

# **Debugging**

**Shannon Pileggi**

# **Getting started**

## Licensing

This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#) (CC BY-SA4.0).

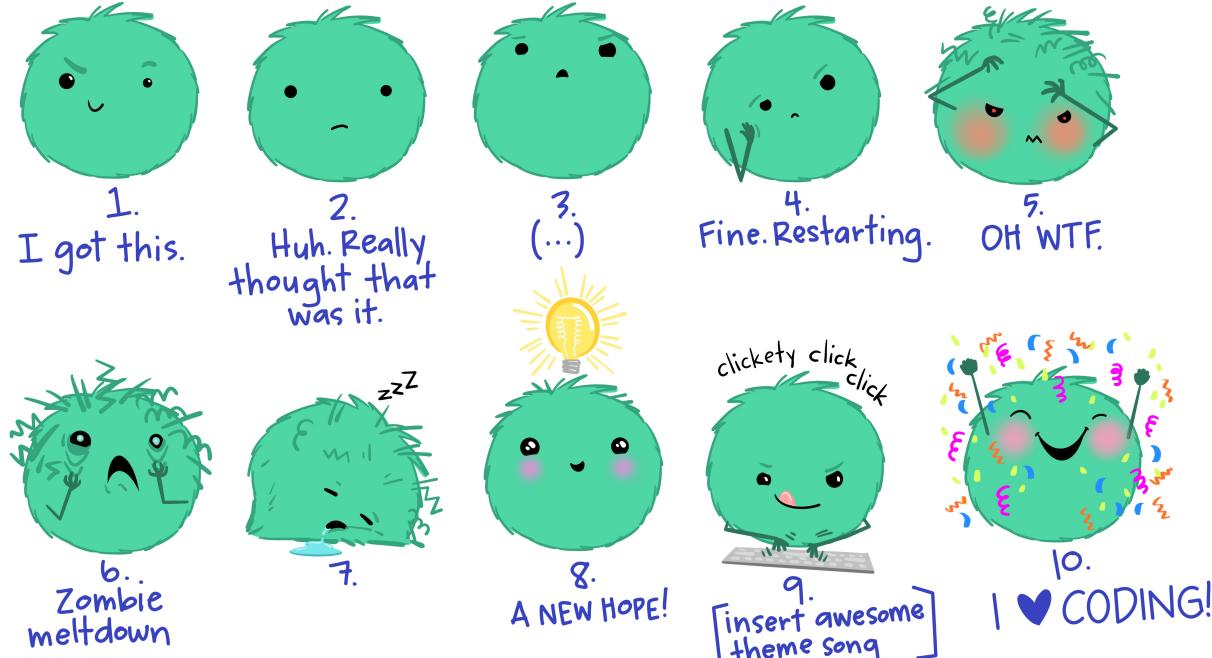
## Checklist

- ✓ 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



@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

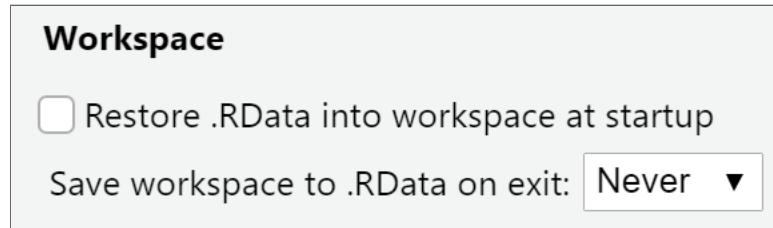
Have you tried turning it **OFF** and **ON** again?

Restart R, especially when things get weird.

