R Bootcamp Day 2

Base R

Christopher T. Kenny
Data-Driven Social Science

DATA-DRIVEN SOCIAL SCIENCE
PRINCETON UNIVERSITY

Table of contents

Today
Functions
Loops and iteration
The apply family
Exercise
Wrap-up

Today

Goals

- Familiarity with major components of “base R”
- Familiarity with functions, iteration, and vectorization
- Practice with using R interactively
- This afternoon: Quarto and tidyverse

Sources

- Wickham, Çetinkaya-Rundel, and Grolemund, *R for Data Science*, 2e (https://r4ds.hadley.nz/)
  - Ch. 25: Functions (especially §25.1-25.2)
  - Ch. 26: Iteration (especially §26.1)
- Harvard IQSS, *Math Prefresher for Political Scientists* (https://iqss.github.io/prefresher/)
  - Ch. 9: Objects, Functions, Loops (especially §§9.4, 9.6, and 9.7)
- R Core Team (https://stat.ethz.ch/R-manual/R-devel/library/base/html/lapply.html)
  - `lapply()` documentation

Functions

What is a function?

Functions make code reusable

Function components:

- Arguments: the inputs to a function
- Body: the code to be run
- Output: what gets returned

Calling a function executes the code on your inputs

```
result ← function_name(argument)
```

Why write a function?

Writing repetitive code is (1) harder to change and (2) easier to break

The calculation is duplicated:

```
income_z ← (states$income - mean(states$income, na.rm = TRUE)) /
  sd(states$income, na.rm = TRUE)

murder_z ← (states$murder_rate - mean(states$murder_rate, na.rm = TRUE)) /
  sd(states$murder_rate, na.rm = TRUE)
```

If you want to change one thing, you need to find every instance of the code

Miss a copy and the results disagree due to stale code

Writing a function

We can define any function we want with `function()`

```
z_score ← function(x) {
  (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
}

income_z ← z_score(states$income)
murder_z ← z_score(states$murder_rate)
head(income_z)
```

```
[1] -1.3211387  3.0582456  0.1533029 -1.7214837  1.1037155  0.7294092
```

Edit `z_score()` directly to fix every later call

Arguments

- Arguments take input values
- Can include defaults (e.g., `arg = 3`)

```
percentage ← function(count, total) {
  100 * count / total
}

percentage(48, 100) # matched by position
percentage(48, total = 100) # can match by name
```

```
[1] 48
[1] 48
```

- Named arguments make calls easier to read
- Defaults can always be overwritten when making a function call

Return values

- Functions return one object which can contain many values
- R returns the last evaluated expression by default

```
quick_summary ← function(x, na.rm = TRUE) {
  c(
    min = min(x, na.rm = na.rm),
    mean = mean(x, na.rm = na.rm),
    max = max(x, na.rm = na.rm)
  )
}

quick_summary(states$income)
```

```
   min   mean    max
3098.0 4435.8 6315.0
```

Local objects inside functions

Values defined within a function don't impact those outside

```
mean_center ← function(x) {
  x_mean ← mean(x)
  x - x_mean
}

x_mean ← 100
mean_center(c(2, 4, 6))
x_mean
```

```
[1] -2  0  2
[1] 100
```

Conditionals

Conditionals allow for forking paths based on logical values

```
classify_change ← function(x) {
  if (x > 0) {
    'positive'
  } else if (x == 0) {
    'zero'
  } else {
    'negative'
  }
}

c(classify_change(5), classify_change(0), classify_change(-2))
```

```
[1] "positive" "zero"     "negative"
```

Vectorized conditionals

- `if` expects one `TRUE` or `FALSE`

```
turnout_change ← c(-2, 0, 4)

if (turnout_change > 0) {
  'positive'
}
```

```
Error in `if (turnout_change > 0) ...`:
! the condition has length > 1
```

- Use `ifelse()` for element-by-element vector comparisons

```
ifelse(turnout_change > 0, 'increase', 'no increase')
```

```
[1] "no increase" "no increase" "increase"
```

