7316 - INTRODUCTION TO R

Module 3: Loops, conditions, functions, text, scrapping

Teacher: Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

This document was generated with R markdown.

1 LOOPS AND CONDITIONS

For tasks that you want to iterate over multiple elements, you may want to think about creating a **loop**. A loop performs a task multiple times, across either a list of objects, a set of index values (with `for`), or as long as a condition is not met (with `while`).

1.1 `for` loops

Syntax:

```
for (indexname in range) {
  do stuff
}
```

- Pay attention to the parenthesis and the braces. They are important.
- You do not need to break lines and indent your code, but this is a usual practice to make your code more readable.

1.1.1 *For loop across named elements*

You can loop over elements instead of values.

For example, you can change the type of a vector of elements in a `data.frame`.

```
# Loading some data
df_1 <- rio::import("data/nlsy97.rds")

# These are the factor variable in the data.frame
factor_vars <- c("personid", "year",
                 "sex", "race",
                 "region", "schooltype")

# For each name in the vector above, apply as.factor in the data.frame
for (i in factor_vars) {
  df_1[, i] <- as.factor(df_1[, i])
}
```

Note the use of `i` in the loop. The `i` is simply a way to name the element being currently manipulated by the loop. You could name it differently if you wish, for example in the case you have multiple loops working together (*i.e.* multiple elements manipulated at the same time).

You can also use the index `i` of the loop as a numeric value:

```
for (i in 1:4) {
  print(i^2)
}
## [1] 1
## [1] 4
## [1] 9
## [1] 16
```

- The `:` in `1:4` means "*all the integers between 1 and 4*". It would be the same as writing `c(1, 2, 3, 4)`.

1.1.2 *`for` loop and vector indexes*

It is often the case that you want to apply an operation to all the element of a vector. You can do it with a loop:

```
for (i in 1:length(df_1$birthyr)) {
  df_1$birthyr[i] <- df_1$birthyr[i] - 1900
}
```

This may be useful when combined with a `if` statement (see below).

1.2 `while` statements

`while` statement works the same way as `for` loops, and perform an operation as long as a condition is not met. Example

```
i <- 10

# While condition is not met
while (i > 0) {
  # do something
  print(i)

  # REMEMBER TO UPDATE THE CONDITION
  i <- i - 1
}
## [1] 10
## [1] 9
## [1] 8
## [1] 7
## [1] 6
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
```

Warning: If your `while` condition is never met, the loop will never stop! Until the end of the universe... Be careful!

1.3 `if` statements

`if` statements are also a useful part of programming, either in conjunction with iteration or separately. An if statement performs operations only if a specified condition is met.

- **Important:** `if` statements evaluate conditions of length one (*i.e.* non-vector arguments).

Syntax:

```
if (condition is TRUE) {
  do stuff
}
```

- In the for loop example, the loop was indexed over only the columns of indicator codes.
- Equally, the loop could be done over all columns with an if-statement to change only the indicator codes.

```
for (i in 1:length(df_1$birthyr)) {
  if (df_1$birthyr[i] > 1900) {
      df_1$birthyr[i] <- df_1$birthyr[i] - 1900
  }
}
```

1.3.1 Multiple conditions

You can encompass several conditions using the `else if` and catch-all `else` control statements.

```
df_1$age_range <- character(length = nrow(df_1))
for (i in 1:length(df_1$birthyr)) {
  if (df_1$age[i] < 20) {
      df_1$age_range[i] <- "below 20"
  } else if (df_1$age[i] > 30) {
      df_1$age_range[i] <- "above 30"
  } else {
    df_1$age_range[i] <- "20-30"
  }
}
```

Note:

- You **cannot** manipulate the element of a vector that does not exist with a `if` statement in a `data.frame`. You need first to create the vector (here, we create an empty `character()`) and then test your condition.
- `nrow()` returns the number of rows in a `data.frame`. This is like `length()` for vectors.

1.4 Vectorized if statements

As alluded to earlier, `if` statements evaluate *only* single-valued objects. Most of the time, you probably want to use conditional statements on vectors. For this, you can:

- combine your `if` statement with a loop, as seen above, or
- you can use the function `ifelse()`

Syntax:

```
ifelse(condition,
       todo if condition is TRUE,
       todo if condition is FALSE)
```

The statements returned can be simple values, but they can also be functions or even further conditions. You can easily nest multiple `ifelse` if desired.

