

# Functions: Parts 1 & 2

---

Daniel Anderson

Week 6

# Agenda

---

- Review take-home midterm
- Everything is a function
- Components of a function
- Function workflows

# Learning objectives

---

- Understand and be able to fluently refer to the three fundamental components of a function
- Understand the workflows that often lead to writing functions, and how you iterate from interactive work to writing a function
- Be able to write a few basic functions

Review Take—  
home

---

# Functions

---

# Functions

---

Anything that carries out an operation in R is a function. For example

```
3 + 5
```

```
## [1] 8
```

The `+` is a function (what's referred to as an *infix* function).

Any ideas on how we could re-write the above to make it look more "function"-y?

```
`+`(3, 5)
```

```
## [1] 8
```

# What about this?

---

```
3 + 5 + 7
```

```
## [1] 15
```

```
`+`(7, `+`(3, 5))
```

```
## [1] 15
```

or

```
library(magrittr)  
`+`(3, 5) %>%  
  `+`(7)
```

```
## [1] 15
```

# What's going on here?

---

- The `+` operator is a function that takes two arguments (both numeric), which it sums.
- The following are also the same (minus what's being assigned)

```
a <- 7  
a
```

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

```
`<-`(a, 5)  
a
```

```
## [1] 5
```

Everything is a function!



# Being devious

---

- Want to introduce a devious bug? Redefine `+`

```
`+` <- function(x, y) {  
  if(runif(1) < 0.01) {  
    sum(x, y) * -1  
  } else {  
    sum(x, y)  
  }  
}  
table(map2_dbl(1:500, 1:500, `+`) > 0)
```

```
##  
## FALSE TRUE  
##      6   494
```

```
rm(`+`, envir = globalenv())  
table(map2_dbl(1:500, 1:500, `+`) > 0)
```

```
##  
## FALSE TRUE  
##      4   496
```

# Tricky...

---

Functions are also (usually) objects!

```
a <- lm  
a(hp ~ drat + wt, data = mtcars)
```

```
##  
## Call:  
## a(formula = hp ~ drat + wt, data = mtcars)  
##  
## Coefficients:  
## (Intercept)          drat           wt  
##      -27.782         5.354        48.244
```

# What does this all mean?

---

- Anything that carries out ANY operation in R is a function
- Functions are generally, but not always, stored in an object (otherwise known as binding the function to a name)

# Anonymous functions

---

- The function for computing the mean is bound the name `mean`
- When running things through loops, you may often want to apply a function without binding it to a name

## Example

```
vapply(mtcars, function(x) length(unique(x)), FUN.VALUE = double
```

```
##   mpg   cyl  disp    hp  drat    wt  qsec    vs  am  gear  carb
##   25     3   27     22   22     29   30     2    2    3     6
```

# Another possibility

---

- If you have a bunch of functions, you might consider storing them all in a list.
- You can then access the functions in the same way you would subset any list

```
funcs <- list(  
  quarter = function(x) x / 4,  
  half = function(x) x / 2,  
  double = function(x) x * 2,  
  quadruple = function(x) x * 4  
)
```

*This is kind of weird...*

```
funcs$quarter(100)
```

```
## [1] 25
```

```
funcs[["half"]](100)
```

```
## [1] 50
```

```
funcs[[4]](100)
```

```
## [1] 400
```

# What does this imply?

---

- If we can store functions in a vector (list), then we can loop through the vector just like any other!

```
smry <- list(  
  n = length,  
  n_miss = function(x) sum(is.na(x)),  
  n_valid = function(x) sum(!is.na(x)),  
  mean = mean,  
  sd = sd  
)
```

```
map_dbl(smry, ~.x(mtcars$mpg))
```

```
##           n      n_miss  n_valid      mean      sd
## 32.000000  0.000000 32.000000 20.090625  6.026948
```

Careful though

This doesn't work

```
map_dbl(smry, mtcars$mpg)
```

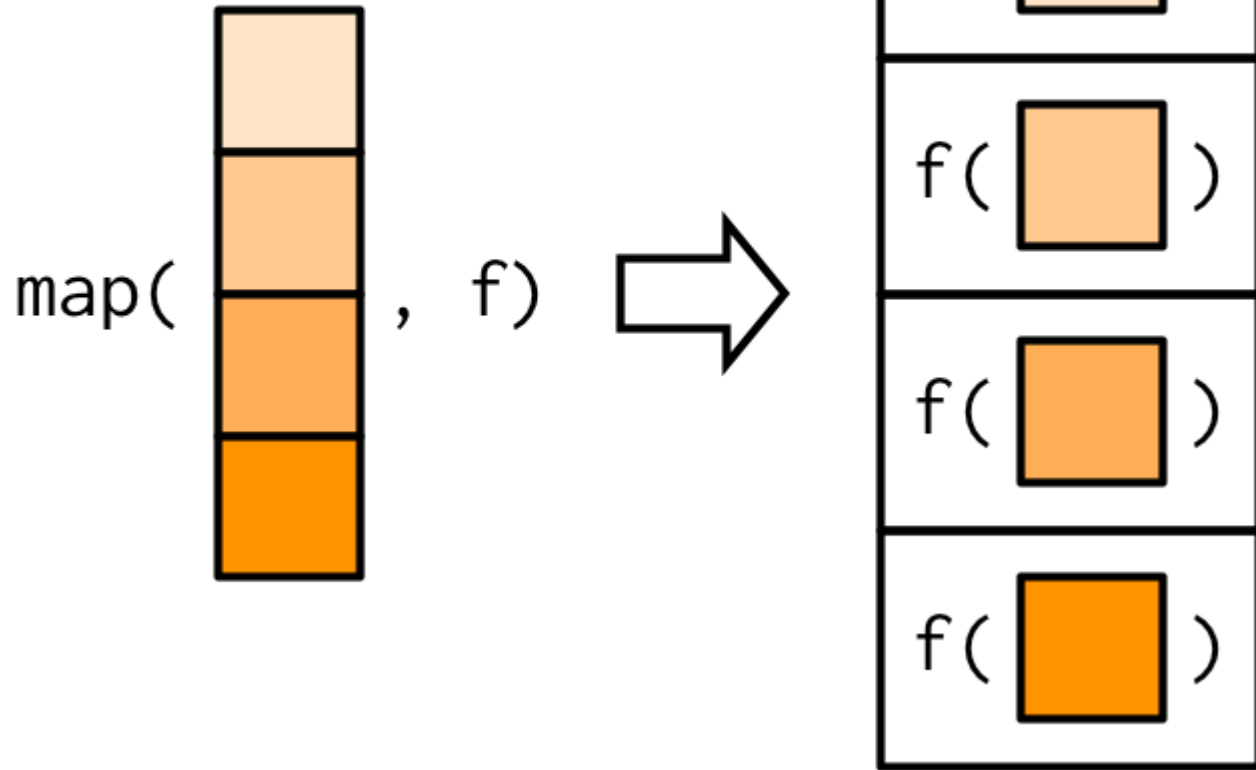
```
## Error: Can't pluck from a builtin
```

