Practice writing R functions

That are relevant to simulation studies

Ian Hussey

06 Mai, 2024

Contents

Primer on functions

General structure of a function

Note that this is pseudo-code only: chunk is set not to run (`eval=FALSE`).

```
# define function
function_name <- function(argument_1, # first arugment is often the data, if the function takes a data
                          argument_2 = "default", # arguments can have defaults
                          argument_3) {
  # required packages
  require(dplyr)

  # checks
  # well written functions contain checks.
  # e.g., if the function assumes that argument_1 is a data frame, check that this is the case.
```

```
  # note that it is more useful to write the function first and add checks later.
  if(!is.data.frame(argument_1)){
    stop("argument_1 must be a data frame")
  }

  # code that does things
  object_to_be_returned <- input_data_frame |>
    # do things
    mutate(value = value + 1)

  # object to be returned
  return(object_to_be_returned)
}
```

```
# define function
interpret_cohens_d_tidyverse <- function(cohens_d) {
  # checks
  if (!is.numeric(cohens_d)) {
    stop("The Cohen's d value must be numeric.")
  }

  # requirements
  require(dplyr)
  require(tibble)

  # do stuff
  result <- data.frame(d = abs(cohens_d)) |>
    mutate(interpretation = case_when(d < .2 ~ "very small",
                                      d >= .2 & d < .5 ~ "small",
                                      d >= .5 & d < .8 ~ "medium",
                                      d >= .8 ~ "large")) |>
    pull(interpretation)

  # return result
  return(result)
}

# call function
# lets do this with lots of different inputs to make sure it gives us the result we expect
interpret_cohens_d_tidyverse(0.1)
```

```
## Lade nötiges Paket: dplyr
```

```
##
## Attache Paket: 'dplyr'
```

```
## Die folgenden Objekte sind maskiert von 'package:stats':
##
##     filter, lag
```

```
## Die folgenden Objekte sind maskiert von 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
## Lade nötiges Paket: tibble
```

```
## [1] "very small"
```

```r
interpret_cohens_d_tidyverse(0.2)
```

```
## [1] "small"
```

```r
interpret_cohens_d_tidyverse(0.5)
```

```
## [1] "medium"
```

```r
interpret_cohens_d_tidyverse(0.8)
```

```
## [1] "large"
```

```r
interpret_cohens_d_tidyverse(0.99)
```

```
## [1] "large"
```

```r
interpret_cohens_d_tidyverse(-0.55)
```

```
## [1] "medium"
```

```r
#interpret_cohens_d_tidyverse("should throw error with this input")
```

Remember:

- The function must be present in your environment to be usable, and must be called to be used
- Build the 'do stuff' part outside of a function first!
- If you can't immediately write the code, write pseudo-code first!
- Check that your function actually works as you expect, not just that it runs. Give it lots of different input values that should and should not work, and check you get the correct outputs.

Ideas for useful functions to practice writing

Many of these functions have already been implemented by existing R packages (e.g., packages in the {easystats} universe). However, our goal is not merely to find an existing solution, but to write one ourselves for practice.

Relevant to generating data

- Generate data from a uniform distribution and return a data frame.
- Generate bounded data, e.g., responses that are continuous but must be between 1 and 10.
- Generate Likert data, e.g., responses that are whole numbers between 1 and 7, but which follow an underlying normal distribution.
- Simulate outliers, e.g., from careless responding or bots.
- Generate item-level data for cross sectional studies [useful but too complicated]

Relevant to analyzing data

- Convert a p value's significance to create APA-format table stars (i.e., "***" vs. **" vs. "" vs "ns")
- Convert a Cohen's d estimate to an interpretation
- Fit a correlation test and extract the p value and correlation
- Fit a regression and extract key results (p values, Beta estimates, etc)
- Fit and extract Cronbach's alpha [requires item level data]
- Choose another kind of test, fit it and extract its key information (eg estimate, p value), such as an assumption test, so that we could simulate the utility of tests of assumptions.
- Simulate publication bias by labeling a given study as "published" or "unpublished" based on a combination of its p value and a defined probability of (non)significant studies being published or not.

Relevant to summarizing simation results across iterations

- Summarize a column of data into a string that summarizes its mean and SD, which could be pasted directly into a manuscript. i.e., taking the form "M = XX.X (SD = XX.X)", with rounding and retention of lagging zeros.
- A function that rounds all numeric variables in a data frame by a given number of places.

