ILLINOIS

R Flow Control and Functions

Lecture 10

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)

Department of Statistics, University of Illinois

Flow Control:

Conditional Execution

Outline

- `if`/`else`, `switch`
- `for`, `while`
- `function`
- simple functional programming: `*apply`, `Map`, `Filter`

Overview

- `if` is the typical *conditional* execution
- Expression is evaluated, can contain an assignment
- If it is TRUE, subsequent expression is evaluated
- The { and } curly braces are used to group several statments

```
R> a <- 10
R> b <- 12
R> if (a > b) cat("Is bigger\n")
R> if (a < b) cat("Is smaller\n")
# Is smaller
```

`else` provides an alternative branch

```
R> a <- 10
R> b <- 12
R> if (a > b) {
+     cat("Is bigger\n")
+ } else {
+     cat("Is smaller\n")
+ }
# Is smaller
```

Anything wrong with this?

Also note that the five lines were pasted at once. You can't type them. More on that later.

We can nest with another `if` after `else`:

```
R> a <- 10
R> b <- 12
R> if (a > b) {
+     cat("Is bigger\n")
+ } else if (a < b) {
+     cat("Is smaller\n")
+ } else {
+     cat("Are equal\n")
+ }
# Is smaller
```

Takeaways

- Good practice to *always* have the `{` on the same line as `if`
  - When parsing text input, R operates line-by-line
  - A `{` alone on the subsequent line would be an error
  - (Not an issue if *files* are parsed or sourced all at once though)
- But defensive and careful programming:
  - Does not hurt to have `{` directly after `if`
  - Good practive to always have `{ ... }`
  - Even though not needed on single statements

Used for multiple alternatives

- The `switch` statement is an alternative to nested `if/else`
- Comparison to `integer` or `character`

```
R> a <- 10
R> b <- 12
R> txt <- switch(as.character(sign(a - b)),
+                "1" = "Is bigger",
+                "0" = "Are equal",
+                "-1" = "Is smaller")
R> cat(txt, "\n")
# Is smaller
```

- Previous example was unusual:
  - `switch` matches to either text, position
  - but `integer` values -1, 0, 1 do not work ... unless re-mapped

```
R> a <- 10
R> b <- 12
R> txt <- switch(sign(a - b) + 2, # add 2 to create 1:3
+                "Is bigger",
+                "Are equal",
+                "Is smaller")
R> cat(txt, "\n")
# Is bigger
```

Works—but not a style of code we recommend. Clarity first!

Flow Control:

Repeated Exceution

Basic loop construct

- Basic form: `for (i in someset) { ... }`
  - `for (i in 1:5) { print(i) }`
- The set over which `for` enumerates can be a variable
  - `for (v in LETTERS) { print(v) }`
  - where `LETTERS` is one of many built-in datasets
- The set over which `for` enumerates can be an expression
  - `for (v in someFunction()) { doSomething(v) }`
- We will get to alternatives later

Common error

- Usage such as `for (i in 1:n)`
  - where `n` may be zero or negative – surprise!
- Better: `for (i in seq_len(n))`
  - as `seq_len()` covers edge cases
  - try `seq_len(5)` and `seq_len(0)`
- Functional programming alternatives can be helpful
- But `for` is easy to *understand* and easy to *debug*
- So *do not be afraid* (or ashamed) to use loops!

Another loop construct

- `while` tests a condition and enters expression if `TRUE`
- `while` does *not* auto-increment a counter
- this is *your* responsibility:
  - if you do not do it ...
  - ... an *endless loop* is created
- mock example:
  - `while (someExpressionTrue) { doSomething(); }`
  - need to ensure the expression changes as a function of the code

Functions

The most important programming building block

- Basic notation:
  - `f <- function(a, b, c) { code here }`
- Functions
  - take arguments
  - arguments can have names
  - and can have default values
  - call with arguments by position, and/or name
- Play with: `f <- function(x = 1, y = 2) { x + Y }`
  - How many different calling patterns can you think of?

Return Values

- The last expression is returned via `invisible(x)`
  - *I.e.* value of `x` not printed by default
  - but can be assigned in calling expression
- Return can be enforced as well via `return(x)`
- As this can be combined: `return(invisible(x))`

Lazy Evaluation (a somewhat technical point)

- Most languages evaluate arguments at time of function call
- When function called, already-evaluated expression is passed
- But R is different here as it evaluates *lazily*
- In R, an expression is evaluated
    - not at the time the call is made
    - but only when the expression is used
    - (used for passing expressions to plot functions for labels etc)
- We can generally ignore this issue as *R does the right thing*

Funcional Programming Basics

In a Nutshell

- Given something that can be iterated over
- Apply given function to each iteration's element
- Simple example:

```
R> myvec <- 1:4
R> sapply(myvec, sqrt)
# [1] 1.00000 1.41421 1.73205 2.00000
```

Using `sapply`

```
R> myvec <- 1:4
R> res <- sapply(myvec, sqrt)
R> res
# [1] 1.00000 1.41421 1.73205 2.00000
```

Same outcome via *implicit* loop.

Using `for`

```
R> myvec <- 1:4
R> res <- myvec * 0  # create 0 vector
R> for (i in seq_along(myvec)) {
+     res[i] <- sqrt(myvec[i])
+ }
R> res
# [1] 1.00000 1.41421 1.73205 2.00000
```

Neither form is “better” or “worse”: they produce identical results at (essentially) identical resource use.

Minimal overview

- `lapply` returns a list
  - most general
  - may require second step collapsing list
- `sapply` is a simplified version
  - generally tries to return a vector
  - useful, less general
- More specialised and less commonly-used
  - `vapply` can specify a return object
  - `mapply` can use multiple arguments
  - `rapply` can be applied recursively

R Base also has

- `Filter` applies `f` to each element and returns `TRUE` subset
- `Reduce` can 'fold' or 'accumulate' results
- `Map` wraps `mapply` and does not simplify

These can be very powerful and expressive.

- Some people give the impressive that
  - functional programming is somewhat superior or better
  - that loops should be avoided
- Don't listen to that – our goals are
  - *comprehensible* code that achieves its objective: *clarity first*
  - real life use rarely gives extra points for "style"
- Important that code can be understood and trusted by
  - current user(s) ("you") and
  - future user(s) ("you" too, maybe others)
- This may mean different things for different people
  - there are *always* different ways and tradeoffs
  - with experience you get better at judging these

ILLINOIS

Control Flow and Functions

- We reviewed `if` and `switch`
- We looked at `for` and `while`
- We learned at `functions`
- Practice makes perfect