Biostatistics II: Introduction to R

Control Flow and Functions

Nicole S. Erler
Department of Biostatistics, Erasmus Medical Center
n.erler@erasmusmc.nl
@N_Erler

Avoiding Repetition

Sometimes we want to perform a particular action/manipulation multiple times and/or on several objects.

For example,

- plot the standardized residuals against each of the covariates,
- extract certain elements from a fitted model and create a results table,
- calculate a particular transformation of multiple variables.

Avoiding Repetition

Sometimes we want to perform a particular action/manipulation multiple times and/or on several objects.

For example,

- plot the standardized residuals against each of the covariates,
- extract certain elements from a fitted model and create a results table,
- calculate a particular transformation of multiple variables.

Option 1:

Copy & Paste

- a lot of work
- susceptible to mistakes

Option 2:

Functions and/or **Loops**

Iteration: Loops

To repeat the same action

- **`for`** each element of a `vector`, a `list`, ..., or
- **`while`** a particular condition is fulfilled

we can specify a **loop**.

The `for`-loop

The **`for`-loop** evaluates an expression **for each element** in a sequence.

It has the structure:

```
for (var in seq) {
  expr
}
```

- **`var`**: the name of a variable that acts as the index
- **`seq`**: the sequence of values `var` takes
- **`expr`**: an "expression", i.e., a piece of R syntax

The `for`-loop

The **`for`-loop** evaluates an expression **for each element** in a sequence.

It has the structure:

```
for (var in seq) {
  expr
}
```

- **`var`**: the name of a variable that acts as the index
- **`seq`**: the sequence of values `var` takes
- **`expr`**: an "expression", i.e., a piece of R syntax

For example:

```
for (i in 1:5) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

The `for`-loop

The **`for`-loop** evaluates an expression **for each element** in a sequence.

It has the structure:

```
for (var in seq) {
  expr
}
```

- **`var`**: the name of a variable that acts as the index
- **`seq`**: the sequence of values `var` takes
- **`expr`**: an "expression", i.e., a piece of R syntax

For example:

```
for (i in 1:5) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
for (i in 1:5) {
  print('test')
}
```

```
## [1] "test"
## [1] "test"
## [1] "test"
## [1] "test"
## [1] "test"
```

The `for`-loop

- The **index** variable **`var`** can be any character string.
- The **sequence `seq`** can be any sequence of numeric values, strings, objects, ...

For example, we could have loops with

- `for (index in c(1, 5, 9, 3))`
- `for (month in c("January", "February", "March", "April"))`
- `for (variable_name in seq_along(names(mydata)))`
- `for (elmt in fitted_model)`
  where `fitted_model` is a fitted model object

The `for`-loop

Loops return the value `NULL`:

```
for (var in names(nhanes)) {
  is.numeric(nhanes[[var]])
}
```

The `for`-loop

Loops return the value `NULL`:

```
for (var in names(nhanes)) {
  is.numeric(nhanes[[var]])
}
```

`print()` will send the output to the console (but only if it is last in the expression):

```
for (var in names(nhanes)) {
  print(is.numeric(nhanes[[var]]))
}
```

```
## [1] TRUE
## [1] FALSE
## [1] TRUE
## [1] FALSE
## [1] TRUE
## [1] FALSE
## [...]
```

The `for`-loop

We can save the output in a pre-specified object:

```
variable_numeric <- logical(ncol(nhanes))
names(variable_numeric) <- names(nhanes)
```

```
variable_numeric
```

```
##      SBP   gender      age     race       WC      alc     educ    creat
##    FALSE    FALSE    FALSE    FALSE    FALSE    FALSE    FALSE    FALSE  [...]
```

The `for`-loop

We can save the output in a pre-specified object:

```
variable_numeric <- logical(ncol(nhanes))
names(variable_numeric) <- names(nhanes)
```

