

Functions

Back at 10:20am

<http://bit.ly/advr16-1>

January 2017

Hadley Wickham

[@hadleywickham](#)

Chief Scientist, RStudio

Warmups

What does this function return?

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

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

What does this function return the 1st time it's run?

The 2nd time?

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

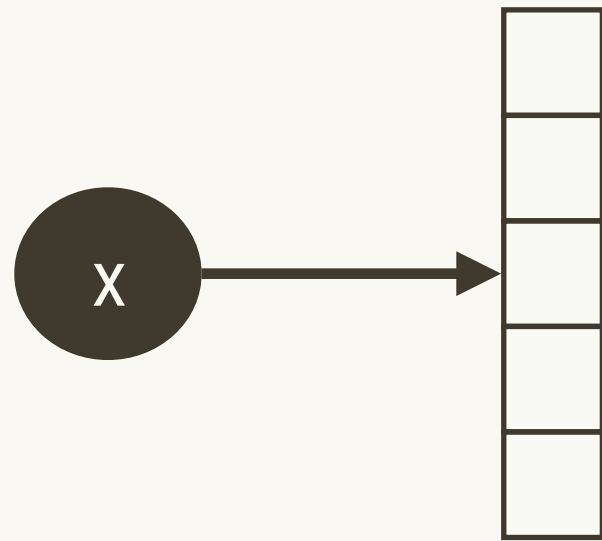
```
j <- function() {  
  if (!exists("a")) {  
    a <- 5  
  } else {  
    a <- a + 1  
  }  
  a  
}  
j()  
#> [1] 5  
j()  
#> [1] 5
```

Your turn

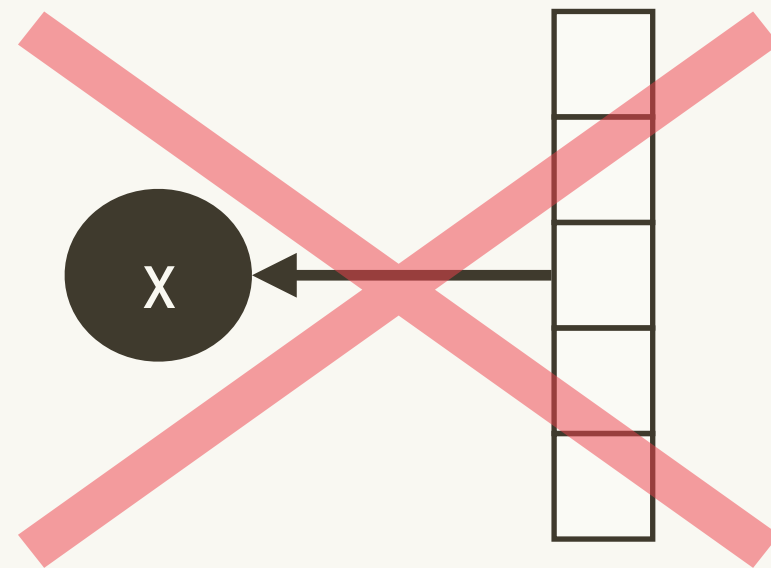
Every function has **three** key properties that defines its behaviour. What are they?

```
add <- function(x, y) {  
  x + y  
}  
formals(add)  
body(add)  
environment(add)  
# Environment controls scoping  
  
# Is the name important?  
# How many names can a function have?
```


Remember:



A name “has” an object



An object doesn't have a name

Your turn

How would you normally write this code?

```
`+` (1, `*` (2, 3))
```

```
#> [1] 7
```

In R, any function can use **prefix** form

$f(1, g(2, 3))$

``+`(1, `*`(2, 3))`

$1 + 2 * 3$

This lets you do crazy things like redefine +

```
`+` <- function(x, y) {  
  if (runif(1) < 0.01) {  
    sum(x, y) * 1.1  
  } else {  
    sum(x, y)  
  }  
}
```

To understand computations in R,
two slogans are helpful:

- Everything that **exists** is an **object**.
- Everything that **happens** is a **function call**.

—*John Chambers*

How to write a function

Your turn

Which is easier?

- a) Figure out how to solve a problem in general, then use that understanding to solve a specific problem?
- b) Solve an example and then figure out how to generalise?

Challenge

Given two vectors (of the same length) how many positions have an NA in the same place in both vectors?



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

Your turn

```
# Write code to determine how many positions  
# have an NA in both vectors.
```

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

```
# Correct answer is 1
```

```
# If you finish early make a function,  
# then think about what should happen if  
# x and y aren't the same lengths
```

There are two approaches to solve this problem:

Think about positions

```
length(intersect(which(is.na(x)), which(is.na(y))))  
length(which(which(is.na(x)) %in% which(is.na(y))))  
sum(which(is.na(x)) %in% which(is.na(y)))
```

Boolean algebra

```
is.na(x) & is.na(y)  
sum(is.na(x) & is.na(y))
```

Create a function **after** you've solved the problem!

```
both_na <- function(x, y) {  
  sum(is.na(x) & is.na(y))  
}
```

```
both_na(x, y)
```

```
# What happens if x and y aren't the same length?
```

Why write a
function?