Session -> Restart R or

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

**Tools -> Global Options -> Workspace**



### 3. Reprex

minimum **reproducible example**



<https://reprex.tidyverse.org/>

minimum

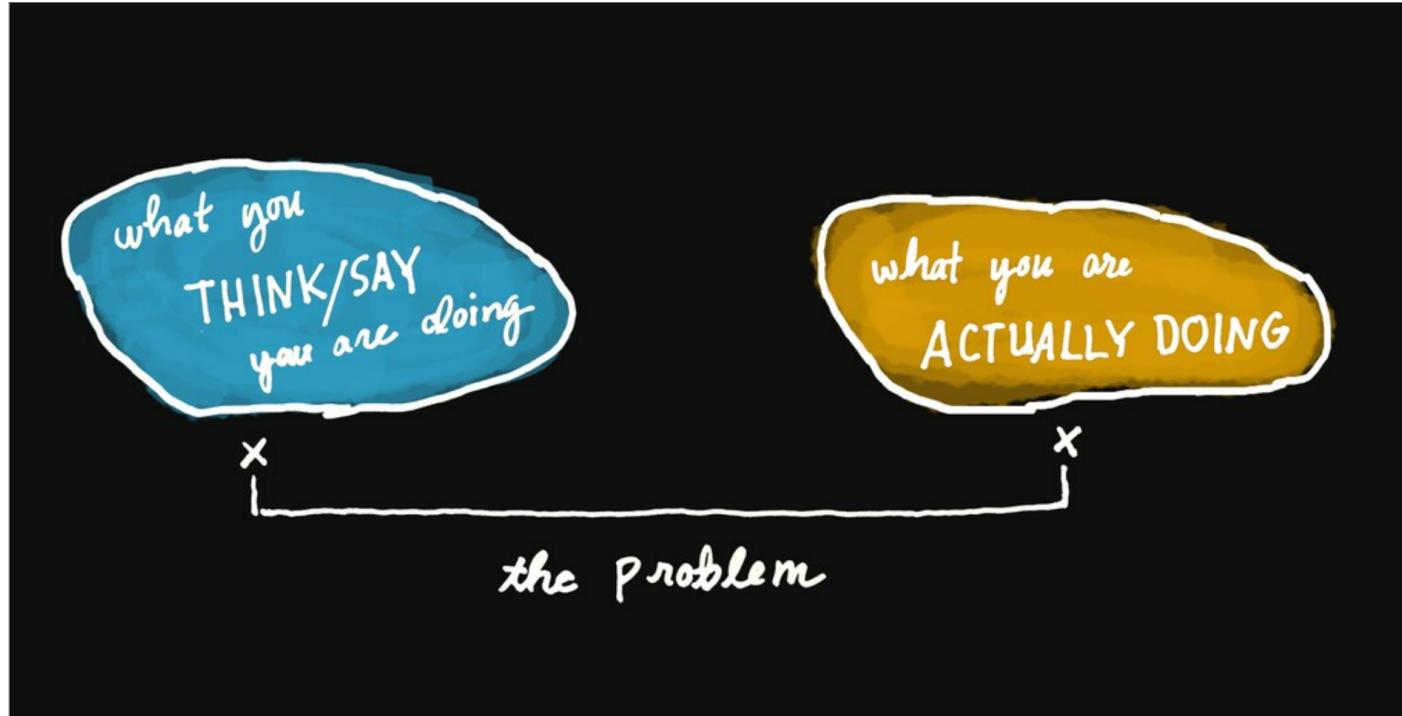
To FIND a NEEDLE



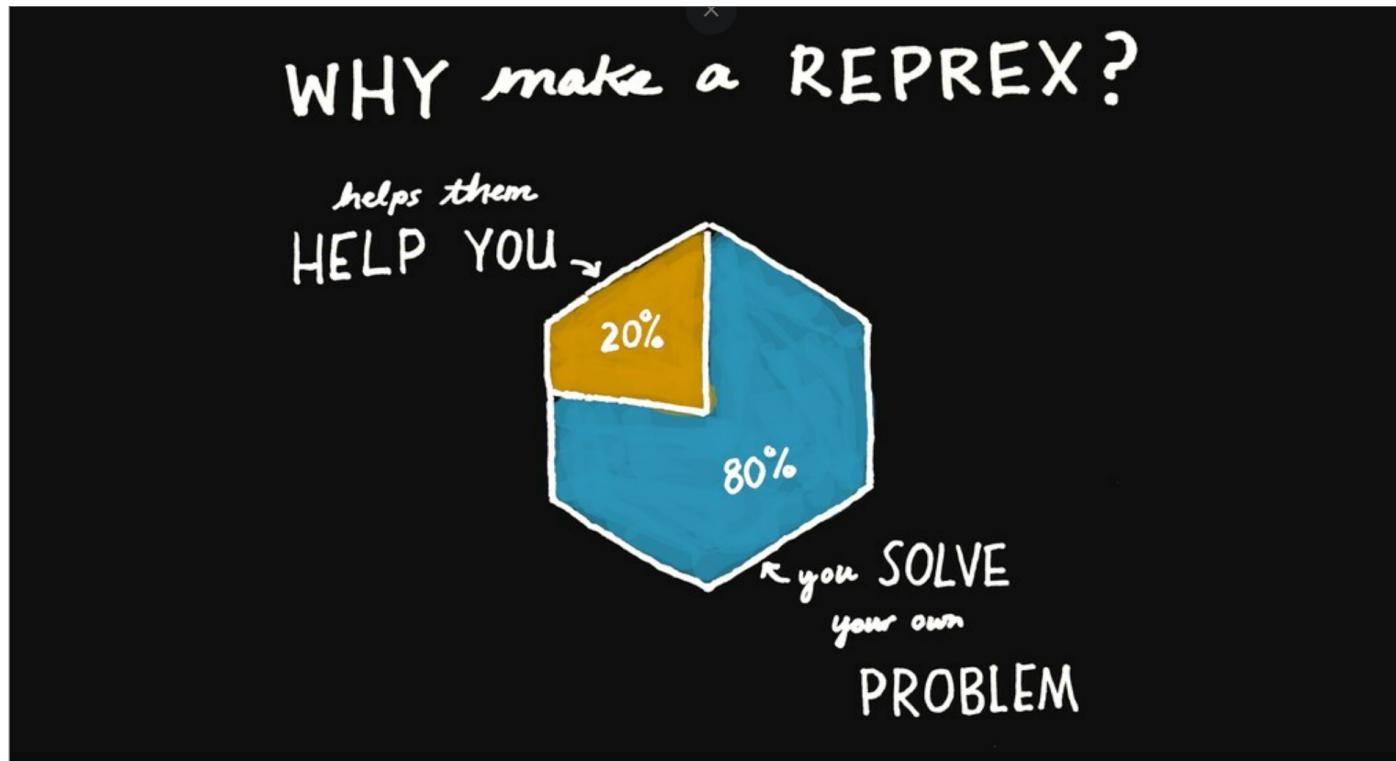
Try looking in a  
SMALLER STACK



## reproducible example



why reprex



## **Review**

Troubleshooting strategies:

1. Search
2. Reset
3. Reprex

But what if



#AgentBinky

## **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

	A traceback	⌚ interactive debugger	Q your code	↗ their code
Functions	 Q print() / cat() / message()			output diagnostic information in code
	 Q ↗ traceback()			locate error
	 Q browser()			open interactive debugger at browser() location
	 Q ↗ debug() / debugonce()			automatically open interactive debugger when function called
	 Q ↗ trace()			start debugger at specific location in function
options()	 Q ↗ error = rlang::entrace			provides richer traceback & error handling
	 Q ↗ error = recover			select frame from traceback to enter interactive debugger
	 ↗ warn = 2			upgrade warnings to errors
