UNIT-V

Control Structures

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some "logic" into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly. Commonly used control structures are,

- 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 interation of a loop

if-else

The if-else combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it's true or false.

Syntax:

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

The above code does nothing if the condition is false. If you have an action you want to execute when the condition is false, then you need an else clause.

Syntax:

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

There may be a series of tests by following the initial if with any number of else ifs.

Syntax:

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

Example of a valid if/else structure

```
## Generate a uniform random number
x <- runif(1, 0, 10)
if(x > 3) {
y <- 10
} else {
y <- 0
}
```

The value of y is set depending on whether x > 3 or not. This expression can also be written a different, but equivalent, way in R.

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

Of course, the else clause is not necessary. You could have a series of if clauses that always get executed if their respective conditions are true.

```
if(<condition1>) {
}
if(<condition2>) {
}
```

for Loops

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in my experience doing data analysis, I've found very few situations where a for loop wasn't sufficient.

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

```
> 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
```

The above loop takes the i variable and in each iteration of the loop gives it values 1, 2, 3, …, 10, executes the code within the curly braces, and then the loop exits. The following three loops all have the same behavior.

```
> x <- c("a", "b", "c", "d")
>
> for(i in 1:4) {
+ ## Print out each element of 'x'
+ print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

The seq_along() function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object (in this case, the object x).

```
> ## Generate a sequence based on length of 'x'
> for(i in seq_along(x)) {
+ print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Note : It is not necessary to use an index-type variable.

```
> for(letter in x) {
+ print(letter)
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

For one line loops, the curly braces are not strictly necessary.

```
> for(i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Nested for loops

for loops can be nested inside of each other.

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

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists). Nesting beyond 2 to 3 levels often makes it difficult to read/understand the code. If you find yourself in need of a large number of nested loops, you may want to break up the loops by using functions (discussed later).

while Loops

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, until the condition is false, after which the loop exits.

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

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

```
+ z <- z + 1
+ } else {
+ z <- z - 1
+ }
+ }
> print(z)
[1] 2
```

Conditions are always evaluated from left to right. For example, in the above code, if z were less than 3, the second test would not have been evaluated.

repeat Loops

repeat initiates an infinite loop right from the start. These are not commonly used in statistical or data analysis applications but they do have their uses. The only way to exit a repeat loop is to call break.

```
repeat {
## repeat statements
if(abs(x1 - x0) < tol) { ## Close enough?
break
} else {
x0 <- x1
}
}
```

The loop above is a bit dangerous because there's no guarantee it will stop. You could get in a situation where the values of x0 and x1 oscillate back and forth and never converge. Better to set a hard limit on the number of iterations by using a for loop and then report whether convergence was achieved or not.

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
}
```

}

Functions

Writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere "user" to a developer who creates new functionality for R. Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions. Functions are also often written when code must be shared with others or the public.

Functions in R

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

- Functions can be passed as arguments to other functions. This is very handy for the various apply funtions, like lapply() and sapply().
- Functions can be nested, so that you can define a function inside of another function

R Functions are used same as common language like C, these features might appear a bit strange. However, they are really important in R and can be useful for data analysis.

My First Function

Functions are defined using the function() directive and are stored as R objects just like anything else. In particular, they are R objects of class "function". Here's a simple function that takes no arguments and does nothing.

```
> f <- function() {
+ ## This is an empty function
+ }
> ## Functions have their own class
> class(f)
[1] "function"
> ## Execute this function
> f()
NULL
```

Not very interesting, but it's a start. The next thing we can do is create a function that actually has a non-trivial *function body*.

```
> f <- function() {
+ cat("Hello, world!\n")
+ }
> f()
Hello, world!
```

The last aspect of a basic function is the *function arguments*. For this basic function, we can add an argument that determines how many times "Hello, world!" is printed to the console.

```
> f <- function(num) {
+ for(i in seq_len(num)) {
```

```
+ cat("Hello, world!\n")
+ }
+ }
> f(3)
Hello, world!
Hello, world!
Hello, world!
```

Obviously, we could have just cut-and-pasted the cat("Hello, world!\n") code three times to achieve the same effect, but then we wouldn't be programming, would we?

Finally, the function above doesn't *return* anything. It just prints "Hello, world!" to the console num number of times and then exits. But often it is useful if a function returns something that perhaps can be fed into another section of code.

This next function returns the total number of characters printed to the console.

```
> f <- function(num) {
+ hello <- "Hello, world!\n"
+ for(i in seq_len(num)) {
+ cat(hello)
+ }
+ chars <- nchar(hello) * num
+ chars
+ }
> meaningoflife <- f(3)
Hello, world!
Hello, world!
Hello, world!
> print(meaningoflife)
[1] 42
```

In the above function, we didn't have to indicate anything special in order for the function to return the number of characters. In R, the return value of a function is always the very last expression that is evaluated. Because the chars variable is the last expression that is evaluated in this function, that becomes the return value of the function.

Note that there is a return() function that can be used to return an explicitly value from a function, but it is rarely used in R (we will discuss it a bit later in this chapter).

Finally, in the above function, the user must specify the value of the argument num. If it is not specified by the user, R will throw an error.

```
> f()  ## Error in f(): argument "num" is missing, with no default
```

We can modify this behavior by setting a *default value* for the argument num. Any function argument can have a default value, if you wish to specify it. Sometimes, argument values are rarely modified (except in special cases) and it makes sense to set a default value for that argument. This relieves the user from having to specify the value of that argument every single time the function is called.

Here, for example, we could set the default value for num to be 1, so that if the function is called without the num argument being explicitly specified, then it will print "Hello, world!" to the console once.

