Debugging

Shannon Pileggi

Getting started

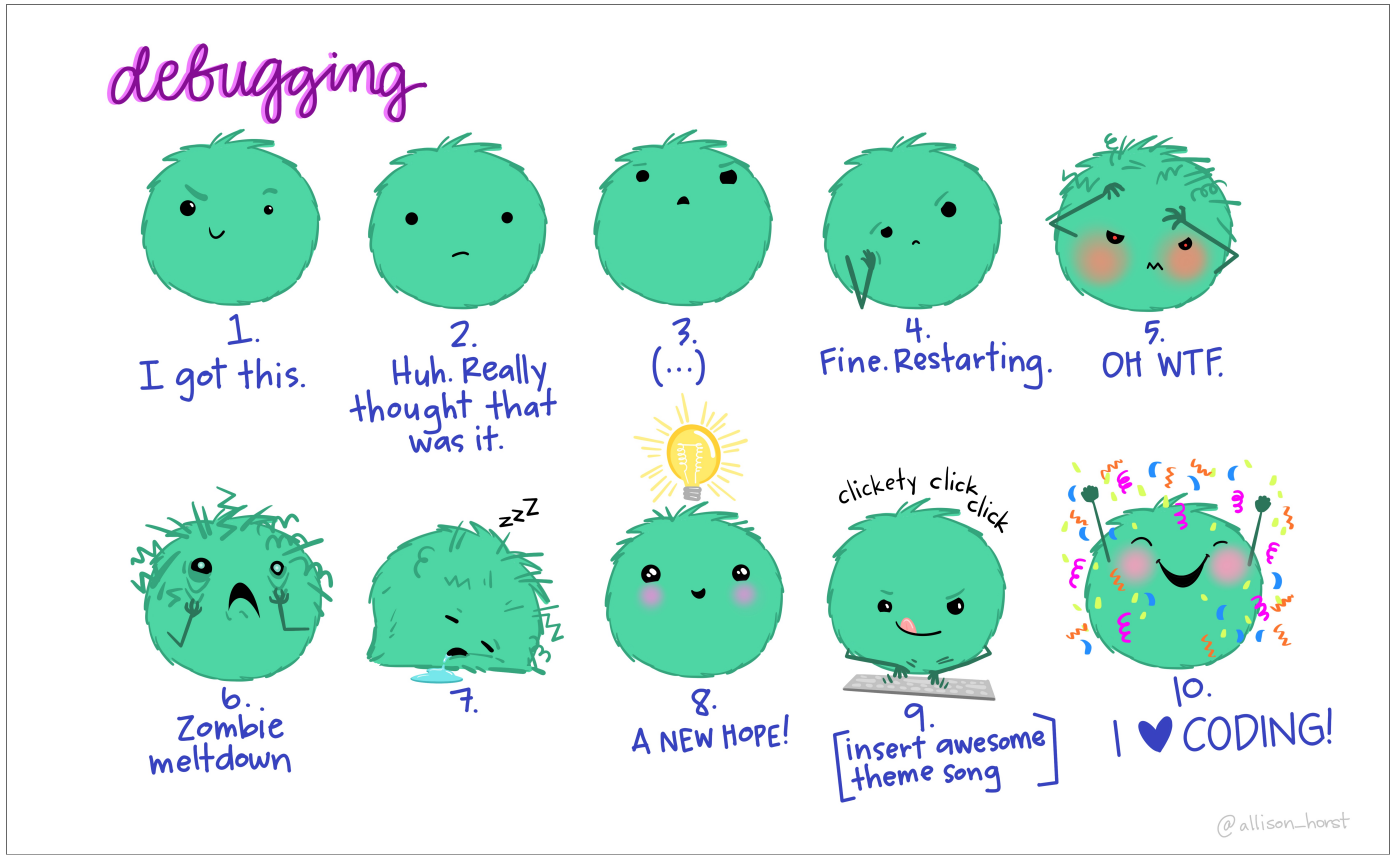
Licensing

Checklist

- ✓ R installed? Pretty recent?
  Recommended >= 4.2.0
- ✓ RStudio installed?
  I'm on 2022.02.3+492
- ✓ Ready to build packages?
  `devtools::has_devel()`
  Your system is ready to build packages!

Additional resources

- WTF Ch 11 *Debugging R code*
  https://rstats.wtf/debugging-r-code.html
- Advanced R Ch 22 *Debugging*
  https://adv-r.hadley.nz/debugging.html
- Jenny Bryan 2020 RStudio Conf Keynote *Object of type closure is not subsettable*
  https://github.com/jennybc/debugging#readme
- Amanda Gadrow 2018 Webinar *Debugging techniques in RStudio*
  https://www.rstudio.com/resources/webinars/debugging-techniques-in-rstudio/
- Jim Hester 2019 *Introduction to debugging in R and RStudio*
  https://www.jimhester.com/talk/2019-crug-debugging/
- Maëlle Salmon 2021 *How to become a better R code detective?*
  https://masalmon.eu/2021/07/13/code-detective/
- Kara Woo 2019 RStudio Conf *Box plots A case study in debugging and perseverance*
  https://www.rstudio.com/resources/rstudioconf-2019/box-plots-a-case-study-in-debugging-and-perseverance/


debugging
1.
I got this.
2.
Huh. Really thought that was it.
3.
(...)
4.
Fine. Restarting.
5.
OH WTF.
6..
Zombie meltdown
zzZ
7.
8.
A NEW HOPE!
clickety click click
9.
[insert awesome theme song]
10.
I ♥ CODING!
@allison_horst

Troubleshooting

1. Search

- Google *exact* error message
- keyword search on **RStudio community**
- keyword search on **stackoverflow**, `[r]` tag

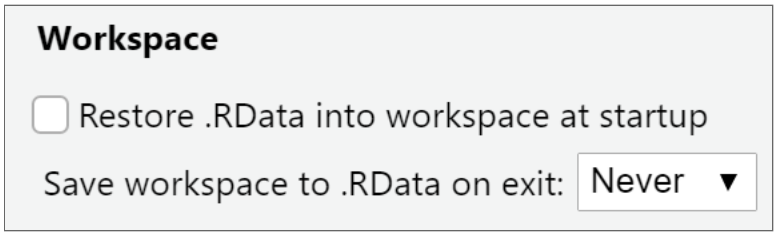
Samantha Csik 2022 R-Ladies St. Louis Workshop *Teach Me How To Google*: **slides**, **recording**

2. Reset

Have you tried turning it `OFF` and `ON` again?
Restart R, especially when things get weird.
Session -> Restart R or

- Ctrl + Shift + F10 (Windows),
- Cmd + Shift + 0 / Cmd + Shift + F10 (Mac)