```
variable_numeric
```

```
##      SBP   gender      age     race       WC      alc     educ    creat
##    FALSE    FALSE    FALSE    FALSE    FALSE    FALSE    FALSE    FALSE  [...]
```

```
for (var in names(nhanes)) {
  variable_numeric[var] <- is.numeric(nhanes[[var]])
}
```

```
variable_numeric
```

```
##      SBP   gender      age     race       WC      alc     educ    creat
##     TRUE    FALSE     TRUE    FALSE     TRUE    FALSE    FALSE     TRUE  [...]
```

The `while`-loop

The **`while`-loop** repeatedly evaluates an expression **while a condition is fulfilled**:

```
while (cond) {
  expr
}
```

- **`cond`**: a **length-one** logical vector that is not `NA`.
- **`expr`**: an "expression", i.e., a piece of R syntax

The `while`-loop

The **`while`-loop** repeatedly evaluates an expression **while a condition is fulfilled**:

```
while (cond) {
  expr
}
```

- **`cond`**: a **length-one** logical vector that is not `NA`.
- **`expr`**: an "expression", i.e., a piece of R syntax

Careful!

If your condition is never `FALSE` this will run forever!!! (or until you stop it manually)

The `while`-loop

The **`while`-loop** repeatedly evaluates an expression **while a condition is fulfilled**:

```
while (cond) {
  expr
}
```

- **`cond`**: a **length-one** logical vector that is not `NA`.
- **`expr`**: an "expression", i.e., a piece of R syntax

Careful!

If your condition is never `FALSE` this will run forever!!! (or until you stop it manually)

Runs forever:

```
i <- 0
while(i < 5) {
  print(i)
}
```

Stops after 5 iterations:

```
i <- 0
while(i < 5) {
  i <- i + 1
  print(i)
}
```

The `while`-loop: Example

Example: Use a `while`-loop to perform a power calculation.

Set-up starting values:

```
N <- 100  ## sample size
power <- power.t.test(n = N, delta = 0.2, sd = 1, sig.level = 0.05)$power
```

The `while`-loop: Example

Example: Use a `while`-loop to perform a power calculation.

Set-up starting values:

```
N <- 100  ## sample size
power <- power.t.test(n = N, delta = 0.2, sd = 1, sig.level = 0.05)$power
```

While the power is less than 0.8, increase `N` by 10 and re-calculate:

```
while (power < 0.8) {
  N <- N + 10
  power <- power.t.test(n = N, delta = 0.2, sd = 1, sig.level = 0.05)$power
}
```

The `while`-loop: Example

Example: Use a `while`-loop to perform a power calculation.

Set-up starting values:

```
N <- 100  ## sample size
power <- power.t.test(n = N, delta = 0.2, sd = 1, sig.level = 0.05)$power
```

While the power is less than 0.8, increase `N` by 10 and re-calculate:

```
while (power < 0.8) {
  N <- N + 10
  power <- power.t.test(n = N, delta = 0.2, sd = 1, sig.level = 0.05)$power
}
```

The resulting sample size and power:

```
N
```

```
## [1] 400
```

```
power
```

```
## [1] 0.8065
```

Conditional Evaluation: `if()`

Sometimes, we may want to execute code only **`if`** a certain condition is fulfilled.

```
if (cond) {
  expr
}
```

- **`cond`**: a **length-one** logical vector that is not **`NA`**.
- **`expr`**: an "expression", i.e., a piece of R syntax

If the condition is **not fulfilled,** **`NULL`** is returned.

Conditional Evaluation: `if()`

Sometimes, we may want to execute code only **`if`** a certain condition is fulfilled.