Example:

```
numbers <- sample(1:30, 7)

df <- data.frame(numbers     = numbers,
                 even_or_odd = ifelse(numbers %% 2 == 0,
                                      "even",
                                      "odd"))

View(df)
```

Question: What if we tried a normal if statement instead ?

```
numbers <- sample(1:30, 7)

df <- data.frame(numbers     = numbers)

# Warning: with if statement, you need to initialize your vector in the datafr
ame BEFORE manipulating its elements
df$even_or_odd <- character(length = nrow(df))
```

```r
for (i in 1:nrow(df)) {
  if (df$numbers[i] %% 2 == 0) {
    df$even_or_odd[i] <- "even"
  } else {
    df$even_or_odd[i] <- "odd"
  }
}

View(df)
```

1.4.1 *Multiple vectorized if statements*

A better alternative to multiple nested `ifelse` statements is the tidyverse case_when function.

Syntax:

```r
dplyr::case_when(
  condition1 ~ todo1,
  condition2 ~ todo2,
  condition3 ~ todo3,
)
```

Example:

```r
df$range <- dplyr::case_when(
  (df$numbers > 0 & df$numbers <= 10) ~ "1-10",
  (df$numbers > 10 & df$numbers <= 20) ~ "10-20",
  (df$numbers > 20 & df$numbers <= 30) ~ "20-30"
)

View(df)
```

2 Functions

Functions are very useful to repeat a specific set of operations on multiple objects. If you find yourself performing the same specific steps more than a couple of times (perhaps with slight variations), then you should consider writing a function.

A function can serve essentially as a wrapper for a series of steps, where you define generalized inputs/arguments.
There are 4 ingredients in a function:

1. Function name: *the way to invoke the function*
2. Arguments: *the values the function takes as input*
3. Function body: *the steps to perform on the arguments*
4. Output: *(optional) the objects that the function returns*

Syntax:

```r
function_name <- function(arg1, arg2, ...){
  do stuff

  return(output)
}
```

Note: the `return()` is optional.

Example:
Let's turn the calculation of even or odd that was completed earlier into a function:

```
# Make odd function
odd <- function(x){
  output <- ifelse(x %% 2 == 0, "even", "odd")

  return(output)
}


# Or you could more simply write (same as above):
odd <- function(x){
  ifelse(x %% 2 == 0, "even", "odd")
}
```

- Note that `x` here is a descriptive placeholder name for the data object to be supplied as an argument for the function.
- You can then save the return of the function in an object, or print it in the console.

```
odd(numbers)
## [1] "even" "odd"  "odd"  "even" "even" "odd"  "even"
```

Warning: objects within the function body and outside are not the same, although they may have the same name, *i.e.* if you have an `x` variable in your environment, it does not have to correspond to the `x` that you used as an argument.

3 Merging data

In the first module, we learnt how to add a vector to a `data.frame`. By doing this, we assume that:

1. the vector is of the same size as the `data.frame`, and
2. the observations are in the same order in the vector and in the `data.frame`.

Let's assume is it not the case. For example, you have the two following `data.frame`, with a unique identifier and a variable, and you want to `merge` them:

```
df_1 <- data.frame(id    = c(1, 2, 3, 4, 5, 6, 7),
                   var_1 = c("a", "a", "b", "a", "d", "e", "e"))

df_2 <- data.frame(identifier = c(2, 12, 7, 8, 9, 10, 11, 3, 13),
                   var_2      = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE,
                                  FALSE, TRUE, FALSE))
```

You note that the two data.frame contain a unique identifier (named differently) and a variable each, but they do not have the same number of rows, and the unique identifier do not match line per line.

The starting point for any merge is to enumerate the column or columns that uniquely identify observations in the dataset. This is often the case to have unique identifiers in your data, such as respondent id, organization number, municipality, *etc.* For panel data, this will typically be both the personal/group identifier and a timing variable, for example Sweden in 2015 in a cross-country analysis.
Note that in the example above, the variable names of the unique identifier do not match accross data.frame. It does not matter: with the merge function, you can provide the name of the identifier for each data.frame. It also does not matter if it is the first column of the data.frame or not.

