UTSA®

ALVAREZ
College of Business
The University of Texas at San Antonio

Introduction to Programming in R

Module 6:
Functions & Loops

Learning Objectives

- Using pipes
- Creating user-defined functions
- Writing if-else (conditional) statements
- Structuring loops

Pipes %>%

- A sequence of multiple operations.
- Forwards the value/result of an expression to the next expression.
- Part of "magrittr" package.
  - First *install.packages("magrittr")*
  - Load *library(magrittr)*

Tidyverse takes care of this

- Pipes makes your code more efficient and easier to understand.
  - Reduces unnecessary intermediate objects.
- For e.g. *filter(data, variable == numeric_value)*

  or *data %>% filter(variable == numeric_value)*.
  - Same result in both cases: filtering the data for a variable that matches a value.

Example

- Load a built-in dataset called mtcars. Use *data("mtcars").*
    - Data frame of 32 observations and 11 variables.
    - help("mtcars") for more details.
- Task: Find the average mpg of cars with more than 1 carburetor, grouped by the number of cylinders, and report it in descending order.
- Possible ways to solve it:
    - Nested functions using indentations
    - Multiple intermediary objects creation
    - Pipes %>%

Example: Nested Functions

- Task: Find the average mpg of cars with more than 1 carburetor, grouped by the number of cylinders, and report it in descending order.
- Traditional way!
- Not clear to read and understand.

```
arrange(
   summarize(
       group_by(
           filter(mtcars, carb > 1),
           cyl
          ),
       Avg_mpg = mean(mpg)
      ),
   desc(Avg_mpg)
 )
## Source: local data frame [3 x 2]
##
##      cyl Avg_mpg
##    (dbl)   (dbl)
## 1      4   25.90
## 2      6   19.74
## 3      8   15.10
```

Example: Multiple Intermediary Objects

- Task: Find the average mpg of cars with more than 1 carburetor, grouped by the number of cylinders, and report it in descending order.
- Useless intermediary objects
- Inefficient and uses up memory.

```
a <- filter(mtcars, carb > 1)
b <- group_by(a, cyl)
c <- summarise(b, Avg_mpg = mean(mpg))
d <- arrange(c, desc(Avg_mpg))
print(d)
## Source: local data frame [3 x 2]
##
##     cyl Avg_mpg
##   (dbl)   (dbl)
## 1     4   25.90
## 2     6   19.74
## 3     8   15.10
```

Figure from https://uc-r.github.io/pipe

Example: Pipes

- Task: Find the average mpg of cars with more than 1 carburetor, grouped by the number of cylinders, and report it in descending order.
- Merges nested and intermediary objects frameworks.
- Efficient and easy to read.
- Read %>% as next…

```
library(magrittr)
library(dplyr)

mtcars %>%
        filter(carb > 1) %>%
        group_by(cyl) %>%
        summarise(Avg_mpg = mean(mpg)) %>%
        arrange(desc(Avg_mpg))
## Source: local data frame [3 x 2]
##
##      cyl Avg_mpg
##    (dbl)   (dbl)
## 1      4   25.90
## 2      6   19.74
## 3      8   15.10
```

Figure from https://uc-r.github.io/pipe

Example: Pipes (cont.)

- Without pipes

```
> mutate(filter(group_by(mtcars, cyl, gear, carb), mpg>27), MileagePerCylinder = mpg/cyl)
# A tibble: 5 x 12
# Groups:   cyl, gear, carb [3]
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb MileagePerCylinder
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>              <dbl>
1  32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1               8.1 
2  30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2               7.6 
3  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1               8.48
4  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1               6.82
5  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2               7.6 
```

- Find out the mileage per cylinder of cars grouped by cylinders, gears, carburetors with mileage > 27mpg.

- With pipes

```
> mtcars %>% group_by(cyl, gear, carb) %>% filter(mpg > 27) %>% mutate(MileagePerCylinder = mpg/cyl)
# A tibble: 5 x 12
# Groups:   cyl, gear, carb [3]
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb MileagePerCylinder
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>              <dbl>
1  32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1               8.1 
2  30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2               7.6 
3  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1               8.48
4  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1               6.82
5  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2               7.6 
```

Pipes

Sometimes pipes are not a good idea -

- If your pipes are longer than 10 steps, it's better to create intermediate objects with descriptive names.
  - Easier to debug.
- If you have multiple inputs and outputs, i.e. more than one object going in or out.
- Pipes are sequential and linear. So, if you have a complex interconnected network of functions, pipes is not a good idea.

magrittr package also provides a few other special pipes that can be helpful.