Why?



Remember what {purrr}  
does

---



# With `map_df`

---

```
map_df(smry, ~.x(mtcars$mpg))
```

```
## # A tibble: 1 × 5
##       n n_miss n_valid   mean   sd
##   <int> <int>   <int>   <dbl> <dbl>
## 1    32      0     32 20.09062 6.026948
```

```
map_df(smry, ~.x(mtcars$cyl))
```

```
## # A tibble: 1 × 5
##       n n_miss n_valid   mean   sd
##   <int> <int>   <int>   <dbl> <dbl>
## 1    32      0     32  6.1875 1.785922
```

What if we wanted this for all columns?

# Challenge

- Can you extend the previous looping to supply the summary for every column? Hint: You'll need to make a nested loop (loop each function through each column)

```
map_df(mtcars, function(col) map_df(smry, ~.x(col)),  
       .id = "column")
```

```
## # A tibble: 11 × 6  
##   column      n n_miss n_valid      mean      sd  
##   <chr>  <int>  <int>   <int>    <dbl>    <dbl>  
## 1 mpg      32      0      32  20.09062  6.026948  
## 2 cyl      32      0      32   6.1875   1.785922  
## 3 disp     32      0      32 230.7219 123.9387  
## 4 hp       32      0      32 146.6875  68.56287  
## 5 drat     32      0      32   3.596562 0.5346787  
## 6 wt       32      0      32   3.21725  0.9784574  
## 7 qsec     32      0      32 17.84875  1.786943  
## 8 vs       32      0      32   0.4375   0.5040161  
## 9 am       32      0      32   0.40625  0.4989909  
## 10 gear    32      0      32   3.6875   0.7378041  
## 11 carb    32      0      32   2.8125   1.615200
```

05:00

# Easier \* more readable

---

- Avoid nested loops
- Instead, turn (at least) one of the loops into a function

```
summarize_col <- function(column) {  
  map_df(smry, ~.x(column))  
}
```

Now we can just loop this function through each column

```
map_df(mtcars, summarize_col, .id = "column")
```

```
## # A tibble: 11 × 6
##   column      n n_miss n_valid      mean      sd
##   <chr>  <int>  <int>   <int>    <dbl>    <dbl>
## 1 mpg      32      0      32  20.09062  6.026948
## 2 cyl      32      0      32   6.1875   1.785922
## 3 disp     32      0      32 230.7219 123.9387
## 4 hp       32      0      32 146.6875  68.56287
## 5 drat     32      0      32   3.596562 0.5346787
## 6 wt       32      0      32   3.21725   0.9784574
## 7 qsec     32      0      32  17.84875  1.786943
## 8 vs       32      0      32   0.4375    0.5040161
## 9 am       32      0      32   0.40625   0.4989909
## 10 gear    32      0      32   3.6875    0.7378041
## 11 carb    32      0      32   2.8125    1.615200
```

# Wrap the whole thing in a function

---

```
summarize_df <- function(df) {  
  map_df(df, summarize_col, .id = "column")  
}
```

```
summarize_df(airquality)
```

```
## # A tibble: 6 × 6  
##   column      n n_miss n_valid    mean      sd  
##   <chr>   <int>  <int>   <int>   <dbl>   <dbl>  
## 1 Ozone    153    37    116 NA      NA  
## 2 Solar.R  153     7    146 NA      NA  
## 3 Wind     153     0    153  9.957516  3.523001  
## 4 Temp     153     0    153  77.88235  9.465270  
## 5 Month    153     0    153  6.993464  1.416522  
## 6 Day      153     0    153 15.80392  8.864520
```

Notice the missing data. Why? What should we do?

# Deep breaths

---

So far, this has just been a high-level overview.

I wanted to show *why* you might want to write functions first.

Let's dig into the particulars

# Function

# components

---




# Three components

---

- `body()`
- `formals()`
- `environment()` (we won't focus so much here for now)

```
poly <- function(x, power) {  
  z <- x^power  
  return(z)  
}
```



The diagram illustrates the components of the function `poly`. A green curly brace on the right side of the function body (the lines `z <- x^power` and `return(z)`) is labeled "Body". A green arrow points from the label "Formals" to the arguments `x` and `power` in the function signature.