`Tools -> Global Options -> Workspace`

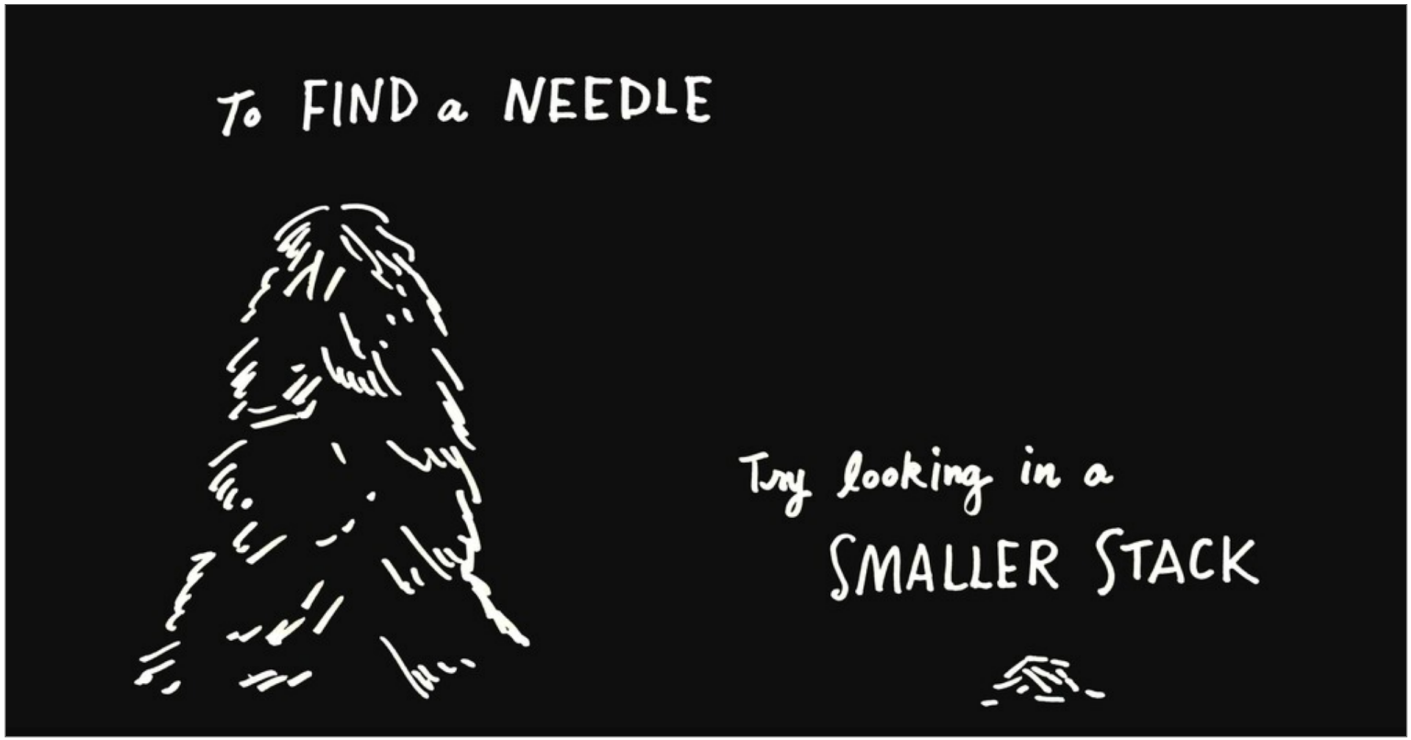


3. Reprex

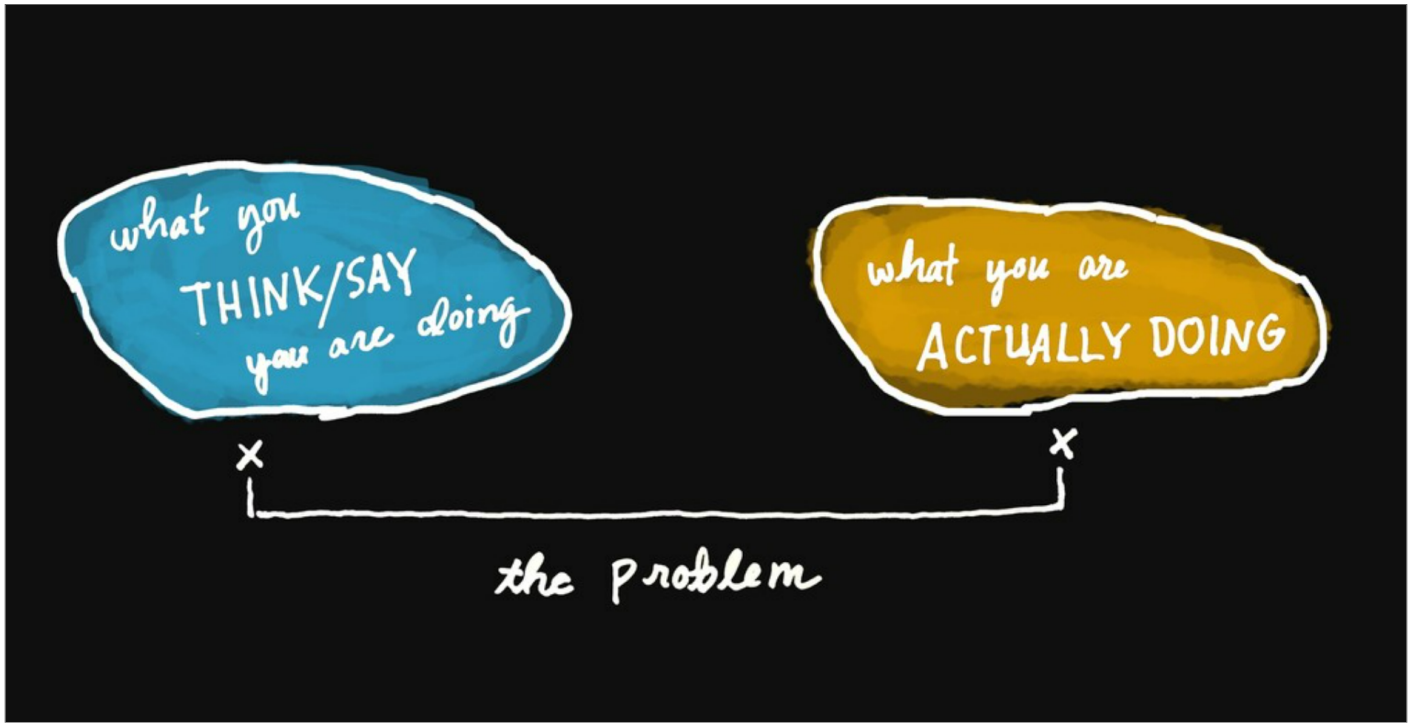
minimum reproducible example

https://reprex.tidyverse.org/

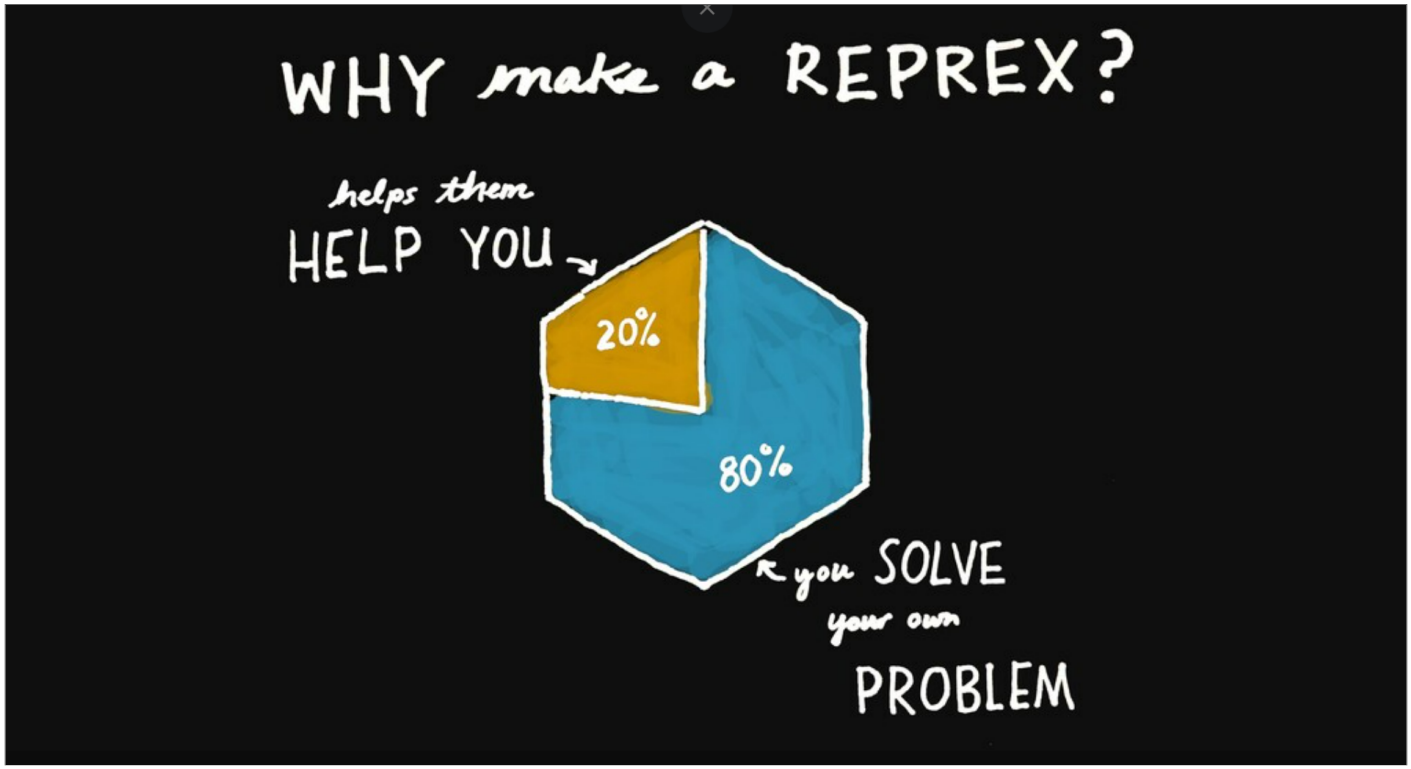
minimum



reproducible example



why reprex


WHY make a REPREX?
helps them
HELP YOU
20%
80%
you SOLVE
your own
PROBLEM

Review

Troubleshooting strategies:

1. Search
2. Reset
3. Reprex

But what if

Moving on

from troubleshooting to...

formal `debugging` techniques.