	 Q ↗ rlang_backtrace_on_error = "branch"			displays simplified backtrace
IDE	 Q breakpoint			open interactive debugger at breakpoint location
	 Q ↗ Debug -> On Error -> Error Inspector			provides "Show Traceback" & "Rerun with Debug" console options
	 Q ↗ Debug -> On Error -> Break in code			open interactive debugger on error

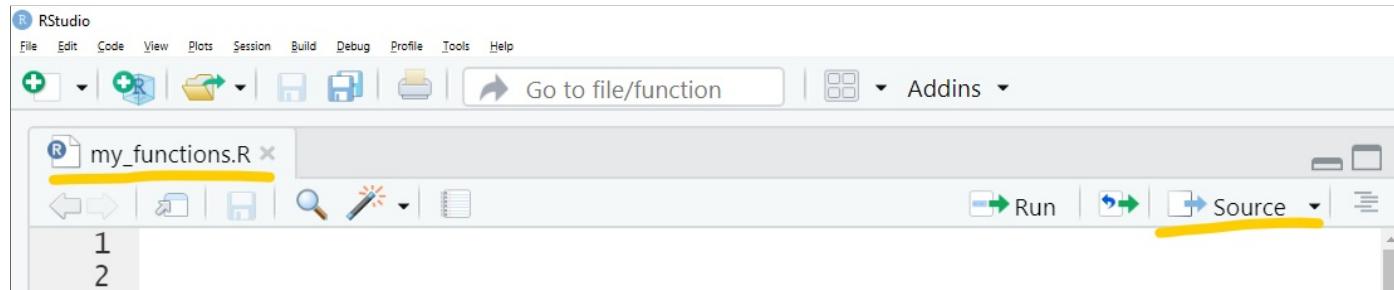
This is a lot.

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

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

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

## **Debugging your code**

## print()

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

```
1 g("a")
```

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

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

## cat()

```
1 f <- function(x) {  
2   cat("f()\n")  
3   cat("x =", x, "\n")  
4   x + 1  
5 }  
6  
7 g <- function(x) f(x)  
8  
9 g("a")
```

```
f()  
x = a
```

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

## traceback()

```
1 source("demo/my_functions.R")
2 g("a")
3 traceback()
```

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

shows the sequence of calls that lead to the error.

