Control Structures

Control Structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are:

- if, else: testing a condition
- for: execute a loop a fixed number of times
- while: execute a loop while a condition is true
- repeat: execute an infinite loop
- break: break the execution of a loop
- next: skip an iteration of a loop
- return: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.

if

This is a valid if/else structure.

```
x<-1
if(x > 3)
  {
  y <- 10
  } else {
  y <- 0
  }
print(y)
```

```
## [1] 0
```

So is this one.

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

```
## [1] 0
```

##for

for loops take an iterator variable and assigns it successive values from a sequence or vector. For loops are most commonly used fro iterating over the elements of an object (list, vector, etc.)

```
for(i in 1:10) {
  print(i)
  }
```

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

This loop takes the ***i*** variable and in each iteration of the loop gives it values *1, 2, 3, .., 10*, and then exits.

These four loops have the same behavior.

```
x <- c("a", "b", "c", "d")
```

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

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

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

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

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

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

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

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

Nested for loops

for loops can be nested.

```
x <- matrix(1:6, 2, 3,)
for(i in seq_len(nrow(x))) {
```

```
  for(j in seq_len(ncol(x))) {
    print(x[j])
  }
}
```

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

```
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Be careful with nesting though. Nesting beyond 2-3 levels is difficult to read/understand.

while

While loops begin by testing a condition. If it is true, then they execute the loop body. once the loop body is executed, the condition is tested again, and so forth.

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

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

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

Sometimes there will be more than one condition in the test.

```
z <- 5
while(z >= 3 && z<= 10) {
  print(z)
  coin <- rbinom(1, 1, 0.5)
  if(coin == 1)
  {
    ## random walk
    z <- z + 1
  } else {
      z <- z - 1
```

```
  }
  }
```

```
## [1] 5
## [1] 6
## [1] 5
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 6
## [1] 7
## [1] 6
## [1] 7
## [1] 8
## [1] 7
## [1] 8
## [1] 7
## [1] 6
## [1] 5
## [1] 4
## [1] 5
## [1] 4
## [1] 5
## [1] 4
## [1] 5
## [1] 4
## [1] 3
## [1] 4
## [1] 3
```

Conditions are always evaluated from left to right.

repeat

repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a repeat loops is to call break.

```
computeEstimate<- function()
{

  #A dummy function
  0
}
x0 <- 1
tot <- 1e-8
repeat {
  x1 <- computeEstimate()
  if(abs(x1 - x0) < tot) {
    break
  } else {
      x0 <- x1
      }}
```

The previous loop is a bit dangerous because there's no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

next, return

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
  }
```

return signals that a function should exit and return a given value.

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 they are technically correct
- Control structures mentioned here are primarily useful for writing programs; for command-line interactive work, the *apply functions are more useful.

Functions

Functions in R are created using the ***function()*** directive and are stored as R objects just like anything else. In particular, they are R objects of class ***"function"***.

```
#f <- function(<arguments>) {
  ## Do something here
 # }
```

Functions in R are "first class objects", which means that thDey can be treated much like any other R object. Importantly,

- Function can be ***passed as arguments*** to other function
- Functions can be ***nested***, so that you can define a function inside of another function. the return value of a function is the last express in the function body to be evaluated.

Function Arguments

Functions have named arguments which potentially have default values.

- The formal arguments are the arguments included in the function definition
- The formals function return a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be missing or might have default values

Argument Matching

R function arguments can be matched ***positionally or by name.*** So the following calls to ***sd*** are all equivalent.

> mydata < -rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)

Even though it's legal, I don't recommend messing around with the order of the arguments too much since it can lead to some confusion.

You can mix positional matching with matching by name. When an argument is matched by name, it is "**taken out**" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

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

The following two calls are equivalent.

lm(data = mydata, y - x, model = FALSE, 1:100)
lm(y - x, mydata, 1:100, model = FALSE)

- Most of the time, names arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
- Named arguments also help if you cannot remember the name of the argument and its position on the argument list (plotting is a good example)

Function arguments can also be potentially matched, which is useful for interactive work. the order of operations when given an argument is

1. Check for an exact match for a named argument
2. Check for a potential match (type part of the name)
3. Check for a positional match

Defining a Function

In addition to not specifying a default value, you can also set an argument to NULL.

f <- function(a, b = 1, c = 2, d = NULL) {

}

Lazy Evaluation

Arguments to functions are evaluated lazily, so they are evaluated only as needed.

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

f(2)
```

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

This function never actually uses the argument b, so calling ***f(2)*** will not produce and error because the 2 gets positionally matched to a.

