Writing good functions

Hadley Wickham

Chief Scientist

RStudio

August 2015

1. What is a function?
2. How do you write a function?
3. Why write a function?
4. How do you write a good function?
5. Debugging

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 return?

```
y <- 10
g <- function() {
  x <- 5
  c(x = x, y = y)
}
g()
```

```
x <- 5
g <- function() {
  y <- 10
  c(x = x, y = y)
}
g()
```

```
x  y
5 10
```

```
z <- 10
h <- function() {
  y <- 10
  i <- function() {
    x <- 5
    c(x = x, y = y, z = z)
  }
  i()
}
h()
```

What does this function return?

```
z <- 10
h <- function() {
  y <- 10
  i <- function() {
    x <- 5
    c(x = x, y = y, z = z)
  }
  i()
}
h()
```

```
x  y  z
5 10 20
```

```
j <- function() {
  if (!exists("a")) {
    a <- 5
  } else {
    a <- a + 1
  }
  print(a)
}
j()

j()
```

What does this function return the first time you run it? The second time?

```
j <- function() {
  if (!exists("a")) {
    a <- 5
  } else {
    a <- a + 1
  }
  print(a)
}
j()
[1] 5
j()
[1] 5
```

What does this function return the first time you run it? The second time?

```r
# What does this function do? How do you use it?
`last<-` <- function(x, value) {
  x[length(x)] <- value
  x
}
```

```r
# What does this function do? How do you use it?
`%+%` <- function(a, b) {
  paste0(a, b)
}
```

```r
# 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"
```

How to write a function

Your turn

Which is easier?

a) Figure out how to solve a problem in general, then use that understand to solve a specific problem?

b) Solve an example and then figure out how to generalise?

Challenge

Given two vectors (of the same length) find how many positions have an NA in both locations.

```
both_na <- function(x, y) {
  # starting writing code
}
```

```r
x <- c(1, 1, NA, NA)
y <- c(1, NA, 1, NA)
```

```r
# Your turn
# Generate code that returns the correct answer
```

```
# Think about positions
length(which(which(is.na(x)) %in% which(is.na(y))))
```

```
# Boolean algebra
sum(is.na(x) & is.na(y))
```

```
# Once you've solved the problem, you
# can create a function
both_na <- function(x, y) {
  sum(is.na(x) & is.na(y))
}

both_na(x, y)
```

Why write a function?

```
sum(is.na(df$age1) & is.na(df$age2))
sum(is.na(df$trt1) & is.na(df$trt2))
sum(is.na(df$year1) & is.na(df$year2))
sum(is.na(df$sex1) & is.na(df$sex2))
sum(is.na(df$bar1) & is.na(df$bar2))
sum(is.na(df$foobar1) & is.na(df$foobar2))
sum(is.na(df$xyz1) & is.na(df$xyz2))
sum(is.na(df$abc1) & is.na(df$abc2))
sum(is.na(df$def1) & is.na(df$def2))
sum(is.na(df$ghi1) & is.na(df$ghi2))
```