- Some functions don't return a result, such as *plot()*, so the pipe gets terminated.
  - Tee-pipes %T>% overcome this – returns the left-hand side of the pipe instead of right-hand side.

```
> rnorm(100) %>% matrix(ncol = 2) %>% plot() %>% str()
NULL
```

vs.

```
> rnorm(100) %>% matrix(ncol = 2) %T>% plot() %>% str()
 num [1:50, 1:2] 0.542 -0.752 -0.205 -2.122 -0.184 ...
```

Example 1

Load *magrittr* pkg*.* Get the *mtcars* dataframe. Using pipes, write an expression that

1. Plots the scatterplot matrix and
2. Shows summary statistics for

mpg, cyl, disp, and *hp* for cars with more than 1 carburetor.

3. Compute the correlation between *mpg* and *hp*.

Using the *diamonds* dataset and pipes, write expressions that

4. Calculate the average price for each cut of “I” colored diamonds.
5. Calculate the standard deviation of carats for each color of ideal cut diamonds.
6. Creates a new column called dims, with x, y, and z combined and separated by commas.

Functions

- A set of statements organized together to perform a specific task.
- Takes operations and arguments, performs its task and returns the results.
- Created by using the keyword **function**.
  - *function_name <- function(arg_1, arg_2, …){function body}*
- Reduces code duplication. Can reuse multiple times in the same script.
  - Improve computing time and reduces the amount of code writing required
  - E.g.

```
> df <-tibble(
+       a = rnorm(10),
+       b = rnorm(10),
+       c = rnorm(10),
+       d = rnorm(10)
+ )
```

Without functions

```
df$a <- (df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
    (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
    (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
    (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

With functions

```
> rescale01 <- function(x) {
+     rng <- range(x, na.rm = TRUE)
+     (x - rng[1]) / (rng[2] - rng[1])
+ }
```

```
> df$a <- rescale01(df$a)
> df$b <- rescale01(df$b)
> df$c <- rescale01(df$c)
> df$d <- rescale01(df$d)
```

```
> print(df)
# A tibble: 10 x 4
       a      b     c     d
   <dbl>  <dbl> <dbl> <dbl>
 1 0.358 0.753  0.206 0.635
 2 0     0      0.705 0.174
 3 0.443 0.225  0.467 0.285
 4 0.642 0.847  1     0.533
 5 0.513 1.59   0.269 1
 6 0.670 0.778  0.970 0
 7 1     0.594  0.386 0.658
 8 0.732 0.248  0.730 0.654
 9 0.485 0.0749 0.892 0.314
10 0.595 0.183  0     0.746
```

Functions (cont.)

- Create a data frame with 4 columns, each column comes with 3 observations.

```
> (df = data.frame(a = c(1:3), b = c(4:6), c = c(7:9), d = c(10:12)))
  a b c  d
1 1 4 7 10
2 2 5 8 11
3 3 6 9 12
```

- Task: Convert elements to z-scores , i.e. .
- Possible ways to solve it:
  - Apply multiple blocks of repeated code to each column.
  - Define a general function and apply the function to each column.

Functions (cont..)

Task: Convert to z-scores

Copy-paste:

- Lots of code redundancy!
- Easy to make mistakes each time we copied and pasted same blocks of code.

Functions:

- Code is more readable.
- Easy to modify code blocks.
- Function can be reused later in the script.

```
> (df$a - mean(df$a))/sd(df$a)
[1] -1  0  1
> (df$b - mean(df$b))/sd(df$b)
[1] -1  0  1
> (df$c - mean(df$c))/sd(df$c)
[1] -1  0  1
> (df$d - mean(df$d))/sd(df$d)
[1] -1  0  1
```

```
> zscore <- function(x) {
+     z <- (x - mean(x))/sd(x)
+     return(z)
+ }
> zscore(df$a)
[1] -1  0  1
> zscore(df$b)
[1] -1  0  1
> zscore(df$c)
[1] -1  0  1
> zscore(df$d)
[1] -1  0  1
```

Functions (cont…)

- Functions also allows for easier modifications.
  - For e.g., if the data has NAs or Infinity values.

```
> (df = data.frame(a = c(1:3), b = c(4,5,Inf), c = c(7,8,NA), d = c(10:12)))
  a   b  c  d
1 1   4  7 10
2 2   5  8 11
3 3 Inf NA 12
```

- Then, the previous *zscore()* fails

```
> zscore(df$a)
[1] -1  0  1
> zscore(df$b)
[1] NaN NaN NaN
```

```
> zscore(df$c)
[1] NA NA NA
> zscore(df$d)
[1] -1  0  1
```

- But if we modify *zscore()* as

