Control Structures

David Li

Commonly used control structures

- if and else: testing a condition and acting on it
- for: execute a loop a fixed number of times
- while: execute a loop *while* a condition is true
- repeat: execute an infinite loop (must break out of it to stop)
- break: break the execution of a loop
- next: skip an iteration of a loop

if-else

-
  ```
  if(<condition>) {
  ## do something
  }
  ## Continue with rest of code
  ```

-
  ```
  if(<condition>) {
  ## do something
  }
  else {
  ## do something else
  }
  ```

if-else {if-else}

```
if(<condition1>) {
## do something
} else if(<condition2>) {
## do something different
} else {
## do something different
}
#-----------------------
if(<condition1>) {
}
if(<condition2>) {
}
```

Example

```
x <- runif(1, 0, 10)
```

-
  ```
  if(x > 3) {
     y <- 10
   } else {
     y <- 0
    }
  ```

-
  ```
  y <- if(x > 3) {
     10
    } else {
       0
    }
  ```

-
  ```
  y <- ifelse(x>3, 10, 0)
  ```

ifelse()

- x <- c(6:-4)
- sqrt(x) #- gives warning
- sqrt(ifelse(x >= 0, x, NA)) # no warning
- ## Note: the following also gives the warning !
- ifelse(x >= 0, sqrt(x), NA)

- ## example of different return modes:

```
yes <- 1:3
no <- pi^(0:3)
typeof(ifelse(NA,    yes, no)) # logical
typeof(ifelse(TRUE,  yes, no)) # integer
typeof(ifelse(FALSE, yes, no)) # double
```

for Loops

-
  ```
  for(i in 1:10) {
   print(i)
  }
  x <- c("a", "b", "c", "d")
  ```

-
  ```
  for(i in 1:4) {
  ## Print out each element of 'x'
   print(x[i])
  }
  ```

for Loops (cont’ d)

- seq_along() function is commonly used in conjunction with for loops

```
for(i in seq_along(x)) {
    print(x[i])
}
```

- It is not necessary to use an index-type variable

```
 for(letter in x) {
    print(letter)
  }
```

- One line loops (curly braces are not required)

```
for(i in 1:4) print(x[i])
```

Nested for loops

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
    for(j in seq_len(ncol(x))) {
        print(x[i, j])
    }
}
```

while Loops

```
while (<condition>) {
    ## do something
}
```

Example:

```
count <- 0
while(count < 10) {
  print(count)
  count <- count + 1
}
```

While loops can potentially result in infinite loops if not written properly. Use with care!

repeat

```
x0 <- 1
tol <- 1e-8
repeat {
    x1 <- computeEstimate()
    if(abs(x1 - x0) < tol) { ## Close enough?
      break
    } else {
     x0 <- x1
    }
}
```

next, break

- next is used to skip an iteration of a loop.

```
for(i in 1:100) {
 if(i <= 20) {
   ## Skip the first 20 iterations
   next
  }
## Do something here
}
```

- break is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100) {
  print(i)
  if(i > 20) {
   ## Stop loop after 20 iterations
   break
  }
}
```

Summary

- Control structures like if, while, and for allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if (you believe) they are theoretically correct.
- Control structures mentioned here are primarily useful for writing programs; for commandline interactive work, the “apply” functions are more useful.
- It is more efficient to use built-in functions rather than control structures whenever possible.

Functions

David Li

Functions in R

- A core activity of an R programmer.
- "user" → developer
- When to write a function
  - Encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions.
  - Code must be shared with others or the public
- Create an interface to the code: via a set of parameters.
- This interface provides an abstraction of the code to potential users.
  - Ex: sort()

Your First Function

```
f <- function() {
   ## This is an empty function
}
## Functions have their own class
class(f)
# Execute this function
 f()
```

```
#with a parameter
f <- function(num) {
 for(i in seq_len(num)) {
  cat("Hello, world!\n")
  }
}
f(3)
```

```
#more fun
f <- function() {
   cat("Hello, world!\n")
}
f()
```

```
# with return value
f <- function(num) {
  hello <- "Hello, world!\n"
   for(i in seq_len(num)) {
   cat(hello)
   }
   chars <- nchar(hello) * num
   chars
}
meaningoflife <- f(3)
```

```
#return the very last expression that is evaluated.
```

Default value

```
f()
f <- function(num = 1) {
  hello <- "Hello, world!\n"
  for(i in seq_len(num)) {
    cat(hello)
  }
  chars <- nchar(hello) * num
  chars
}
f() ## Use default value for 'num   ‘
f(2)
f(num=2) #specified using argument name
```

So far, we have written a function that

- has one *formal argument* named num with a *default value* of 1.
- prints the message “Hello, world!” to the console a number of times indicated by the argument num
- *returns* the number of characters printed to the console

Argument Matching

- R functions arguments can be matched *positionally* or by name.
- Positional matching just means that R assigns the first value to the first argument, the second value to second argument, etc.

```
> str(rnorm)
function (n, mean = 0, sd = 1)
> set.seed(0)
> mydata <- rnorm(100, 2, 1) ## Generate some data
```

100 is assigned to the n argument, 2 is assigned to the mean argument, and 1 is assigned to the sd argument, all by positional matching.

Specifying arguments by name

- Order doesn't matter then

```
> sd(na.rm = FALSE, mydata)
```

Here, the mydata object is assigned to the x argument, because it's the only argument not yet specified.

- Function arguments can also be *partially* matched

- The order of operations when given an argument
  1. Check for exact match for a named argument
  2. Check for a partial match
  3. Check for a positional match

Example

```
> args(lm)
function (formula, data, subset, weights, na.action, method = "qr",
model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
contrasts = NULL, offset, ...)
NULL
```

The following two calls are equivalent.

```
set.seed(0)
mydata =  data.frame(y=rnorm(20), x=rnorm(20))
lm(data = mydata, y ~ x, model = FALSE, 1:20)
lm(y ~ x, mydata, 1:20, model = FALSE)
```

Lazy Evaluation

- Arguments to functions are evaluated *lazily*, so they are evaluated only as needed in the body of the function.

```
> f <- function(a, b) {
  a^2
 }
> f(2)
```

```
> f <- function(a, b) {
  print(a)
  print(b)
}
> f(45)
```

The ... Argument

- A special argument in R
- Indicate a variable number of arguments that are usually passed on to other functions.

```
myplot <- function(x, y, type = "l", ...) {
  plot(x, y, type = type, ...) ## Pass '...' to 'plot' function
}
```

- The … argument is necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL
> args(cat)
function (..., file = "", sep = " ", fill = FALSE, labels = NULL, append = FALSE)
NULL
```

Arguments Coming After the ... Argument

- One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched or matched positionally.

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL

paste("a", "b", sep = ":")

paste("a", "b", se = ":")
```

Summary

- Functions can be defined using the function() directive and are assigned to R objects just like any other R object
- Functions have can be defined with named arguments; these function arguments can have default values
- Functions arguments can be specified by name or by position in the argument list
- Functions always return the last expression evaluated in the function body
- A variable number of arguments can be specified using the special … argument in a function definition.

Next week

- Loop functions
- Homework 2 due
- Quiz 2

Computing Lab Ex

- Lab 3