Duplicated code hides intent

```
sum(is.na(df$age1) & is.na(df$age2))  
sum(is.na(df$year1) & is.na(df$year2))  
sum(is.na(df$sex1) & is.na(df$year2))  
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))
```

Duplicated code = opportunities for errors

Rule of thumb: **3 copies** is ok

```
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))
```

Time to write a function:

```
sum(is.na(df$abc1) & is.na(df$abc1))  
sum(is.na(df$def1) & is.na(df$def2))  
sum(is.na(df$ghi1) & is.na(df$ghi2))  
sum(is.na(df$jkl1) & is.na(df$jkl2))
```

Duplicated code = opportunities for errors

Rule of thumb: **3 copies** is ok

```
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))
```

Time to write a function:

```
both_na(df$abc1, df$abc2)  
both_na(df$def1, df$def2)  
both_na(df$ghi1, df$ghi2)  
both_na(df$jkl1, df$jkl2)
```

(We'll learn how to deal with duplicated function calls in the next section)

What does this code do?

```
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
```

...

What does this code do?

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

What variables do you need?

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

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

What variables do you need?

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

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

What should you call them?

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

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

What should you call them?

```
min_x <- min(x, na.rm=T)
```

```
rng_x <- max(x, na.rm=T) - min_x
```

```
sum(wt * ((x - min_x) / rng_x), na.rm=T) / sum(wt[!is.na(x)])
```

How can you improve this function?

```
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)])  
}
```

What's the intent of `na.rm = TRUE`?

```
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)])  
}
```


Is this better?

```
rescale01 <- function(x) {  
  rng <- range(x, na.rm = TRUE)  
  (x - rng[1]) / (rng[2] - rng[1])  
}  
  
f <- function(x, wt) {  
  wt <- wt[!is.na(x)]  
  x <- x[!is.na(x)]  
  
  sum(wt * rescale01(x)) / sum(wt)  
}
```

Is this better?

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

```
f <- function(x, wt) {  
  # First, remove rows correspond to missing x  
  wt <- wt[!is.na(x)]  
  x <- x[!is.na(x)]  
  
  sum(wt * rescale01(x)) / sum(wt)  
}
```

Is this better?

```
rescale01 <- function(x) {  
  rng <- range(x, na.rm = TRUE)  
  (x - rng[1]) / (rng[2] - rng[1])  
}  
  
f <- function(x, wt) {  
  wt_not_miss <- wt[!is.na(x)]  
  x_not_miss <- x[!is.na(x)]  
  
  sum(wt_not_miss * rescale01(x_not_miss)) /  
    sum(wt_not_miss)  
}
```

Is this better?

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

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

Is this better?

```
rescale01 <- function(x) {  
  rng <- range(x, na.rm = TRUE)  
  (x - rng[1]) / (rng[2] - rng[1])  
}  
  
f <- function(x, wt) {  
  missing <- is.na(x) | is.na(wt)  
  x <- x[!missing]  
  wt <- wt[!missing]  
  
  sum(wt * rescale01(x)) / sum(wt)  
}
```

Is this better?

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

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

Is this better?

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

```
weighted_rescaled_mean <- function(x, wt) {  
  weighted.mean(rescale01(x), wt, na.rm = TRUE)  
}
```

```
weighted_rescaled_mean <- function(x, wt) {  
  weighted.mean(rescale01(x), wt, na.rm = TRUE)  
}  
df$new_var <- weighted_rescaled_mean(df$x, df$w)  
df <- mutate(df, weighted_rescaled_mean(x, w))
```

```
weighted_rescaled_mean <- function(data, x_name,  
wt_name) {  
  weighted.mean(rescale01(data[["x_name"]]), wt,  
na.rm = TRUE)  
}  
df <- weighted_rescaled_mean(df, "x", "w")
```


How do you write
a good function?

Your turn

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

What makes a good function?

Correct




**Obviously
correct**

Understandable

Fast (enough)