Exercise 1: proportion above a cutoff

Write `prop_above()` to compute the proportion of values above a cutoff

```
prop_above ← function(x, cutoff, na.rm = TRUE) {
  # your code here
}
```

It should handle this input:

```
turnout ← c(0.71, 0.88, NA, 0.93, 0.65)
prop_above(turnout, cutoff = 0.80)
```

A Solution

```
prop_above ← function(x, cutoff, na.rm = TRUE) {
  mean(x > cutoff, na.rm = na.rm)
}

turnout ← c(0.71, 0.88, NA, 0.93, 0.65)
prop_above(turnout, cutoff = 0.80)
```

```
[1] 0.5
```

Why does this work?

```
turnout > 0.80
```

```
[1] FALSE  TRUE    NA  TRUE FALSE
```

R treats `TRUE` as 1 and `FALSE` as 0 when computing the mean

Loops and iteration

Vectorization

- R is vectorized: you can apply one operation to the full vector
- No iteration needed

```
turnout ← c(0.52, 0.61, 0.68)

100 * turnout
```

```
[1] 52 61 68
```

Math is vectorized, as are most functions in R

What if I need to work on elements one-by-one?

for loops

- Process one value at a time
- Repeat the body of the loop for each value

```
states_of_interest ← c('California', 'Massachusetts', 'New Hampshire')

for (state in states_of_interest) {
  print(paste('Selected state:', state))
}
```

```
[1] "Selected state: California"
[1] "Selected state: Massachusetts"
[1] "Selected state: New Hampshire"
```

```
state
```

```
[1] "New Hampshire"
```

Saving results

- Create a vector to store the outputs
- Use the index to fill one position on each iteration

```
columns ← states[c('income', 'hs_grad', 'murder_rate')]
column_means ← numeric(length(columns))
```

```
for (i in seq_along(columns)) {
  column_means[i] ← mean(columns[[i]])
}
```

Give it names:

```
names(column_means) ← names(columns)
column_means
```

```
     income     hs_grad murder_rate
   4435.800      53.108       7.378
```

while loops

- Repeat while **something** remains `TRUE`
- Useful when the number of iterations is unknown
- The condition must eventually become `FALSE` or your computer will die

```
daily_interviews ← c(18, 27, 24, 35, 28, 31)
interviews ← 0
day ← 0

while (interviews < 75) {
  day ← day + 1
  interviews ← interviews + daily_interviews[day]
}

c(days = day, interviews = interviews)
```

```
      days interviews
         4        104
```

The apply family

lapply(): your new best friend

- Apply a function to each element of a vector, list, etc.
- Return one **list** element for each input element
- Preserve input names

```
columns ← states[c('income', 'hs_grad', 'murder_rate')]
lapply(columns, mean)
```

```
$income
[1] 4435.8

$hs_grad
[1] 53.108

$murder_rate
[1] 7.378
```

Note: we pass the function as `mean`, not `mean()`

Additional arguments

Arguments after the function are passed to every call

```
trust_by_wave ← list(
  wave_1 = c(2, 3, NA, 4),
  wave_2 = c(3, NA, 4, 5)
)

lapply(trust_by_wave, mean, na.rm = TRUE)
```

```
$wave_1
[1] 3

$wave_2
[1] 4
```

⚠ Warning!

This is *not* advised, but is valid code and works!

Anonymous functions

- Define a function where it is used once

```
lapply(
  columns,
  function(x) c(mean = mean(x), sd = sd(x))
)
```

```
$income
     mean        sd
4435.8000  614.4699

$hs_grad
     mean        sd
53.108000  8.076998

$murder_rate
   mean      sd
7.37800 3.69154
```

Anonymous functions

- A second way to write it

```
lapply(
  columns,
  \(x) c(mean = mean(x), sd = sd(x))
)
```

```
$income
     mean        sd
4435.8000  614.4699

$hs_grad
     mean        sd
53.108000  8.076998

$murder_rate
   mean      sd
7.37800 3.69154
```