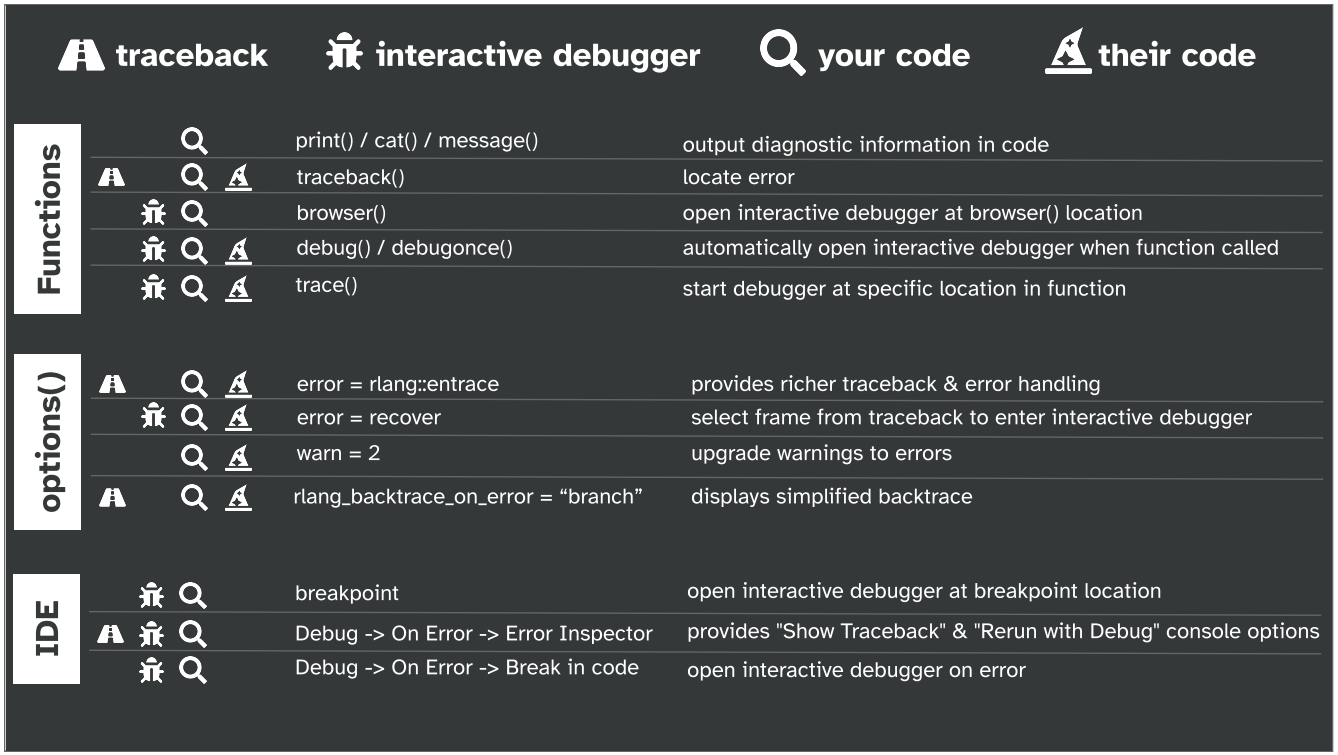
Key concepts

traceback
`location` where did the error occur

interactive debugger
`context` why did the error occur

your code vs their code

Debugging tools



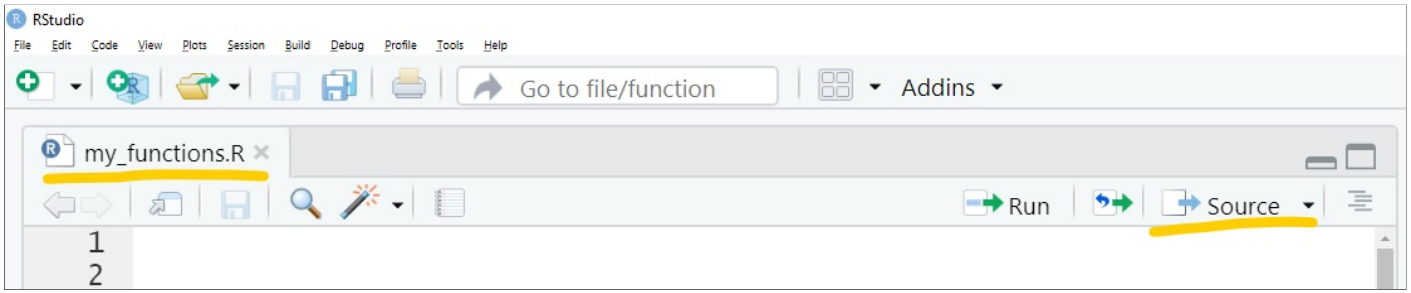
This is a lot.

These tools achieve similar objectives slightly differently.

People don’t generally use *all* of these tools at once. They pick and choose the ones they like.

Sourcing

⚠️ name your script with functions
⚠️ source your script with functions
for the best debugging experience 😎



Set up

```
f <- function(x) {
  x + 1
}

g <- function(x) f(x)

g("a")
```

```
Error in x + 1: non-numeric argument to binary operator
```

Debugging your code

print()

```r
f <- function(x) {
  print(x)
  x + 1
}

g <- function(x) f(x)
```

```r
g("a")
```

```
[1] "a"

Error in x + 1: non-numeric argument to binary operator
```

cat()

```r
f <- function(x) {
  cat("f()\n")
  cat("x =", x, "\n")
  x + 1
}

g <- function(x) f(x)

g("a")
```

```
f()
x = a

Error in x + 1: non-numeric argument to binary operator
```

traceback()

```r
source("demo/my_functions.R")
g("a")
traceback()
```

```
2: f(x) at my_functions.R#5
1: g("a")
```

shows the sequence of calls that lead to the error.

Richer traceback

```r
source("demo/my_functions.R")
options(error = rlang::entrace)
g("a")
rlang::last_error()
```

```
<error/rlang_error>
Error:
! non-numeric argument to binary operator
---
Backtrace:
 1. global g("a")
 2. global f(x)
      at wtf-debugging-slides/demo/my_functions.R:5:5
Run `rlang::last_trace()` to see the full context.
```

Richer traceback

```
1 source("demo/my_functions.R")
2 options(error = rlang::entrace)
3 g("a")
4 rlang::last_error()
5 rlang::last_trace()
```

```
<error/rlang_error>
Error:
! non-numeric argument to binary operator
---
Backtrace:
    ▆
 1. └─global g("a")
 2.   └─global f(x) at wtf-debugging-slides/demo/my_functions.R:5:5
```

`options(error = rlang::entrace)`
could go in your `.Rprofile`

traceback vs rlang functions

Numbering and ordering differs between `traceback()` and rlang functions.

`traceback()`	`where`	`recover()`	rlang functions
`5: stop("...")`			
`4: i(c)`	`where 1: i(c)`	`1: f()`	`1. └─global::f(10)`
`3: h(b)`	`where 2: h(b)`	`2: g(a)`	`2. └─global::g(a)`
`2: g(a)`	`where 3: g(a)`	`3: h(b)`	`3. └─global::h(b)`
`1: f("a")`	`where 4: f("a")`	`4: i("a")`	`4. └─global::i("a")`

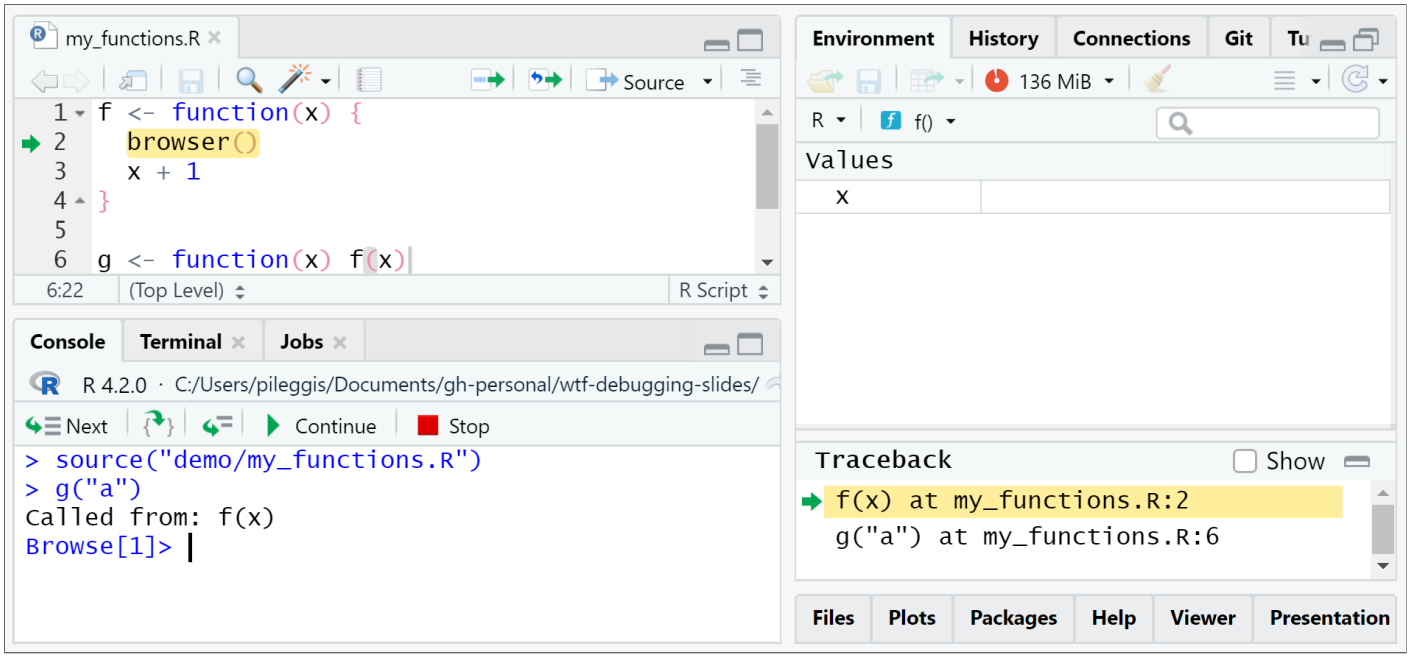
browser()

```
1 f <- function(x) {
2   browser(x)
3   x + 1
4 }
5
6 g <- function(x) f(x)
```

`browser()` opens the interactive debugger.