Duplication

- Duplication is the enemy
- More code = more places to make mistakes
- A good rule of thumb is if you copy-and-paste something more than two times (i.e. 3 in total), you should attack the problem differently

```
passionn <- min(comp$passion,na.rm=T)
passionx <- max(comp$passion,na.rm=T)-passionn

leadershipn <- min(comp$leadership,na.rm=T)
leadershipx <- max(comp$leadership,na.rm=T)-leadershipn

loyaltyn <- min(comp$loyalty,na.rm=T)
loyaltyx <- max(comp$loyalty,na.rm=T)-loyaltyn

basicServn <- min(comp$basicServ,na.rm=T)
basicServx <- max(comp$basicServ,na.rm=T)-basicServn

educationn <- min(comp$education,na.rm=T)
educationx <- max(comp$education,na.rm=T)-educationn

safetyn <- min(comp$safety,na.rm=T)
safetyx <- max(comp$safety,na.rm=T)-safetyn

...
```

```
cityagg <- ddply(dat,.(city),summarise,
  wt=sum(svywt),
  people=length(svywt),
  passion=sum(svywt*((passion-passionn)/passionx),na.rm=T)/sum(svywt[!is.na(
  leadership=sum(svywt*((leadership-leadershipn)/leadershipx),na.rm=T)/sum(s
  loyalty=sum(svywt*((loyalty-loyaltyn)/loyaltyx),na.rm=T)/sum(svywt[!is.na(
  basicServ=sum(svywt*((basicServ-basicServn)/basicServx),na.rm=T)/sum(svywt
  education=sum(svywt*((education-educationn)/educationx),na.rm=T)/sum(svywt
  safety=sum(svywt*((safety-safetyn)/safetyx),na.rm=T)/sum(svywt[!is.na(safe
  aesthetic=sum(svywt*((aesthetic-aestheticn)/aestheticx),na.rm=T)/sum(svywt
  economy=sum(svywt*((economy-economyn)/economyx),na.rm=T)/sum(svywt[!is.na(
  socialOff=sum(svywt*((socialOff-socialOffn)/socialOffx),na.rm=T)/sum(svywt
  civicInv=sum(svywt*((civicInv-civicInvn)/civicInvx),na.rm=T)/sum(svywt[!is
  openness=sum(svywt*((openness-opennessn)/opennessx),na.rm=T)/sum(svywt[!is
  socialCap=sum(svywt*((socialCap-socialCapn)/socialCapx),na.rm=T)/sum(svywt
  domains=sum(svywt*((domains-domainsn)/domainsx),na.rm=T)/sum(svywt[!is.na(
  comOff=sum(svywt*((comOff-comOffn)/comOffx),na.rm=T)/sum(svywt[!is.na(comO
  comAttach=sum(svywt*((comAttach-comAttachn)/comAttachx),na.rm=T)/sum(svywt
)
```

```
# Your turn: turn this into a function.
# What variables do you need?

passionn <- min(comp$passion,na.rm=T)
passionx <- max(comp$passion,na.rm=T)-passionn

sum(svywt*((comp$passion-passionn)/
passionx),na.rm=T)/sum(svywt[!is.na(comp$passion)])
```

```r
f <- function(x, wt) {
  min_x <- min(x, na.rm = TRUE)
  rng_x <- max(x, na.rm = TRUE) - min_x
  sum(wt * ((x - min_x)/rng_x), na.rm = TRUE) /
    sum(wt[!is.na(x)])
}

# How could you make the intent clearer?
```

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1]
}
```

```
f <- function(x, wt) {
  sum(wt * rescale01(x), na.rm = TRUE) /
    sum(wt[!is.na(x)])
}
```

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1]
}

# Is this better?
f <- function(x, wt) {
  wt <- wt[!is.na(x)]
  x <- x[!is.na(x)]

  sum(wt * rescale01(x)) / sum(wt)
}
```

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1]
}
```

```
# Is this better?
f <- function(x, wt) {
  weighted.mean(rescale01(x), wt, na.rm = TRUE)
}
```

```
f <- function(x, y, z) {
  if (x) {
    y
  } else {
    z
  }
}
```

Focus on the essentials

```
f <- function(x, y, z) {
  if (x) {
    out <- y
  } else {
    out <- z
  }
  return(out)
}
```

```r
# Other common complications
x[which(is.na(x))]
x[is.na(x)]

x[-which(is.na(x))]
x[!is.na(x)]

x == TRUE
x

x == FALSE
!x

y == "a" | y == "b" | y == "c"
y %in% c("a", "b", "c")
```

Emphasise the structure

```
f <- function(x, y, z) if (x) y else z
```

How do you write a good function?

Your turn

With your neighbours, brainstorm what makes a good (or bad!) function.

My thoughts

- Correct
- **Obviously correct**
- Fast
- General/complex vs. specific/simple
- Concise/clever vs. long/easy

Readability 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.
- Use comments to explain why, not what or how.
- 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]) {
    i <- i + 1
  }
  out[j] <- i
}
out
```

```r
# Too clever?
cumsum(x) + !x[1]

# Little less clever
cumsum(x) + if(!x[1]) 1 else 0
```

```r
# Reasonably obvious & has place for comment
grp <- cumsum(x)
if (!x[1]) # first group should start at 1
  grp <- grp + 1
```

```r
# What does this code do?
paste0(
  "Good ",
  if (time <= 12) "morning" else "afternoon",
  ".",
  if (some_var) "This is extra text."
)

# What does if (FALSE) 3 return?
# What does paste0("x", NULL) return?
```

