Introduction to R

Session 03: Basics of programming

Álvaro Pérez[1]

Instituto Tecnológico Autónomo de México

Fall 2024

[1] Based on PhD Romero Londoño's notes.

For loops

- Basic idea: you have a vector of values, and you have an "iterator" variable
- You go through the vector, setting the iterator variable to each value one at a time
- Then you run a chunk of that code with the iterator variable set

```
for (iteratorvariable in vector) {
  code chunk
}
```

For loops

```r
data(mtcars)
abovemed <- mtcars %>% filter(cyl >= median(cyl))
belowmed <- mtcars %>% filter(cyl < median(cyl))
for (i in c('mpg','disp','hp','wt')) {
  print(mean(abovemed[[i]])-mean(belowmed[[i]]))
}
```

For loops

- Sometimes, you want R to do something only in some cases
- The basic syntax to do this is an "if" statement

```
if (test_expression) {
statement
}
```

- The "test_expression" usually depends on logical operators
  - '&' is AND
  - '|' is OR
  - To check equality use '==', not '='
  - '>=' is greater than OR equal to, similarly for '<='

For loops

- Sometimes is useful to “break” (stop) a loop, or skip an iteration
- Do this with “break” and “next”
- Often “break” and “next” are nested within an “if” statement

```
#printing odd numbers
m=20
for (k in 1:m){
  if (!k %% 2)
    next
    print(k)
}
```

Example: loop combined with if and break

```
#finding the first number divisible by 13 greater than
    100
for (k in 100:100000){
  if (!k %% 13){ #checks if number is odd
    break
  }
}
print(k)
```

Example: finding prime numbers

```
for(num in 1:100){
  # Program to check if the input number is prime or not
  flag = 0
  # prime numbers are greater than 1
  if(num > 1) {
    # check for factors
    flag = 1
    for(i in 2:(num-1)) {
      if ((num %% i) == 0) {
        flag = 0 #if number is divisible, then not prime
        break #and we can break the loop
      }
    }
  }
  if(num == 2)    flag = 1
  if(flag == 1) {
    print(paste(num,"is a prime number"))
  } else {
    print(paste(num,"is not a prime number"))
  }
}
```

Functions

- Functions are used when you have to repeat the same operation multiple times
- Avoids coding mistakes (and if there is one you only need to fix it once)
- Let's you break code into simpler parts which become easy to maintain and understand
- It's pretty straightforward to create your own function in R programming.

```r
func_name <- function (argument) {
statement
return(whatyouwant)
}
```

Example: finding prime numbers

```r
# Identifying prime number
prime_num<-function(num){
  # Program to check if the input number is prime or not
  flag = 0
  # prime numbers are greater than 1
  if(num > 1) {
    # check for factors
    flag = 1
    for(i in 2:(num-1)) {
      if ((num %% i) == 0) {
        flag = 0 #if number is divisible, then not prime
        break #and we can break the loop
      }
    }
  }
  if(num == 2)     flag = 1
  return(flag)
}
```

Example: finding prime numbers

```
prime_num(2)
prime_num(3)
prime_num(4)
prime_num(6131)
prime_num(4684561123)
```

Functions

- Functions can have multiple arguments.
- Functions can have default values for arguments.
- Functions can output vectors, list, or any other object.

Example: power function

```r
pow <- function(x, y = 2) {
  result1 <- x^y
  return(result1)
}
pow(3)
pow(3,3)
```

Some basic guiding principles

- Automate everything that can be automated.
- Design your workflow so that when you add new data or tweak something you don't need to manually recreate anything
- Avoid copying and pasting figures/tables from one software into another
  - e.g., impossible to automate: copying a graph from R into Word
- General workflow: raw data $\rightarrow$ clean data $\rightarrow$ analysis $\rightarrow$ (pretty) table or figure

Don't forget to look at the data

- Use `the R studio browser` or excel
- Scroll through the variables and accompany yourself with what you've got visually

Look for missing observations

- Check the size of your dataset using `dim`
- Check the structure of your data using `str`
- Check for missing variables
  - Sometimes the data has -99 for missing
  - Others is truly missing (i.e., NA)
  - Others is an empty string

Always check your merges (combining data sets)

- During a stage of arranging datasets, you will likely merge
- Make sure you count before and after you merge so you can figure out what went wrong, if anything
- Also make sure you understand what type of merge you are using/need
  - many to one
  - one to many
  - one to one
  - many to many (please no!!!)

Good practices

- Beyond checking the data, here are a few things that will prevent errors

1. Organized subdirectories
2. Automation
3. Naming conventions
4. Version control

Naming conventions

- Things to keep in mind when naming:
  1. variables,
  2. datasets
  3. scripts
- Avoid
  - meaningless words (e.g., `lmb2`)
  - dating (e.g., `temp05012020`), except to save old version in the archive
  - numbering (e.g., `outcome25`)
  - any of these will confuse your future self

Naming conventions for variables

- Variables should be readable to a stranger
  - Say that you want to create the product of two variables. Use an underscore and mash the two together
  - price_mpg <- price * mpg
- Name the variable exactly what it is when possible
  - bmi <- weight / (height*height * 703)

Headers

- Always start your programming scripts with a header
- The header is a big comment on the code and says:
  - The purpose of the script
  - The author(s)
  - The date the script was created
  - The last date in which it was updated/modified (and by whom)

Automating Tables and Figures (more on visualization below)

- Goal is to make "beautiful tables/figures" that are never edited post-production
- Tables/figures should be readable on their own (with the table/figure notes)
- Large fixed costs learning commands like "stargazer" in R or "estout" in R

Eliminate junk

- Maximize the data-to-ink ratio by using as little ink as possible to show your data
- Remove non data ink, e.g., extra gridlines
- Remove redundant data
- Remove indicators you don't need
- Display only as many decimal places as are relevant

Choose the type of visualization based on the information you want to convey

- Use a table only if you can't use a figure. Usually when
  - You need to show exact numerical values
  - You want to allow for multiple localized comparisons
  - You have relatively few numbers to show
- Avoid pie charts
  - It is difficult for the audience to visually distinguish the size of each wedge
  - A good alternative for showing shares of a whole is a stacked bar chart

Other random tips

- Make the font larger than the Stata/R preset
- Label your axis
- The Y-axis tickmarks ideally should be horizontal (R puts them sideways)
- Use legends if you have multiple lines
- Avoid having two-axis
- Avoid pie-charts

Example

- Add x-axis tick marks
- We only want people aged 5-18
- Make the y-axis in percent terms
- Use a more appealing color scheme
- The y-axis is misleading, lets make it go to 100
- Add a legend to explain what the colors mean
- Make the numbers in the y-axis easier to read and remove legend box