1. Modify the function by inserting a `browser()` statement.
2. Source the function.
3. Execute the function.

Interactive debugger



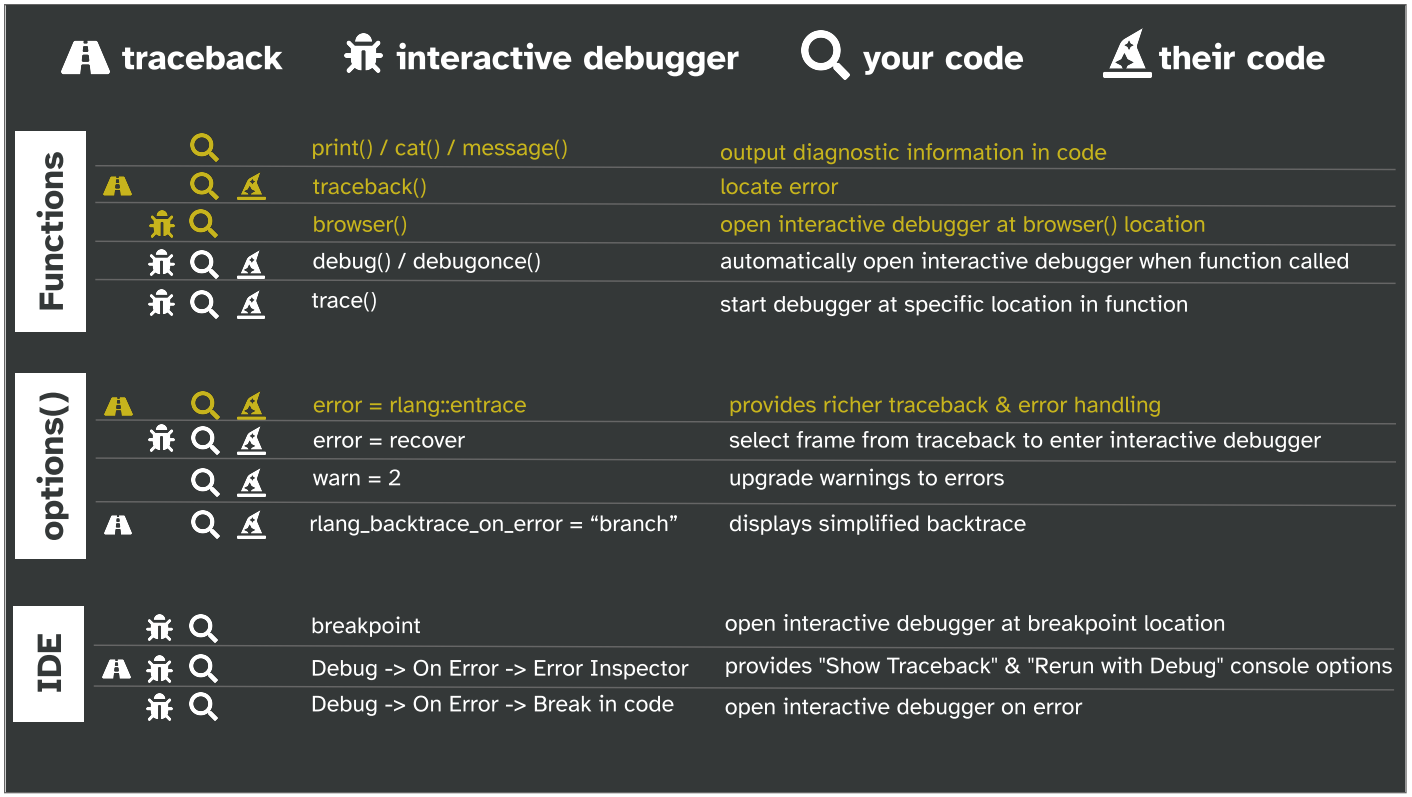
Interactive debugger tips

Investigate objects: `ls()`, `str()`, `ls.str()`, `print()`
Control execution:

`n`	next statement
`c`	continue
`s`	step into function call
`f`	finish loop / function
`where`	show previous calls
`Q`	quit debugger

:::

Debugging your code



Your turn, exercise 01

```
usethis::use_course("rstats-wtf/wtf-debugging")
```

Complete 01_exercise to practice debugging your own code.

Check out the `README.md` for some getting started tips!

- `01_debugging_spartan.R` (directions to explore without suggested code)
- `01_debugging_comfy.R` (directions to explore with suggested code)
- `01_debugging_solution.R` (directions to explore with code solutions)

15:00

Debugging in RStudio

Editor breakpoints

red circle = breakpoint = `browser()`
(but you don't have to change your code)

Set / re-set an editor breakpoint:

- click to the left of the line number in the source file
- press Shift+F9 with your cursor on the line

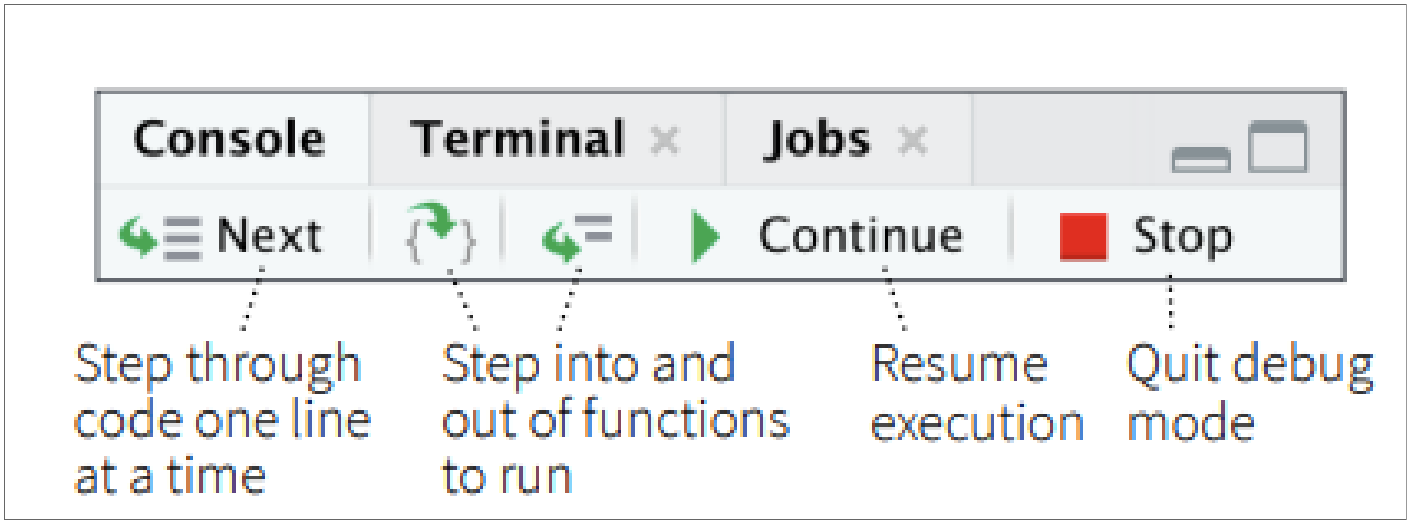
Editor breakpoints

red circle = breakpoint = `browser()`
(but you don't have to change your code)

To activate, either

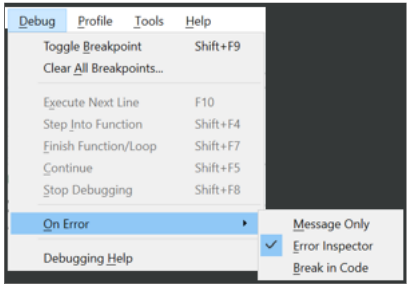
- click IDE Source button, or
- `debugSource("demo/my_functions.R")`

Debugging console



IDE on error

Automatically invoke actions on error.



New set up for demonstration.