Important to think about other tensions

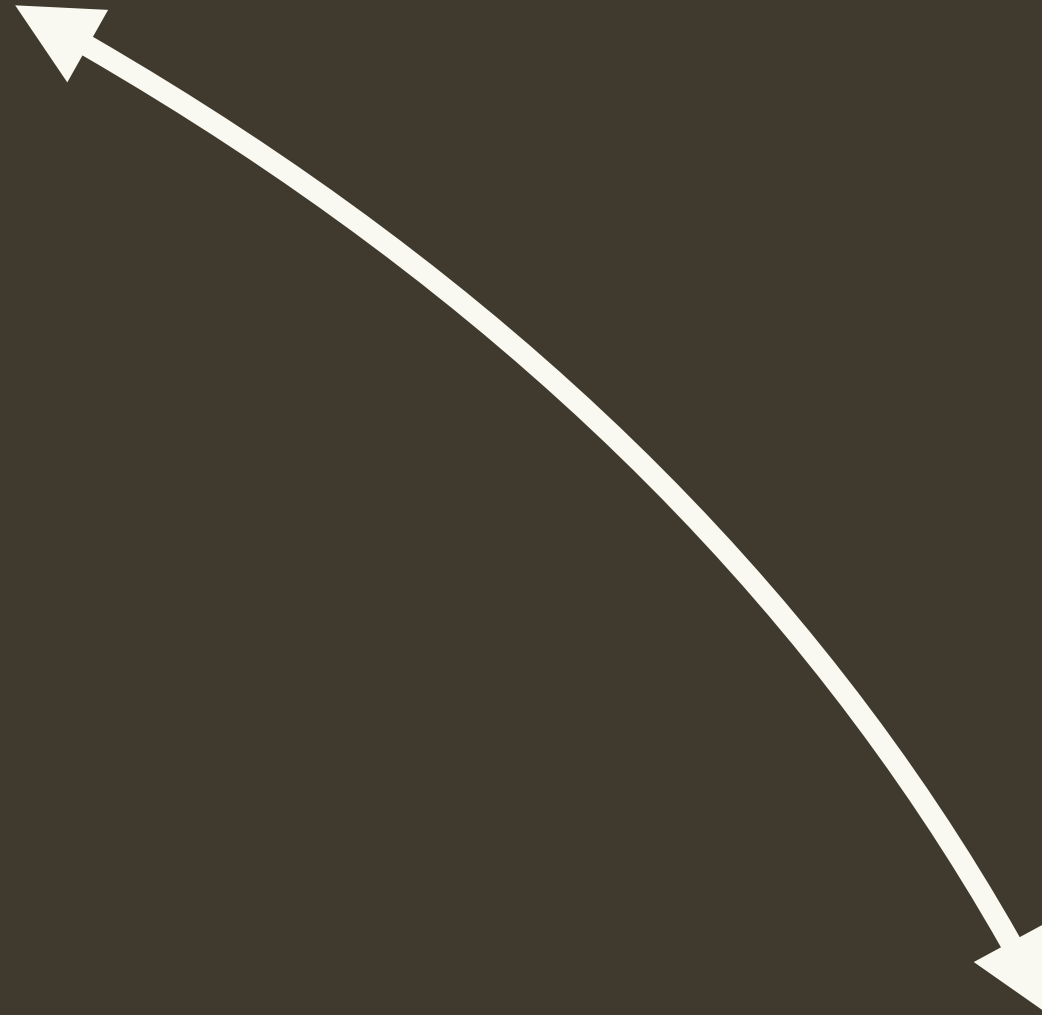
Helpful  Strict

Verbose  Clever

Specialised  General

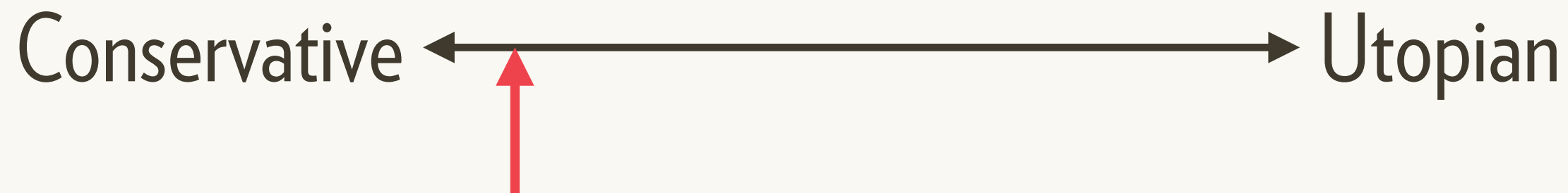
Isolated  Contextual

Helpful



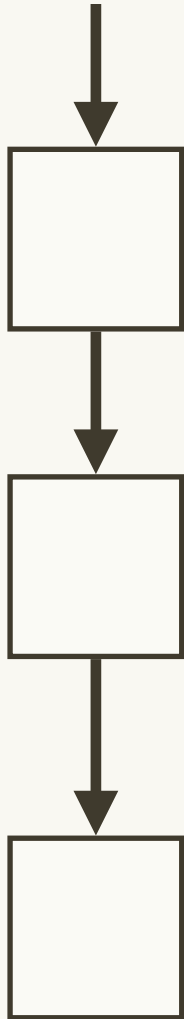
Strict

Two important tensions for understanding base R



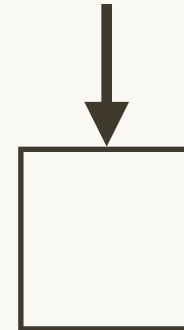
For robust code, fail early

Bad input



Uninformative error

Bad input



Useful error

Best

Right answer

Useful error

Not useful error

Worst

Wrong answer


```
both_na <- function(x, y) {  
  stopifnot(length(x) == length(y))  
  
  sum(is.na(x) & is.na(y))  
}
```

```
both_na <- function(x, y) {  
  if (length(x) != length(y))  
    stop("`x` and `y` must be the same length.")  
  stop("`x` and `y` are different lengths.")  
  
  sum(is.na(x) & is.na(y))  
}
```

```
both_na(c(TRUE, FALSE), c(T, F, F, T))
```