Robust code

- Spend time now to save time later
- Be explicit, e.g. `TRUE` and `FALSE`, not `T` and `F`
- Avoid functions that have different types of output (avoid `sapply`, beware `[`)
- Avoid functions that use non-standard evaluation (no `subset`, `with`, `transform`)
- Check preconditions and fail fast

```r
# Your turn: what's wrong with this function?
# How could you improve the error message?

impute_na1 <- function(x) {
  for (i in 1:length(x - 1)) {
    if (x[i] == "NA") {
      x[i] <- (x[i - 1] + x[i + 1]) / 2
    } else {
      x[i] <- x[i]
    }
  }
  x
}
impute_na1(c(1, 4, 5, "NA", 10, 13, 10))
```

Your turn

For what other inputs with `input_na()` fail? (Think about boundaries)

```
impute_na2(c(1, 4, 5, "NA", 10, 13, 10))
```

```
impute_na2(c(1, 4, 5, NA, 10, 13, 10))
```

```
# What should the answers be?

impute_na2(numeric())
impute_na2(NA_real_)

impute_na2(c(1, NA))
impute_na2(c(1, NA, NA, 2))
impute_na2(c(NA, 2))
```

```r
# A more general implementation

impute_na3 <- function(x) {
  miss <- is.na(x)
  interp <- approxfun(seq_along(x)[!miss], x[!miss])

  x[miss] <- interp(seq_along(x)[miss])
  x
}
```

Your turn

Discuss how the function works.

Argue about the behaviour of:

```
impute_na3(c(NA, NA))

impute_na3(c(NA, NA, 1))

impute_na3(c(NA, NA, 1, 2))
```

Vocabulary

- A broad R vocabulary lets you make use of existing R functions
- Existing functions are documented, better tested, often more general, ...
- But more importantly they will often have a standard name

“A rose by any other name
would smell as sweet.”
— Shakespeare

“A function by any other name
would not smell as sweet.”
— Hadley Wickham

Code = communication

- Rewrite important code: your first attempt isn’t usually the best approach.
- Consider the audience; what vocabulary should you assume?
- Being obviously correct is better than just being correct, but it may take a lot of time to get there.

Debugging

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

“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

Steps

1. Realise that you have a bug
2. Make it repeatable
3. Figure out where it is
4. Fix it and test the fix

Tools

1. RStudio error inspector/ `traceback()`
2. RStudio's rerun with debug/ `options(error = browser)`
3. RStudio's breakpoints/`browser()`

See the call stack

traceback()

```
> 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)
```

```
f <- function(x, y) g(x * y, y)
g <- function(x, y) j(y) + h(x)
h <- function(x) i(x - 5) * 2
i <- function(x) j(x) + 1
j <- function(x) {
  if (x < 0) {
    stop("`x` must be positive")
  }

  log(x)
}

f(10, 10)
f(2, 2)
f(10, -3)
```

Post-mortem

```
> f(10)
```

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

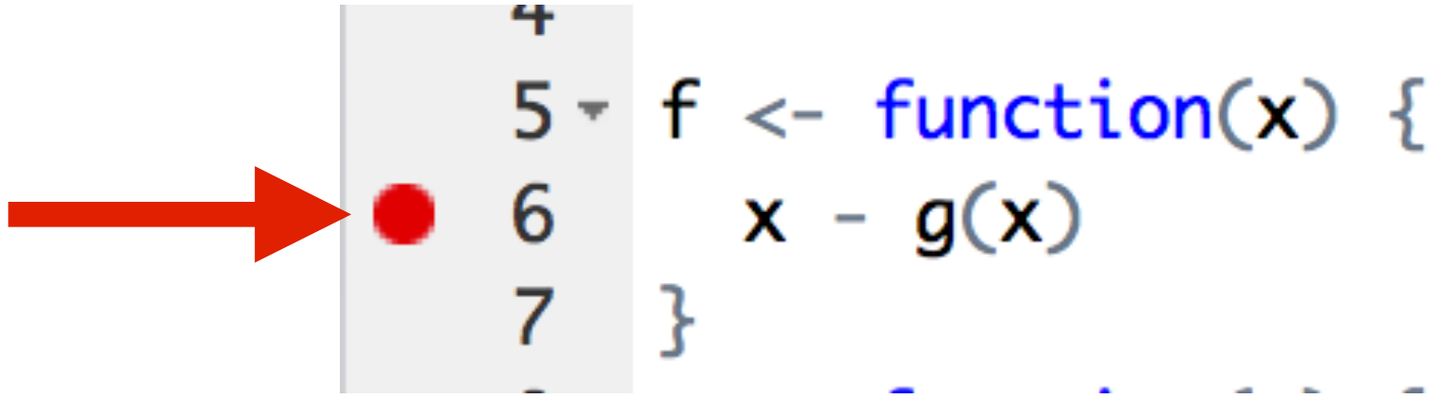
Show Traceback

Rerun with Debug

or

```
browseOnce <- function() {
  old <- getOption("error")
  function() {
    options(error = old)
    browser()
  }
}
options(error = browseOnce())
```