'Do it lots of times': calling functions built to take a single input value on multiple input values

I.e., just as we need to do in our simulations.

Method 1: using `lapply()`

One of base R's `apply()` functions for exactly this.

Note that the input is a vector and the output is a list of numerics, which is tricky to work with.

```
# define vector with multiple valid values of cohen's d
cohens_d_values <- c(0.1, 0.2, 0.5, 0.8, 0.99, -0.55)

# use lapply to apply the function to each element of the vector
lapply(X = cohens_d_values, FUN = interpret_cohens_d_tidyverse)
```

```
## [[1]]
## [1] "very small"
##
## [[2]]
## [1] "small"
##
## [[3]]
## [1] "medium"
##
## [[4]]
## [1] "large"
##
## [[5]]
## [1] "large"
##
## [[6]]
## [1] "medium"
```

Method 2: using `purrr::map()`

Which is a version of `purrr::pmap()`, which we use in our simulations.

Note that the input is a vector and the output is a list of numerics, which is tricky to work with.

```
library(purrr)
# use map to map the function onto each element of the vector
map(cohens_d_values, interpret_cohens_d_tidyverse)
```

```
## [[1]]
## [1] "very small"
##
## [[2]]
```

```
## [1] "small"
##
## [[3]]
## [1] "medium"
##
## [[4]]
## [1] "large"
##
## [[5]]
## [1] "large"
##
## [[6]]
## [1] "medium"
```

NB: same result as lapply, but it is built to be used in more tidyverse like workflows.

Method 3: using `purrr::pmap()`

As in our simulations. Note that the input and output are a data frame, which is easier to work with further within a simulation workflow.

```
# generate data
dat <- tibble(cohens_d = cohens_d_values)

# apply function
dat |>
  mutate(interpretation = pmap(list(cohens_d),
                               interpret_cohens_d_tidyverse),
         # convert to character rather than unnest(), as used in the simulations, because reasons that
         interpretation = as.character(interpretation))
```

```
## # A tibble: 6 x 2
##   cohens_d interpretation
##      <dbl> <chr>
## 1     0.1  very small
## 2     0.2  small
## 3     0.5  medium
## 4     0.8  large
## 5     0.99 large
## 6    -0.55 medium
```

NB: input and output are now data frames, passed via pipes, so pmap() can be used in a fully tidyverse workflow. We saw in previous weeks how the combination of `expand_grid()` to build the simulation conditions and interations and `pmap()` to apply data generation and data analysis functions to those parameters create a fully tidyverse workflow for building simulations.

What makes writing functions difficult?

You have to learn and do multiple things at once:

- Problem conceptualization
- Formal logic
- R syntax

AI for writing code

OpenAI's ChatGPT, GitHub's Copilot, etc. can be great for writing functions and you should learn to use them *effectively*.

However, they cannot remove the need for understanding from the coder: you need to know how to formulate your question/need, how to check the suggested code for errors and omissions, and it is you rather than the AI who will get the blame if the code is wrong. AI can help you write code that you could have written but faster (good). It can cause problems when you have it write code you could not have written yourself.

Solutions to the above

Session info

```
sessionInfo()
```

```
## R version 4.3.2 (2023-10-31 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 11 x64 (build 22631)
##
## Matrix products: default
##
##
## locale:
## [1] LC_COLLATE=German_Switzerland.utf8  LC_CTYPE=German_Switzerland.utf8
## [3] LC_MONETARY=German_Switzerland.utf8 LC_NUMERIC=C
## [5] LC_TIME=German_Switzerland.utf8
##
## time zone: Europe/Zurich
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] purrr_1.0.2  tibble_3.2.1 dplyr_1.1.4
##
## loaded via a namespace (and not attached):
##  [1] digest_0.6.35     utf8_1.2.4        R6_2.5.1          fastmap_1.1.1
##  [5] tidyselect_1.2.1  xfun_0.43         magrittr_2.0.3    glue_1.7.0
##  [9] knitr_1.46        pkgconfig_2.0.3   htmltools_0.5.8.1 rmarkdown_2.26
## [13] generics_0.1.3    lifecycle_1.0.4   cli_3.6.2         fansi_1.0.6
## [17] vctrs_0.6.5       withr_3.0.0       compiler_4.3.2    rstudioapi_0.16.0
## [21] tools_4.3.2       pillar_1.9.0      evaluate_0.23     yaml_2.3.8
## [25] rlang_1.1.3
```