Repetitive Tasks and Functional Programming

HES 505 Fall 2022: Session 4

Matt Williamson

Objectives

1. Describe the basic components of functions
2. Introduce the `apply` and `map` family of functions
3. Practice designing functions for repetitive tasks

What are functions?

- A specific `class` of R object (can call `function` inside `functions`)

```
rg <- paste("The range of mpg is", sum(mean(mtcars$mpg), sd(mtcars$mpg)), "
rg
```

```
[1] "The range of mpg is 26.1175730520891 - 14.0636769479109"
```

- A self-contained (i.e., modular) piece of code that performs a specific task
- Allows powerful customization and extension of R

Why use functions?

- Copy-and-paste and repetitive typing are *prone to errors*
- Evocative names and modular code make your analysis more tractable
- Update in one place!

If you are copy-and-pasting more than 2x, consider a function!

Designing Functions

Getting started

- Sketch out the steps in the algorithm (pseudocode!)
- Develop working code for each step
- Anonymize

```
do_something <- function(arg1, arg2, arg3){
  intermediate_process <- manipulate(arg1,arg2, arg3)
  clean_output <- cleanup(intermediate_process)
  return(clean_output)
}
```

Structure of functions: Names

- What will your function do?
- Short, but clear!
- Avoid using reserved words or functions that already exist
- Use `snake_case`

```
something <- function(...)
}
```

Not Great

```
do_something_ultraspecific
}
```

Better

```
do_something <- function(.
}
```

Pretty good

Structure of functions: Arguments

- Provide the data that the function will work on
- Provide other arguments that control the details of the computation (often with defaults)
- Called by name or position (names should be descriptive)

```
nums <- rnorm(n = 1000, mean=2, sd=1.5)
```

Same As

```
nums <- rnorm(1000, 2, 1.5)
```

Structure of functions: Body

- The body of the function appears between the `{}`
- This is where the function does its work

```r
# Compute confidence interval around mean using normal approximation
mean_ci <- function(x, conf = 0.95) {
  se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
  mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}

x <- runif(100)
mean_ci(x)
```

```
[1] 0.4642289 0.5863640
```

```r
mean_ci(x, conf = 0.99)
```

```
[1] 0.4450401 0.6055528
```

Structure of functions: Return

- Default is to return the last argument evaluated
- Can use `return()` to return an earlier value
- Can use `list` to return multiple values
- A note on the Environment

```
mean_ci <- function(x, conf = 0.95) {
  se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
  ci <- mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
  myresults <- list(alpha = alpha, ci = ci, se = se)
  return(myresults)
}

ci_result <- mean_ci(x)
```

Structure of functions: Return

```
str(ci_result)
```

```
List of 3
 $ alpha: num 0.05
 $ ci   : num [1:2] 0.464 0.586
 $ se   : num 0.0312
```

Repetitive Tasks

Iteration

- Another tool for reducing code duplication
- **Iteration** for when you need to repeat the same task on different columns or datasets
- **Imperative** iteration uses loops (`for` and `while`)
- **Functional** iteration combines functions with the `apply` family to break computational challenges into independent pieces.

Loops

- Use *counters* (`for`) or *conditionals* (`while`) to repeat a set of tasks
- 3 key components
  - **Output** - before you can loop, you need a place to store the results
  - **Sequence** - defines what you are looping over
  - **Body** - defines what the code is actually doing

Loops

```r
library(tidyverse)
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
```

```
[1]  0.68400529 -0.10006700 -0.37078755 -0.06756608
```

```
#> [1] -0.24576245 -0.28730721 -0.05669771  0.14426335
```

The `apply` family

- Vectorized functions that eliminate explicit `for` loops
- Differ by the `class` they work on and the output they return
- `apply`, `lapply` are most common; extensions for parallel processing (e.g., `parallel::mclapply`)

The `apply` family

- `apply` for vectors and data frames
- Args: `X` for the data, `MARGIN` how will the function be applied, (1=rows, 2=columns), `FUN` for your function, `...` for other arguments to the function

```
apply(mtcars, 2, mean)
```

```
       mpg        cyl       disp         hp       drat         wt       qsec 
 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750 
        vs         am       gear       carb 
  0.437500   0.406250   3.687500   2.812500 
```

The `apply` family

- `lapply` for lists (either input or output)
- Args: `X` for the data, `FUN` for your function, `...` for other arguments to the function

```
data <- list(item1 = 1:4,
             item2 = rnorm(10),
             item3 = rnorm(20, 1),
             item4 = rnorm(100, 5))

# get the mean of each list item
lapply(data, mean)
```

```
$item1
[1] 2.5

$item2
[1] 0.3090117

$item3
```

```
[1] 1.058706

$item4
[1] 4.904276
```

The `map` family

- Similar to `apply`, but more consistent input/output
- All take a vector for input
- Difference is based on the output you expect
- Integrates with `tidyverse`

The `map` family

- `map()`: output is a list
- `map_int()`: output is an integer vector
- `map_lgl()`: output is a logical vector
- `map_dbl()`: output is a double vector
- `map_chr()`: output is a character vector
- `map_df()`, `map_dfr()`, `map_dfc()`: output is a dataframe (`r` and `c` specify how to combine the data)

Some parting thoughts

- Transparency vs. speed
- Testing
- Moving forward

Back to our example