```
> zscore <- function(x) {
+     x = x[which(x < Inf)]
+     z <- (x - mean(x, na.rm = TRUE))/sd(x, na.rm = TRUE)
+     return(z)
+ }
> zscore(df$a); zscore(df$b); zscore(df$c); zscore(df$d)
[1] -1  0  1
[1] -0.7071068  0.7071068
[1] -0.7071068  0.7071068
[1] -1  0  1
```

- By default, a function will return the last computed value, unless specified by *return()*.

Storing Functions

- In most cases, we write functions into scripts, most likely at the top of the script.
- This can get untidy if there are many functions.
- Instead, you can store functions separately and load them using *source()*.
- Open a new R script file and put the z-score function there.

zscore.R

Source on Save

```r
zscore <- function(x) {

  z = x - mean(x)/(sd(x))

  return(z)
}
```

✓ Reusable
✓ Easy to maintain
✓ Shareable

- Then whenever you need it in any script, just use

*source("*pathname including filename of the function*").*

Example 2

1. Create a function that takes a variable and normalizes it. This is very common in data science.

$$normalize = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Load the *airquality* data, write piped expressions that

2. Uses your normalize function on the *Ozone* variable and computes monthly grouped means of the normalized *Ozone*.

```
# A tibble: 5 x 2
  Month month_norm_ozone
  <int>            <dbl>
1     5            0.135
2     6            0.170
3     7            0.348
4     8            0.353
5     9            0.182
```

3. Repeat Q2 with z-score function.

Conditionals

- IF statements are one of the most useful available functions.
- Format:

  if (condition) {
  code executed when condition is true
  } else {
  code executed when condition is false
  }
- You can use || (or) and && (and) to combine multiple conditions.
  - || will return true as soon as the first true is found
  - && will return false as soon as the first false is found.
  - Note that the rest of the expression is not even evaluated, as soon as the condition is met.
- Do not use single | or & within IF statements. They work with vectorized data, as you saw with *filter()*.
  - If your condition involves a vector, collapse it to a single value using *any()* or *all()*.

Conditionals (cont.)

- Be careful when checking for equality as == works on vectorized data, i.e. you'll most likely get a vector output.
- You can use *any()* and *all()* to collapse the resultant vector.
- Use *identical()* that provides a scalar output.
    - *identical()* is very strict – doesn't coerce data types.
- == also has issues with floating point numbers.

```
> if(c(1:3) == c(1,3,5)) {
+     sum(1:5)
+ }else{
+         sum(1:2)
+     }
[1] 15
Warning message:
In if (c(1:3) == c(1, 3, 5)) { :
  the condition has length > 1 and only the first element will be used
```

```
> if(all(c(1:3) == c(1,3,5))) {
+     sum(1:5)
+ }else{
+         sum(1:2)
+     }
[1] 3
```

```
> if(identical(c(1:3),c(1,3,5))) {
+     sum(1:5)
+ }else{
+         sum(1:2)
+     }
[1] 3
```

```
> sqrt(2)^2 == 2
[1] FALSE
```

Multiple Conditionals

- IF-ELSEIF-ELSE statements can be used to combine multiple conditionals.

  ***if** (condition 1) {*

  do this

  *} **else if** (condition 2) {*

  do something else

  *} **else** {*

  do that

  }

- If you have too many ELSEIFs, consider using *switch()*.
- *switch()* can evaluate code based on position or name.

```
> switch(2,"red","green","blue")
[1] "green"
> switch("color", "color" = "red",
+         "shape" = "square", "length" = 5)
[1] "red"
```

Debugging advice:

- You can place stop("warning message") inside a condition that checks for errors.

Examples

1. Write a function called *myfun* that accepts numeric vectors and returns the squared elements as a vector.
   - If other data types are entered, it should display a warning stating "*Wrong data format!*"

Hint: *?stop* and *?is.vector* and use conditionals.

2. Write a function called *mydist* that generates *n* random numbers for a specified distribution: gamma, exponential, or normal.
   - Accept strings "gamma", "exp" or "norm".
   - If other distributions are entered an error is printed stating "distribution must be gamma, exponential or normal".
   - For gamma distributions, a shape parameter must also be entered, otherwise error.

Hint: use *switch()*.

Loops

- A multi-step process with organized sequences of actions that need to be repeated.
- Execute repetitive code statements for a specified number of times
- Loops makes repeating calculations and actions more efficient and easier to understand.
- For loop

```
for (i in 1:n) { <execute this>}
```

- While loop

```
counter <- 1
while(test_expression){statement counter <- counter + 1 }
```

For Loop

- Create a data frame with 4 columns, each column has 10 obs of random numbers normally distributed (like before).