```
if (cond) {
  expr
}
```

- **`cond`**: a **length-one** logical vector that is not **`NA`**.
- **`expr`**: an "expression", i.e., a piece of R syntax

If the condition is **not fulfilled, `NULL`** is returned.

Example:
Calculate the mean of a variable in a dataset only if the variable is numeric:

```
if (is.numeric(nhanes$SBP)) {
  mean(nhanes$SBP, na.rm = TRUE)
}
```

```
## [1] 119.3
```

```
if (is.numeric(nhanes$gender)) {
  mean(nhanes$gender, na.rm = TRUE)
}
```

Conditional Evaluation: `if()`

The example makes more sense in combination with a `for`-loop:

```
nhanes_means <- list()

for (var in names(nhanes)) {

  nhanes_means[var] <- if (is.numeric(nhanes[[var]])) {

    mean(nhanes[[var]], na.rm = TRUE)

  }

}
```

```
unlist(nhanes_means)
```

```
##       SBP       age        WC     creat      albu uricacid      bili
## 119.2957   43.5108   94.9989    0.8438    4.3455    5.3528    0.7208
```

Conditional Evaluation: `if() ... else`

We can also specify an expression that is evaluated **if the condition is not fulfilled**:

```
if (cond) {
  cons.expr
} else {
  alt.expr
}
```

Conditional Evaluation: `if() ... else`

We can also specify an expression that is evaluated **if the condition is not fulfilled**:

```
if (cond) {
  cons.expr
} else {
  alt.expr
}
```

For example:

```
if (is.numeric(nhanes[[var]])) {

  mean(nhanes[[var]], na.rm = TRUE)

} else {

  unique(nhanes[[var]])

}
```

Conditional Evaluation: `if() ... else`

```
nhanes_summary <- list()

for (var in names(nhanes)) {

  nhanes_summary[[var]] <- if (is.numeric(nhanes[[var]])) {

    mean(nhanes[[var]], na.rm = TRUE)

  } else {

    unique(nhanes[[var]])
  }
}
```

```
nhanes_summary
```

```
## $SBP
## [1] 119.3
##
## $gender
## [1] male   female
## Levels: male female
## [...]
```

Conditional Element Selection: `ifelse()`

A similar function is **`ifelse()`**, which performs **conditional element selection**:

```
ifelse(test, yes, no)
```

- **`test`**: a logical vector
- **`yes`**: return values for `TRUE` elements of `test`
- **`no`**: return values for `FALSE` elements of `test`

Conditional Element Selection: `ifelse()`

A similar function is **`ifelse()`**, which performs **conditional element selection**:

```
ifelse(test, yes, no)
```

- **`test`**: a logical vector
- **`yes`**: return values for `TRUE` elements of `test`
- **`no`**: return values for `FALSE` elements of `test`

For example:

```
ifelse(nhanes$gender == "male", nhanes$WC, nhanes$WC * 1.1)
```

```
##   [1]  99.00  82.70 104.39  82.40  93.10 105.40  88.00  99.11 133.60 105.82
##  [11] 100.60 105.82  90.20  93.00  94.60 124.00  91.00 112.00 102.30 110.00 [...]
```

Note:

- `cond` in `if() ... else` is **a single** `TRUE` or `FALSE`
- `test` in `ifelse()` is a **vector** of `TRUE` and `FALSE` values

Conditional Element Selection: `ifelse()`

The arguments `yes` and `no` can be `vectors` or `lists`:

```
ifelse(test = c(TRUE, FALSE, FALSE),
       yes = c("ABC", "DEF", "GHI"),
       no = list("abc", "def", c("g", "h", "i")))
```

```
## [[1]]
## [1] "ABC"
##
## [[2]]
## [1] "def"
##
## [[3]]
## [1] "g" "h" "i"
```

Conditional Element Selection: `ifelse()`

The arguments `yes` and `no` can be `vectors` or `lists`:

```
ifelse(test = c(TRUE, FALSE, FALSE),
       yes = c("ABC", "DEF", "GHI"),
       no = list("abc", "def", c("g", "h", "i")))
```