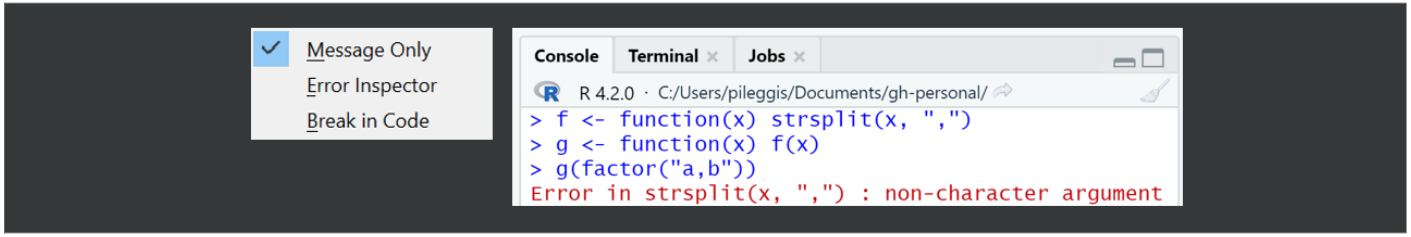
```
1 f <- function(x) { strsplit(x, ",") }
2 g <- function(x) { f(x) }
3 g("a,b")
```

```
[[1]]
[1] "a" "b"
```

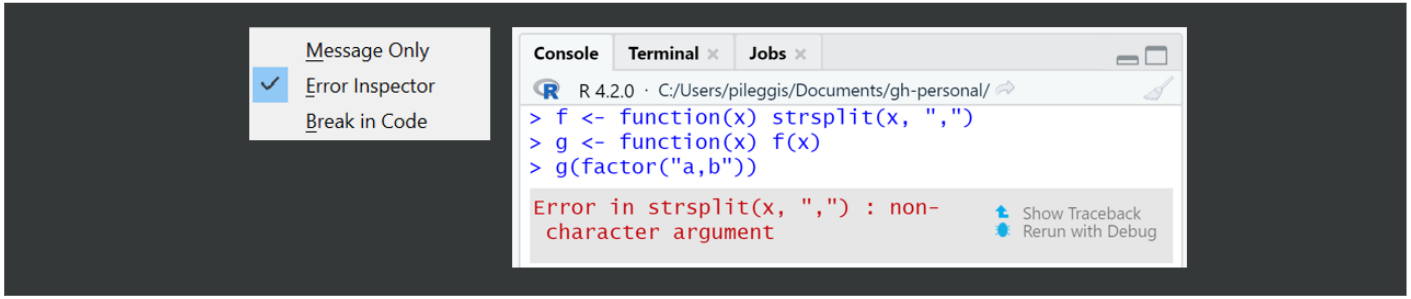
```
1 g(factor("a,b"))
```

```
Error in strsplit(x, ","): non-character argument
```

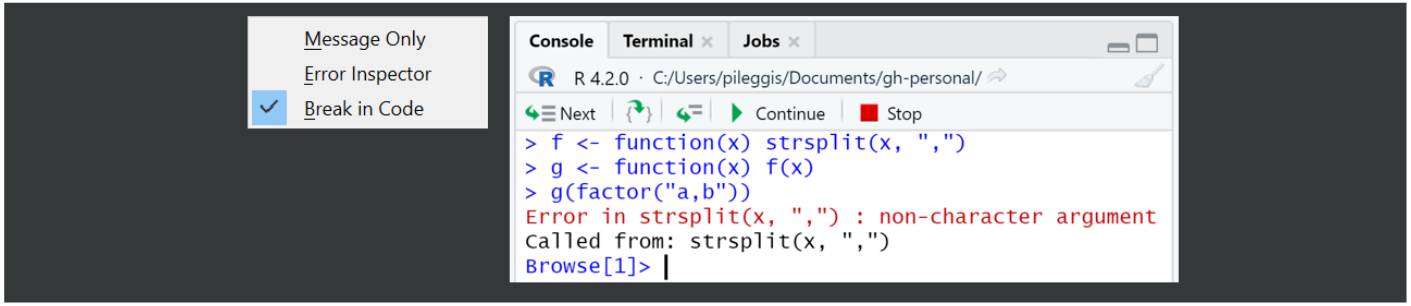
IDE message only



IDE error inspector



IDE break in code

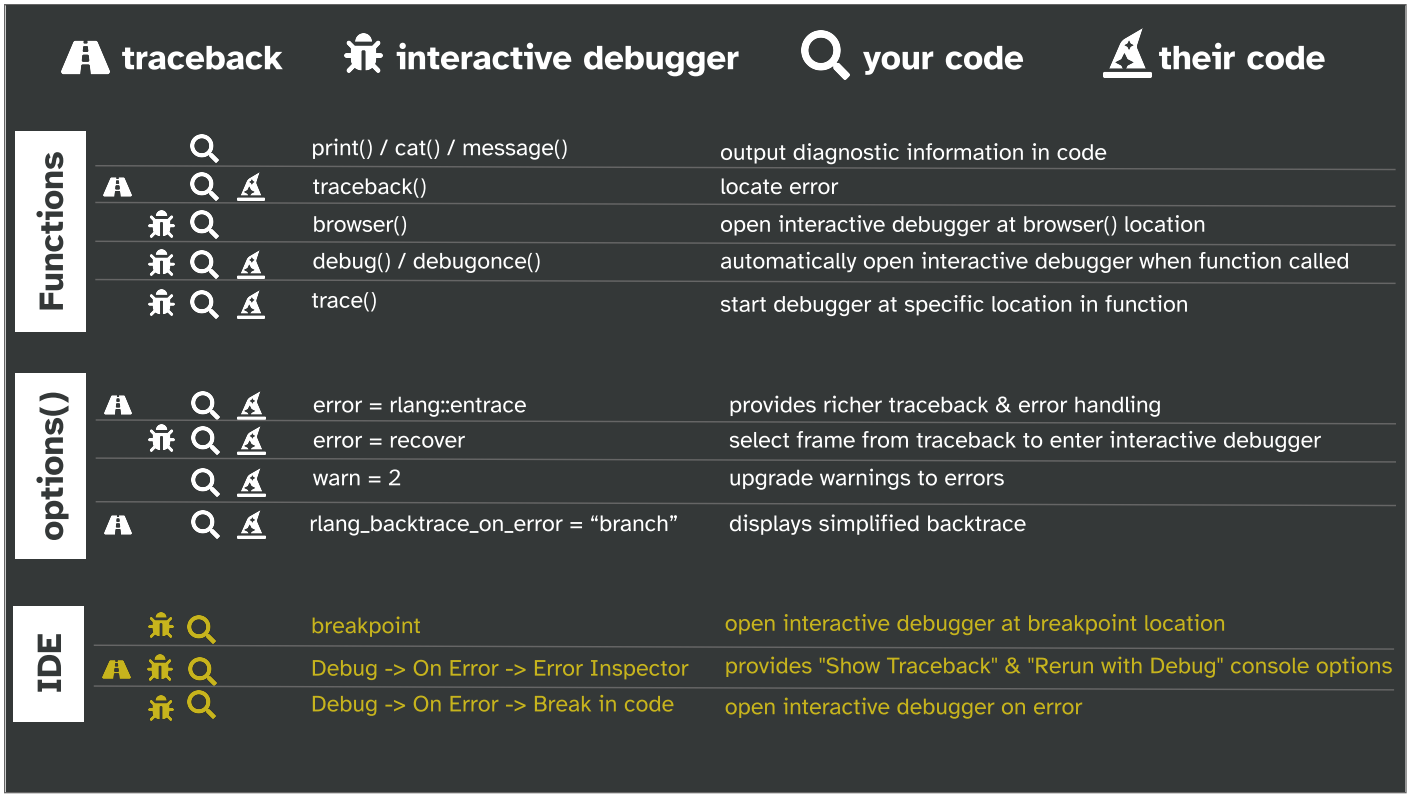


Error inspector

```r
# IDE Error Inspector not triggered
f <- function(x) x + 1
g <- function(x) f(x)
g("a")

# IDE Error Inspector not triggered
strsplit(factor("a,b"), ",")

# IDE Error Inspector not triggered
f <- function(x) strsplit(x, ",")
f(factor("a,b"))

# IDE Error Inspector yes triggered
g <- function(x) f(x)
g(factor("a,b"))
```

The error inspector is only invoked if
your code is involved.

Debugging in the RStudio IDE



Your turn, exercise 02

Complete 02_exercise to practice debugging a slightly different version of your own code, using features from the RStudio IDE.

07:00

Debugging their code

debug()

`debug()` = `browser()`/ `breakpoint` in first line of function

```
# source if code is yours, otherwise not necessary
source("demo/my_functions.R")
# set debugging flag
debug("g")
g("a")
# turn off debugging flag
undebug("g")
```

- interactive debugger is initiated every time `g()` is executed, until `undebug("g")`
- depending on function internals, this can... trap you in the debugger 😬

debugonce()

`debugonce()` = `browser()`/ `breakpoint` in first line of function

```
# source if code is yours, otherwise not necessary
source("demo/my_functions.R")
# set debugging flag
debugonce("g")
g("a")
```

- one time only!
- interactive debugger initiated a **single** time when `g()` is executed

options(error = recover)

✓ we already discussed
`options(error = rlang::entrace)`
for a richer traceback on error

🤠 `options(error = recover)`

1. displays an interactive prompt with frames
2. you select the frame to enter the debugger

recover example

```
1 source("demo/my_functions.R")
2 options(error = recover)
3 g("a")
4 options(error = NULL)
```

```
> options(error = recover)
> g("a")
Error in x + 1 : non-numeric argument to binary operator

Enter a frame number, or 0 to exit

1: g("a")
2: #1: f(x)

Selection:
```

trace() overview

`trace(what = fun, tracer = browser)`
is equivalent to

- inserting `broswer()` in first line of function
- `debug(fun)`

you an also insert *any* code at *any* location in function

- `trace(what = fun, tracer = browser, at = 2)`
  inserts `browser()` at second step of `fun`

`untrace(fun)` cancels the tracing

trace() without a specified step

```
1 trace(what = colSums, tracer = browser)
2 colSums(1:3)
3 # ls.str() in browser
4 untrace(colSums)
```