## Richer traceback

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

<code>traceback()</code>	<code>where</code>	<code>recover()</code>	<code>rlang functions</code>
5: <code>stop("...")</code>			
4: <code>i(c)</code>	where 1: <code>i(c)</code>	1: <code>f()</code>	1. <code>\_\_global::f(10)</code>
3: <code>h(b)</code>	where 2: <code>h(b)</code>	2: <code>g(a)</code>	2. <code>\_\_global::g(a)</code>
2: <code>g(a)</code>	where 3: <code>g(a)</code>	3: <code>h(b)</code>	3. <code>\_\_global::h(b)</code>
1: <code>f("a")</code>	where 4: <code>f("a")</code>	4: <code>i("a")</code>	4. <code>\_\_global::i("a")</code>

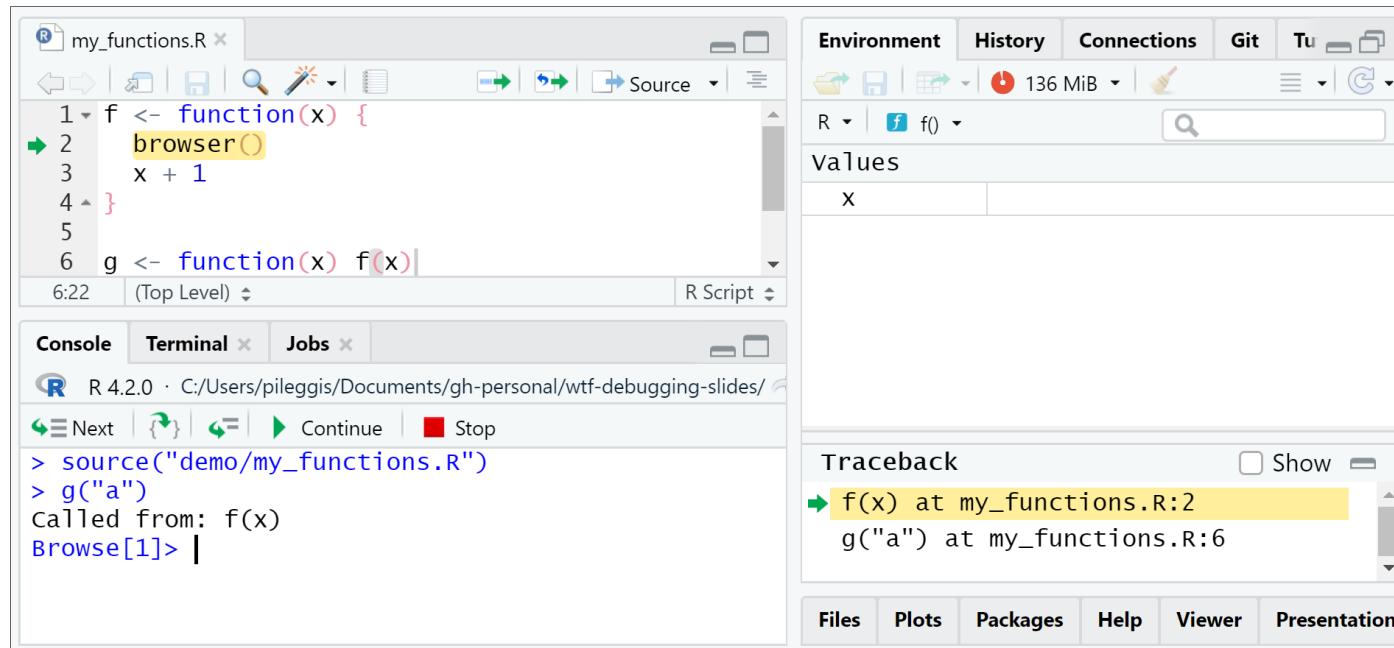
## browser()

```
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:

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

# Debugging your code

	A traceback	I interactive debugger	Q your code	A their code
Functions	   print() / cat() / message()	output diagnostic information in code		
	   traceback()	locate error		
	  browser()	open interactive debugger at browser() location		
	   debug() / debugonce()	automatically open interactive debugger when function called		
	   trace()	start debugger at specific location in function		
options	   error = rlang::entrance	provides richer traceback & error handling		
	   error = recover	select frame from traceback to enter interactive debugger		
	  warn = 2	upgrade warnings to errors		
	   rlang_backtrace_on_error = "branch"	displays simplified backtrace		
IDE	  breakpoint	open interactive debugger at breakpoint location		
	   Debug -> On Error -> Error Inspector	provides "Show Traceback" & "Rerun with Debug" console options		
	  Debug -> On Error -> Break in code	open interactive debugger on error		

## Your turn, exercise 01

```
1 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)