Some specialised packages exist to make this easier

```
gapminder %>%
```

```
  verify(nrow(.) == 1704) %>%
```

```
  verify(ncol(.) == 6) %>%
```

```
  verify(is.factor(.$continent)) %>%
```

```
  verify(length(levels(.$continent)) == 5)
```

assertr

assertthat

assertive

ensurer

testthat

valdecece

Data frames are helpful

```
df <- data.frame(xy = c("x", "y"))  
# Character vectors work hard to work with for a long  
# time, so R helpfully converts to a factor for you:  
class(df$xy)  
  
# If you're only selecting a single column, R tries to  
# be helpful and give you that column, rather than  
# giving you a single column data frame  
class(df[, "xy"])  
  
# If you have long variable names, R is "helpful" and  
# lets you select them with a unique prefix  
df$x
```

tibbles are strict

```
library(tibble)
```

```
df2 <- tibble(xy = c("x", "y"))
```

```
# Never coerces
```

```
class(df2$xy)
```

```
# Always returns another tibble
```

```
class(df2[, "xy"])
```

```
# Complains if variable doesn't exist
```

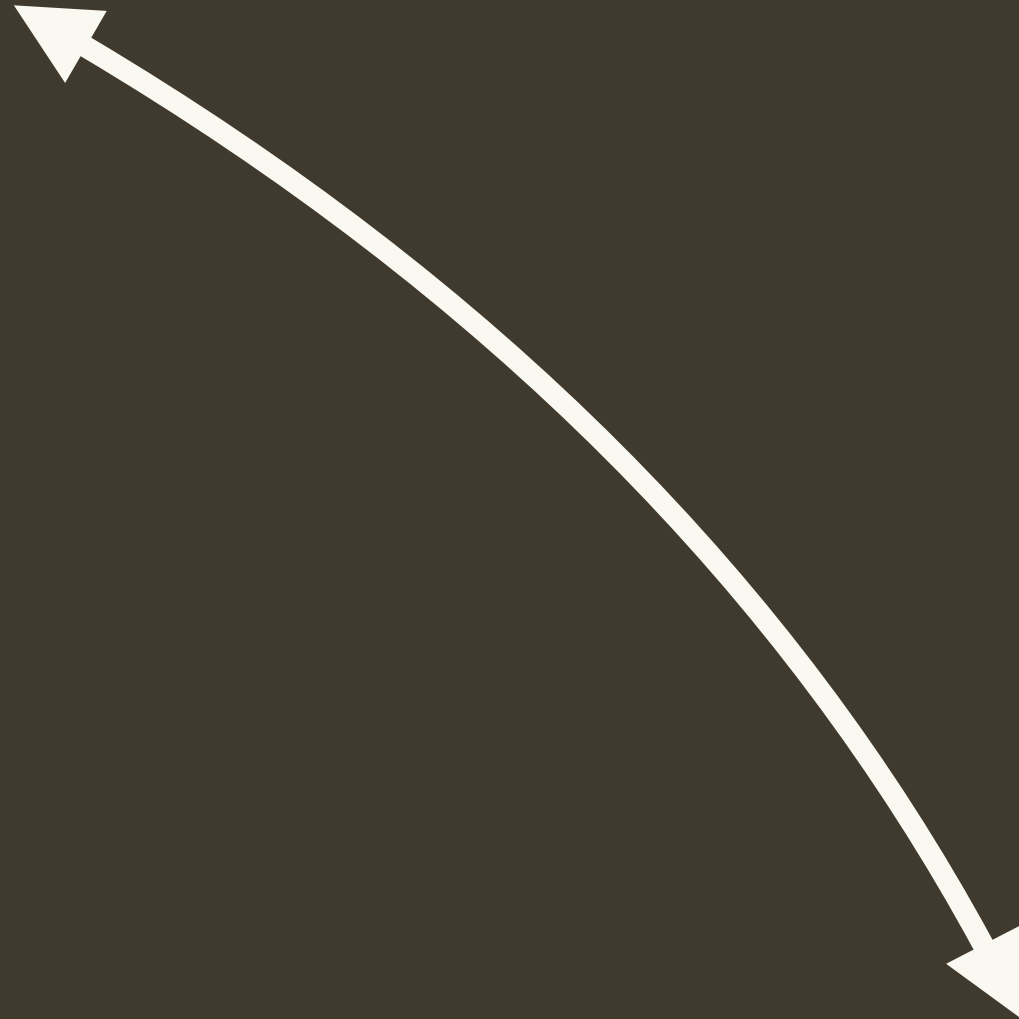
```
df2$x
```

```
df2$z
```

```
tribble(  
  ~x, ~y, ~z,  
    10, 1, "a",  
    5, 2, 1  
)  
as_tibble()
```

```
data.frame(  
  x = c(10, 5),  
  y = c(1, 2),  
  z = c(3, 4)  
)
```

Verbose



Clever

```
# 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 <- c(F, F, T, F, F, F, F, T, F, T)  
# want to get:  
y <- c(1, 1, 2, 2, 2, 2, 2, 3, 3, 4)
```

```
# Brainstorm for 1 minute
```

