

Writing good functions

Hadley Wickham

Chief Scientist
RStudio

1. What is a function?

2. What is a good function?

3. How can you write good functions?

**What is a
function?**

Your turn

Every function has three key properties that defines its behaviour. What are they?

```
add <- function(x, y) {  
  x + y  
}  
formals(add)  
body(add)  
environment(add)
```

```
# Environment is important part of scoping.  
# If can't find names inside function, next  
# looks in the parent environment.
```

```
# Is the name of a function important?
```

What does this function do? How do you use it?

```
`last<-` <- function(x, value) {  
  x[length(x)] <- value  
  x  
}
```

What does this function do? How do you use it?

```
`%+%` <- function(a, b) {  
  paste0(a, b)  
}
```

```
# A replacement function
`last<-` <- function(x, value) {
  x[length(x)] <- value
  x
}
x <- runif(sample(10))
last(x) <- 10

# An infix function
`%+%` <- function(a, b) {
  paste0(a, b)
}
"this" +% " makes" +% " one string"
```

Your turn

How do you find the source code of a function? What are two common roadblocks to finding the source? How can you overcome them?


```
# Can usually just type the name  
nrow
```

```
# But functions might be implemented in C  
sum
```

```
pryr::show_c_source(.Primitive("sum"))
```

```
# Or might be a method of a generic  
mean
```

```
mean.default
```

```
getS3method("mean", "default")
```

**What is a
good
function?**

Your turn

With your neighbours, brainstorm what makes a good function.

(If that's too hard, think about what makes a bad function)

My thoughts

- Correct
- Maintainable
- Fast
- General vs. simple
- Does one thing well vs. solves entire class of problems
- Concise vs. explicit

Correct code

<http://adv-r.had.co.nz/Exceptions-Debugging.html>

Testing

- Will focus on systematic unit testing tomorrow afternoon – this is so important!
- Today we'll discuss basic debugging techniques and ways to make your code more robust.

Your turn

How do you find out where an error occurred?

How do you start R's interactive debugger?

How do you raise an error in your own code?

`traceback()`

`browser()`

`options(error = recover)`

`options(error = NULL)`

`debugonce(f)`

`setBreakpoint(...)`


`# RStudio: breakpoints`

`stop("This is an error!")`


```
> f(10)
```

```
Error in "a" + d : non-numeric argument to binary operator
```


 Show Traceback

 Rerun with Debug

```
> f(10)
```

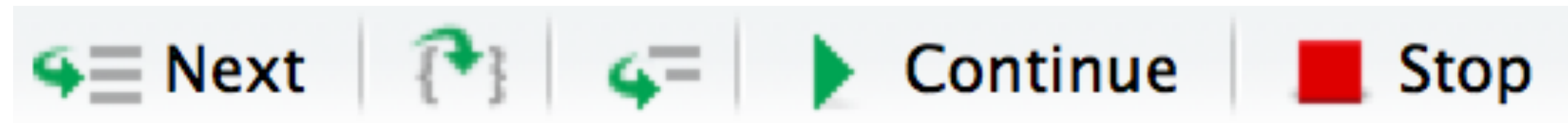
```
Error in "a" + d : non-numeric argument to binary operator
```

 Hide Traceback

 Rerun with Debug

```
4 i(c) at exceptions-example.R#3
3 h(b) at exceptions-example.R#2
2 g(a) at exceptions-example.R#1
1 f(10)
```

Step into



Step out of

```
f <- function() g()
g <- function() h()
h <- function() i()
i <- function() j()
j <- function() stop("Error!")
```

```
f()
```

```
h <- function() {
  x <- 2
  y <- 3
  browser()
  i()
}
```

```
f()
```

```
f <- function() g()  
g <- function() h()  
h <- function() i()  
i <- function() j()  
j <- function() stop("Error!")
```

```
options(error = recover)  
j()  
options(error = NULL)
```

```
# In batch R process ----
dump_and_quit <- function() {
  # Save debugging info to file last.dump.rda
  dump.frames(to.file = TRUE)
  # Quit R with error status
  q(status = 1)
}
options(error = dump_and_quit)

# In a later interactive session ----
load("last.dump.rda")
debugger()
```

Robust code

- Spend time now to save time later
- Be explicit: use `TRUE` and `FALSE`, not `T` and `F`; use `foo::bar()`
- Avoid functions that use non-standard evaluation (no `subset`, `with`, `transform`)
- Avoid functions that have different types of output (avoid `sapply`, beware `[]`)

Check preconditions

Always best to fail early - as soon as you know something is wrong.