```
## [[1]]
## [1] "ABC"
##
## [[2]]
## [1] "def"
##
## [[3]]
## [1] "g" "h" "i"
```

If `yes` and/or `no` are shorter than `test`, they are recycled:

```
ifelse(test = c(TRUE, FALSE, FALSE, TRUE, FALSE, TRUE),
       yes = "yes",
       no = c("1st", "2nd", "3rd"))
```

```
## [1] "yes" "2nd" "3rd" "yes" "2nd" "yes"
```

Functions

What are functions?

- a group of (organized) R commands
- a (small) program with flexible (= not pre-specified) input

Functions

What are functions?

- a group of (organized) R commands
- a (small) program with flexible (= not pre-specified) input

Almost all commands in R are functions!

Some examples:

- `mean()`
- `sum()`
- `plot()`
- …

```
class(mean)
## [1] "function"
class(sum)
## [1] "function"
class(plot)
## [1] "function"
```

Writing Functions

To write your own function:

```
function(arglist) {
  expr
}
```

- **`arglist`**: one or more names of arguments (optional)
- **`expr`**: an "expression", i.e., a piece of R syntax

Writing Functions

To write your own function:

```
function(arglist) {
  expr
}
```

- **`arglist`**: one or more names of arguments (optional)
- **`expr`**: an "expression", i.e., a piece of R syntax

For example:

```
square <- function(x) {
  x^2
}
```

```
square(3)
```

```
## [1] 9
```

Writing Functions: Arguments

Functions do not always need an argument:

```
random <- function() {
  rnorm(n = 1)
}
```

```
random()
## [1] 0.4471
```

Writing Functions: Arguments

Functions do not always need an argument:

```
random <- function() {
  rnorm(n = 1)
}
```

```
random()
## [1] 0.4471
```

Functions can have **multiple arguments**:

```
difference <- function(x, y) {
  x - y
}
```

```
difference(x = 5.2, y = 3.3)
```

```
## [1] 1.9
```

Writing Functions: Arguments

Multiple arguments are interpreted in the **pre-defined order**, unless they are named:

```
difference(5.2, 1.2)
```

```
## [1] 4
```

is equivalent to

```
difference(x = 5.2, y = 1.2)
```

```
## [1] 4
```

Writing Functions: Arguments

Multiple arguments are interpreted in the **pre-defined order**, unless they are named:

```
difference(5.2, 1.2)
```

```
## [1] 4
```

is equivalent to

```
difference(x = 5.2, y = 1.2)
```

```
## [1] 4
```

But this is different:

```
difference(y = 5.2, x = 1.2)
```

```
## [1] -4
```

Writing Functions: Arguments

We can also define **default values** for arguments.

```
multiply <- function(x, y = 2) {
  x * y
}
```

The default value is used when the user does not specify a value for that argument:

```
multiply(x = 3, y = 3)
```

```
## [1] 9
```

```
multiply(x = 3)
```

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

Writing Functions: Example

```
summarize_data <- function(dataset) {

  smry_list <- list()

  for (var in names(dataset)) {

    smry_list[[var]] <- if (is.numeric(dataset[[var]])) {

      mean(dataset[[var]], na.rm = TRUE)

    } else {

      unique(dataset[[var]])
    }
  }

  # return the result
  smry_list
}
```

Writing Functions: Example

```
summarize_data(nhanes)
```

```
## $SBP
## [1] 119.3
##
## $gender
## [1] male   female
## Levels: male female
##
## $age
## [1] 43.51
##
## $race
## [1] Non-Hispanic Black Other Hispanic
## [3] Non-Hispanic White other
## [5] Mexican American
## 5 Levels: Mexican American ... other
## [...]
```

```
summarize_data(airquality)
```

```
## $Ozone
## [1] 42.13
##
## $Solar.R
## [1] 185.9
##
## $Wind
## [1] 9.958
##
## $Temp
## [1] 77.88
##
## $Month
## [1] 6.993
##
## [...]
```