Uses very simple ideas, but many places to make mistakes

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



```
# Too clever?
```

```
cumsum(x) + !x[1]
```

```
# Little less clever
```

```
cumsum(x) + if (x[1]) 0 else 1
```

```
# Reasonably obvious & has place for comment
```

```
grp <- cumsum(x)
```

```
if (!x[1]) {
```

```
  # first group should start at 1
```

```
  grp <- grp + 1
```

```
}
```

Use `explicit return()` only for early exit

```
f <- function(x, y) {  
  if (x == 0) {  
    return(NA_integer_)  
  }
```

```
  # ...
```

```
  # ...
```

```
  # ...
```

```
  y / x
```

```
}
```

A useful alternative to if with long case

```
f <- function(x, y) {  
  if (x == 0) {  
    NA  
  } else { # x != 0  
    # Many lines  
    # of complicated  
    # code go  
    # ...  
    # ...  
    # ...  
    # ...  
    # ...  
    # ...  
  }  
}
```

```
# . . .  
# . . .  
# . . .  
# . . .  
# . . .  
# . . .  
# . . .  
# here  
# even more than this  
y / x  
} # end x != 0  
}
```

Other common complications

```
x[-which(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")
```

Isolated



Contextual

Principle: less context needed = clearer code

```
baz <- foo(bar, qux)
```

vs

```
df2 <- arrange(df, qux)
```

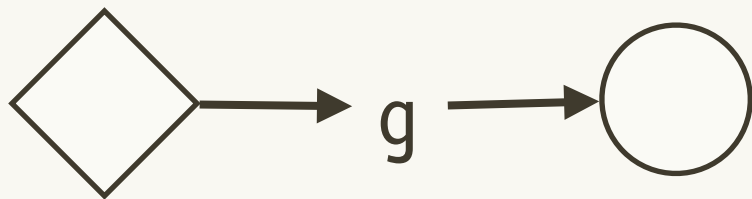
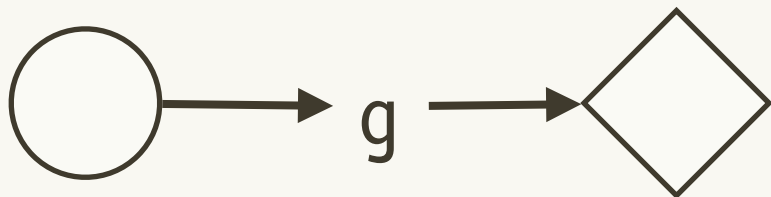
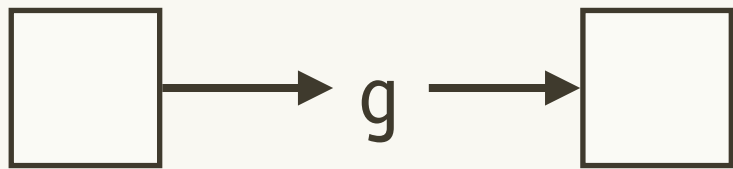
Your turn: what will these calls do?

```
df[, vars]
```

```
filter(df, x == y)
```

```
data.frame(x = "a")
```


Three classes of surprises



Unstable types

```
filter(df, x == y)
```

Non-standard evaluation

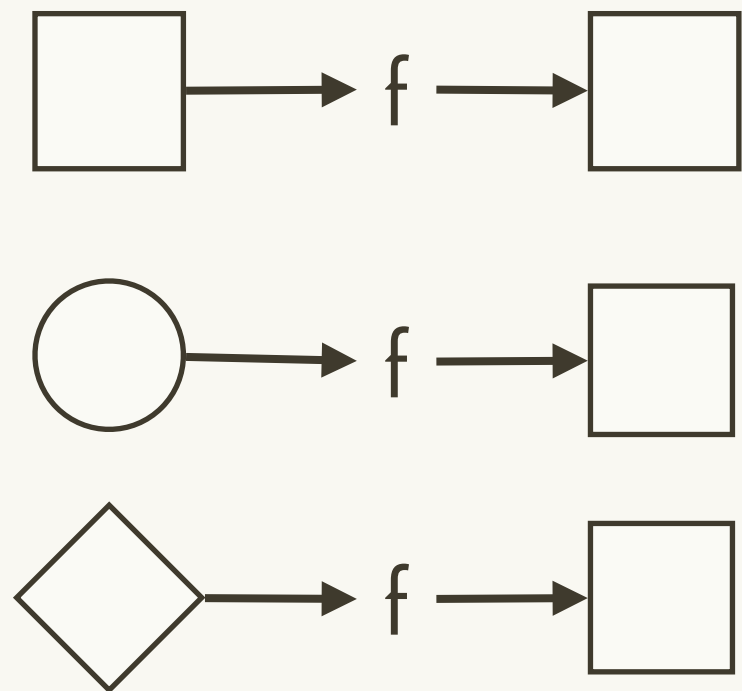
```
getOption("stringsAsFactors")
```

```
data.frame()
```