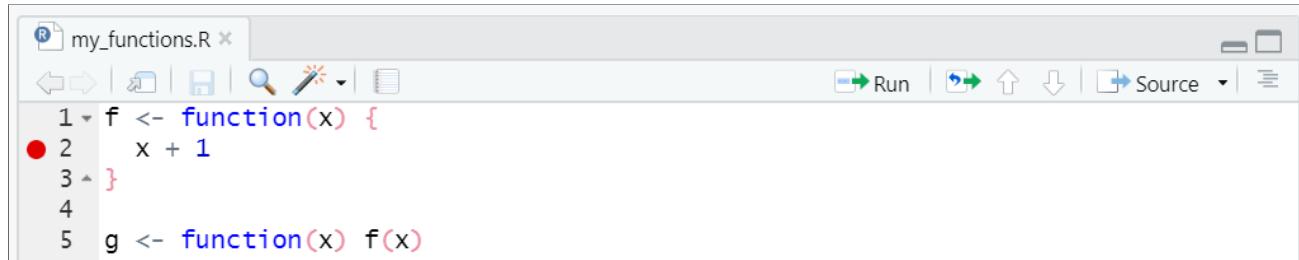
15:00

# Debugging in RStudio

## Editor breakpoints

red circle = breakpoint = `browser()`

(but you don't have to change your code)



The screenshot shows the RStudio IDE interface with a source editor window titled "my\_functions.R x". The code contains five lines:

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

A red circular breakpoint marker is placed on the line number 2 of the first function definition. The RStudio toolbar at the top includes icons for file operations, search, and run.

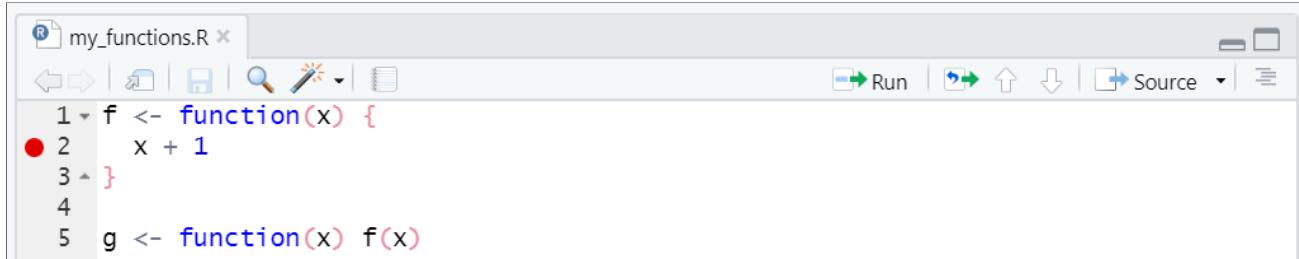
Set / re-set an editor breakpoint:

- 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)



The screenshot shows the RStudio IDE interface with a file named "my\_functions.R" open. The code editor displays the following R script:

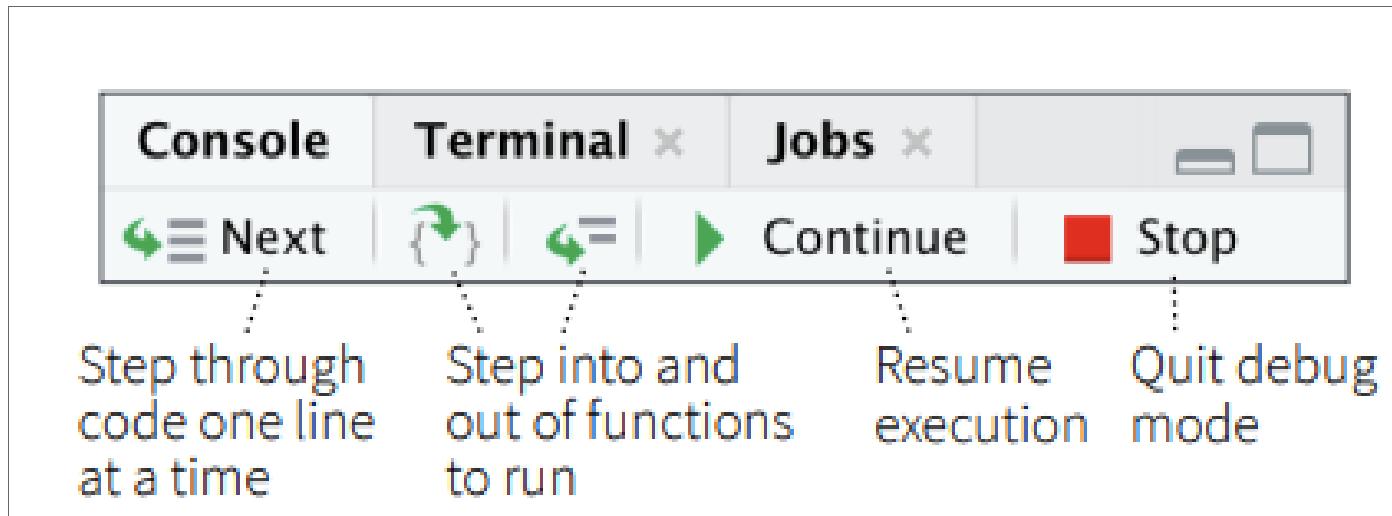
```
1 f <- function(x) {  
● 2   x + 1  
3 }  
4  
5 g <- function(x) f(x)
```

A red circular breakpoint marker is positioned at the start of line 2. The IDE's toolbar and menu bar are visible at the top.

