7 Control flow

(AST230) R for Data Science

Md Rasel Biswas

Control flow

- In R, there are two primary tools of control flow: **choices** and **loops**
- Choices allow you to run different code depending on the input
  - e.g. `if` statements and `switch`
- Loops allow you to repeatedly run code, typically with changing options
  - e.g. `for` and `while`

Choices

- The basic form of an `if` statement in R as follows

```
if (condition) true_action
if (condition) true_action else false_action
```

- `true_action` corresponds to the action for which the `condition` is `true`
- `false_action` corresponds to the action for which the `condition` is `false`

Choices

```
x <- sample(1:10, 1)
if (x >= 5) {
  "x is greater than or equals 5"
} else {
  "x is smaller than 5"
}
```

```
[1] "x is greater than or equals 5"
```

When R evaluates the condition inside `if` statements, it is looking for a logical element, i.e., `TRUE` or `FALSE`

```
x  <-  4 == 3
if (x) {
  "Nothing will print as condition is FALSE"
}
```

```
if (TRUE) {"Understand?"}
```

```
[1] "Understand?"
```

Choices

- Write a function that takes the raw score obtained in an exam and returns the grade corresponding to the number.

$$y = \begin{cases} A & \text{if } x >= 90 \\ B & \text{if } 80 \leq x < 90 \\ C & \text{if } 60 \leq x < 80 \\ F & \text{if } x < 60 \end{cases}$$

Choices

```
grade <- function(x) {
  if (x >= 90) y = "A"
  else if (x >= 80) y = "B"
  else if (x >= 60) y = "C"
  else y = "D"
  return(y)
}
```

```
grade(77)
```

```
[1] "C"
```

```
grade(95)
```

```
[1] "A"
```

Choices

- The `dplyr` package has a function `case_when()` to categorize a variable

```
library(dplyr)
grade_c <- function(x) {
  y <- case_when(x >= 90 ~ "A",
                 x >= 80 | x < 90 ~ "B",
                 x >= 60 | x < 80 ~ "C",
                 x < 60 ~ "D")
  return(y)
}
```

```
grade_c(77)
```

```
[1] "B"
```

```
grade_c(95)
```

```
[1] "A"
```

Choices

- `if()` function is used for single values.
- To sort a list of values into two categories, you can use the base R function `ifelse()` or the `dplyr` package's `if_else()` function
- The syntax of `if_else()`

```
if_else(condition, true_action, false_action)
```

```
x <- 1:10
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
dplyr::if_else(x %% 2 == 0, "even", "odd")
```

```
 [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

Loops

- `for` loops are used to iterate over items in a vector

```
for (item in vector) {
  perform_action
}
```

```
for (i in 1:3) {
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
```

```
v <- c("this", "is", "a", "for", "loop")
for(i in v){
  print(i)
}
```

```
[1] "this"
[1] "is"
[1] "a"
[1] "for"
[1] "loop"
```

Loops

- The `next` statement skips the current iteration of the loop and starts the loop from the next iteration
- The `break` statement terminate the execution of the loop.

```
for (i in 1:10) {
  if (i < 3) next
  if (i >= 5) break
  print(i)
}
```

```
[1] 3
[1] 4
```

Loops

- Sometimes you will find yourself needing to repeat an operation until a certain condition is met, rather than doing it for a specific number of times.

```
z <- 1
while (z < 4) {
  z <- z + 1
  print(z)
}
```

```
[1] 2
[1] 3
[1] 4
```

Base functionals

> A functional is a function that takes a function as an input and returns a vector as output.

- `apply()` function takes a matrix or array as an input and return a lower dimensional summary

```
apply(X, MARGIN, FUN)
```

- `X` → a matrix or an array
- `MARGIN` → a vector giving the subscripts which the function will be applied over, e.g., for a matrix, 1 indicates a row and 2 indicates a column
- `FUN` → the function to be applied on each MARGIN
- A data frame should not be used as an input in `apply()`

Base functionals

```
mat <- matrix(1:15, nrow = 3)
mat
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

- Row sum using loops

```
rsum <- rep(NA, nrow(mat))
for (i in 1:nrow(mat)) {
  rsum[i] <- sum(mat[i, ])
}
rsum
```

```
[1] 35 40 45
```

- Row sum using `apply()`

```
apply(mat, 1, sum)
```

```
[1] 35 40 45
```

- Similarly, column sum

```
apply(mat, 2, sum)
```

```
[1]  6 15 24 33 42
```

Base functionals

- `lapply()` returns a list after applying a function on each element of an R object

```
lapply(X, FUN, ...)
```

- `sapply()` is similar to `lapply()` but it returns a atomic vector instead of a list

Base functionals

```
lres <- list(x = 1:3, y = matrix(1:4, 2))
lres
```

```
$x
[1] 1 2 3

$y
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
lapply(lres, sum)
```

```
$x
[1] 6

$y
[1] 10
```

```
sapply(lres, sum)
```

```
 x  y
 6 10
```