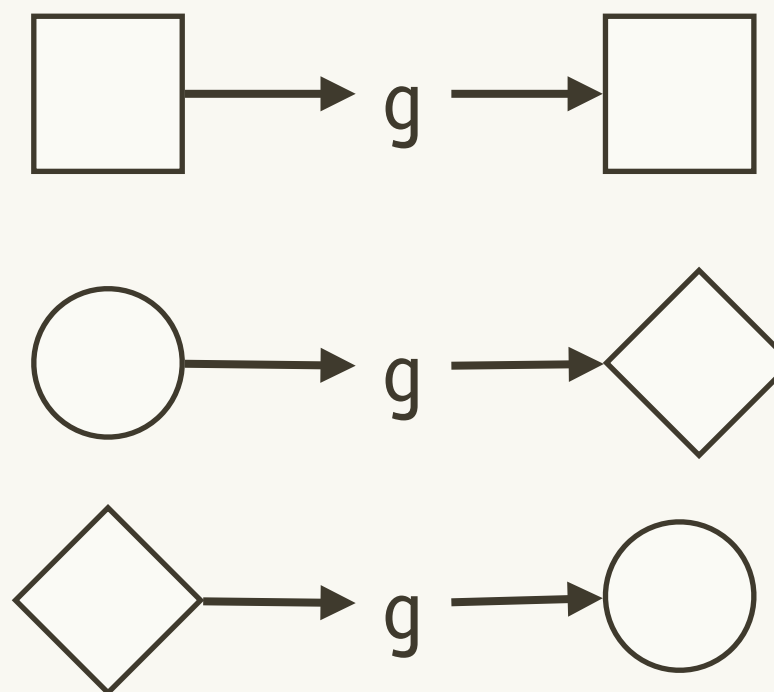
Hidden arguments

Unstable types

Type-stable functions



Regardless of the input, a **type-stable** function gives the same type of output



A **type-unstable** function is like a box of chocolates...

This function uses two type-unstable functions

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

We'll come back to this code later

Non-standard evaluation

How can this function fail?

```
big_x <- function(df, threshold) {  
  dplyr::filter(df, x > threshold)  
}
```

```
filter(df, x > threshold)  
df[df$x > df$threshold, , drop = FALSE]
```

Two modes of **silent failure**

```
x <- 1
```

```
big_x(mtcars, 10)
```

```
df <- dplyr::data_frame(x = 1:10, threshold = 100)
```

```
big_x(df, 5)
```

Currently have to do

```
big_x <- function(df, threshold) {  
  if (!"x" %in% names(df))  
    stop("`df` must contain variable called `x`.", call. = FALSE)  
  
  if ("threshold" %in% names(df))  
    stop("`df` must not contain variable called `threshold`.",  
        call. = FALSE)  
  
  dplyr::filter(df, x > threshold)  
}
```


In future version:

```
big_x <- function(df, threshold) {  
  dplyr::filter(df, .data$x > .env$threshold)  
}
```

See <http://rpubs.com/hadley/lazyeval> for more details

Alternative in base R

```
big_x <- function(df, threshold) {  
  rows <- df[["x"]] > threshold  
  df[rows & !is.na(rows), , drop = FALSE]  
}  
  
# But have to remember all the details of  
# exactly what filter() does; and remember  
# that $ is helpful
```