Pre-mortem

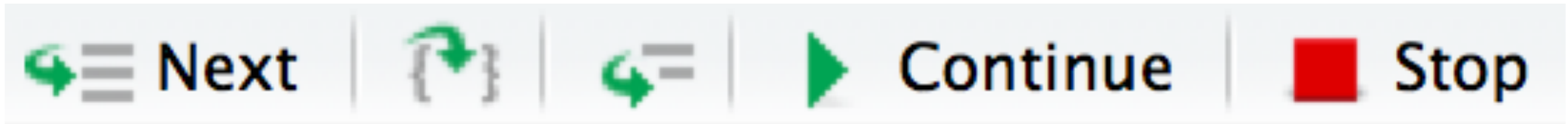


or

```
5 f <- function(x) {
6   browser()
7   x - g(x)
8 }
```

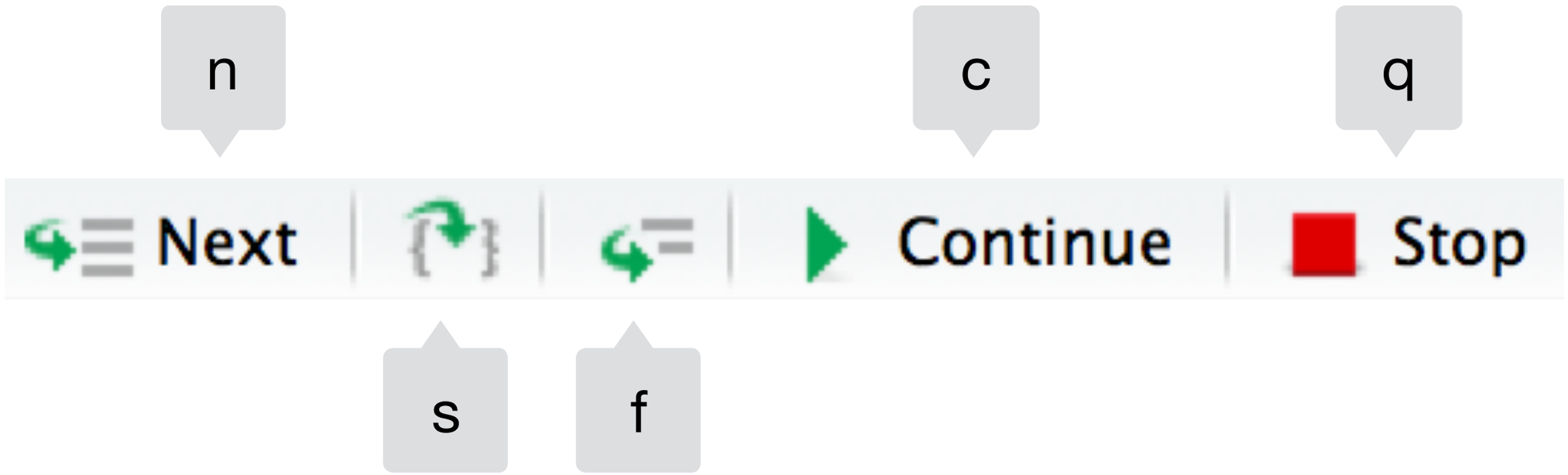
Controlling the debugger: RStudio

Step into



Step out of

Controlling the debugger: Keyboard



By default, enter is also equivalent to n.
Disable with `options(browserNLdisabled = TRUE)`

Post-mortem debugging on another server

```
# 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()
```

Testing

- Debugging gets it working now; testing ensure that it keeps working in the future. Really important!
- Recommend that you learn how to use `testthat`: http://r-pkgs.had.co.nz/tests.html

Reducing duplication

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
- **Meta**programming: break all the rules!