3.1 Using `merge()`

3.1.1 *Case 1: left join*

You left join when you want all the observations from the first dataset, but not the second (missing values will be filled with `NA`:

```
df_left <- merge(x = df_1, y = df_2,
                 by.x = "id", by.y = "identifier",
                 all.x = TRUE, all.y = FALSE)
```

- where `x` and `y` are the names of the `data.frames` to merge.
- `by.x` and `by.y` provide the names of the unique identifiers in the first and in the second `data.frame`, respectively.
- `all.x = TRUE` indicates that we want to keep ALL the observations from the first `data.frame` (even if there is no corresponding observations in the second `data.frame`)
- `all.y = FALSE` indicates that we do NOT want to keep the observations from the second `data.frame` that do not also appear in the first `data.frame`

3.1.2 *Case 2: right join*

It is the same, but keeping all observations from the second dataset:

```
df_right <- merge(x = df_1, y = df_2,
                  by.x = "id", by.y = "identifier",
                  all.x = FALSE, all.y = TRUE)
```

- Pay attention to the different number of observations in `df_left` and `df_right`.

3.1.3 *Case 3 and 4: all observations, or only observations in common*

- To keep all observations from any data.frame (most inclusive):

```
df_all <- merge(x = df_1, y = df_2,
                by.x = "id", by.y = "identifier",
                all.x = TRUE, all.y = TRUE)
```

- To keep only the common observations (most exclusive):

```
df_all <- merge(x = df_1, y = df_2,
                by.x = "id", by.y = "identifier",
                all.x = FALSE, all.y = FALSE)
```

3.2 Extract missing rows

Sometimes, you may want to keep only observations that do NOT appear in both dataset. This is called `anti_join()` in the tidyverse. You can do this with R base:

```
# df_1 observations that are NOT in df_2
df1_not_df2 <- df_1[!c(df_1$id %in% df_2$identifier),]

# df_2 observations that are NOT in df_1
df2_not_df1 <- df_2[!c(df_2$identifier %in% df_1$id),]
```

Reminder: `!` means NOT. The way to read the content of the brackets `!c(df_1$id %in% df_2$identifier)` means NOT `df_1$id` in `df_2$identifier`.

3.3 Appending data

Finally, instead of joining different datasets for the same individuals, sometimes you want to join together files that are for different individuals within the same dataset. When join data where the variables for each dataset are the same, but the observations are different, this is called *appending* data.

If you want to append `df2_not_df1` to `df1_not_df2`, you can use `plyr::rbind.fill()`. Make sure that the column in common have the same names. The missing columns will be filled with `NA`:

```r
# Make sure the same column have the same name
names(df2_not_df1)[names(df2_not_df1) == "identifier"] <- "id"

df <- plyr::rbind.fill(df1_not_df2, df2_not_df1)
```

4 MANIPULATING TEXT

4.1 Concatenating strings

The last type of data preparation that we will cover in this course is manipulating string data.

The simplest string manipulation may be concatenating (*i.e.* combining) strings. For this, you can use `paste()` and `paste0()`:

- `paste()` let you concatenate strings and separate them with a chosen character or a space.
- `paste0()` sticks the strings together without any intermediary character.

Example:

```r
# Creating string
string_1 <- "This is a text."
string_2 <- "This is another text."

# We paste with a space in the middle
string_3 <- paste(string_1, string_2)

# We paste with a semicolumn in the middle
string_3bis <- paste(string_1, string_2, sep = ";")

# We paste with nothing in the middle
string_4 <- paste0(string_1, string_2)
```

You can also add the content of numerical values in the string. For example:

```r
# Creating string
df <- data.frame(user = c("user 1", "user 2"),
                 age  = c(23, 24))

for (i in 1:nrow(df)) {
  print(
    paste0("The age of ", df$user[i], " is ", df$age[i], ".")
  )
}
## [1] "The age of user 1 is 23."
## [1] "The age of user 2 is 24."
```

4.2 Extracting and replacing parts of a string using `stringr`

Other common string manipulating tasks include extracting or replacing parts of a string. These are accomplished via the `str_extract()` and `stringr::str_replace()` from stringr package.