Hidden parameters

Biggest offenders

stringsAsFactors

System language

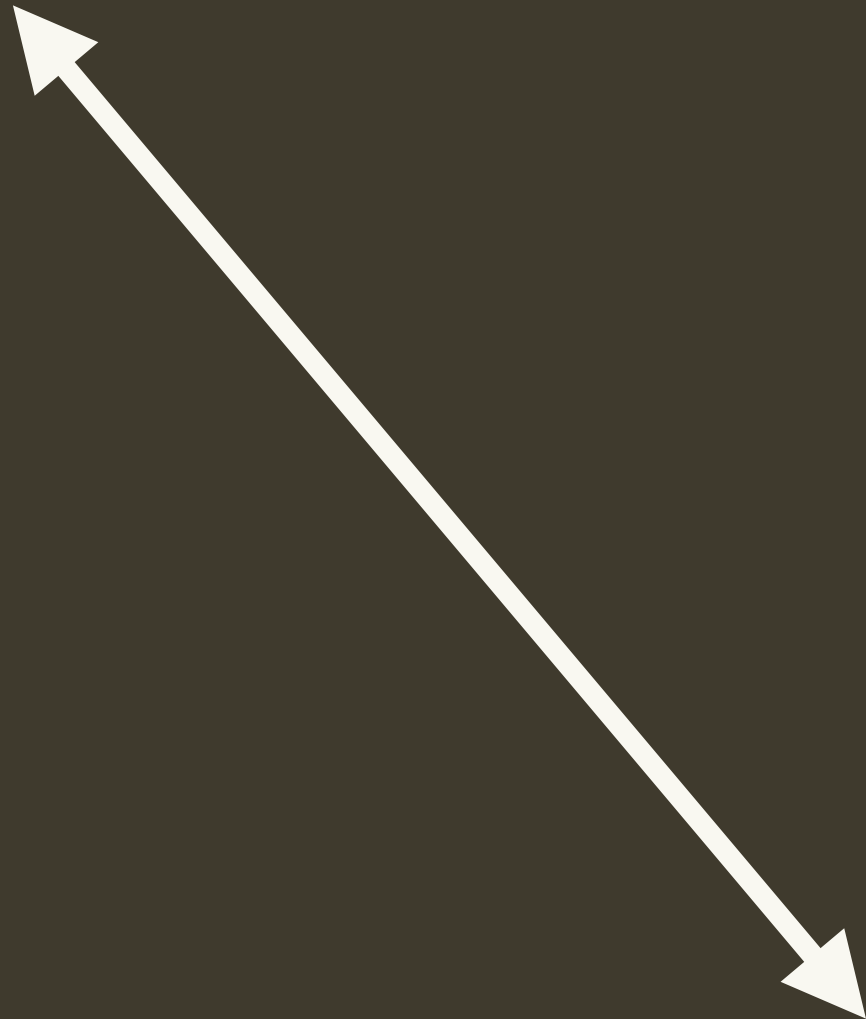
na.action

Time zone

Default text
encoding

Only use `options()` to control side-effects of a function. The value of an option should **never** affect the return value of a function

Specialised



General

What's the goal of this function?

```
impute_na1 <- function(x) {  
  for (i in 2:length(x)) {  
    if (x[i] == "NA") {  
      x[i] <- (x[i - 1] + x[i + 1]) / 2  
    }  
  }  
  x  
}
```

What's wrong with this function?

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

When will this function fail?

```
impute_na2 <- function(x) {  
  for (i in 2:(length(x) - 1)) {  
    if (is.na(x[i])) {  
      x[i] <- (x[i - 1] + x[i + 1]) / 2  
    }  
  }  
  x  
}  
impute_na2(c(1, 4, 5, NA, 10, 13, 10))
```


Your turn

Modify `input_na2()` so that it works for all inputs.
Think about the ideas on the next slide