`trace(what = colSums, tracer = browser)`
is equivalent to

- inserting `broswer()` in first line of `colSums`
  if we had the source code
- `debug(colSums)`

navigating function steps

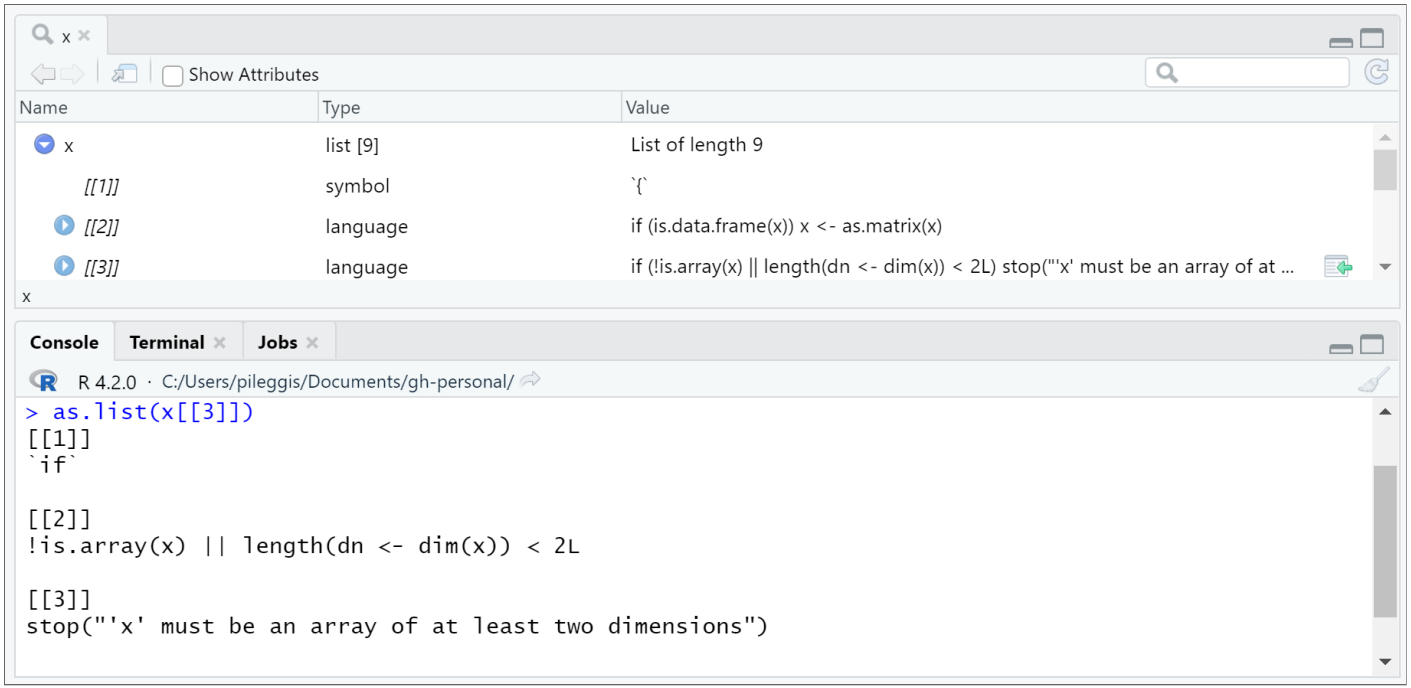
investigate 🧐 the function with `as.list()` + `body()`

```
1  x <- as.list(body(colSums))
2  View(x)
```

```
> colSums
function (x, na.rm = FALSE, dims = 1L)
{
    if (is.data.frame(x))
        x <- as.matrix(x)
    if (!is.array(x) || length(dn <- dim(x)) < 2L)
        stop("'x' must be an array of at least two dimensions")
    if (dims < 1L || dims > length(dn) - 1L)
        stop("invalid 'dims'")
    n <- prod(dn[id <- seq_len(dims)])
    dn <- dn[-id]
    z <- if (is.complex(x))
        .Internal(colSums(Re(x), n, prod(dn), na.rm)) + (0+1i) *
            .Internal(colSums(Im(x), n, prod(dn), na.rm))
    else .Internal(colSums(x, n, prod(dn), na.rm))
    if (length(dn) > 1L) {
        dim(z) <- dn
        dimnames(z) <- dimnames(x)[-id]
    }
    else names(z) <- dimnames(x)[[dims + 1L]]
    z
}
<bytecode: 0x000001b47ed54c48>
<environment: namespace:base>
```

Name	Type	Value
x	list [9]	List of length 9
[[1]]	symbol	`{`
[[2]]	language	if (is.data.frame(x)) x <- as.matrix(x)
[[3]]	language	if (!is.array(x) || length(dn <- dim(x)) < 2L) stop("'x' must be an array of at ...
[[4]]	language	if (dims < 1L || dims > length(dn) - 1L) stop("invalid 'dims'")
[[5]]	language	n <- prod(dn[id <- seq_len(dims)])
[[6]]	language	dn <- dn[-id]
[[7]]	language	z <- if (is.complex(x)) .Internal(colSums(Re(x), n, prod(dn), na.rm)) + (0+ ...
[[8]]	language	if (length(dn) > 1L) { dim(z) <- dn dimnames(z) <- dimnames(x)[-id] } el ...
[[9]]	symbol	`z`

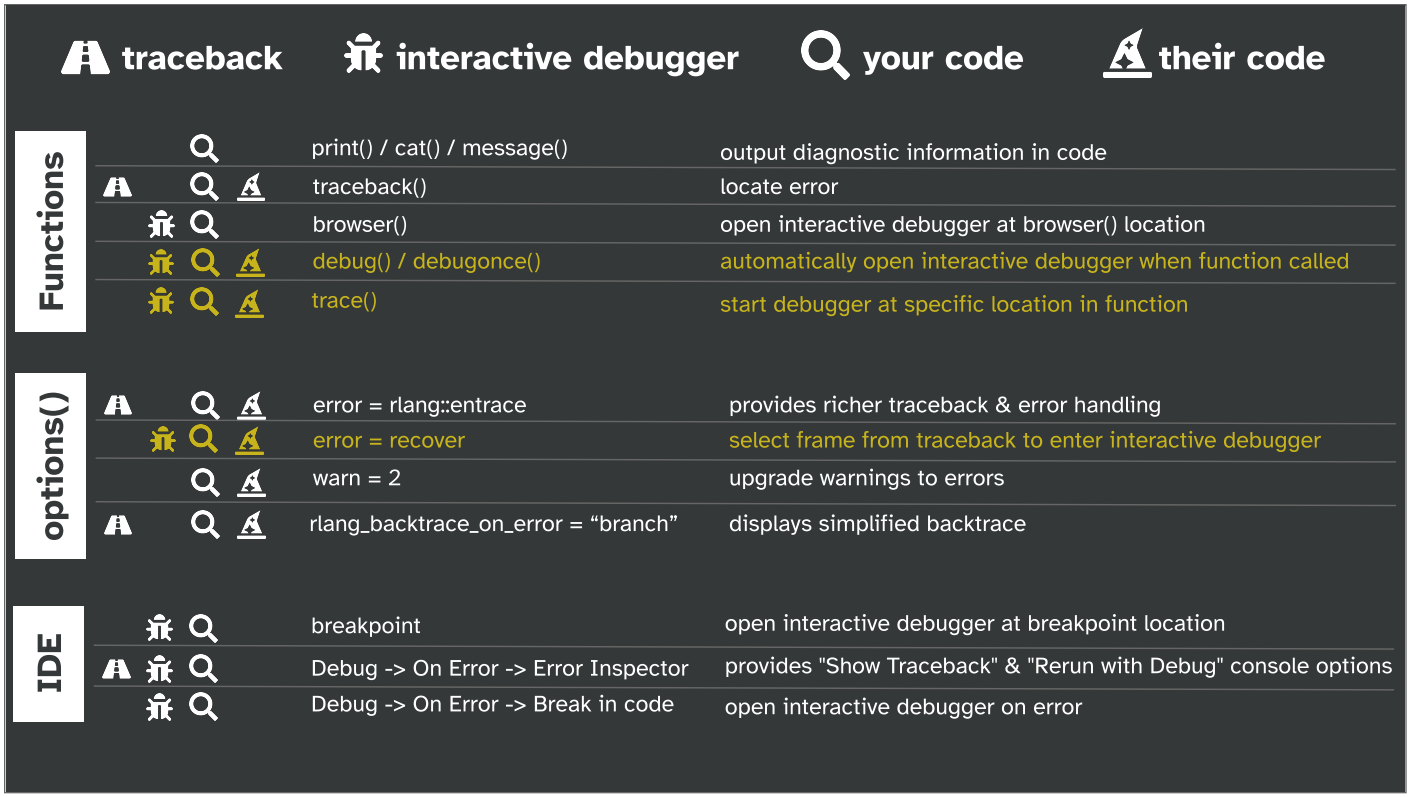
identify function step



trace() at specified step

```
# identify spot to insert code
as.list(x[[3]][[1]])
# equivalent notation
as.list(x[[c(3, 1)]])

# insert browser at step 3.1
trace(colSums, browser, at = list(c(3, 1)))
# execute function
colSums(1:3)
# ls.str() handy in browser
# cancel tracing
untrace(colSums)
```