# Formals

---

- The arguments supplied to the function
- What's one way to identify the formals for a function – say, `lm`?

`?`: Help documentation!

Alternative – use a function!

```
formals(lm)
```

```
## $formula
##
##
## $data
##
##
## $subset
##
##
## $weights
```

# How do you see the body?

---

- In RStudio: Super (command on mac, cntrl on windows) + click!

[demo]

- Alternative – just print to screen

# Or use **body**

---

```
body(lm)
```

```
## {
##   ret.x <- x
##   ret.y <- y
##   cl <- match.call()
##   mf <- match.call(expand.dots = FALSE)
##   m <- match(c("formula", "data", "subset", "weights", "na.action",
##               "offset"), names(mf), 0L)
##   mf <- mf[c(1L, m)]
##   mf$drop.unused.levels <- TRUE
##   mf[[1L]] <- quote(stats::model.frame)
##   mf <- eval(mf, parent.frame())
##   if (method == "model.frame")
##     return(mf)
##   else if (method != "qr")
##     warning(gettextf("method = '%s' is not supported. Using 'qr'",
##                     method), domain = NA)
##   mt <- attr(mf, "terms")
##   y <- model.response(mf, "numeric")
##   w <- as.vector(model.weights(mf))
##   if (!is.null(w) && !is.numeric(w))
##     stop("'weights' must be a numeric vector")
##   offset <- model.offset(mf)
##   mlm <- is.matrix(y)
##   ny <- if (mlm)
```

# Environment

---

- As I mentioned, we won't focus on this too much, but if you get deep into programming it's pretty important

```
double <- function(x) x*2  
environment(double)
```

```
## <environment: 0x7ff1dd310cf0>
```

```
environment(lm)
```

```
## <environment: namespace:stats>
```

# Why this matters

---

What will the following return?

```
x <- 10
f1 <- function() {
  x <- 20
  x
}

f1()
```

```
## [1] 20
```

# What will this return?

---

```
x <- 10
y <- 20
f2 <- function() {
  x <- 1
  y <- 2
  sum(x, y)
}
f2()
```

```
## [1] 3
```

# Last one

---

What do each of the following return?

```
x <- 2
f3 <- function() {
  y <- 1
  sum(x, y)
}
```

```
f3()
```

```
y
```

```
## [1] 3
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```



# Environment summary

---

- The previous examples are part of *lexical scoping*.
- Generally, you won't have to worry too much about it
- If you end up with unexpected results, this could be part of why

# Scoping

---

- Part of what's interesting about these scoping rules is that your functions can, and very often do, depend upon things in your global workspace, or your specific environment.
- If this is the case, the function will be a "one-off", and unlikely to be useful in any other script
- Note that our `summarize_df()` function depended on the global `smry` list object.

A few real  
examples

---

# Example 1

---

Return the item "scores" for a differential item functioning analysis

```
extract_grades <- function(dif_mod, items) {  
  item_names <- names(items)  
  delta <- -2.35 * log(dif_mod$alphaMH)  
  grades <- symnum(  
    abs(delta),  
    c(0, 1, 1.5, Inf),  
    symbols = c("A", "B", "C")  
  )  
  
  tibble(item = item_names, delta, grades) %>%  
    mutate(grades = as.character(grades))  
}
```

# Example 2

---

## Reading in data

```
read_sub_files <- function(filepath) {  
  read_csv(filepath) %>%  
    mutate(  
      content_area = str_extract(  
        file, "[Ee][Ll][Aa]|[Rr]dg|[Ww]ri|[Mm]ath|[Ss]ci"  
      ),  
      grade = gsub("."+g("\\d\\d*").+", "\\1", filepath),  
      grade = as.numeric(grade)  
    ) %>%  
    select(content_area, grade, everything()) %>%  
    clean_names()  
}  
  
ifiles <- map_df(filepaths, read_sub_files)
```

# Simple example

---

Please follow along

Pull out specific coefficients

```
mods <- mtcars %>%  
  group_by(cyl) %>%  
  nest() %>%  
  mutate(  
    model = map(  
      data, ~lm(mpg ~ disp + hp + drat, data = .x)  
    )  
  )  
mods
```

```
## # A tibble: 3 × 3  
## # Groups:   cyl [3]  
##   cyl data          model  
##   <dbl> <list>          <list>  
## 1     6 <tibble [7 × 10]> <lm>  
## 2     4 <tibble [11 × 10]> <lm>  
## 3     8 <tibble [14 × 10]> <lm>
```

# Pull a specific coef

---

Find the solution for one model

```
# pull just the first model
m <- mods$model[[1]]

# extract all coefs
coef(m)
```

```
## (Intercept)          disp           hp          drat
## 6.284507434 0.026354099 0.006229086 2.193576546
```

```
# extract specific coefs
coef(m)["disp"]
```

```
##          disp
## 0.0263541
```

```
coef(m)["(Intercept)"]
```

```
## (Intercept)
## 6.284507
```

# Challenge

---

Can you write a function that returns a *specific* coefficient from a model?

Try using the code on the previous slide as you guide

04:00



# Generalize it

---

```
pull_coef <- function(model, coef_name) {  
  coef(model)[coef_name]  
}
```

```
mods %>%  
  mutate(intercept = map_dbl(model, pull_coef, "(Intercept)"),  
         disp      = map_dbl(model, pull_coef, "disp"),  
         hp        = map_dbl(model, pull_coef, "hp"),  
         drat      = map_dbl(model, pull_coef, "drat"))
```

```
## # A tibble: 3 × 7  
## # Groups:   cyl [3]  
##   cyl data          model intercept      disp      hp  
##   <dbl> <list>         <list>      <dbl>      <dbl>      <dbl>  
## 1     6 <tibble [7 × 10]> <lm>      6.284507  0.02635410  0.006229086  2.  
## 2     4 <tibble [11 × 10]> <lm>     46.08662 -0.1225361 -0.04937771 -0.  
## 3     8 <tibble [14 × 10]> <lm>     19.00162 -0.01671461 -0.02140236  2.
```

# Make it more flexible

---

- Since the intercept is a little difficult to pull out, we could have it return that by default.

