Week-5: Code-along

Dawn Cheung

2023-09-09

II. Code to edit and execute using the Code-along.Rmd file

A. Writing a function

1. Write a function to print a "Hello" message (Slide #14)

```
# Enter code here
name <- "Kashif"
hello_generator <- function(x) { # is the generic placeholder
  print(paste0("Hello ", x, "!"))
}
hello_generator(name)
```

```
## [1] "Hello Kashif!"
```

```
#??hello_generator
```

2. Function call with different input names (Slide #15)

```
# Enter code here
name <- "Slay"
hello_generator(name)
```

```
## [1] "Hello Slay!"
```

```
name <- "Unslay"
hello_generator(name)
```

```
## [1] "Hello Unslay!"
```

3. typeof primitive functions (Slide #16)

```
# Enter code here
typeof(`+`)
```

```
## [1] "builtin"
```

```
typeof(mean)
```

```
## [1] "closure"
```

```
typeof(sum)
```

```
## [1] "builtin"
```

4. typeof user-defined functions (Slide #17)

```
# Enter code here
typeof(hello_generator)
```

```
## [1] "closure"
```

5. Function to calculate mean of a sample (Slide #19)

```
# Enter code here
mean(rnorm(100))
```

```
## [1] 0.05048804
```

```
mean(rnorm(3000))
```

```
## [1] -0.005227069
```

```
calc_sample_mean <- function(sample_size) {
  mean(rnorm(sample_size))
} #no need return
```

6. Test your function (Slide #22)

```
# With one input
calc_sample_mean(90)
```

```
## [1] -0.03311139
```

```
calc_sample_mean(90)
```

```
## [1] 0.03771719
```

```
# With vector input
calc_sample_mean(c(200, 399, 100))
```

7. Customizing the function to suit input (Slide #23)

(Dealing with unvectorised functions)

```
# Enter code here
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.2     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.3     ✔ tibble    3.2.1
## ✔ lubridate 1.9.2     ✔ tidyr     1.3.0
## ✔ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be
come errors
```

```
sample_tibble <- tibble(sample_sizes = c(100, 300, 3000))
#tibbles are lists where all the columns or vairables have the same number of entries
sample_tibble %>% group_by(sample_sizes) %>%
  mutate(sample_means = calc_sample_mean(sample_sizes)) #calc_sample_mean is our own function
```

```
## # A tibble: 3 × 2
## # Groups:   sample_sizes [3]
##   sample_sizes sample_means
##          <dbl>        <dbl>
## 1          100    -0.0599
## 2          300    -0.0588
## 3         3000     0.000573
```

8. Setting defaults (Slide #25)

```
# First define the function
calc_sample_mean <- function(sample_size,
                             our_mean = 0,
                             our_sd = 1) {
  sample <- rnorm(sample_size,
                  mean = our_mean,
                  sd = our_sd)
  mean(sample)
}
# Call the function
calc_sample_mean(sample_size = 10) #the order of arguments matter
```

```
## [1] -0.1745392
```

9. Different input combinations (Slide #26)

```
# Enter code here
calc_sample_mean(10, our_sd = 2)
```

```
## [1] 0.10707
```

```
calc_sample_mean(10, our_mean = 6)
```

```
## [1] 5.474105
```

```
calc_sample_mean(10, 6, 2) #sample_size, our_mean, our_sd in order
```

```
## [1] 6.090785
```

```
#sample_size die die needs to be there bc we did not set a defalt value for it
```

10. Different input combinations (Slide #27)

```
# set error=TRUE to see the error message in the output
# Enter code here
calc_sample_mean(our_mean = 6)
```

```
## Error in calc_sample_mean(our_mean = 6): argument "sample_size" is missing, with no defaul
t
```

11. Some more examples (Slide #28)

```
# Enter code here
add_two <- function(x) {
  x+2
}
add_two(4)
```

```
## [1] 6
```

```
add_two(-34)
```

```
## [1] -32
```

```
add_two(5.784)
```

```
## [1] 7.784
```

```
# the 'return' function can only return 1 value at a time ie return(x,y) cannot be done
#the idea of local and global variables apply for R's function
```

B. Scoping (Variable scopes: global vs local variables)

12. Multiple assignment of z (Slide #36)

```
# Enter code here
z <- 1
sprintf("the value assigned to z outside the function is %d", z)
```

```
## [1] "the value assigned to z outside the function is 1"
```

```
#set z to the default value of 2
foo <- function(z = 2) {
  #reassigning z
  z <- 3
  return(z+3)
}
foo()
```

```
## [1] 6
```

13. Multiple assignment of z (Slide #37)

```r
# Enter code here
z <- 1
sprintf("the value assigned to z outside the function is %d", z)
```

```
## [1] "the value assigned to z outside the function is 1"
```

```r
foo <- function(z = 2) {
  z <- 3
  return(z+3)
}
# yet another reassignment of z
foo(z = 4)
```

```
## [1] 6
```

```r
sprintf("the FINAL value assigned to z after reassigning it to a different value inside the f
unction is %d", z) #u get 1 bc that's the global variable
```

```
## [1] "the FINAL value assigned to z after reassigning it to a different value inside the fu
nction is 1"
```