Programming with R

Rohit Jain

Objective:

- Write an if-else expression
- Write a for loop, a while loop, and a repeat loop
- Define a function in R and specify its return value
- Define what lexical scoping is with respect to how the value of free variables are resolved in R
- Describe the difference between lexical scoping and dynamic scoping rules
- Convert a character string representing a date/time into an R datetime object

Control Structures:

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

- if, else: testing a condition.
- for: executing a loop a fixed number of time.
- while: executing 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.

Control Structure - if,else:

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

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

Control Structure - for:

for loop takes an iterator varibale and assign it successive values from a sequence or a vector. For loops are most commonly used for iterating over the elements of an R 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
```

```
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"
```

Nested for:

```
m <- matrix(1:6,2,3)
m
```

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

```
for(i in 1:nrow(m)){
  for(j in 1:ncol(m)){
    print(m[i,j])
  }
}
```

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

Control Structure - while:

While loop begins by testing a condition.If it is true 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
```

Control Structure - Repeat,Next,Break:

- repeat intiates an infinite loop. The only way to exit a repeat loop is to call break.
- next is used to skip an iteration of a loop

```
for(i in 1:10){
  if(i <= 5){
    #skip first five iteration
    next
  }
  print(i)
}
```

```
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

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

Functions:

- Functions in R are created using the function() directive and stored as R objects. In particular they are R-objects of class "function".
- Function can be passed as an arguments to other functions.
- Functions can be nested.
- The return value of a function is the last expression in the function body to be evaluated.

Function Arguments:

- Functions have named arguments which potentially have a default values.
- Formal arguments are the arguments which are included in the function definition.
- the formals() function is used to list of all the formal arguments of the function.

- Function argument can be missing or might have default values.

Argument Matching:

- R funtions arguments can be matched positionally or by name. example:

```
mydata <- rnorm(100)
args(sd)
```

```
## function (x, na.rm = FALSE)
## NULL
```

```
formals(sd)
```

```
## $x
##
##
## $na.rm
## [1] FALSE
```

```
sd(mydata)
```

```
## [1] 0.9181317
```

```
sd(x = mydata)
```

```
## [1] 0.9181317
```

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

```
## [1] 0.9181317
```

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

```
## [1] 0.9181317
```

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

```
## [1] 0.9181317
```

defining a Function:

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

}

## lazy evalution
fl <- function(a,b){
  a^2
}
fl(2)
```

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

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 orginal function.

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

}
```

Scoping Rules:

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

```
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.

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

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.

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.

Lexical Scoping:

```
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
##         }
## <environment: 0x5040290>
```

```
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:

```
 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 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.

Date and Times in R

Dates are represented by the Date class and can be coerced from a character string using the as.Date() function. This is a common way to end up with a Date object in R.

```
## Coerce a 'Date' object from character
x <- as.Date("1970-01-01")
x
```

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

You can see the internal representation of a Date object by using the unclass() function.

```
unclass(x)
```

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

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

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

Times are represented by the POSIXct or the POSIXlt class. POSIXct is just a very large integer under the hood. It use 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. This is useful when you need that kind of information.

There are a number of generic functions that work on dates and times to help you extract pieces of dates and/or times

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

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

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

```
## [1] "2018-10-23 09:31:46 EDT"
```

```
class(x)   ## 'POSIXct' object
```

```
## [1] "POSIXct" "POSIXt"
```

The POSIXlt object contains some useful metadata.

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

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

```
p$wday     ## day of the week
```

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

You can also use the POSIXct format.

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

```
## [1] "2018-10-23 09:31:46 EDT"
```

```
unclass(x)    ## Internal representation
```

```
## [1] 1540301507
```

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

```
## [1] 46.76795
```

Finally, there is the strptime() function in case your dates are written in a different format. strptime() takes a character vector that has dates and times and converts them into to a POSIXlt object.

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

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

```
class(x)
```

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

Operations on Date and Time:

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 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
x <- as.POSIXlt(x)
x-y
```

```
## Time difference of 356.3095 days
```

Coding standards for R

- Always use text files/text editor.
- Indent code
- limt the column width.

Thank You