```
pull_coef <- function(model, coef_name = "(Intercept)") {  
  coef(model)[coef_name]  
}  
mods %>%  
  mutate(intercept = map_dbl(model, pull_coef))
```

```
## # A tibble: 3 × 4  
## # Groups:   cyl [3]  
##   cyl data          model intercept  
##   <dbl> <list>         <list>      <dbl>  
## 1     6 <tibble [7 × 10]> <lm>        6.284507  
## 2     4 <tibble [11 × 10]> <lm>       46.08662  
## 3     8 <tibble [14 × 10]> <lm>       19.00162
```

# Return all coefficients

---

First, figure it out for a single case

```
coef(m)
```

```
## (Intercept)      disp      hp      drat  
## 6.284507434 0.026354099 0.006229086 2.193576546
```

```
names(coef(m))
```

```
## [1] "(Intercept)" "disp"      "hp"      "drat"
```

Put them in a data frame

```
tibble(  
  term = names(coef(m)),  
  coefficient = coef(m)  
)
```

```
## # A tibble: 4 × 2  
##   term      coefficient  
##   <chr>          <dbl>  
## 1 (Intercept)  6.284507  
## 2 disp         0.02635410  
## 3 hp           0.006229086  
## 4 drat         2.193577
```

# Challenge

---

Can you generalize this to a function?

```
pull_coef <- function(model) {  
  coefs <- coef(model)  
  tibble(  
    term = names(coefs),  
    coefficient = coefs  
  )  
}
```

# Test it out

---

```
mods %>%  
  mutate(coefs = map(model, pull_coef))
```

```
## # A tibble: 3 × 4  
## # Groups:   cyl [3]  
##   cyl data          model coefs  
##   <dbl> <list>         <list> <list>  
## 1     6 <tibble [7 × 10]> <lm>   <tibble [4 × 2]>  
## 2     4 <tibble [11 × 10]> <lm>   <tibble [4 × 2]>  
## 3     8 <tibble [14 × 10]> <lm>   <tibble [4 × 2]>
```

# unnest

```
mods %>%  
  mutate(coefs = map(model, pull_coef)) %>%  
  unnest(coefs)
```

```
## # A tibble: 12 × 5  
## # Groups:   cyl [3]  
##     cyl data          model term          coefficient  
##   <dbl> <list>         <list> <chr>          <dbl>  
## 1     6 <tibble [7 × 10]> <lm>   (Intercept)    6.284507  
## 2     6 <tibble [7 × 10]> <lm>   disp           0.02635410  
## 3     6 <tibble [7 × 10]> <lm>   hp             0.006229086  
## 4     6 <tibble [7 × 10]> <lm>   drat           2.193577  
## 5     4 <tibble [11 × 10]> <lm>   (Intercept)   46.08662  
## 6     4 <tibble [11 × 10]> <lm>   disp          -0.1225361  
## 7     4 <tibble [11 × 10]> <lm>   hp            -0.04937771  
## 8     4 <tibble [11 × 10]> <lm>   drat          -0.6041857  
## 9     8 <tibble [14 × 10]> <lm>   (Intercept)   19.00162  
## 10    8 <tibble [14 × 10]> <lm>   disp          -0.01671461  
## 11    8 <tibble [14 × 10]> <lm>   hp            -0.02140236  
## 12    8 <tibble [14 × 10]> <lm>   drat           2.006011
```

# Slightly nicer

---

```
mods %>%  
  mutate(coefs = map(model, pull_coef)) %>%  
  select(cyl, coefs) %>%  
  unnest(coefs)
```

```
## # A tibble: 12 × 3  
## # Groups:   cyl [3]  
##   cyl term      coefficient  
##   <dbl> <chr>      <dbl>  
## 1     6 (Intercept)  6.284507  
## 2     6 disp      0.02635410  
## 3     6 hp       0.006229086  
## 4     6 drat     2.193577  
## 5     4 (Intercept) 46.08662  
## 6     4 disp     -0.1225361  
## 7     4 hp      -0.04937771  
## 8     4 drat     -0.6041857  
## 9     8 (Intercept) 19.00162  
## 10    8 disp     -0.01671461  
## 11    8 hp      -0.02140236  
## 12    8 drat     2.006011
```



# Create nice table

---

```
mods %>%  
  mutate(coefs = map(model, pull_coef)) %>%  
  select(cyl, coefs) %>%  
  unnest(coefs) %>%  
  pivot_wider(  
    names_from = "term",  
    values_from = "coefficient"  
  ) %>%  
  arrange(cyl)
```

```
## # A tibble: 3 × 5  
## # Groups:   cyl [3]  
##   cyl `(Intercept)`      disp      hp      drat  
##   <dbl>      <dbl>      <dbl>      <dbl>      <dbl>  
## 1     4      46.08662 -0.1225361 -0.04937771 -0.6041857  
## 2     6       6.284507  0.02635410  0.006229086  2.193577  
## 3     8      19.00162 -0.01671461 -0.02140236  2.006011
```

When to write  
a function?

---

# Example

---

```
set.seed(42)
df <- tibble::tibble(
  a = rnorm(10, 100, 150),
  b = rnorm(10, 100, 150),
  c = rnorm(10, 100, 150),
  d = rnorm(10, 100, 150)
)
```

df

