Functions

Packages for this section

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.2     v readr     2.1.4
v forcats   0.5.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

```
library(broom)
```

Don't repeat yourself

- See this:

```
a <- 50
b <- 11
d <- 3
as <- sqrt(a - 1)
as
```

```
[1] 7
```

```r
bs <- sqrt(b - 1)
bs
```

```
[1] 3.162278
```

```r
ds <- sqrt(d - 1)
ds
```

```
[1] 1.414214
```

What's the problem?

- Same calculation done three different times, by copying, pasting and editing.
- Dangerous: what if you forget to change something after you pasted?
- Programming principle: "don't repeat yourself".
- Hadley Wickham: don't copy-paste more than twice.
- Instead: *write a function.*

Anatomy of function

- Header line with function name and input value(s).
- Body with calculation of values to output/return.
- Return value: the output from function. In our case:

```r
sqrt_minus_1 <- function(x) {
  ans <- sqrt(x - 1)
  return(ans)
}
```

or more simply ("the R way", better style)

```r
sqrt_minus_1 <- function(x) {
  sqrt(x - 1)
}
```

If last line of function calculates value without saving it, that value is returned.

About the input; testing

- The input to a function can be called anything. Here we called it `x`. This is the name used inside the function.
- The function is a "machine" for calculating square-root-minus-1. It doesn't do anything until you call it:

```
sqrt_minus_1(50)
```

```
[1] 7
```

```
sqrt_minus_1(11)
```

```
[1] 3.162278
```

```
sqrt_minus_1(3)
```

```
[1] 1.414214
```

- It works!

Vectorization 1/2

- We conceived our function to work on numbers:

```
sqrt_minus_1(3.25)
```

```
[1] 1.5
```

- but it actually works on vectors too, as a free bonus of R:

```
sqrt_minus_1(c(50, 11, 3))
```

```
[1] 7.000000 3.162278 1.414214
```

- or… (over)

Vectorization 2/2

- or even data frames:

```
d <- tibble(x = 1:2, y = 3:4)
sqrt_minus_1(d)
```

```
  x        y
1 0 1.414214
2 1 1.732051
```

More than one input

- Allow the value to be subtracted, before taking square root, to be input to function as well, thus:

```
sqrt_minus_value <- function(x, d) {
  sqrt(x - d)
}
```

- Call the function with the x and d inputs in the right order:

```
sqrt_minus_value(51, 2)
```

```
[1] 7
```

- or give the inputs names, in which case they can be in *any order*:

```
sqrt_minus_value(d = 2, x = 51)
```

```
[1] 7
```

Defaults 1/2

- Many R functions have values that you can change if you want to, but usually you don't want to, for example:

```
x <- c(3, 4, 5, NA, 6, 7)
mean(x)
```

```
[1] NA
```

```
mean(x, na.rm = TRUE)
```

```
[1] 5
```

- By default, the mean of data with a missing value is missing, but if you specify `na.rm=TRUE`, the missing values are removed before the mean is calculated.
- That is, `na.rm` has a default value of `FALSE`: that's what it will be unless you change it.

Defaults 2/2

- In our function, set a default value for d like this:

```
sqrt_minus_value <- function(x, d = 1) {
  sqrt(x - d)
}
```

- If you specify a value for d, it will be used. If you don't, 1 will be used instead:

```
sqrt_minus_value(51, 2)
```

```
[1] 7
```

```
sqrt_minus_value(51)
```

```
[1] 7.071068
```

Catching errors before they happen

- What happened here?

```
sqrt_minus_value(6, 8)
```

```
Warning in sqrt(x - d): NaNs produced
```

```
[1] NaN
```

- Message not helpful. Actually, function tried to take square root of negative number.
- In fact, not even error, just warning.
- Check that the square root will be OK first. Here's how:

```r
sqrt_minus_value <- function(x, d = 1) {
  stopifnot(x - d >= 0)
  sqrt(x - d)
}
```

What happens with `stopifnot`

- This should be good, and is:

```r
sqrt_minus_value(8, 6)
```

```
[1] 1.414214
```

- This should fail, and see how it does:

```r
sqrt_minus_value(6, 8)
```

```
Error in sqrt_minus_value(6, 8): x - d >= 0 is not TRUE
```

- Where the function fails, we get informative error, but if everything good, the `stopifnot` does nothing.
- `stopifnot` contains one or more logical conditions, and all of them have to be true for function to work. So put in everything that you want to be true.