To activate, either

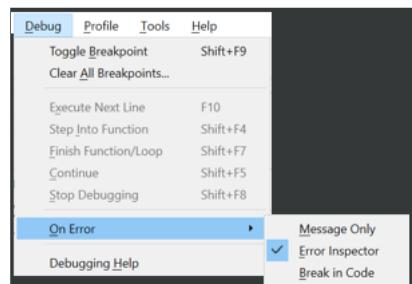
- click IDE Source button, or
- `debugSource("demo/my_functions.R")`

## Debugging console



## IDE on error

Automatically invoke actions on error.



New set up for demonstration.

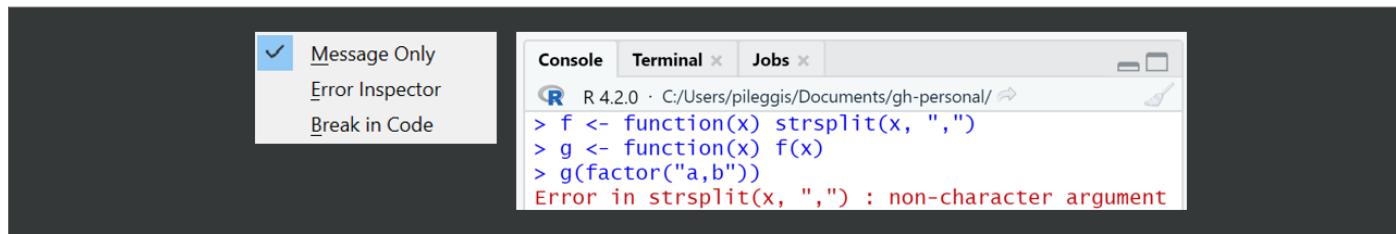
```
1 f <- function(x) { strsplit(x, ",") }
2 g <- function(x) { f(x) }
3 g("a,b")
```