Debugging their code



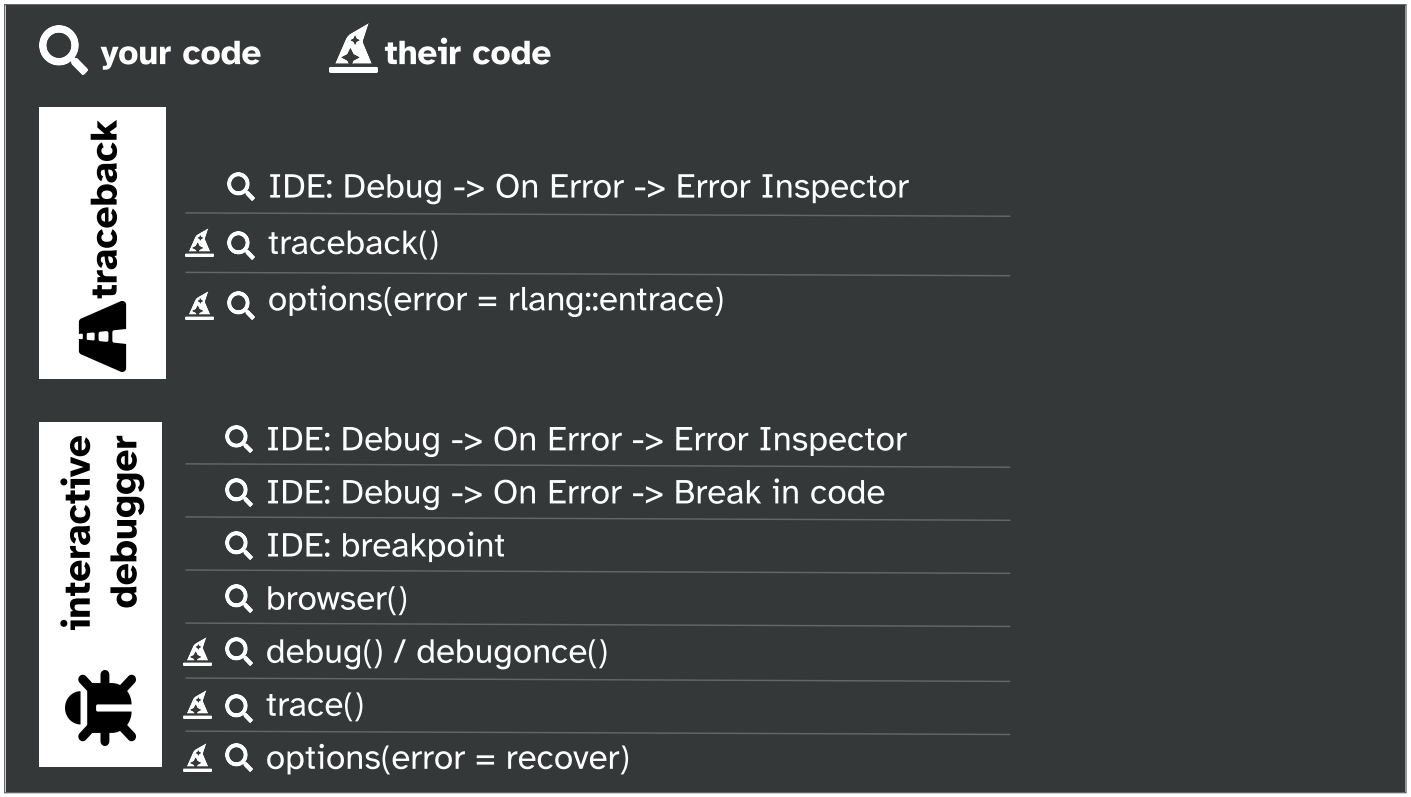
Your turn, exercise 03

```
# install.packages("devtools")
devtools::install_github("rstats-wtf/wtfdbg")
```

Complete 03_exercise to practice debugging others' code.

10:00

Pick your favorite



Special cases

Warnings

If you want to dig deeper into a warning, you can convert them to errors to initiate debugging tools.

```
?options
options(warn = 0) # default, stores warnings until top-level function returns
options(warn = 1) # warnings are printed as they occur
options(warn = 2) # upgrades warnings to errors


# initiate recover on warning
options(warn = 2, error = recover)
# restore original settings
options(warn = 0, error = NULL)
```

Piped expressions

tracebacks can be verbose with pipes

```
options(error = rlang::entrace, rlang_backtrace_on_error = "branch")
```

gives trimmed tracebacks when using pipes

- more contributed answers on RStudio Community
- Matt Dray 2019 blog post Fix leaky pipes in R

Rmarkdown

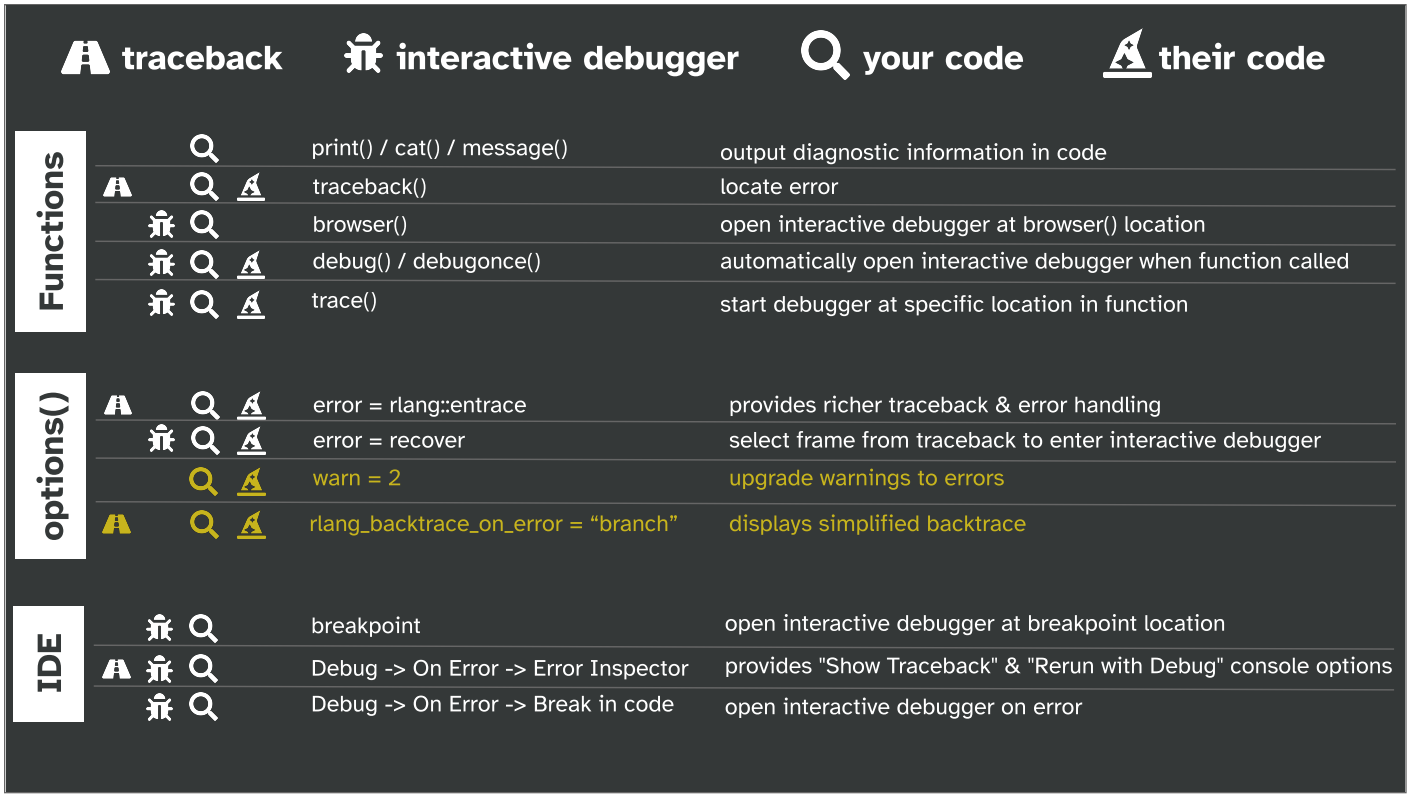
Troubleshooting:

1. rmarkdown chunk option `error = TRUE` enables knitting with errors
2. insert `knitr::knit_exit()` and interactively work through .Rmd

Debugging:

1. Adv R Ch 22.5.3 RMarkdown https://adv-r.hadley.nz/debugging.html#rmarkdown
2. WTF Ch 11.4 Debugging in Rmarkdown documents https://rstats.wtf/debugging-r-code.html#debugging-in-r-markdown-documents