Using R's built-ins

- When you write a function, you can use anything built-in to R, or even any functions that you defined before.
- For example, if you will be calculating a lot of regression-line slopes, you don't have to do this from scratch: you can use R's regression calculations, like this:

```r
my_df <- tibble(x = 1:4, y = c(10, 11, 10, 14))
# my_df
my_df.1 <- lm(y ~ x, data = my_df)
# summary(my_df.1)
tidy(my_df.1)
```

```
# A tibble: 2 x 5
  term        estimate std.error statistic p.value
  <chr>          <dbl>     <dbl>     <dbl>   <dbl>
1 (Intercept)      8.5     1.88       4.53  0.0455
2 x                1.1     0.686      1.60  0.250
```

Pulling out just the slope

Use `pluck`:

```
tidy(my_df.1) %>% pluck("estimate", 2)
```

```
[1] 1.1
```

Making this into a function

- First step: make sure you have it working without a function (we do)
- Inputs: two, an `x` and a `y`.
- Output: just the slope, a number. Thus:

```
slope <- function(xx, yy) {
  y.1 <- lm(yy ~ xx)
  tidy(y.1) %>% pluck("estimate", 2)
}
```

- Check using our data from before: correct:

```
with(my_df, slope(x, y))
```

```
[1] 1.1
```

Passing things on

- `lm` has a lot of options, with defaults, that we might want to change. Instead of intercepting all the possibilities and passing them on, we can do this:

```
slope <- function(xx, yy, ...) {
  y.1 <- lm(yy ~ xx, ...)
  tidy(y.1) %>% pluck("estimate", 2)
```

```
}
```

- The `...` in the header line means "accept any other input", and the `...` in the `lm` line means "pass anything other than `x` and `y` straight on to `lm`".

Using `...`

- One of the things `lm` will accept is a vector called `subset` containing the list of observations to include in the regression.
- So we should be able to do this:

```
with(my_df, slope(x, y, subset = 3:4))
```

```
[1] 4
```

- Just uses the last two observations in `x` and `y`:

```
my_df %>% slice(3:4)
```

```
# A tibble: 2 x 2
      x     y
  <int> <dbl>
1     3    10
2     4    14
```

- so the slope should be $(14 - 10)/(4 - 3) = 4$ and is.

Running a function for each of several inputs

- Suppose we have a data frame containing several different `x`'s to use in regressions, along with the `y` we had before:

```
(d <- tibble(x1 = 1:4, x2 = c(8, 7, 6, 5), x3 = c(2, 4, 6, 9)))
```

```
# A tibble: 4 x 3
     x1    x2    x3
  <int> <dbl> <dbl>
1     1     8     2
2     2     7     4
```

```
3     3     6     6
4     4     5     9
```

- Want to use these as different x's for a regression with `y` from `my_df` as the response, and collect together the three different slopes.
- Python-like way: a `for` loop.
- R-like way: `map_dbl`: less coding, but more thinking.

The loop way

- "Pull out" column `i` of data frame `d` as `d %>% pull(i)`.
- Create empty vector `slopes` to store the slopes.
- Looping variable `i` goes from 1 to 3 (3 columns, thus 3 slopes):

```
slopes <- numeric(3)
for (i in 1:3) {
  d %>% pull(i) -> xx
  slopes[i] <- slope(xx, my_df$y)
}
slopes
```

```
[1]  1.1000000 -1.1000000  0.5140187
```

- Check this by doing the three `lm`s, one at a time.

The `map_dbl` way

- In words: for each of these (columns of `d`), run function (`slope`) with inputs "it" and `y`), and collect together the answers.
- Since slope returns a decimal number (a `dbl`), appropriate function-running function is `map_dbl`:

```
map_dbl(d, \(d) slope(d, my_df$y))
```

```
       x1         x2         x3 
 1.1000000 -1.1000000  0.5140187
```

- Same as loop, with a lot less coding.