```
f <- function(a, b) {
  #print(b) # This line is commented only to generate pdf. Before testing uncomment it.
  print(a)}

f(45)
```

```
## [1] 45
```

Notice that "45" got printed first before the error was triggered. This is because b did not have to be evaluated until after print(a). Once the function tried to evaluate print(b) it had to throw an error.

The '...' Argument

The ... argument indicate a variable number of arguments that are usually passed on to other functions.

`...` is often used when extending another function and you don't want to copy the entire argument list of the original function

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

Generic functions use `...` so that extra arguments can be passed to methods (more on this later).

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x00000288aa9e0428>
## <environment: namespace:base>
```

```
function(x, ...)
UseMethod("mean")
```

```
## function(x, ...)
## UseMethod("mean")
```

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

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

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

one catch with ... is that any arguments that appear after ... on the argument list must be named explicitly and cannot be partially matched.

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

> paste("a", "b", sep = ":")
[1] "a:b"

> paste(“a”, “b”, se = “:”)
[1] “a b :”

Your First R Function

Add two numbers together.

```
add2 <- function(x, y)
{
  x + y
}
add2(3, 5)
```

```
## [1] 8
```

Take a vector and return a subset of numbers larger than 10.

```
aboveTen <- function(x) {
  use <- x > 10   ## logical vector
  x[use]
  }

aboveTen(1:20)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

Take a vector and let the user define the number (but default to 10).

```
above <- function(x, n = 10) {
  use <- x > n
   x[use]
   }
above(1:20, 12)
```

```
## [1] 13 14 15 16 17 18 19 20
```

Take a matrix or a dataframe and calculate the mean of each column. Include an optional argument to keep or remove NA values in the data.

```
columnMean <- function(x, removeNA = TRUE) {
  nc <- ncol(x)
  means <- numeric(nc)
  for(i in 1:nc) {
    means[i] <- mean(x[,i], na.rm = removeNA)
    }
  means
  }
columnMean(airquality)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

[1] 42.129310 185.931507 9.957516 77.882353 6.993464 15.803922

Coding Standards

1. Use a text editor and save your file with the ASCII character set.

2. Indent your code

3. Limit the width of your code (80 columns?)

4. Limit the length of individual functions (one basic activity)

Scoping Rules

Binding Values to a symbol

How does R know which value to assign to which symbol? When I type

```
lm <- function(x) {
  x * x
}

lm
```

```
## function(x) {
##   x * x
## }
```

How does R know what value to assign to the symbol lm? why doesn't it give it the value of lm that is in the stats package?

When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly:

1. Search the global environment for a symbol matching the one requested

2. Search the namespaces of each of the packages on the search list

The search list can be found using the search function

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

4. The global environment or the user's workspace is always the first element of the search list and the base package is always the last

5. The order of the packages on the search list matter!

6. User's can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available

7. When a user loads a package with library the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list

8. Note that R has separate namespaces for functions and non-functions so it's possible to have an object named c and a function named c

Scoping rules

The scoping rules for R are the main feature that make it different from the original S language.

1. The scoping rules determine how a value is associated with a free variable in a function

2. R uses lexical scoping or static scoping. A common alternative is dynamic scoping

3. Related to the scoping rules is how R uses the search list to bind a value to a symbol

4. Lexical scoping turns out to be particularly useful for simplifying statistical computations

Lexical Scoping

Consider the following function.

```
f <- function(x, y) {
  x^2 + y / z
  }
```

This function has 2 formal arguments x and y. In the body of the function there is another symbol z. In this case z is called a free variable. The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned inside the function body).

Lexical scoping in R means that the values of free variables are searched for in the environment in which the function was defined.

What is an environment?

1. An environment is a collection of (symbol, value) pairs *i.e.* x is a symbol and 3.14 might be its value

2. Every environment has a parent environment; it is possible for an environment to have multiple "children"

3. The only environment without a parent is the empty environment

4. A function + an environment = a closure or function closure

Searching for the value for a free variable:

1. If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment

2. The search continues down the sequence of parent environments until we hit the top-level environment; this usually the global environment (workspace) or the namespace of a package

3. After the top-level environment, the search continues down the search list until we hit the empty environment. If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown

Why does it matter?

- Typically a function is defined int the global environment so that the values of free variables are found in the user's workspace
- This behavior is logical for most people and is usually the "right thing" to do
- However, in R you can have functions defined inside other functions
- Languages like C don't let you do this

Now things get interesting – In this case the environment in which a function is defined is the body of another function!

Example:

```
make.power <- function(n) {
  pow <- function(x) {
    x^n
  }
  pow}
```

This function returns another function as its value

```
cube <- make.power(3)
square <- make.power(2)
cube(3)
```

```
## [1] 27
```

```
square(3)
```

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

Exploring a Function Closure

What's in a function's environment?

```
ls(environment(cube))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(cube))
```

```
## [1] 3
```

```
 ls(environment(square))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(square))
```

```
## [1] 2
```

Lexical vs. Dynamic Scoping

What is the value of f(3)?

```
y <- 10
f <- function(x) {
  y <- 2
  y^2 +  g(x)
  }
g <- function(x) {
  x*y}

f(3)
```

```
## [1] 34
```

- With lexical scoping the value of y in the function g is looked up in the environment in which the function was defined, in this case the global environment, so the value of y is 10.
- With dynamic scoping, the value of y is looked up in the environment from which the function is called (sometimes referred to as the calling environment.
  - In R the calling environment is known as the parent frame
- so the value of y would be 2