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

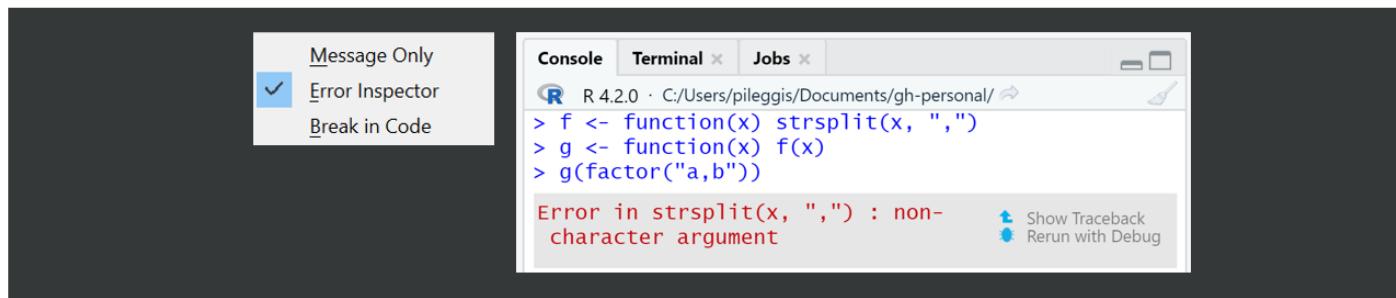
```
1 g(factor("a,b"))
```

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

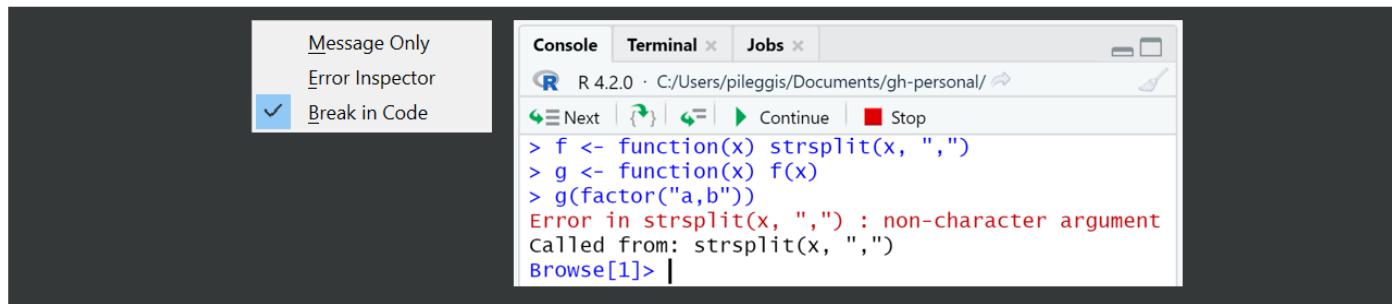
## IDE message only



## IDE error inspector



## IDE break in code



## Error inspector

```
1 # IDE Error Inspector not triggered
2 f <- function(x) x + 1
3 g <- function(x) f(x)
4 g("a")
5
6 # IDE Error Inspector not triggered
7 strsplit(factor("a,b"), ",")
8
9 # IDE Error Inspector not triggered
10 f <- function(x) strsplit(x, ",")
11 f(factor("a,b"))
12
13 # IDE Error Inspector yes triggered
14 g <- function(x) f(x)
15 g(factor("a,b"))
```

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

# Debugging in the RStudio IDE

	A traceback	⌚ interactive debugger	🔍 your code	↗ their code
<b>Functions</b>	  	print() / cat() / message() traceback() browser() debug() / debugonce() trace()	output diagnostic information in code locate error open interactive debugger at browser() location automatically open interactive debugger when function called start debugger at specific location in function	
<b>options()</b>	     	error = rlang::entrance error = recover warn = 2 rlang_backtrace_on_error = "branch"	provides richer traceback & error handling select frame from traceback to enter interactive debugger upgrade warnings to errors displays simplified backtrace	
<b>IDE</b>	      	breakpoint Debug -> On Error -> Error Inspector Debug -> On Error -> Break in code	open interactive debugger at breakpoint location provides "Show Traceback" & "Rerun with Debug" console options open interactive debugger on error	

## 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

```
1 # source if code is yours, otherwise not necessary
2 source("demo/my_functions.R")
3 # set debugging flag
4 debug("g")
5 g("a")
6 # turn off debugging flag
7 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

```
1 # source if code is yours, otherwise not necessary
2 source("demo/my_functions.R")
3 # set debugging flag
4 debugonce("g")
5 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 `browser()` in first line of function
- `debug(fun)`

you can 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

The screenshot shows the RStudio interface with two main panes: the Environment pane at the top and the Console pane below it.

**Environment Pane:**

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 ...")

**Console Pane:**

```
R 4.2.0 · C:/Users/pileggis/Documents/gh-personal/ ↵
> as.list(x[[3]])
[[1]]
`if`  
[[2]]
!is.array(x) || length(dn <- dim(x)) < 2L  
[[3]]
stop("'x' must be an array of at least two dimensions")
```

## trace() at specified step

```
1 # identify spot to insert code
2 as.list(x[[3]][[1]])
3 # equivalent notation
4 as.list(x[[c(3, 1)]])
5
6 # insert browser at step 3.1
7 trace(colSums, browser, at = list(c(3, 1)))
8 # execute function
9 colSums(1:3)
10 # ls.str() handy in browser
11 # cancel tracing
12 untrace(colSums)
```

# Debugging their code

	A traceback	⌚ interactive debugger	Q your code	👤 their code
Functions	  	print() / cat() / message() traceback() browser() debug() / debugonce() trace()	output diagnostic information in code locate error open interactive debugger at browser() location automatically open interactive debugger when function called start debugger at specific location in function	
options()	     	error = rlang::entrance error = recover warn = 2 rlang_backtrace_on_error = "branch"	provides richer traceback & error handling select frame from traceback to enter interactive debugger upgrade warnings to errors displays simplified backtrace	
IDE	      	breakpoint Debug -> On Error -> Error Inspector Debug -> On Error -> Break in code	open interactive debugger at breakpoint location provides "Show Traceback" & "Rerun with Debug" console options open interactive debugger on error	

## Your turn, exercise 03

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

Complete 03\_exercise to practice debugging others' code.