The arguments for each function are:

```
stringr::str_extract(string_object, "pattern_to_match")
stringr::str_replace(string_object, "pattern_to_match", "replacement_text")
```

By default, both function operate on the first match of the specified pattern. To operate on *all* matchs, add "_all" to the function name, as in:

```
stringr::str_extract_all(string_object, "pattern_to_match")
```

Example:

```
# Text copied from Wikipedia: https://en.wikipedia.org/wiki/Logic_gate
string_1 <- "This logic diagram of a full adder shows how logic gates can be
used in a digital circuit to add two binary inputs (i.e., two input bits), al
ong with a carry-input bit (typically the result of a previous addition), res
ulting in a final \"sum\" bit and a carry-output bit. This particular circuit
is implemented with two XOR gates, two AND gates and one OR gate, although eq
uivalent circuits may be composed of only NAND gates or certain combinations
of other gates."

stringr::str_extract(string_1, "logic")
## [1] "logic"
stringr::str_extract_all(string_1, "logic")
## [[1]]
## [1] "logic" "logic"
stringr::str_replace(string_1, "logic", "illogic")
## [1] "This illogic diagram of a full adder shows how logic gates can be use
d in a digital circuit to add two binary inputs (i.e., two input bits), along
with a carry-input bit (typically the result of a previous addition), resulti
ng in a final \"sum\" bit and a carry-output bit. This particular circuit is i
mplemented with two XOR gates, two AND gates and one OR gate, although equiva
lent circuits may be composed of only NAND gates or certain combinations of o
ther gates."
stringr::str_replace_all(string_1, "logic", "illogic")
## [1] "This illogic diagram of a full adder shows how illogic gates can be u
sed in a digital circuit to add two binary inputs (i.e., two input bits), alo
ng with a carry-input bit (typically the result of a previous addition), resu
lting in a final \"sum\" bit and a carry-output bit. This particular circuit i
s implemented with two XOR gates, two AND gates and one OR gate, although equ
ivalent circuits may be composed of only NAND gates or certain combinations o
f other gates."
```

4.3 `gsub` and `grepl`

Another convenient command, from base R, to replace content in a string is `gsub`. It works as `stringr::str_replace_all()`, but the argument are in a different order.

```
gsub(pattern = "Logic", replacement = "illogic",
     x = string_1, ignore.case = TRUE)
## [1] "This illogic diagram of a full adder shows how illogic gates can be u
sed in a digital circuit to add two binary inputs (i.e., two input bits), alo
ng with a carry-input bit (typically the result of a previous addition), resu
lting in a final \"sum\" bit and a carry-output bit. This particular circuit i
s implemented with two XOR gates, two AND gates and one OR gate, although equ
```

```
ivalent circuits may be composed of only NAND gates or certain combinations o
f other gates."
```

Note the `ignore.case = TRUE` argument: in this case, gsub will match both upper and lower case pattern. If you want to match *exactly* the pattern you wrote, you can use the argument `fixed = TRUE`:

```
gsub(pattern = "Logic", replacement = "illogic",
     x = string_1, fixed = TRUE)
## [1] "This logic diagram of a full adder shows how logic gates can be used
in a digital circuit to add two binary inputs (i.e., two input bits), along w
ith a carry-input bit (typically the result of a previous addition), resultin
g in a final \"sum\" bit and a carry-output bit. This particular circuit is im
plemented with two XOR gates, two AND gates and one OR gate, although equival
ent circuits may be composed of only NAND gates or certain combinations of ot
her gates."
```

`grepl` allows used to check wheter a pattern appear in a string: the output is `TRUE` or `FALSE`.

```
grepl(pattern = "Logic", x = string_1, fixed = TRUE)

grepl(pattern = "Logic", x = string_1, fixed = FALSE)

grepl(pattern = "Logic", x = string_1, ignore.case = TRUE)
```

4.4 Using *regular expressions* in patterns

Often we want to modify strings based on a pattern rather than an exact expression, as seen with examples above. Patterns are specified in R (as in many other languages) using a syntax known as **regular expressions** or **regex**. Today, we will very briefly introduce some regular expressions.

- To match "one of" several elements, refer to them in square brackets, *e.g.*: `"[abc]"`
- To match one of a range of values, use a hyphen to indicate the range: *e.g.* `"[A-Z]"`, `"[a-z]"`, `"[0-9]"`
- To match either of a couple of patterns/expressions, use the `|` operator, *e.g.*: `"2017|2018"`
- `"^text"` match text at the beginning of the string
- `"text$"` match text at the end of the string
- There are also abbreviation for one of specific types of characters *e.g.*: `[:digit:]` for numbers, `[:alpha:]` for letters, `[:punct:]` for punctuation, and `.` for every character.
- See the RStudio cheat sheet on stringr for more examples (and in general, as a brilliant reference to *regex*)

4.4.1 *How many times to match?*

Aside from specifiying the characters to match, such as `"[0-9]"`, another important component of regular expressions is how many time should the characters appear.