```
> df <- tibble(
+       a = rnorm(10),
+       b = rnorm(10),
+       c = rnorm(10),
+       d = rnorm(10)
+ )
```

```
> print(df)
# A tibble: 10 x 4
        a      b      c      d
    <dbl>  <dbl>  <dbl>  <dbl>
 1  0.782  -0.595 1.10    1.34
 2  0.0282 -1.13  1.63   -0.304
 3  0.741  -0.240 0.943   0.394
 4  0.412   0.958 1.83   -0.935
 5  0.778  -1.40  0.0382 -0.394
 6 -2.05    0.543 -1.21  -0.928
 7  0.143   0.743 -0.316 -1.25
 8 -1.69    0.0630 -0.560 0.819
 9 -1.78    0.254 -1.99   0.174
10 -2.00   -0.315 0.557   1.18
```

- Task: Find median of a, b, c, and d.
- Copy-paste:

```
> median(df$a); median(df$b); median(df$c); median(df$d)
[1] 0.08575371
[1] -0.08858712
[1] 0.2976156
[1] -0.06507679
```

- For loop option:

Initialize output for allocating memory

```
> output <- vector("double", ncol(df))
> for (i in seq_along(df)) {
+       output[[i]] <- median(df[[i]])
+ }
> output
[1]  0.08575371 -0.08858712  0.29761557 -0.06507679
```

Sequence by *seq_along()* - same as *length()* with minor improvements

Body of loop

While Loop

- Sometimes, we do not know how long the iteration should run for.
- We just know we want to iterate until a condition is met.
- While loop format: *while (condition) {*body of loop*}*
- A while loop is more general than for loop.
  - Can always write a for loop as a while loop, but not the other way.
- Coin Flip E.g. – how many coin flips does it take to get 3 heads in a row.

for (i in seq_along(x)) {
body of loop
}

same result

i <- 1
while (i <= length(x)) {
body of loop
i <- i + 1
}

```
> flip <- function() sample(c("T", "H"), 1)
>
> flips <- 0
> nheads <- 0
>
> while (nheads < 3) {
+     if (flip() == "H") {
+         nheads <- nheads + 1
+     } else {
+         nheads <- 0
+     }
+     flips <- flips + 1
+ }
> flips
```

Loop vs. Functions

- Let’s work with the same data frame.
- We want to compute column means.
- Now, also compute median and standard deviation:
  - Copy-paste: but that’s not efficient or clean.
  - Create a function with the loop!

```
> col_summary <- function(df, fun) {
+     out <- vector("double", length(df))
+     for (i in seq_along(df)) {
+         out[i] <- fun(df[[i]])
+     }
+     out
+ }
> col_summary(df, median)
[1] -0.02159531  0.41958823 -0.37241157 -0.59274634
> col_summary(df, mean)
[1] -0.03102625  0.46680961 -0.06090380 -0.43116568
> col_summary(df, sd)
[1] 0.8167022 0.8408837 1.1741349 0.8807663
```

Loops vs. Functions

- Some functions work like loops. Such functions accept a function (fun) argument to execute on elements in vectors, matrices, etc.
- *apply(x, margin, fun)* outputs a vector, list, or array
  - x is data frame or matrix
  - margin = 1 row, margin = 2 column, margin = c(1,2) for both.

```
> (m1 <- matrix(C<-(1:10),nrow=5, ncol=6))
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6    1    6    1    6
[2,]    2    7    2    7    2    7
[3,]    3    8    3    8    3    8
[4,]    4    9    4    9    4    9
[5,]    5   10    5   10    5   10
> (a_m1 <- apply(m1, 2, sum))
[1] 15 40 15 40 15 40
```

- *lapply(x, fun)* outputs a list
  - *x is list, vector, or data frame*

Same except output

```
> movies <- c("SPYDERMAN","BATMAN","VERTIGO","CHINATOWN")
> movies_lower <-lapply(movies, tolower)
> str(movies_lower)
List of 4
 $ : chr "spyderman"
 $ : chr "batman"
 $ : chr "vertigo"
 $ : chr "chinatown"
```

- *sapply(x, fun)* outputs a vector or matrix
  - x is list, vector, or data frame
- *tapply(x, index, fun)* executes the function on x grouped by the index factors.
  - x is usually a vector
  - index is a list containing factors

```
> data(mtcars)
> tapply(mtcars$mpg, mtcars$cyl, mean)
       4        6        8 
26.66364 19.74286 15.10000 
```

Example 3

There are 4 financial datasets: "FB-2", "GOOG", "^GSPC-2", "^IRX".

1. Write a loop to read in these datasets, use read_csv
    - Use a vector of strings for filenames.
    - If any data is missing for opening price, remove that observation.
    - Add on a variable with the name of the company
    - Concatenate all the datasets for the different companies.
2. Compute a PriceChange variable using opening and closing prices.
3. Compute the correlation in PriceChange between *FB-2* and the remaining companies.
    - The output should look like:

```
[1] "Correlation between daily changes in Facebook stocks and GOOG is 0.5."
[1] "Correlation between daily changes in Facebook stocks and ^GSPC-2 is 0.58."
[1] "Correlation between daily changes in Facebook stocks and ^IRX is -0.5."
```