Square roots

- "Find the square roots of each of the numbers 1 through 10":

```
x <- 1:10
map_dbl(x, \(x) sqrt(x))
```

```
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278
```

Summarizing all columns of a data frame, two ways

- use my `d` from above:

```
map_dbl(d, \(d) mean(d))
```

```
  x1   x2   x3 
2.50 6.50 5.25 
```

```
d %>% summarize(across(everything(), \(x) mean(x)))
```

```
# A tibble: 1 x 3
     x1    x2    x3
  <dbl> <dbl> <dbl>
1   2.5   6.5  5.25
```

The mean of each column, with the columns labelled.

What if summary returns more than one thing?

- For example, finding quartiles:

```
quartiles <- function(x) {
  quantile(x, c(0.25, 0.75))
}
quartiles(1:5)
```

```
25% 75% 
  2   4 
```

- When function returns more than one thing, `map` (or `map_df`) instead of `map_dbl`.

map results

- Try:

```
map(d, \(d) quartiles(d))
```

```
$x1
 25%  75% 
1.75 3.25 

$x2
 25%  75% 
5.75 7.25 

$x3
 25%  75% 
3.50 6.75 
```

- A list.

Or

- Better: pretend output from quartiles is one-column data frame:

```
map_df(d, \(d) quartiles(d))
```

```
# A tibble: 3 x 2
  `25%` `75%`
  <dbl> <dbl>
1  1.75  3.25
2  5.75  7.25
3  3.5   6.75
```

Or even

```
d %>% map_df(\(d) quartiles(d))
```

```
# A tibble: 3 x 2
  `25%` `75%`
  <dbl> <dbl>
1  1.75  3.25
2  5.75  7.25
3  3.5   6.75
```

Comments

- This works because the implicit first thing in map is (the columns of) the data frame that came out of the previous step.
- These are 1st and 3rd quartiles of each column of `d`, according to R's default definition (see help for `quantile`).

`Map` in data frames with `mutate`

- `map` can also be used within data frames to calculate new columns. Let's do the square roots of 1 through 10 again:

```
d <- tibble(x = 1:10)
d %>% mutate(root = map_dbl(x, \(x) sqrt(x)))
```

```
# A tibble: 10 x 2
       x  root
   <int> <dbl>
 1     1  1
 2     2  1.41
 3     3  1.73
 4     4  2
 5     5  2.24
 6     6  2.45
 7     7  2.65
 8     8  2.83
 9     9  3
10    10  3.16
```

Write a function first and then map it

- If the "for each" part is simple, go ahead and use `map_`-whatever.
- If not, write a function to do the complicated thing first.
- Example: "half or triple plus one": if the input is an even number, halve it; if it is an odd number, multiply it by three and add one.
- This is hard to do as a one-liner: first we have to figure out whether the input is odd or even, and then we have to do the right thing with it.

Odd or even?

- Odd or even? Work out the remainder when dividing by 2:

```
6 %% 2
```

```
[1] 0
```

```
5 %% 2
```

```
[1] 1
```

- 5 has remainder 1 so it is odd.

Write the function

- First test for integerness, then test for odd or even, and then do the appropriate calculation:

```
hotpo <- function(x) {
  stopifnot(round(x) == x) # passes if input an integer
  remainder <- x %% 2
  if (remainder == 1) {
    ans <- 3 * x + 1
  }
  else {
    ans <- x %/% 2 # integer division
  }
  ans
}
```

Test it

```r
hotpo(3)
```

```
[1] 10
```

```r
hotpo(12)
```

```
[1] 6
```

```r
hotpo(4.5)
```

```
Error in hotpo(4.5): round(x) == x is not TRUE
```

One through ten

- Use a data frame of numbers 1 through 10 again:

```r
tibble(x = 1:10) %>% mutate(y = map_int(x, \(x) hotpo(x)))
```

```
# A tibble: 10 x 2
       x     y
   <int> <int>
 1     1     4
 2     2     1
 3     3    10
 4     4     2
 5     5    16
 6     6     3
 7     7    22
 8     8     4
 9     9    28
10    10     5
```

Until I get to 1 (if I ever do)

- If I start from a number, find `hotpo` of it, then find `hotpo` of that, and keep going, what happens?
- If I get to 4, 2, 1, 4, 2, 1 I'll repeat for ever, so let's stop when we get to 1:

```
hotpo_seq <- function(x) {
  ans <- x
  while (x != 1) {
    x <- hotpo(x)
    ans <- c(ans, x)
  }
  ans
}
```

- Strategy: keep looping "while `x` is not 1".
- Each new `x`: add to the end of `ans`. When I hit 1, I break out of the `while` and return the whole `ans`.