If your function expects certain types of input, it's a good idea to test that they are as expected. `stopifnot` is a quick and dirty way of doing so.

Your turn

Take the function on the next page and make it work more reliably, or at least identifying why it is failing.


```
col_means <- function(df) {  
  numeric <- sapply(df, is.numeric)  
  numeric_cols <- df[, numeric]  
  
  data.frame(lapply(numeric_cols, mean))  
}
```

```
col_means(mtcars)  
col_means(mtcars[, 0])  
col_means(mtcars[0, ])  
col_means(mtcars[, "mpg", drop = F])  
col_means(1:10)  
col_means(as.matrix(mtcars))  
col_means(as.list(mtcars))
```

```
mtcars2 <- mtcars  
mtcars2[-1] <- lapply(mtcars2[-1], as.character)  
col_means(mtcars2)
```

No peeking until you've
made an attempt!

```
col_means <- function(df) {  
  stopifnot(is.data.frame(df))  
  # OR  
  # df <- as.data.frame(df)  
  
  numeric <- vapply(df, is.numeric, logical(1))  
  numeric_cols <- df[, numeric, drop = FALSE]  
  
  data.frame(lapply(numeric_cols, mean))  
}
```

Maintainable code

<http://adv-r.had.co.nz/Style.html>

Tips

- Code gets faster as computers get faster. It never gets correct by itself, and it never gets more elegant.
- Pick a style guide and stick with it.
- Balance concision and cleverness.

```
# Imagine you have a vector of events that you  
# want to divide into groups. You know when  
# an event ends. How could you generate a  
# unique integer for each group?
```

```
x <- sample(c(TRUE, FALSE), prob = c(0.2, 0.8),  
            100, rep = T)
```

```
# Brainstorm for 2 minutes
```

Uses very simple ideas, but
many places to make mistakes

```
i <- 1
out <- numeric(length(x))
out[1] <- i
for (j in 2:length(x)) {
  if (x[j] != x[j - 1]) {
    i <- i + 1
  }
  out[j] <- i
}
out
```

Very terse, but once you
get the idea, expressive

Too clever?

```
cumsum(c(TRUE, diff(x) != 0))
```

Just right?

```
library(dplyr)
```

```
changed <- x != lag(x, default = !x[1])
```

```
cumsum(changed)
```


Tips

- Rewrite important code - your first attempt will not usually be the best approach.
- Use comments to explain why, not what or how.
- Use git & GitHub
(more on that tomorrow)

Fast code

Figure out what's slow.

Speed it up.

<http://adv-r.had.co.nz/Profiling.html>

What's slow?

Profiling

- R has a sampling profiler that stops execution every few ms and records the functions currently being run
- You can use it directly with `Rprof()`, but I've written a package to make the results a little easier to understand
- Still experimental:
`install_github("hadley/lineprof")`

```
library(lineprof)
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```

```
library(lineprof)
```

```
# Needed so we get line numbers
```

```
source("lineprof-ex.R")
```

```
l <- lineprof(f())
```

```
l
```

```
shine(l)
```

How can you make it
faster?

Speeding up code

- Look for existing solutions
- Do less
- Vectorise
- Parallelise
- Avoid copies
- Byte code compile

But first, setup an
experimental framework

```
library(microbenchmark)
```

```
x <- runif(100)  
microbenchmark(  
  sqrt(x),  
  x ^ 0.5,  
  exp(log(x) / 2)  
)
```

Your turn

What's the fastest way to get the bottom-right value from `mtcars` (i.e. 32nd from top, 11th from left)? e.g. `mtcars[32, 11]`

First brainstorm, then benchmark.

```
library(microbenchmark)
```

```
microbenchmark(  
  mtcars["Volvo 142E", "carb"],  
  mtcars[32, 11],  
  mtcars[[c(11, 32)]],  
  mtcars[["carb"]][32],  
  mtcars[[11]][32],  
  mtcars$carb[32],  
  .subset2(mtcars, 11)[32]  
)
```

Caution

These are microbenchmarks, which test a very very small specific piece of code. You must have correctly identified what is slow before they can be useful.

Your turn

You can find the function used to compute the BIC for linear models with `getS3method("BIC", "default")`. This function is much more complicated than it needs to be. Rewrite to simplify. How do your changes affect performance?

(Hint: if you have time, also look at `getS3method("logLik", "lm")`)

**Doing more
with less
code**

Rest of the day

- **Functional** programming: work with functions that take functions as input
- **Object oriented** programming: make code behave differently based on the type of input
- **Metaprogramming**: break all the rules!

This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.