Functional Programming I: Control Flow

UNIVERSITY OF
SAN FRANCISCO

James D. Wilson

BSDS 100 - Intro to Data Science with R

Functional Programming

The focus of the next few classes will be to turn your existing, informal knowledge of functions into a rigorous understanding of what functions are and how to write efficient code.

Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste.

Outline

- Control Flow: if, while, for, repeat, and switch
- Functions
- Writing Robust Code
- Functionals
- Efficiency of Code

Reference: Chapters 19 and 21 in *R for Data Science* book

Control Flow

We begin talking about functional programming by first exploring *control flow*, which enables execution of statements repetitively, while only executing other statements if certain conditions are met. Control flow can be subdivided into two main topics:

- Repetition and Looping
  - `for()` loops
  - `while()` loops
  - `repeat()` loops
- Conditional Execution
  - `if()`
  - `if() {} else if() {} else {}`
  - `ifelse()`
  - `switch()`

The `for()` loop

- Executes a statement repetitively until a variable's value is no longer contained in the sequence `seq`
- The generic in-line syntax is

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

- A simple example which prints "BSDS 100" 5 times

```
for (i in 1:5) print("BSDS 100")
```

The for() loop

- It is possible to iterate over more complex sequences

```
> myVector <- factor(c("A", "A", "B", "C", "C", "C", "ZzZ"))

> for (k in levels(myVector)) print(k)
[1] "A"
[1] "B"
[1] "C"
[1] "ZzZ"
```

The `for()` loop

- The `seq` in a `for()` loop is evaluated at the start of the loop; changing it subsequently does not affect the loop
- You can make an assignment to the looping variable (e.g., *i*) within the body of the loop, but this will not affect the next iteration
- When the loop terminates, the looping variable contains its latest value

The `while()` loop

- Executes a statement repetitively until a condition is no longer true
- The generic in-line syntax is

```
while (condition) {expression}
```

- A simple example which prints “MSAN” 3 times

```
> n <- 3

> while (n > 0) {print("MSAN"); n <- n - 1}
```

- **Note**: the use of the semi-colon is only required when writing more than one in-line expression. When multiple lines are used, the semi-colon can be omitted.

The `repeat()` loop

- The `repeat()` loop can be used when the terminal condition does not apply at the top of the loop
- A `repeat()` loop must be terminated with a `break` command placed somewhere inside `repeat()` loop
- The `break` command immediately exists the innermost active `for()`, `while()` or `repeat()` loop

The `repeat()` loop

- Example

```
> x <- 7

> repeat{
+   print(x)
+   x <- x + 2
+   if (x > 10) break
+ }
[1] 7
[1] 9
```

The `if()` statement

- The `if()` control structure executes a statement if a given condition is true
- The generic in-line syntax is

```
if (condition) {expression}
```

- A simple example

```
> x <- 3

> if (x > 0) print(paste("x is: ", x, sep = ""))
[1] "x is: 3"
```

The `if()` statement

- The multi-line form for `if()` is

```
if (condition) {
    < expression 1 >
    ...
    < expression n >
}
```

The `if() {} else {}` statement

- The `if() {} else {}` control structure executes a statement if a given `condition` is true
- The generic in-line syntax is

  ```
  if (condition) expression_01 else expression_02
  ```

- A simple example

  ```
  > x <- -3

  > if (x > 0) print("x is positive") else print("x is negative")
  [1] "x is negative"
  ```

The `if() {} else {}` statement

- The multi-line form of `if() {} else {}` is

```
    if (condition) {
        < expressions >
     } else {
        < alternate expressions >
     }
```

The above will run `expressions` if the `condition` is true, but will run `alternate expressions` if the `condition` is false.

Formatting Pitfalls

- This code snippet will run without error

```
  x <- -3

  if (x > 0) {
    print(paste("x is: ", x, sep = ""))
    } else {
      print("x is negative")
    }
[1] "x is negative
```

Formatting Pitfalls

- The code snippet will throw an error

```
  x <- -3

  if (x > 0) {
    print(paste("x is: ", x, sep = ""))
    }
  else {
      print("x is negative")
    }
Error: unexpected ’}’ in "  }"
```

if() {} else if() {} else {}

- The multi-line form of `if() {} else if() {} else {}` is

```
if (condition_01) {
    < expressions 01 >
 } else if (condition_02) {
    < expressions 02 >
 } else {
    < expressions 03 >
```

- As many `else if () {}` clauses may be chained (sequenced) together as desired

`ifelse()` versus `if() {} else {}`

- If a vector $\mathbf{x} : |\mathbf{x}| > 1$ is passed to an `if()` statement, only the first element of the vector will be evaluated for conditional execution; moreover, `R` will throw a warning
- The `ifelse()` construct is a vectorized version of `if() {} else {}` which tests each element of a vector passed to it

ifelse() versus if() {} else {}

```
> x <- c(3, 2, 1)

> if ( x > 2) {print("first element in vector > 2")}
[1] "first element in vector > 2"
Warning message:
In if (x > 2) { :
  the condition has length > 1 and only the first element will be used

> ifelse(x > 2, ">2", "<=2")
[1] ">2"  "<=2" "<=2"
```

The switch() function

- switch() chooses statements from a vector based on the value on an expression
- The multi-line form of switch() is

```
switch(expression,
  condition_01 = command_01,
  condition_02 = command_02,
  ...
  condition_n = command_n,
)
```

  - If the expression passed to switch() is not a character, it is coerced to integer
  - If the expression passed to switch() is a character string, then the string is matched exactly (with some small edge cases, see documentation)

The `switch()` function

```
grades <- c("A", "D", "F")

for (i in grades) {
  print(
    switch(i,
           A = "Well Done",
           B = "Alright",
           C = "C’s get Degrees!",
           D = "Meh",
           F = "Uh-Oh"
           )
  )
}

[1] "Well Done"
[1] "Meh"
[1] "Uh-Oh"
```

In-Class Lab

titanic.csv

1. Using a `for()` loop, recode the entries in the `Survived` variable with `"Survived"` and `"Perished"`
2. Using the `if()` command and loop, create a new variable of type `ordered factor` in the data frame called `ageClass`, and map `Age` to: `"Minor"` if less than 18 yrs; 18 yrs ≤ `"Adult"` ≤ 65 yrs; and `"Senior"` if older than 65 yrs
3. Ordering the passengers in descending order by last name, use a `while()` loop to identify the name of the 100*th* surviving passenger
4. Using a `switch()` statement, identify each passenger class, `Pclass`, as either `"First Class"`, `"Business Class"` or `"Economy"`, and print the results to the console

In-Class Lab - on your own

titanic.csv

- Iterate through the data frame, and for variables that are numeric, create a histogram, for categorical variables create a bar chart, and skip over all others
    1. Be sure to correct and clean the variable types before you run code (e.g., there are only two truly numeric variables)
    2. After creating each graph, be sure to include a pop-up a message that says `"Press Enter for next Graph"` to add a pause in the sequential execution