Output types

- `lapply()` always returns a list
- `sapply()` simplifies when possible

```
lapply(columns, mean)
sapply(columns, mean)
```

```
$income
[1] 4435.8

$hs_grad
[1] 53.108

$murder_rate
[1] 7.378

     income     hs_grad murder_rate
   4435.800      53.108       7.378
```

Output types: A stricter choice

- `vapply()` requires a declared output type

```
vapply(columns, mean, FUN.VALUE = numeric(1))
```

```
    income    hs_grad murder_rate
  4435.800     53.108       7.378
```

```
vapply(columns, mean, FUN.VALUE = character(1))
```

```
Error in `vapply()`:
! values must be type 'character',
 but FUN(X[[1]]) result is type 'double'
```

Split, apply, combine

- Split data into groups
- Apply one function to every group
- Combine the results

```
states_by_region ← split(states, states$region)

mean_income ← lapply(
  states_by_region,
  \(data) mean(data$income)
)

unlist(mean_income)
```

```
    Northeast         South North Central          West
     4570.222      4011.938      4611.083      4702.615
```

Iteration in base R

- These approaches return the same column means
- Choose the form that is easiest to explain and modify

```
# explicit loop
out ← numeric(length(columns))
for (i in seq_along(columns)) {
  out[i] ← mean(columns[[i]])
}
names(out) ← names(columns)

out
sapply(columns, mean)
```

```
    income     hs_grad murder_rate
  4435.800      53.108       7.378
    income     hs_grad murder_rate
  4435.800      53.108       7.378
```

Common mistakes

- Calling the function too early: `lapply(x, mean())`
- Not saving the result at each iteration
- Growing the output on every iteration
- Using `1:length(x)` when `x` may be empty (`1:0`)
- Expecting `lapply()` to return a vector
- Trying to do too much inside one loop

Exercise

Exercise 2: compare regions

Using `states`, compare regions

Create a function `region_summary()` that returns:

- Number of states
- Mean income
- Mean high school graduation rate
- Mean life expectancy

Apply it to the four regions

Write and test the function

```
region_summary ← function(data, na.rm = TRUE) {
  c(
    n = # ...,
    mean_income = # ...,
    mean_hs_grad = # ...,
    mean_life_expectancy = # ...
  )
}
```

Test on one group before adding iteration:

```
northeast ← states[states$region == "Northeast", ]
region_summary(northeast)
```

Split and apply

```
states_by_region ← split(states, states$region)
result ← lapply(states_by_region, region_summary)

class(result)
str(result)
```

- What is the type of `result`?
- What is the type of each element?
- Which names were preserved?

Missing values

- Add an `na.rm` argument
- Pass it to each summary function
- Test the function after adding an `NA`

```
northeast$income[1] ← NA
region_summary(northeast)
```

Does the function still return four values?

A Solution

```
region_summary ← function(data, na.rm = TRUE) {
  c(
    n = nrow(data),
    mean_income = mean(data$income, na.rm = na.rm),
    mean_hs_grad = mean(data$hs_grad, na.rm = na.rm),
    mean_life_expectancy = mean(data$life_expectancy, na.rm = na.rm)
  )
}

states_by_region ← split(states, states$region)
result ← lapply(states_by_region, region_summary)
do.call(rbind, result)
```

```
              n mean_income mean_hs_grad mean_life_expectancy
Northeast     9    4570.222     53.96667              71.26444
South        16    4011.938     44.34375              69.70625
North Central 12    4611.083     54.51667              71.76667
West         13    4702.615     62.00000              71.23462
```

Wrap-up

Summary

- Repeated calculation? Write a function
- Same calculation on a whole vector? Prefer vectorization to iteration
- Same function on each list element? Use `lapply()`
- Need to allow future iterations to depend on past ones? Use `for`
- Do not know the number of repetitions? Use `while`

When to use iteration

- Start with working code
- Identify repetition
- Extract the repeated calculation