Trying it 1/2

- Start at 6:

```
hotpo_seq(6)
```

```
[1]  6  3 10  5 16  8  4  2  1
```

Trying it 2/2

- Start at 27:

```
hotpo_seq(27)
```

```
 [1]   27   82   41  124   62   31   94   47  142   71  214
[12]  107  322  161  484  242  121  364  182   91  274  137
[23]  412  206  103  310  155  466  233  700  350  175  526
[34]  263  790  395 1186  593 1780  890  445 1336  668  334
[45]  167  502  251  754  377 1132  566  283  850  425 1276
[56]  638  319  958  479 1438  719 2158 1079 3238 1619 4858
[67] 2429 7288 3644 1822  911 2734 1367 4102 2051 6154 3077
[78] 9232 4616 2308 1154  577 1732  866  433 1300  650  325
```

```
 [89]  976  488  244  122   61  184   92   46   23   70   35
[100]  106   53  160   80   40   20   10    5   16    8    4
[111]    2    1
```

Which starting points have the longest sequences?

- The `length` of the vector returned from `hotpo_seq` says how long it took to get to 1.
- Out of the starting points 1 to 100, which one has the longest sequence?

Top 10 longest sequences

```
tibble(start = 1:100) %>%
  mutate(seq_length = map_int(
    start, \(start) length(hotpo_seq(start)))) %>%
  slice_max(seq_length, n = 10)
```

```
# A tibble: 10 x 2
   start seq_length
   <int>      <int>
 1    97        119
 2    73        116
 3    54        113
 4    55        113
 5    27        112
 6    82        111
 7    83        111
 8    41        110
 9    62        108
10    63        108
```

- 27 is an unusually low starting point to have such a long sequence.

What happens if we save the entire sequence?

```
tibble(start = 1:7) %>%
  mutate(sequence = map(start, \(start) hotpo_seq(start)))
```

```
# A tibble: 7 x 2
  start sequence
  <int> <list>
1     1 <int [1]>
2     2 <dbl [2]>
3     3 <dbl [8]>
4     4 <dbl [3]>
5     5 <dbl [6]>
6     6 <dbl [9]>
7     7 <dbl [17]>
```

- Each entry in `sequence` is itself a vector. `sequence` is a "list-column".

Using the whole sequence to find its length and its max

```
tibble(start = 1:7) %>%
  mutate(sequence = map(start, \(start) hotpo_seq(start))) %>%
  mutate(
    seq_length = map_int(sequence, \(sequence) length(sequence)),
    seq_max = map_int(sequence, \(sequence) max(sequence))
  )
```

```
# A tibble: 7 x 4
  start sequence   seq_length seq_max
  <int> <list>          <int>   <int>
1     1 <int [1]>           1       1
2     2 <dbl [2]>           2       2
3     3 <dbl [8]>           8      16
4     4 <dbl [3]>           3       4
5     5 <dbl [6]>           6      16
6     6 <dbl [9]>           9      16
7     7 <dbl [17]>         17      52
```

Does it work with `rowwise`?

```
tibble(start=1:7) %>%
  rowwise() %>%
  mutate(sequence = list(hotpo_seq(start))) %>%
  mutate(seq_length = length(sequence)) %>%
  mutate(seq_max = max(sequence))
```

```
# A tibble: 7 x 4
# Rowwise:
  start sequence   seq_length seq_max
  <int> <list>          <int>   <dbl>
1     1 <int [1]>           1       1
2     2 <dbl [2]>           2       2
3     3 <dbl [8]>           8      16
4     4 <dbl [3]>           3       4
5     5 <dbl [6]>           6      16
6     6 <dbl [9]>           9      16
7     7 <dbl [17]>         17      52
```

It does.

Final thoughts on this

- Called the **Collatz conjecture**.
- Nobody knows whether the sequence always gets to 1.
- Nobody has found an n for which it doesn't.
- A tree.