```
## # A tibble: 10 × 4
##           a           b           c           d
##       <dbl>       <dbl>       <dbl>       <dbl>
## 1  305.6438    295.7304    54.00421    168.3175
## 2   15.29527   442.9968  -167.1963    205.7256
## 3  154.4693  -108.3291    74.21240    255.2655
## 4  194.9294    58.18168   282.2012     8.661044
## 5  160.6402    80.00180   384.2790    175.7433
## 6   84.08132   195.3926    35.42963  -157.5513
## 7  326.7283    57.36206    61.40959   -17.66885
## 8   85.80114 -298.4683  -164.4745   -27.63614
## 9  402.7636  -266.0700   169.0146  -262.1311
## 10  90.59289   298.0170     4.000769   105.4184
```

# Rescale each column to 0/1

---

We do this by subtracting the minimum value from each observation, then dividing that by the difference between the min/max values. For example

```
tibble(  
  v1 = c(3, 4, 5),  
  numerator = v1 - 3,  
  denominator = 5 - 3,  
  scaled = numerator / denominator  
)
```

```
## # A tibble: 3 × 4  
##       v1 numerator denominator scaled  
##   <dbl>     <dbl>         <dbl>  <dbl>  
## 1     3         0           2      0  
## 2     4         1           2    0.5  
## 3     5         2           2     1
```

# One column

---

```
df %>%  
  mutate(  
    a = (a - min(a, na.rm = TRUE)) /  
        (max(a, na.rm = TRUE) - min(a, na.rm = TRUE))  
  )
```

```
## # A tibble: 10 × 4  
##           a           b           c           d  
##       <dbl>       <dbl>       <dbl>       <dbl>  
## 1 0.7493478 295.7304    54.00421   168.3175  
## 2 0          442.9968  -167.1963   205.7256  
## 3 0.3591881 -108.3291   74.21240   255.2655  
## 4 0.4636099  58.18168  282.2012    8.661044  
## 5 0.3751145  80.00180  384.2790   175.7433  
## 6 0.1775269 195.3926   35.42963  -157.5513  
## 7 0.8037639  57.36206   61.40959  -17.66885  
## 8 0.1819655 -298.4683 -164.4745  -27.63614  
## 9 1          -266.0700  169.0146  -262.1311  
## 10 0.1943323 298.0170    4.000769  105.4184
```

# Do it for all columns

---

```
df %>%
  mutate(
    a = (a - min(a, na.rm = TRUE)) /
      (max(a, na.rm = TRUE) - min(a, na.rm = TRUE)),
    b = (b - min(b, na.rm = TRUE)) /
      (max(b, na.rm = TRUE) - min(b, na.rm = TRUE)),
    c = (c - min(c, na.rm = TRUE)) /
      (max(c, na.rm = TRUE) - min(c, na.rm = TRUE)),
    d = (d - min(d, na.rm = TRUE)) /
      (max(d, na.rm = TRUE) - min(d, na.rm = TRUE))
  )
```

```
## # A tibble: 10 × 4
##       a          b          c          d
##   <dbl>    <dbl>    <dbl>    <dbl>
## 1 0.7493478 0.8013846 0.4011068 0.8319510
## 2 0          1          0          0.9042516
## 3 0.3591881 0.2564372 0.4377506 1
## 4 0.4636099 0.4810071 0.8149005 0.5233744
## 5 0.3751145 0.5104355 1          0.8463031
## 6 0.1775269 0.6660608 0.3674252 0.2021270
## 7 0.8037639 0.4799017 0.4145351 0.4724852
## 8 0.1819655 0          0.004935493 0.4532209
## 9 1          0.04369494 0.6096572 0
```

# An alternative

---

- What's an alternative we could use *without* writing a function?

```
map_df(df, ~(.x - min(.x, na.rm = TRUE)) /  
          (max(.x, na.rm = TRUE) - min(.x, na.rm = TRUE)))
```

```
## # A tibble: 10 × 4  
##       a          b          c          d  
##   <dbl>    <dbl>    <dbl>    <dbl>  
## 1 0.7493478 0.8013846 0.4011068 0.8319510  
## 2 0          1          0          0.9042516  
## 3 0.3591881 0.2564372 0.4377506 1  
## 4 0.4636099 0.4810071 0.8149005 0.5233744  
## 5 0.3751145 0.5104355 1          0.8463031  
## 6 0.1775269 0.6660608 0.3674252 0.2021270  
## 7 0.8037639 0.4799017 0.4145351 0.4724852  
## 8 0.1819655 0          0.004935493 0.4532209  
## 9 1          0.04369494 0.6096572 0  
## 10 0.1943323 0.8044685 0.3104346 0.7103825
```

# Another alternative

---

Write a function

- What are the arguments going to be?
- What will the body be?

Arguments

- One formal argument – A numeric vector to rescale



# Body

---

- You try first

```
(x - min(x, na.rm = TRUE)) /  
  (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
```

03:00

# Create the function

---

```
rescale01 <- function(x) {  
  (x - min(x, na.rm = TRUE)) /  
    (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))  
}
```

Test it!