What should the answers be?

```
impute_na2(numeric())
```

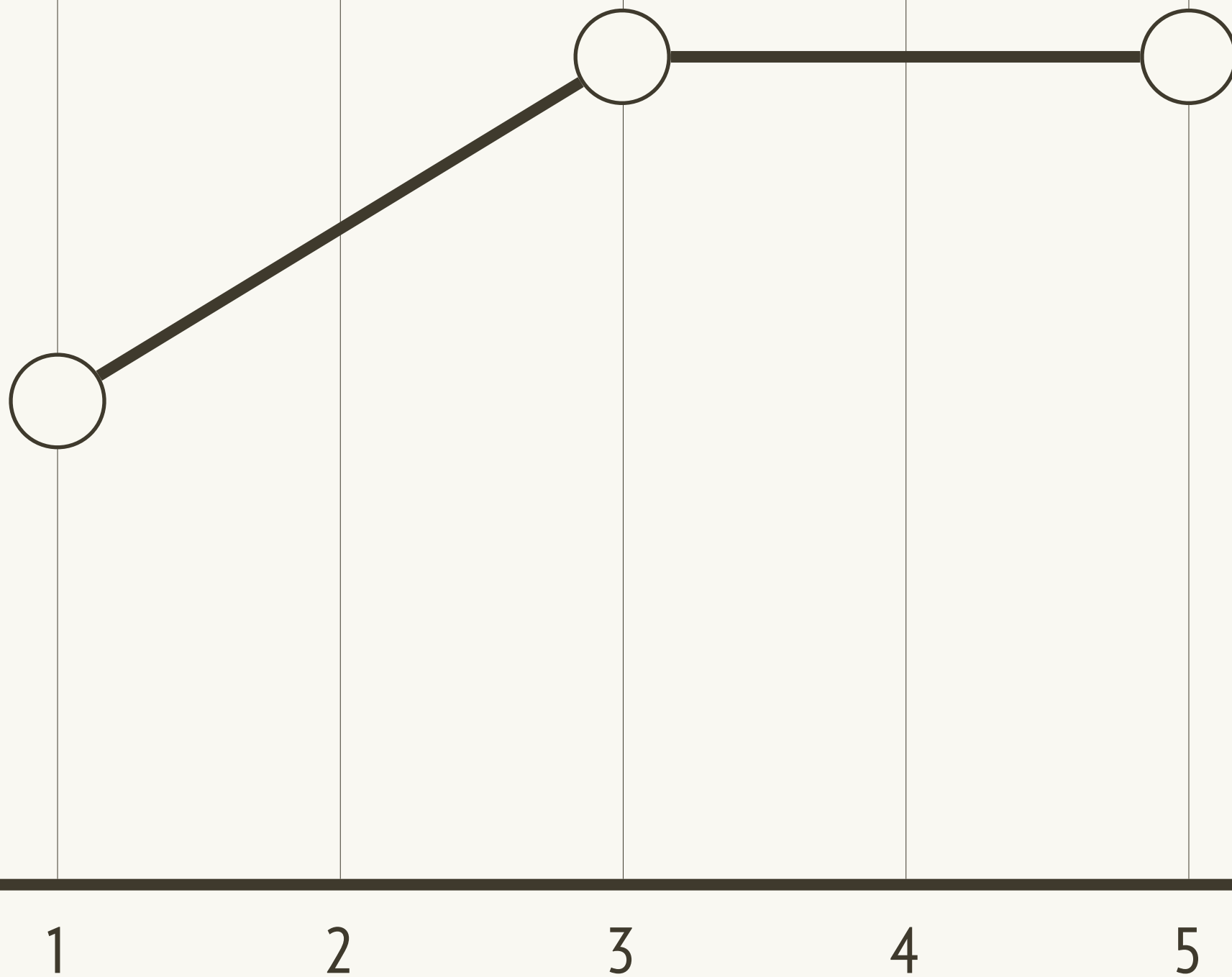
```
impute_na2(NA_real_)
```

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

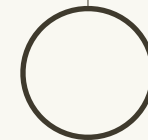
```
impute_na2(c(1, NA, NA, 2))
```

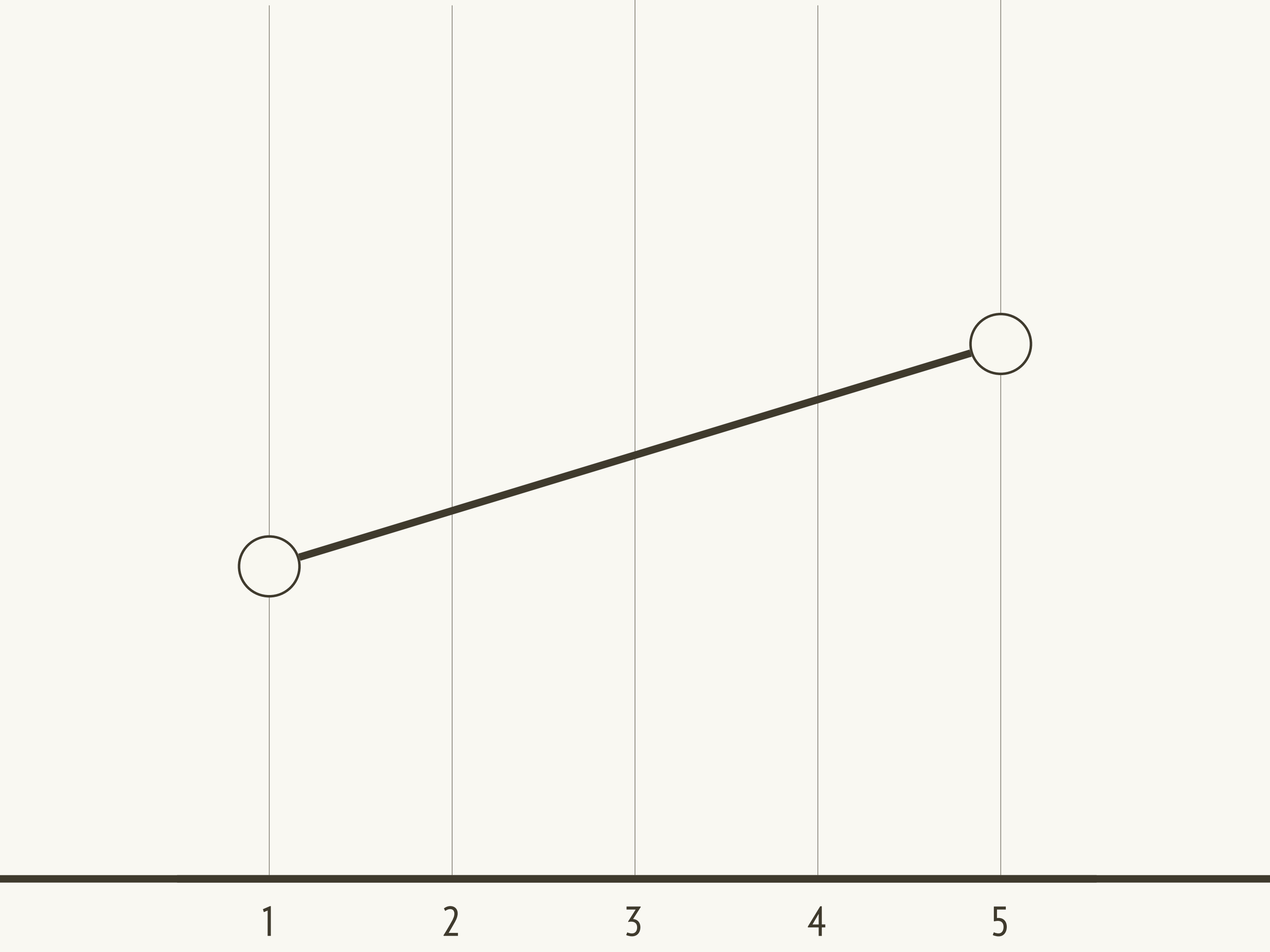
```
impute_na2(c(NA, 2))
```

Sometimes a completely different framing is useful



What should we do if there are multiple missing values?





A more general implementation

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

Vocabulary is important!

Documented Tested

Existing function >> New function

Standard name

More general

Name



Knowledge!

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

— *Shakespeare*

“A **function** by any other name
would **not** smell as sweet.”

— *Hadley Wickham*

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/>