Week 4 Tutorial: Functions in R

POP77001 Computer Programming for Social Scientists

Naming Functions

- Use **lower_case_with_underscores** style for R objects.
- Give short, but descriptive names.
- Try to use verbs in function names:

```
summarise()
get_coefs()
```

rather than:

```
summary()
coefs()
```

- An added benefit is in distinguishing user-defined functions from built-it (e.g. `summary()`, `coef()`)

Code Layout

- Limit all lines to a maximum of 79 characters.
- If the function name and definition are too long, break it up:
  - across *function-indented* lines:

```
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

- across *double-indented* lines:

```
long_function_name <- function(
    a = "a long argument",
    b = "another argument",
    c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

- When it fits, *function-indented* style should be preferred to *double-indented*.

Exercise: Function Definition

- Study the code for calculating a t-test below:

```r
t_test <- function(x, y) {
  # Calculate the means of the two samples
  mean_x <- mean(x)
  mean_y <- mean(y)

  # Calculate the variances of the two samples
  var_x <- var(x)
  var_y <- var(y)

  # Calculate the sample sizes
  n_x <- length(x)
  n_y <- length(y)

  # Calculate the sample standard errors
  se_x <- sqrt(var_x/n_x)
  se_y <- sqrt(var_y/n_y)
  se <- sqrt(se_x^2 + se_y^2)

  # Calculate the t-statistic
  t_stat <- (mean_x - mean_y) / se

  # Calculate the degrees of freedom
  df <- se^4/(se_x^4/(n_x - 1) + se_y^4/(n_y - 1))

  # Calculate the p-value
  p_val <- 2 * (1 - pt(abs(t_stat), df))

  # Create a result list
  res <- list(
    t_statistic = t_stat,
    degrees_of_freedom = df,
    p_value = p_val
  )

  return(res)
}
```

Exercise: Function Components

- What are the components of the function `t_test()`?
- Check each component using access functions.

Exercise: Function Call

- Call the function on 2 random samples from the standard normal distribution (`rnorm()`).
- What is the return object of this function?
- Compare the return object of the `t_test()` with that of the built-in `t.test()`.
- Modify the function code to use implicit return.
- Modify the function to return a named vector instead of a list.

Exercise: Built-in Source Code

- As R is an open source language, you can check the inner workings of all the built-in functions.
- Run `getS3method("t.test", "default")` to study the source code of the built-in `t.test()` function.

Exercise: Functionals

- As R is a functional language, many of iteration routines can be avoided.
- E.g. instead of creating a loop for calculating standard deviations, we can use `apply()` function.
- `apply(<object_name>, 2, <function_name>)` allows to calculate the desired summary statistic for each of the variables.
- Apply this function to the matrix from the exercise above
- Now, change 2 in the function call to 1
- What do you see? What do the current numbers show? Does this summary make sense and why?

```r
# When dealing with random number generation it's always a good idea to mak
# by setting the seed with set.seed(function)
set.seed(2025)
# Here we create a matrix of 30 observations of 5 variables
# where each variable is a random draw from a normal distribution with mean
# and standard deviation drawn from a uniform distribution between 0 and 10
mat <- mapply(
  function(x) cbind(rnorm(n = 30, mean = 0, sd = x)),
  runif(n = 5, min = 0, max = 10)
)
```

```r
mat
```

```
             [,1]        [,2]        [,3]        [,4]         [,5]
 [1,]   0.0780885  3.48089343   2.9553545   3.2248615  -4.66712911
 [2,]  -8.3269637 -2.05680059   6.5670977   8.3727181 -13.86628154
 [3,]   1.2834069 -4.25078791   3.0311714  -3.6736945   2.20685092
 [4,]  12.2189488 12.51048821   2.2500211   5.8992348  12.81071523
 [5,] -14.5079803 -0.83591818  -0.8329266   1.0321989   5.73323413
 [6,]   7.8277431 -0.04931062   1.4101378   6.7058639   2.63359979
 [7,]  -2.5172557  1.27351635  -5.6927903 -12.9850971  -8.19349948
 [8,]  -9.0808601  1.07779111   3.0050245  -4.1673004   0.51438396
 [9,]   6.5582896 -3.70133731   7.1498934  -8.8572673   0.05485376
[10,]   4.1685004 -6.50775660   9.0180872  -4.5555447  -4.32601307
[11,]  10.2156894 -1.84773737   2.4000462   6.2044559  -6.12311135
[12,]  -6.1880010 -4.12393197  -2.1740754  -0.4008144   3.40181041
```

Week 4 Exercise (unassessed)

- Re-visit the code for converting grades into marks:

```
convert_mark_to_grade <- function(mark) {
  if (mark >= 70) {
    grade <- "I"
  } else if (mark >= 60) {
    grade <- "II.1"
  } else if (mark >= 50) {
    grade <- "II.2"
  } else {
    grade <- "F"
  }
  grade
}
```

- As discussed in the lecture, this function only takes a single mark as an input.
- Modify this function such that if takes a vector of marks (longer than 1) as an input and returns a vector of grades.