```
rescale01(c(0, 5, 10))
```

```
## [1] 0.0 0.5 1.0
```

```
rescale01(c(seq(0, 100, 10)))
```

```
## [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

# Make it cleaner

---

- There's nothing inherently "wrong" about the prior function, but it is a bit hard to read
- How could we make it easier to read?
  - Remove missing data once (rather than every time)
  - Don't calculate things multiple times

# A little cleaned up

---

```
rescale01b <- function(x) {  
  z <- na.omit(x)  
  min_z <- min(z)  
  max_z <- max(z)  
  
  (z - min_z) / (max_z - min_z)  
}
```

Test it!

```
rescale01b(c(0, 5, 10))
```

```
## [1] 0.0 0.5 1.0
```

```
rescale01b(c(seq(0, 100, 10)))
```

```
## [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Make sure they give the same output

```
identical(rescale01(c(0, 1e5, .01)), rescale01b(c(0, 1e5, 0.01)))
```

```
## [1] TRUE
```

```
rand <- rnorm(1e3)  
identical(rescale01(rand), rescale01b(rand))
```

```
## [1] TRUE
```

# Final solution

---

Could use **modify** here too

```
map_df(df, rescale01b)
```

```
## # A tibble: 10 × 4
##       a          b          c          d
##   <dbl>    <dbl>    <dbl>    <dbl>
## 1 0.7493478 0.8013846 0.4011068 0.8319510
## 2 0          1          0          0.9042516
## 3 0.3591881 0.2564372 0.4377506 1
## 4 0.4636099 0.4810071 0.8149005 0.5233744
## 5 0.3751145 0.5104355 1          0.8463031
## 6 0.1775269 0.6660608 0.3674252 0.2021270
## 7 0.8037639 0.4799017 0.4145351 0.4724852
## 8 0.1819655 0          0.004935493 0.4532209
## 9 1          0.04369494 0.6096572 0
## 10 0.1943323 0.8044685 0.3104346 0.7103825
```

# Getting more complex

---

- What if you want a function to behave differently depending on the input?

## Add conditions

```
function() {  
  if (condition) {  
  
    # code executed when condition is TRUE  
  
  } else {  
    # code executed when condition is FALSE  
  
  }  
}
```

# Lots of conditions?

---

```
function() {  
  if (this) {  
  
    # do this  
  
  } else if (that) {  
  
    # do that  
  
  } else {  
  
    # something else  
  
  }  
}
```



# Easy example

---

- Given a vector, return the mean if it's numeric, and **NULL** otherwise

```
mean2 <- function(x) {  
  if(is.numeric(x)) {  
    out <- mean(x)  
  }  
  else {  
    return()  
  }  
  out  
}
```

# Test it

---

```
mean2(rnorm(12))
```

```
## [1] 0.1855869
```

```
mean2(c("a", "b", "c"))
```

```
## NULL
```

# Mean for all numeric columns

---

- The `prior` function can now be used within a new function to calculate the mean of all columns of a data frame that are numeric
- First, let's do it "by hand", then we'll wrap it in a function. We'll use `ggplot2::mpg`

```
head(mpg, n = 3)
```

```
## # A tibble: 3 × 11
##   manufacturer model displ  year   cyl trans      drv    cty   hwy fl
##   <chr>         <chr> <dbl> <int> <int> <chr>    <chr> <int> <int> <chr>
## 1 audi         a4      1.8  1999     4 auto(l5)  f       18    29 p
## 2 audi         a4      1.8  1999     4 manual(m5) f       21    29 p
## 3 audi         a4      2    2008     4 manual(m6) f       20    31 p
```

# Calculate all means

---

```
means_mpg <- map(mpg, mean2)
means_mpg
```

```
## $manufacturer
## NULL
##
## $model
## NULL
##
## $displ
## [1] 3.471795
##
## $year
## [1] 2003.5
##
## $cyl
## [1] 5.888889
##
## $trans
## NULL
##
## $drv
## NULL
##
## $cty
```

# Drop **NULL**'s

---

- First identify which are null

```
is_null <- map_lgl(means_mpg, is.null)
is_null
```

## manufacturer	model	displ	year	cyl
## TRUE	TRUE	FALSE	FALSE	FALSE
## fl	class			
## TRUE	TRUE			

# Subset

---

Use the `is_null` object to subset

```
means_mpg[!is_null]
```

```
## $displ
## [1] 3.471795
##
## $year
## [1] 2003.5
##
## $cyl
## [1] 5.888889
##
## $cty
## [1] 16.85897
##
## $hwy
## [1] 23.44017
```

# Transform to a df

---

```
as.data.frame(means_mpg[!is_null])
```

```
##      displ   year      cyl      cty      hwy  
## 1  3.471795 2003.5  5.888889 16.85897 23.44017
```

# Challenge

---

Can you wrap all of the previous into a function?

05:00



```
means_df <- function(df) {  
  means <- map(df, mean2) # calculate means  
  nulls <- map_lgl(means, is.null) # find null values  
  means_l <- means[!nulls] # subset list to remove nulls  
  
  as.data.frame(means_l) # return a df  
}
```

```
head(mpg)
```

```
## # A tibble: 6 × 11
##   manufacturer model displ  year   cyl trans      drv    cty   hwy fl
##   <chr>          <chr> <dbl> <int> <int> <chr>    <chr> <int> <int> <chr>
## 1 audi          a4      1.8  1999     4 auto(l5)  f      18    29 p
## 2 audi          a4      1.8  1999     4 manual(m5) f      21    29 p
## 3 audi          a4      2    2008     4 manual(m6) f      20    31 p
## 4 audi          a4      2    2008     4 auto(av)  f      21    30 p
## 5 audi          a4      2.8  1999     6 auto(l5)  f      16    26 p
## 6 audi          a4      2.8  1999     6 manual(m5) f      18    26 p
```

```
means_df(mpg)
```

```
##      displ  year    cyl    cty    hwy
## 1 3.471795 2003.5 5.888889 16.85897 23.44017
```

# We have a problem though!

---

