HOUSE OF INNOVATION

MODULE 5: PLAYING WITH MORE ADVANCED TOOLS

7316 - INTRODUCTION TO DATA ANALYSIS WITH R

Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

When you do programming with R, it often happens that you want to repeat an operation on multiple objects. To do just this, you can use **loops** and **functions**. Those are massively used tools in any project where you want to use multiple times the same code (you should never need to copy-paste code, rather use functions). Loops and functions are very powerful, but a great power implies great responsibilities… We will see later in this module the limits of these tools and how to avoid the traps of falling in infinite loops or in memory overload.

1. Loops

A loop performs a task multiple times across 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.
  - The list of elements on which to do stuff is written in parentheses after the `for`.
  - The stuff to do are written between braces `{ }`.
- 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` loops 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/7316_module_5_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` loops and vector indexes*

- It is often the case that you want to apply an operation to all the element of a vector. This may be useful when combined with a `if` statement (see below about `if` statements). You can do it with a loop:

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

- You can also do it on all the rows of a `data.frame`:

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

- `nrow()` returns the number of rows in a `data.frame`. This is like `length()` for vectors.

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.

1.4 A general advice: DON'T USE LOOPS! EVER!!!

- Loops are evil! They use a lot of time and resources! They are a **BAD** practice!! Probably every time you use a loop, there is a penguin dying in Antarctica.
  - Maybe I sound too dramatic, but loops in R and really poorly designed and you can easily get your computer stuck, or have memory overload if you do not use them wisely.
  - If you have one million observations, it might take hours to run your loop. This does not work. R is designed to manipulate vectors. As much as you can, manipulate vectors, not loops (we have seen this in module 2).

Example:

- I create a big data.frame, with 100000 observations. I want a third variable, `var_3`, in `df`, that is the product of `var_1` and `var_2`.

```
df <- data.frame(var_1 = rnorm(100000, 1, 5),
                 var_2 = rnorm(100000, 10, 15))
```

- With a loop:

```
df$var_3 <- numeric(length = nrow(df))
starttime <- Sys.time()
```

```r
for (i in 1:nrow(df)) {
  df$var_3[i] <- df$var_1[i] * df$var_2[i]
}
endtime <- Sys.time()

# Time to run
endtime - starttime
```

- The code above would take about **5 mins** to run.
- Now, with a vector manipulation:

```r
starttime <- Sys.time()
df$var_3 <- df$var_1 * df$var_2
endtime <- Sys.time()

# Time to run
endtime - starttime
```

```
Time difference of 0.001748085 secs
```

- The code above takes less than a second to run. **Both codes lead to the exact same results!**

More complex example:

- You might think, what if I need to check each elements one by one? For example, if I want `var_3` to be the product of `var_1` and `var_2` only if `var_1` is above 5, and equals `var_1` otherwise.
- Solution: proceed by steps, and **use vectors**

```r
starttime <- Sys.time()

tmp <- df[df$var_1 >= 5,]
tmp2 <- df[df$var_1 < 5,]

tmp$var_3 <- tmp$var_1 * tmp$var_2
tmp2$var_3 <- tmp2$var_1

df <- rbind(tmp, tmp2)

endtime <- Sys.time()

# Time to run
endtime - starttime
```

```
Time difference of 0.0226841 secs
```

- Here, the time to run is, again, less than a second. If you would use a loop, with a if condition, that would likely take 15 minutes to run.

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:

```
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.

```
numbers <- c(2, 3, 5, 8, 10, 11)

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

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. Manipulating text

3.1 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."
```

3.2 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:

```r
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* matches, add "_all" to the function name, as in:

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

Example:

```r
# 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), along with a
carry-input bit (typically the result of a previous addition), resulting in a fina
l \"sum\" bit and a carry-output bit. This particular circuit is implemented with
two XOR gates, two AND gates and one OR gate, although equivalent circuits may be
composed of only NAND gates or certain combinations of other gates."

stringr::str_extract(string_1, "logic")
```

```
[1] "logic"
```

```r
stringr::str_extract_all(string_1, "logic")
```

```
[[1]]
[1] "logic" "logic"
```

```r
stringr::str_replace(string_1, "logic", "illogic")
```

```
[1] "This illogic diagram of a full adder shows how logic gates can be used in a d
igital circuit to add two binary inputs (i.e., two input bits), along with a carry
-input bit (typically the result of a previous addition), resulting in a final \"s
um\" bit and a carry-output bit. This particular circuit is implemented with two X
OR gates, two AND gates and one OR gate, although equivalent circuits may be compo
sed of only NAND gates or certain combinations of other gates."
```

```r
stringr::str_replace_all(string_1, "logic", "illogic")
```

```
[1] "This illogic diagram of a full adder shows how illogic gates can be used in a
digital circuit to add two binary inputs (i.e., two input bits), along with a carr
y-input bit (typically the result of a previous addition), resulting 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 equivalent circuits may be comp
osed of only NAND gates or certain combinations of other gates."
```

3.3 `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.

```r
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 used in a
digital circuit to add two binary inputs (i.e., two input bits), along with a carr
y-input bit (typically the result of a previous addition), resulting 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 equivalent circuits may be comp
osed of only NAND gates or certain combinations of 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`:

```r
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 dig
ital circuit to add two binary inputs (i.e., two input bits), along with a carry-i
nput bit (typically the result of a previous addition), resulting 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 equivalent circuits may be composed
of only NAND gates or certain combinations of other gates."
```

`grepl` allows used to check whether 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)
```

3.4 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*)

3.4.1 How many times to match?

Aside from specifying 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
```

3.4.2 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.
```

3.5 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"
```

3.6 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.

4. 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.

4.1 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`.
- 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
# Provide the URL
url <- "https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_capit
a"

# Fetch the page
page <- rvest::read_html(url)

# Extract the tables into a list
df <- rvest::html_table(page)

# Select the table of interest
df <- df[[2]]
```

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!

4.1.1 Does `rvest` covers everything?

- rvest is a tool to scrape static page, i.e. pages where content does not vary asynchronously. If you want to harvest asynchronous content (for example, data from webapps), you will need more advanced tools, such as rselenium, that is able to scrape content as we humans would see it on the screen. We will not cover rselenium in this course, because it does not belong to the introductory tools, but you can find more information on their page: https://github.com/ropensci/RSelenium
- Many websites that are open to data collection from user offers a tool called API. If an API is provided, use the **API**, it will generally offers a much more convenient access to data that any other mean.
  - This is the case for reddit, twitter, indiegogo, stackexchange, and many other big platforms.

- Each API is unique and has its own documentation. Usually, data in API come in a nicely formatted `json` files. `json` files are a nested data file format that you can read with R (using `rio`). Here, you can find an example of `json` file containing, as an example, twitter data: https://gist.github.com/hrp/900964.
- some API have a dedicated R package to facilitate data collection. This is the case of twitter, and the package academictwitteR.
- Often, before being able to get data through the API, you need to request access from the platform. Then platform may then grants you rights to access certain data and possibly the documentation to access it.