10:00

## Pick your favorite

Q your code

A their code



interactive debugger



Q IDE: Debug -> On Error -> Error Inspector

A Q traceback()

A Q options(error = rlang::entrace)

Q IDE: Debug -> On Error -> Error Inspector

Q IDE: Debug -> On Error -> Break in code

Q IDE: breakpoint

Q browser()

A Q debug() / debugonce()

A Q trace()

A Q options(error = recover)

## **Special cases**

## Warnings

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

```
1 ?options
2 options(warn = 0) # default, stores warnings until top-level function returns
3 options(warn = 1) # warnings are printed as they occur
4 options(warn = 2) # upgrades warnings to errors
5
6
7 # initiate recover on warning
8 options(warn = 2, error = recover)
9 # restore original settings
10 options(warn = 0, error = NULL)
```

## Piped expressions

tracebacks can be verbose with pipes

```
1 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

	A traceback	I interactive debugger	Q your code	A their code
Functions	  	print() / cat() / message() traceback() browser() debug() / debugonce() trace()	output diagnostic information in code locate error open interactive debugger at browser() location automatically open interactive debugger when function called start debugger at specific location in function	
options()	     	error = rlang::entrance error = recover warn = 2 rlang_backtrace_on_error = "branch"	provides richer traceback & error handling select frame from traceback to enter interactive debugger upgrade warnings to errors displays simplified backtrace	
IDE	      	breakpoint Debug -> On Error -> Error Inspector Debug -> On Error -> Break in code	open interactive debugger at breakpoint location provides "Show Traceback" & "Rerun with Debug" console options open interactive debugger on error	

**Read the source**



## (almost) all source on GitHub

[github.com/search](https://github.com/search)

All of GitHub! Can do advanced searches, too.

---

[github.com/cran](https://github.com/cran)

CRAN packages

---

[github.com/wch/r-source](https://github.com/wch/r-source)

R core

[About searching on GitHub](#)

# google: cran pkgname

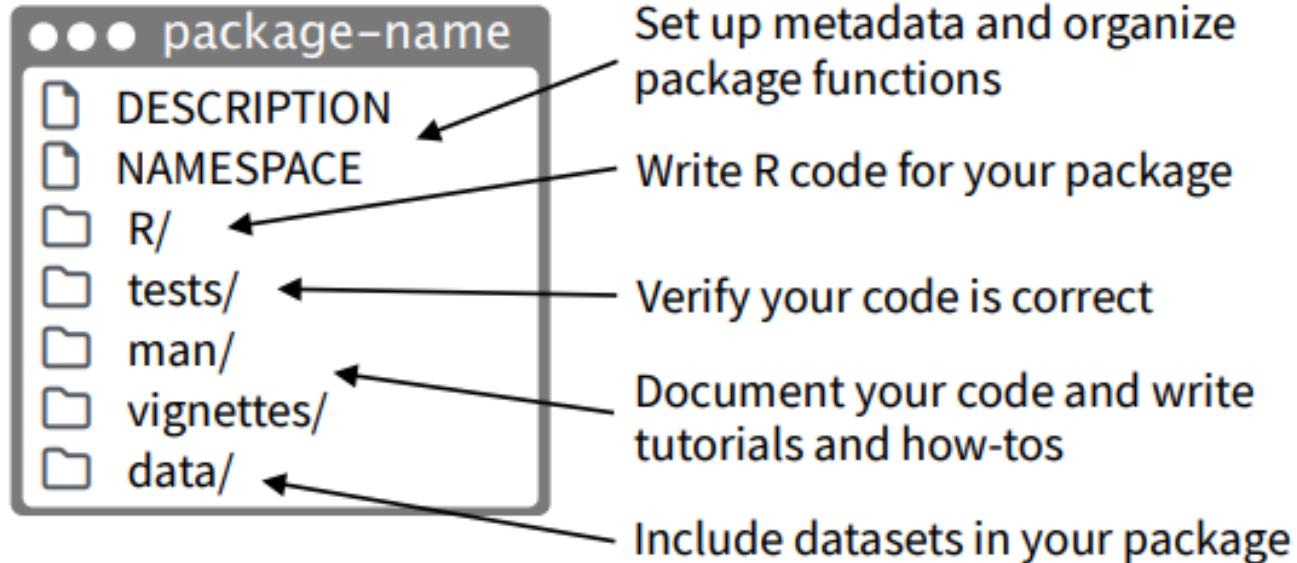
The screenshot shows a web browser window displaying the CRAN package page for 'usethis'. The title bar reads 'CRAN - Package usethis'. The page content includes the package's name, version (2.1.6), and a brief description: 'Automate package and project setup tasks that are otherwise performed manually. This includes setting up unit testing, test coverage, continuous integration, Git, 'GitHub', licenses, 'Rcpp', 'RStudio' projects, and more.' It lists dependencies, imports, and suggests packages, along with author information (Hadley Wickham, Jennifer Bryan, Malcolm Barrett) and a maintainer email. The 'BugReports' link (<https://github.com/r-lib/usethis/issues>) is highlighted with a yellow box. The 'License' is MIT + file LICENSE, and the URL is <https://usethis.r-lib.org>. The 'Downloads' section includes links for package source, Windows binaries, macOS binaries, and old sources, with the 'Old sources' link ([usethis archive](#)) also highlighted with a yellow box.

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](https://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](https://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](https://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](https://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

x