```
head(airquality)
```

```
##      Ozone  Solar.R Wind  Temp Month  Day
## 1      41      190  7.4    67     5    1
## 2      36      118  8.0    72     5    2
## 3      12      149 12.6    74     5    3
## 4      18      313 11.5    62     5    4
## 5      NA       NA 14.3    56     5    5
## 6      28       NA 14.9    66     5    6
```

```
means_df(airquality)
```

```
##      Ozone  Solar.R      Wind      Temp      Month      Day
## 1      NA       NA 9.957516 77.88235 6.993464 15.80392
```

Why is this happening?

How can we fix it?

# Easiest way . . .

---

Pass the dots!

Redefine **means2**

```
mean2 <- function(x, ...) {  
  if(is.numeric(x)) {  
    mean(x, ...)  
  }  
  else {  
    return()  
  }  
}
```

# Redefine means\_df

---

```
means_df <- function(df, ...) {  
  means <- map(df, mean2, ...) # calculate means  
  nulls <- map_lgl(means, is.null) # find null values  
  means_l <- means[!nulls] # subset list to remove nulls  
  
  as.data.frame(means_l) # return a df  
}
```

```
means_df(airquality)
```

```
##      Ozone Solar.R      Wind      Temp      Month      Day
## 1      NA      NA 9.957516 77.88235 6.993464 15.80392
```

```
means_df(airquality, na.rm = TRUE)
```

```
##      Ozone Solar.R      Wind      Temp      Month      Day
## 1 42.12931 185.9315 9.957516 77.88235 6.993464 15.80392
```

# Break

---

Functions: Part 2 upcoming

# Functions: Part 2

---

## Agenda

- Purity (quickly)
- Function conditionals
  - `if (condition) {}`
  - embedding warnings, messages, and errors
- Return values



# Learning objectives

---

- Understand the concept of purity, and why it is often desirable
  - And be able to define a side effect
- Be able to change the behavior of a function based on the input
- Be able to embed warnings/messages/errors

# Purity

---

A function is pure if

1. Its output depends *only* on its inputs
2. It makes no changes to the state of the world

Any behavior that changes the state of the world is referred to as a *side-effect*

Note – state of the world is not a technical term, just the way I think of it

# Common side effect functions

---

- We've talked about a few... what are they?

A couple examples

- `print`
- `plot`
- `write.csv`
- `read.csv`
- `Sys.time`
- `options`
- `library`
- `install.packages`

# Conditionals

---

# Example

---

From an old lab:

Write a function that takes two vectors of the same length and returns the total number of instances where the value is **NA** for both vectors. For example, given the following two vectors

```
c(1, NA, NA, 3, 3, 9, NA)
c(NA, 3, NA, 4, NA, NA, NA)
```

The function should return a value of **2**, because the vectors are both **NA** at the third and seventh locations. Provide at least one additional test that the function works as expected.

# How do you *start* to this problem?

---

04:00

~~Start with writing a function~~

Solve it on a test case, then generalize!

Use the vectors to solve!

```
a <- c(1, NA, NA, 3, 3, 9, NA)
b <- c(NA, 3, NA, 4, NA, NA, NA)
```

You try first. See if you can use these vectors to find how many elements are **NA** in both (should be 2).

# One approach

---

```
is.na(a)
```

```
## [1] FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE
```

```
is.na(b)
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
```

```
is.na(a) & is.na(b)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

```
sum(is.na(a) & is.na(b))
```

```
## [1] 2
```

# Generalize to function

---

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

What happens if not same length?

03:00



# Test it

---

```
both_na(a, b)
```

```
## [1] 2
```

```
both_na(c(a, a), c(b, b))
```

```
## [1] 4
```

```
both_na(a, c(b, b)) # ???
```

```
## [1] 4
```

What's going on here?

# Recycling

---

- R will *recycle* vectors if they are divisible

```
data.frame(nums = 1:4,  
           lets = c("a", "b"))
```

```
##   nums lets  
## 1     1    a  
## 2     2    b  
## 3     3    a  
## 4     4    b
```

- This will not work if they are not divisible

```
data.frame(nums = 1:3,  
           lets = c("a", "b"))
```

```
## Error in data.frame(nums = 1:3, lets = c("a", "b")): arguments imply dif
```

# Unexpected results

---

- In the `both_na` function, recycling can lead to unexpected results, as we saw
- What should we do?
- Check that they are the same length, return an error if not

# Check lengths

---

- Stop the evaluation of a function and return an error message with **stop**, but only if a condition has been met.

## Basic structure

```
both_na <- function(x, y) {  
  if(condition) {  
    stop("message")  
  }  
  sum(is.na(x) & is.na(y))  
}
```

# Challenge

---

Modify the code below to check that the vectors are of the same length. Return a *meaningful* error message if not. Test it out to make sure it works!

```
both_na <- function(x, y) {  
  if(condition) {  
    stop("message")  
  }  
  sum(is.na(x) & is.na(y))  
}
```

02:00

# Attempt 1

---

- Did yours look something like this?

```
both_na <- function(x, y) {  
  if(length(x) != length(y)) {  
    stop("Vectors are of different lengths")  
  }  
  sum(is.na(x) & is.na(y))  
}  
both_na(a, b)
```

```
## [1] 2
```

```
both_na(a, c(b, b))
```

```
## Error in both_na(a, c(b, b)): Vectors are of different lengths
```

# More meaningful error message?

---

What would make it more meaningful?