When a function is defined in the global environment and is subsequently called from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping.

```
g<- function(x) {

  a <- 3
  x + a + y
}

g(2)
```

```
## [1] 15
```

```
 y <-3
 g(2)
```

```
## [1] 8
```

Other languages

Other languages that support lexical scoping:

- Scheme
- Perl
- Python
- Common Lisp (all languages converge to Lisp)

Consequences of Lexical Scoping

- In R all objects must be stored in memory
- All functions must carry a pointer to their respective defining environments, which could be anywhere
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment" of all the functions was the same

Vectorized Operations

Many operations in R are vectorized kaing the code more efficient, concise, and easier to read.

```
x <- 1:4; y <- 6:9
x + y
```

```
## [1]  7  9 11 13
```

```
 x > 2
```

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

```
x >= 2
```

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

```
y == 8
```

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

```
 x * y
```

```
## [1]  6 14 24 36
```

```
 x / y
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

In other languages you might have to run a loop to add two vectors together. But in R x + y acts as you'd expect and sums the two vectors.

```
 x <- matrix(1:4, 2, 2); y <= matrix(rep(10, 4), 2, 2)
```

```
##      [,1] [,2]
## [1,] TRUE TRUE
## [2,] TRUE TRUE
```

```
 x * y  ## element-wise multiplication
```

```
##      [,1] [,2]
## [1,]    6   24
## [2,]   14   36
```

```
 x / y
```

```
##           [,1]      [,2]
## [1,] 0.1666667 0.3750000
## [2,] 0.2857143 0.4444444
```

```
 #x %*% y ## true matrix multiplication # This line is commented only to generate pdf. Before testing u
```

Dates and Time

R has developed a special representation of dates and times.

- Dates are represented by the Date class
- Times are represented by the POSIXct and POSIXlt class
- Dates are stored internally as the number of days since 1970-01-01
- Times are stored internally as the number of seconds since 1970-01-01

Dates in R

Dates are represented by the Date and can be coerced from a character string using the as.Date() function.

```
x <- as.Date("1970-01-01")
x
```

```
## [1] "1970-01-01"
```

```
unclass(x)
```

```
## [1] 0
```

```
unclass(as.Date("1970-01-02"))
```

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

Times in R

Times are represented using the POSIXct or the POSIXlt class.

- POSIXct is just a very large integer under the hood; it uses a useful class when you want to store times in something like a data frame
- POSIXlt is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month

There are a number of generic functions that work on the dates and times.

- weekdays: give the day of the week
- months: give the month name
- quarters: give the quarter number("Q1", "Q2", "Q3", "Q4")

Times can be coerced from a character string using the as.POSIXlt or as.POSIXct function.

```
 x <- Sys.time()
 x
```

```
## [1] "2024-09-11 10:55:19 IST"
```

```
p <- as.POSIXlt(x)
p
```

```
## [1] "2024-09-11 10:55:19 IST"
```

```
names(unclass(p))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
##  [9] "isdst"  "zone"   "gmtoff"
```

```
p$sec
```

```
## [1] 19.44243
```

You can also use the POSIXct format:

```
x <- Sys.time()
x ## Already in `POSIXct` format
```

```
## [1] "2024-09-11 10:55:19 IST"
```

```
unclass(x)
```

```
## [1] 1726032319
```

```
#x$sec # This line is commented only to generate pdf. Before testing uncomment it.
```

```
p <- as.POSIXlt(x)
p$sec
```

```
## [1] 19.45938
```

Finally there is the ***strptime*** function in case your dates are written in a different format.

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011 10:40")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2012-01-10 10:40:00 IST" "2011-12-09 10:40:00 IST"
```

```
class(x)
```

```
## [1] "POSIXlt" "POSIXt"
```

Check ***?strptime*** for details the formatting strings.

Operations on Dates and Times

You can use mathematical operations on dates and times. Well really just + and -. you can do comparisons too (*i.e.* ==, <=)

```
x <- as.Date("2012-01-01")
y <- strptime("9 January 2011 11:34:21", "%d %b %Y %H:%M:%S")
#x-y # This line is commented only to generate pdf. Before testing uncomment it.
```

```
x <- as.POSIXlt(x)
x-y
```

```
## Time difference of 356.747 days
```

Time difference of 356.XXX days

Even keep track of leap years, leap seconds, daylight savings, and time zones.

```
x <- as.Date("2012-03-01"); y <- as.Date("2012-02-28")
x-y
```

```
## Time difference of 2 days
```

```
x <- as.POSIXct("2012-10-25 01:00:00")
y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
y-x
```

```
## Time difference of 10.5 hours
```

Summary

- Dates and times have special classes in R that allow for numerical and statistical calculations
- Dates use the Date class
- Times use the POSIXct and POSIXlt class
- Character strings can be coerced to Date/Time classes using the strptime function or the as.Date, as.POSIXlt, or as.POSIXct functions