Debugging special cases



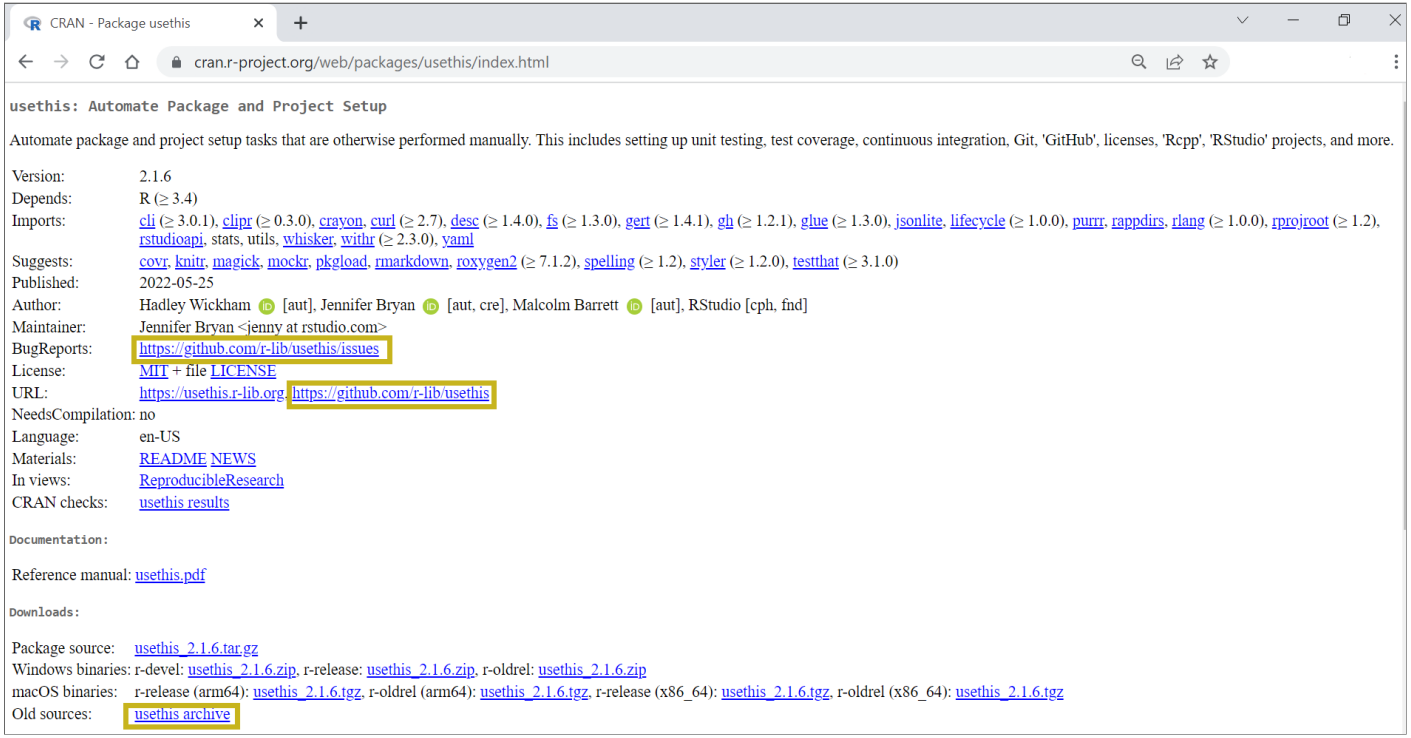
Read the source

(almost) all source on GitHub

github.com/search	All of GitHub! Can do advanced searches, too.
github.com/cran	CRAN packages
github.com/wch/r-source	R core

About searching on GitHub

google: cran pkgname



usethis: Automate Package and Project Setup

Automate package and project setup tasks that are otherwise performed manually. This includes setting up unit testing, test coverage, continuous integration, Git, 'GitHub', licenses, 'Rcpp', 'RStudio' projects, and more.

Version:	2.1.6
Depends:	R (≥ 3.4)
Imports:	cli (≥ 3.0.1), clipr (≥ 0.3.0), crayon, curl (≥ 2.7), desc (≥ 1.4.0), fs (≥ 1.3.0), gert (≥ 1.4.1), gh (≥ 1.2.1), glue (≥ 1.3.0), jsonlite, lifecycle (≥ 1.0.0), purrr, rappdirs, rlang (≥ 1.0.0), rprojroot (≥ 1.2), rstudioapi, stats, utils, whisker, withr (≥ 2.3.0), yaml
Suggests:	covr, knitr, magick, mockr, pkgload, rmarkdown, roxygen2 (≥ 7.1.2), spelling (≥ 1.2), styler (≥ 1.2.0), testthat (≥ 3.1.0)
Published:	2022-05-25
Author:	Hadley Wickham [aut], Jennifer Bryan [aut, cre], Malcolm Barrett [aut], RStudio [cph, fnd]
Maintainer:	Jennifer Bryan <jenny at rstudio.com>
BugReports:	https://github.com/r-lib/usethis/issues
License:	MIT + file LICENSE
URL:	https://usethis.r-lib.org, https://github.com/r-lib/usethis
NeedsCompilation:	no
Language:	en-US
Materials:	README NEWS
In views:	ReproducibleResearch
CRAN checks:	usethis results

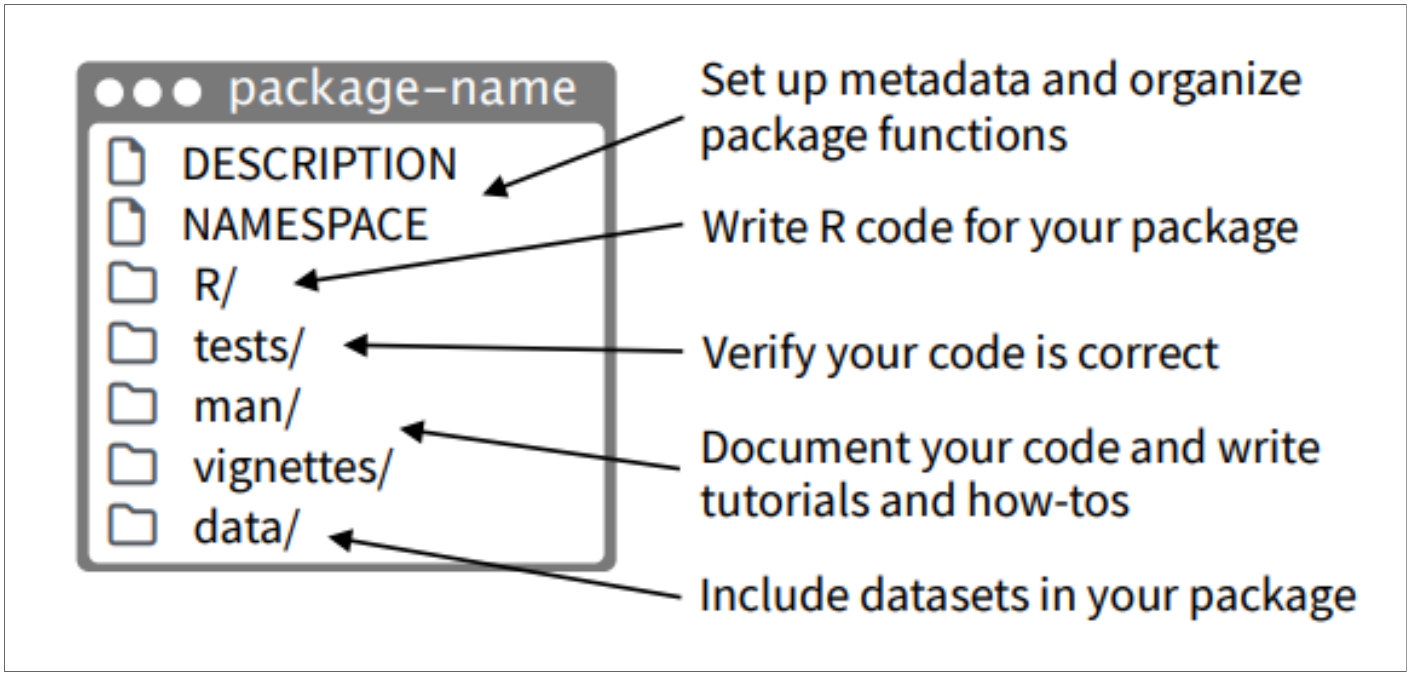
Documentation:

Reference manual: usethis.pdf

Downloads:

Package source:	usethis_2.1.6.tar.gz
Windows binaries:	r-devel: usethis_2.1.6.zip, r-release: usethis_2.1.6.zip, r-oldrel: usethis_2.1.6.zip
macOS binaries:	r-release (arm64): usethis_2.1.6.tgz, r-oldrel (arm64): usethis_2.1.6.tgz, r-release (x86_64): usethis_2.1.6.tgz, r-oldrel (x86_64): usethis_2.1.6.tgz
Old sources:	usethis archive

github.com/cran



In R console

`pkg::fnc` to see exported functions

`pkg:::fnc` to see non-exported functions

Your turn, investigating a CRAN package

At a high level,
how does readr::read_lines() work?

03:00

github.com/wch/r-source

src/main	base R
src/library/	packages included in base R
doc/manual	R manuals & documentation

Your turn, investigating source code

1. When was the trimws() function added to base R?
2. How does is.na() work?

Hints on next slides.

05:00

Investigating source code hints

1. When was the trimws() function added to base R?

Try restricting search to only commit messages in **github.com/wch/r-source**.

Investigating source code hints

> 2. How does is.na() work?

1. Look at the function.

```
1 is.na
```

```
function (x)  .Primitive("is.na")
```

2. Look in **github.com/wch/r-source/src/main/names.c** to find the name of the **internal function**. Or in GitHub search bar: `is.na filename:names.c`.
3. In GitHub search bar: `"do_isna" language:c`

Go forth,
and learn from your bugs!

Troubleshoot
Debug
Read the source

Debugging

Error