.R

# --- Reset workspace before debugging
# See `?dump.frames` and `?debugger` for details

load("dump.rda")  # use an already prepared dump
debugger()
```

```
Available environments had calls:
...
4: eval(ei, envir)
5: dump_frames_intro.R#25: tryLog(f(2), ...
...
13: dump_frames_intro.R#9: ff(value)
14: dump_frames_intro.R#13: fff(y + 2)
15: dump_frames_intro.R#18: stop("z is too big\n")
...
Enter an environment number, or 0 to exit
```

Determine the frame causing the error by entering
the call frame (call number minus one!) and checking the variables...

```
Selection: 14
Browsing in the environment with call:
   dump_frames_intro.R#13: fff(y + 2)
Called from: debugger.look(ind)
Browse[1]> ls()
[1] "z"
Browse[1]> z
[1] 5
Browse[1]> z > 4
[1] TRUE
Browse[1]>
```

Little detour: Environment, frames and (en)closures

See: ?environment:

- **Environments** consist of
  - a **frame (**collection of named objects)
  - and a pointer to an enclosing environment.
  Example: **Frame of variables local to a function call.**
  Its **enclosure** is the environment where the function was defined.
- The enclosing environment is distinguished from the **parent frame**: the latter (returned by `parent.frame`) refers to **the environment of the caller of a function**.

Since confusion is so easy, it is best never to use ‘parent’ in connection with an environment (despite the presence of the function parent.env).

Confused? ;-)

8) Post-mortem debugging of a shiny app

```
# File: app.R

[...]

server <- function(input, output) {
    library(futile.logger)
    output$distPlot <- renderPlot({
        flog.info("Begin of renderPlot (bins=%i)",
                  input$bins)
        # generate bins based on input$bins
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out =
                    input$bins + 1)
    [...]
```

```
# File: script_with_error_and_tryLog_and_dump_file.R
library(tryCatchLog)

options("tryCatchLog.write.error.dump.file" = TRUE)
options(tryCatchLog.include.full.call.stack = FALSE)

tryLog( {
          if (input$bins > 40) {
              stop("Too many bins!")
        }
        print(log(-1))  # just provoke a warning
})
```

```
> load("dump.rda")
> debugger(last.dump)
166: func()
167: ..stacktraceon..(renderPlot())
168: renderPlot()
169: app.R#60: source("script_with_error_and_tryLog_and_dump
...
173: script_with_error_and_tryLog_and_dump_file.R#12: tryLog({
    if (input$bins > 40) {
        stop("Too many bins!")
174: tryLog.R#57: tryCatchLog(expr = expr...
...
180: script_with_error_and_tryLog_and_dump_file.R#15:
     stop("Too many bins!")
...
Enter an environment number, or 0 to exit
Selection: 172
Browsing in the environment with call:
   eval(ei, envir)
Called from: debugger.look(ind)
Browse[1]> ls()
[1] "bins" "x"
```

Oops, where is the “input” variable? => “input” is a “reactiveValues” class (an “environment”) and shiny hides it quite well.

Best practice: Always use a local variable to store the required variables of “input” (or “output”) to ease post-mortem analysis!

When to use which approach to hunt bugs?

	Logging	Debugging	Post-mortem analysis
Preparation efforts	**High:** Logging functions must be added into code first	**Low:** No code changes required normally	**Medium:** Each reactive function requires at least one “error handler”
Available during DEV vs when deployed on shiny server (e.g. PRD ;-)	Local machine + server	local machine only (R does not support remote debugging of processes)	Local machine + server
Advantages	▪ Logging granularity can be modified without changing code ▪ Always active when something happens	▪ step-by-step code execution ▪ interactive examination of (large) data ▪ no code changes required	▪ Sporadic (non-reproducible) bugs can be analyzed (killer feature ;-) ▪ Captures exact code location and call stack of an error ▪ Allows interactive examination of (large) data
Restrictions	▪ Logging of larger data like data frames not practical ▪ Code without logging calls is a black box ▪ Requires code changes and possibly a roll-out if adding/modifying logging calls are required to hunt a bug ▪ possible performance overhead ▪ risk of introducing a bug (e.g. RTE) in a logging call	▪ No remote debugging on the server (e.g. if a bug happens only on a certain infrastructure) ▪ Non-reproducible bugs are difficult to discover	▪ no (easy) step-by-step code execution (“debugging”) ▪ memory dumps may flood the server HDD/SSD until it is full ;-) ▪ “input” and “output” env vars are not visible

Summary: Steps to enable logging and post-mortem

- Use the **packages** `tryCatchLog` and a logging framework like `futile.logger`
- **Set global options** for `tryCatchLog` and your logging framework in your `app.R`
- **Surround every reactive function with a `tryLog()` block** (which also catches conditions in all called functions within the block)
- *Optionally:*
  Add `try/tryCatch/tryLog/tryCatchLog` condition handler arround every expression where you want to handle expected conditions in a special way (ignore, retry, graceful recovery…)
- **Add INFO or DEBUG severity logging** output calls with relevant variables at the beginning and end of each of your functions (and other significant code locations)
- **Store variables of input and output in local variables** to make them visible in dump files for post-mortem analysis

```
# app.R

library(shiny)
library(tryCatchLog)
library(futile.logger)  # “flog”

flog.threshold(INFO) # or FATAL, ERROR, WARN, INFO
# flog.appender(appender.file("my_app.log"))

# Keep the file name and line numbers of sourced files
options(keep.source = TRUE)

# creates a dump file for each error
options(tryCatchLog.write.error.dump.file = TRUE)

# stack trace shall contain only code with known line numbers
options(tryCatchLog.include.full.call.stack = FALSE)

[…]

    output$distPlot <- renderPlot({
        tryLog({
            flog.info("Begin of renderPlot (bins=%i)", input$bins)
            bins <- input$bins
            […]
            flog.info("End of renderPlot")
        })
    })
```

Questions?

Links

- Slides:
  https://github.com/aryoda/R_shiny_post_mortem_analysis_training
- tryCatchLog package:
  https://github.com/aryoda/tryCatchLog
- Excellent shiny intro:
  https://debruine.github.io/shinyintro
- Debugging shiny applications (official doc):
  https://shiny.rstudio.com/articles/debugging.html
- List of shiny options:
  https://shiny.rstudio.com/reference/shiny/latest/shinyoptions
- Debugging hints:
  https://stackoverflow.com/questions/31920286/effectively-debugging-shiny-apps
  https://stackoverflow.com/questions/32222935/find-source-of-warning-in-shiny-app

Things not mentioned but worth to do it...

- Shiny Server trace output will be placed in a log file on the server under:
  `/var/log/shiny-server/*.log`
- `options(shiny.fullstacktrace = TRUE)  # see also shiny-internal calls!`
- `...`