```
> 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'
Hello, world!
[1] 14
> f(2) ## Use user-specified value
Hello, world!
Hello, world!
[1] 28
```

Remember that the function still returns the number of characters printed to the console.

The ... Argument

There is a special argument in R known as the ... argument, which indicate a variable number of arguments that are usually passed on to other functions. The ... argument is often used when extending another function and you don't want to copy the entire argument list of the original function

For example, a custom plotting function may want to make use of the default plot() function along with its entire argument list. The function below changes the default for the type argument to the value type = "l" (the original default was type = "p").

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

Generic functions use ... so that extra arguments can be passed to methods. The ... argument is necessary when the number of arguments passed to the function cannot be known in advance.

Example : This is clear in functions like paste() and cat().

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

Because both paste() and cat() print out text to the console by combining multiple character vectors together, it is impossible for those functions to know in advance how many character vectors will be passed to the function by the user. So the first argument to either function is ....

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. Take a look at the arguments to the paste() function.

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

With the paste() function, the arguments sep and collapse must be named explicitly and in full if the default values are not going to be used.

Here I specify that I want “a” and “b” to be pasted together and separated by a colon.

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

If I don’t specify the ‘sep’ argument in full and attempt to rely on partial matching, I don’t get the expected result.

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

Scoping Rules of R - A Diversion on Binding Values to 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 }
```

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 in which things occur is roughly

1. Search the global environment (i.e. your workspace) for a symbol name matching the one requested.
2. Search the namespaces of each of the packages on the search list

The search list can be found by using the search() function.

```
> search()
[1] ".GlobalEnv" "package:knitr" "package:stats"
[4] "package:graphics" "package:grDevices" "package:utils"
[7] "package:datasets" "Autoloads" "package:base"
```

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. For better or for worse, the order of the packages on the search list matters, particularly if there are multiple objects with the same name in different packages.

Users can configure which packages get loaded on startup so if you are writing a function (or a package), you cannot assume that there will be a set list of packages available in a given order.

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.

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 features that make it different from the original S language (in case you care about that). This may seem like an esoteric aspect of R, but it's one of its more interesting and useful features.

The scoping rules of a language determine how a value is associated with a *free variable* in a function. R uses *lexical scoping* or *static scoping*. An alternative to lexical scoping is *dynamic scoping* which is implemented by some languages. Lexical scoping turns out to be particularly useful for simplifying statistical computations

Related to the scoping rules is how R uses the *search list* to bind a value to a symbol 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*. Okay then, what is an environment?

An *environment* is a collection of (symbol, value) pairs, i.e. x is a symbol and 3.14 might be its value. Every environment has a parent environment and it is possible for an environment to have multiple "children". The only environment without a parent is the *empty environment*.

A function, together with an environment, makes up what is called a *closure* or *function closure*. Most of the time we don't need to think too much about a function and its associated environment (making up the closure), but occasionally, this setup can be very useful. The function closure model can be used to create functions that "carry around" data with them.

How do we associate a value to a free variable? There is a search process that occurs that goes as follows:

- 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*.
- 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.
- 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.

One implication of this search process is that it can be affected by the number of packages you have attached to the search list. The more packages you have attached, the more symbols R has to sort through in order to assign a value.

Lexical Scoping: Why Does It Matter?

Typically, a function is defined in the global environment, so that the values of free variables are just 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!

Here is an example of a function that returns another function as its return value. Remember, in R functions are treated like any other object and so this is perfectly valid.

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

The make.power() function is a kind of “constructor function” that can be used to construct other functions.

```
> cube <- make.power(3)
> square <- make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Let’s take a look at the cube() function’s code.

```
> cube
function(x) {
x^n
}
```

Notice that cube() has a free variable n. What is the value of n here? Well, its value is taken from the environment where the function was defined. When I defined the cube() function it was when I called make.power(3), so the value of n at that time was 3.

We can explore the environment of a function to see what objects are there and their values.

```
> ls(environment(cube))
[1] "n" "pow"
> get("n", environment(cube))
[1] 3
```

We can also take a look at the square() function.

```
> ls(environment(square))
[1] "n" "pow"
> get("n", environment(square))
[1] 2
```

Lexical vs. Dynamic Scoping

We can use the following example to demonstrate the difference between lexical and dynamic scoping rules.

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

What is the value of the following expression? f(3)

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 was *called* (sometimes referred to as the *calling environment*).

In R the calling environment is known as the *parent frame*. In this case, 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.

Consider following example:

```
> g <- function(x) {
+ a <- 3
+ x+a+y
+ ## 'y' is a free variable
+ }
> g(2)
```

Error **in** g(2): object 'y' not found

```
> y <- 3
> g(2)
[1] 8
```

Here, y is defined in the global environment, which also happens to be where the function g() is defined.

There are numerous other languages that support lexical scoping, including

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

Lexical scoping in R has consequences beyond how free variables are looked up. In particular, it's the reason that all objects must be stored in memory in R. This is because all functions must carry a pointer to their respective defining environments, which could be *anywhere*.

In the S language (R's close cousin), free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment" of all functions is the same.

Loop Functions - Looping on the Command Line

Writing for and while loops is useful when programming but not particularly easy when working interactively on the command line. Multi-line expressions with curly braces are just not that easy to sort through when working on the command line. R has some functions which implement looping in a compact form to make your life easier.

- lapply(): Loop over a list and evaluate a function on each element
- sapply(): Same as lapply but try to simplify the result
- apply(): Apply a function over the margins of an array
- tapply(): Apply a function over subsets of a vector
- mapply(): Multivariate version of lapply