State the lengths of each

```
both_na <- function(x, y) {  
  if(length(x) != length(y)) {  
    v_lengths <- paste0("x = ", length(x),  
                        ", y = ", length(y))  
    stop("Vectors are of different lengths:", v_lengths)  
  }  
  sum(is.na(x) & is.na(y))  
}  
both_na(a, c(b, b))
```

```
## Error in both_na(a, c(b, b)): Vectors are of different lengths:x = 7, y
```

# Clean up

---

- Often we don't need/want to echo the function. Set `call. = FALSE`
- We also can state the lengths on new lines

```
both_na <- function(x, y) {  
  if(length(x) != length(y)) {  
    v_lengths <- paste0("x = ", length(x),  
                        "\n",  
                        "y = ", length(y))  
    stop(  
      "Vectors are of different lengths:\n",  
      v_lengths,  
      call. = FALSE  
    )  
  }  
  sum(is.na(x) & is.na(y))  
}
```



```
both_na(a, c(b, b))
```

```
## Error: Vectors are of different lengths:  
## x = 7  
## y = 14
```

# Quick error messages

---

- For quick checks, with usually less than optimal messages, use `stopifnot`
- Often useful if the function is just for you

```
z_score <- function(x) {  
  stopifnot(is.numeric(x))  
  x <- x[!is.na(x)]  
  (x - mean(x)) / sd(x)  
}  
z_score(c("a", "b", "c"))
```

```
## Error in z_score(c("a", "b", "c")): is.numeric(x) is not TRUE
```

```
z_score(c(100, 115, 112))
```

```
## [1] -1.1338934  0.7559289  0.3779645
```

# warnings

---

If you want to embed a warning, just swap out `stop()` for `warning()`

# Extension

---

- Let's build out a warning if the vectors are different lengths, but the ARE recyclable.

## Modulo operator

`%%` returns the remainder in a division problem. So `8 %% 2` and `8 %% 4` both return zero (because there is no remainder), while `7 %% 2` returns 1 and `7 %% 4` returns 3.

# One approach

---

Note the double condition

```
both_na <- function(x, y) {  
  if(length(x) != length(y)) {  
    lx <- length(x)  
    ly <- length(y)  
  
    v_lengths <- paste0("x = ", lx, ", y = ", ly)  
  
    if(lx %% ly == 0 | ly %% lx == 0) {  
      warning("Vectors were recycled (", v_lengths, ")")  
    }  
    else {  
      stop("Vectors are of different lengths and are not recycled",  
           v_lengths)  
    }  
  }  
  sum(is.na(x) & is.na(y))  
}
```

# Test it

---

```
both_na(a, c(b, b))
```

```
## Warning in both_na(a, c(b, b)): Vectors were recycled (x = 7, y = 14)
```

```
## [1] 4
```

```
both_na(a, c(b, b)[-1])
```

```
## Error in both_na(a, c(b, b)[-1]): Vectors are of different lengths and a
```

# Refactoring

---

- Refactoring means changing the internals, but the output stays the same
- Next week, we'll refactor this function to make it more readable



Step back  
from the  
ledge

---



# How important is this?

---

- For most of the work you do? Not very
- Develop a package? Very!
- Develop functions that others use, even if not through a package? Sort of.

# Return values

---

# Thinking more about return values

---

- By default the function will return the last thing that is evaluated
- Override this behavior with `return`
- This allows the return of your function to be conditional
- Generally the last thing evaluated should be the "default", or most common return value

# Pop quiz

---

- What will the following return?

```
add_two <- function(x) {  
  result <- x + 2  
}
```

Answer: Nothing! Why?

```
add_two(7)  
add_two(5)
```

# Specify the return value

---

The below are all equivalent, and all result in the same function behavior

```
add_two.1 <- function(x) {  
  result <- x + 2  
  result  
}  
add_two.2 <- function(x) {  
  x + 2  
}
```

```
add_two.3 <- function(x) {  
  result <- x + 2  
  return(result)  
}
```

# When to use `return`?

---

Generally reserve `return` for you're returning a value prior to the full evaluation of the function. Otherwise, use `.1` or `.2` methods from prior slide.

# Thinking about function names

---

Which of these is most intuitive?

```
f <- function(x) {  
  x <- sort(x)  
  data.frame(value = x,  
             p = ecdf(x)(x))  
}  
  
ptile <- function(x) {  
  x <- sort(x)  
  data.frame(value = x,  
             ptile = ecdf(x)(x))  
}  
  
percentile_df <- function(x) {  
  x <- sort(x)  
  data.frame(value = x,  
             percentile = ecdf(x)(x))  
}
```

# Output

---

- The descriptive nature of the output can also help
- Maybe a little too tricky but...

```
percentile_df <- function(x) {  
  arg <- as.list(match.call())  
  x <- sort(x)  
  d <- data.frame(value = x,  
                  percentile = ecdf(x)(x))  
  
  names(d)[1] <- paste0(as.character(arg$x), collapse = "_")  
  d  
}
```



```
random_vector <- rnorm(100)
tail(percentile_df(random_vector))
```

```
##      random_vector percentile
## 95      1.777044      0.95
## 96      1.827628      0.96
## 97      1.905176      0.97
## 98      2.222762      0.98
## 99      2.602469      0.99
## 100     2.633710      1.00
```

```
head(percentile_df(rnorm(50)))
```

```
##      rnorm_50 percentile
## 1 -2.454277      0.02
## 2 -2.428808      0.04
## 3 -2.035993      0.06
## 4 -1.508518      0.08
## 5 -1.226605      0.10
## 6 -1.114624      0.12
```

# How do we do this?

---

- I often debug functions and/or figure out how to do something within the function by changing the return value & re-running the function multiple times

[demo]

# Thinking about dependencies

---

- What's the purpose of the function?
  - Just your use? Never needed again? Don't worry about it at all.
  - Mass scale? Worry a fair bit, but make informed decisions.
- What's the likelihood of needing to reproduce the results in the future?
  - If high, worry more.
- Consider using name spacing (::)

# Next time

---

Lab 3