- `"[0-9]"` will match any part of a string composed of exactly *1* number.
- `"[0-9]+"` will match any part of a string composed of *1 or more* numbers.
- `"[0-9]{4}"` will match any part of a string composed of exactly *4* numbers.
- `"[0-9]*"` will match any part of a string composed of zero or more numbers.

Examples:

```
years <- c("This was in 1999", "This was in 2000.", "This was in 1850")

grepl("(19|20)[0-9]{2}$", years[1])
## [1] TRUE
grepl("(19|20)[0-9]{2}$", years[2])
## [1] FALSE
grepl("(19|20)[0-9]{2}$", years[3])
## [1] FALSE
```

4.4.2 *Escaping special characters*

Often, special characters can cause problems when working with strings. For example, trying to add a quote can result in R thinking you are trying to close the string. For most characters, you can "*escape*" (cause R to read as part of the string) special characters by prepending them with a backslash.

Example:

```
quote <- "\"Without data, you're just another person with an opinion.\"
- W. Edwards Deming."

writeLines(quote)
## "Without data, you're just another person with an opinion."
## - W. Edwards Deming.
```

4.5 Matching strings that precede or follow specific patterns

To match part of a string that occurs before or after a specific other pattern, you can also specify "lookarounds", the pattern the match should precede or follow:

To match a string pattern x, preceded or followed by y:

- **y precedes x:** `"(?<=y)x"`
- **y follows x:** `"x(?=y)"`

Example:

```
price_info <- ("The price is 5 dollars")

stringr::str_extract(price_info, "(?<=(The price is )).+")
## [1] "5 dollars"
stringr::str_extract(price_info, ".+(?=( dollars))")
## [1] "The price is 5"
stringr::str_extract(price_info, "(?<=(The price is )).+(?=( dollars))")
## [1] "5"
```

4.6 Trimming a string

When working with formatted text, a third common task is to remove extra spaces before or after the string text. This is done with the `stringr::str_trim()` function. The syntax is:

```
stringr::str_trim("an  extra space ")
## [1] "an  extra space"
```

Note, when printing a string, any formatting characters are shown. To view how the string looks formatted, use the `ViewLines()` function.

5 Web Scraping with `rvest`

"Scraping" data from the web - that is, automating the retrieval of data displayed online (other than through API) is an increasingly common data analysis task.

- Today, we will briefly explore very rudimentary web scraping, using the `rvest` package.
- The specific focus today is only on scraping data structured as a table on a webpage. The basic method highlighted will work much of the time, but does not work for every table.

5.1 Using rvest to scrape a table

- The starting point for scraping a web table with `rvest` is the `rvest::read_html()` function, where the URL to the page with data should go.
- After reading the webpage, the table should be parsed. For many tables, the `rvest::read_html` can be piped directly into the `rvest::html_table()` function.
  - If this works, the data should then be converted from a list into a dataframe/tibble.
- If `rvest::html_table()` does not work, a more robust option is to first pipe `rvest::read_html` into `rvest::html_nodes(xpath = "//table")` and then into `rvest::html_table(fill = TRUE)`
  - `rvest::html_nodes(xpath = "//table")` looks for all HTML objects coded as a table, hence

Example:

```r
tech_stock_names <- c("MSFT", "AMZN", "GOOGL",
                      "AAPL", "FB", "INTC", "CSCO")

scrape_yahoo <- function(x) {
  url <- paste0("https://finance.yahoo.com/quote/",
                x,
                "/history")
  html_page <- rvest::html_table(rvest::read_html(url))
  stock_table <- as.data.frame(html_page)
  stock_table$stock <- x

  return(stock_table)
}

tech_stocks <- scrape_yahoo(tech_stock_names[1])

for (j in 2:length(tech_stock_names)) {
  tmp <- scrape_yahoo(tech_stock_names[j])

  tech_stocks <- plyr::rbind.fill(tech_stocks, tmp)
}
```

WARNING: Scraping is often tolerated, but sending many request to the same server will likely lead you to be banned. If you scrape data, do not overload server! Make sure you collect data that you are allowed to collect!

6 TO DO BEFORE THE NEXT CLASS

1. Go through this material again and try other cases: run all the codes for yourself, change the values, change the parameters, see what happens.
2. If you haven't finished during the class, finish the practice assignment of Module 3.

Next time, we will move on, and consider the knowledge provided here is mastered.