programming structures in R

R is a block-structured language in the context of the ALGOL-descendant family (C, C++, Python, Perl, etc.). Blocks are identified by braces **[ ]**; however, braces optional if a block contains a single statement. Statements are separated by newline characters or by semicolons (optional).

loops in R

Assume the following illustration that returns the squared value of each element **n** in vector **x**:

```
> x <- c(5,12,13)
> for (n in x) print(n^2)
[1] 25
[1] 144
[1] 169
```

Concretely, **for** every element **n** in vector **x**, the function **print(n^2)** returns the squared value of each element individually until the entire vector is passed through. The latter is achieved through looped iterations—the first iteration returns the function **print(n^2)** applied against **n = x[1]**, the second iteration returns the function **print(n^2)** applied against **n = x[2]**, and the third iterations returns the **n = [3]**.

Equally available for looping in R is with the **while** and **repeat** functions; completed with **break** (causing control to exit the loop). The follow illustrates the use of the latter three functions:

```
i <- 1
> while (i <= 10) i <- i+4    #i is assigned 1, 5, 9, and then 13 before breaking
> i
[1] 13

> i <- 1
> while(TRUE) {          #break plays a key role to break the loop after i > 10 = TRUE
+     i <- i+4
+     if (i > 10) break
+ }
> i
[1] 13

> i <- 1
> repeat {     #Note the lack of Boolean exit condition with repeat (break is required
+     i <- i+4
+     if (i > 10) break
+ }
> i
[1] 13
```

Additionally, the **next** function can be used to ***skip*** the remainder of the current iteration of a loop and proceed as follows. The advantage lies in avoiding complexity in nested **if-then-else** statements that can make code overly verbose.

The **for** constructs applies to ***any*** vector, regardless of class (mode). For example, filenames can be looped over to read their respective contents and perform a function or series of operations on.

looping over nonvector sets in R

R does not directly support iterative functions over nonvector sets. Some examples of indirect iterations to nonvector sets are as follows:

… **lapply( )** is applicable, assuming the iterations of the loop are independent of each other; allowing the order of operation to be irrelevant.
… **get( )** takes a character string as an argument representative of the name of some object and returns the object of the representative name.

Assuming two given matrices **u** and **v**, containing statistical data to become the object of R's linear regression function **lm( )** to each of the latter:

```
> u <- matrix(c(1,2,3,1,2,4), nrow = 3)
> v <- matrix(c(8,12,20,15,10,2), nrow =  3)
> u
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    4

> v
     [,1] [,2]
[1,]    8   15
[2,]   12   10
[3,]   20    2

> for(m in c("u", "v")) {
+     z <- get(m)
+     print(lm(z[,2] ~ z[,1]))
+ }

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept)       z[, 1]
    -0.6667       1.5000

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept)       z[, 1]
     23.286       -1.071
```

The above illustration initially sets **m** equal to **u**; the lines then assign matrix **u** to **z**, allowing **lm( )** call on **u**.

```
> z <- get(m)
> print(lm(z[,2] ~ z[,1]))

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept)       z[, 1]
     23.286       -1.071
```

The loop is then iterated over on matrix **v**, performing the same set of operations; calling **lm( )** on **v**.

if-else statements in R

The **if-else** syntax can be found as follows:

```
> if (r == 4) {
+     x <- 1
+ } else {
+     x <- 3
+     y <- 4
+ }
```

Simplicity aside, it is noted that there is just a ***single statement*** within the **if** function (**x <- 1**). It is thus implied that the braces **{ }** around the initial **if** statement are unnecessary. However, the right brace **}** is used by the R parser to identify the statement as an **if-else**, rather than an **if** alone. An **if-else** statement works as a call to a function, returning the last value assigned:

```
> v <- if(cond) expression1 else expression2if (r == 4)
```

The above sets **v** to the result of **expression1** or **expression2** depending on whether **cond** is **TRUE**.

```
> x <- 2
> y <- if(x == 2) x else x+1
> y
[1] 2

> x <- 3
> y <- if(x == 2) x else x+1
> y
[1] 4
```

As applications become more complex, **expression1** and **expression2** are likely to take on calls to functions. It is noted to not allow compactness to take priority over clarity. When applying **if-else** statements to vectors, the **ifelse( )** function should be applied to produce faster running code.

arithmatic and boolean operators and values in R

a list of basic operators in R programming:

Operation	Description
x + y	Addition
x - y	Subtraction
x * y	Multiplication
x / y	Division
x ^ y	Exponentiation
x %% y	Modular arithmetic
x %/% y	Integer division
x == y	Test for equality
x <= y	Test for less than or equal to
x >= y	Test for greater than or equal to
x && y	Boolean AND for scalars
x || y	Boolean OR for scalars
x & y	Boolean AND for vectors (vector x, y, result)
x | y	Boolean OR for vectors (vector x, y, result)
!x	Boolean negation

R technically does not employ scalar data types; scalars are treated as one-element vectors. However, the above illustrates various Boolean operators for both scalar and vector cases. The example on the proceeding page illustrates the need for the latter distinction:

```
> x <- c(TRUE, FALSE, TRUE)       #create logical vector x
> y <- c(TRUE, TRUE, FALSE)       #create logical vector y
> x
[1]  TRUE FALSE  TRUE

> y
[1]  TRUE  TRUE FALSE

> x & y                           #test Boolean AND for vectors x AND y
[1]  TRUE FALSE FALSE

> x[1] && y[1]                    #test Boolean AND for scalars in x[1] AND y[1]
[1] TRUE

> x && y                          #looks at just the first elements in vector x AND y
[1] TRUE

> if( x[1] && y[1]) print("both TRUE")
[1] "both TRUE"

> if(x & y) print("both TRUE")
[1] "both TRUE"
Warning message:
In if (x & y) print("both TRUE") :
  the condition has length > 1 and only the first element will be used
> if(x & y) print("both TRUE")
```

The illustration above demonstrates the unique property of an **if** function ***requiring a single Boolean*** in evaluation, opposed to a vector of Booleans. The **warning** result in example where **&** is applied is proof.

The Boolean logicals **TRUE** and **FALSE** can be abbreviated as **T** and **F** (capitalized). When applied in arithmetic, the values of **T** and **F** are represented as **1** and **0**:

```
> 1 < 2                   #1 is less than 2
[1] TRUE

> (1 < 2) * (3 < 4)       #1 is less than 2 and 3 is less than 4 (T * T = 1 * 1 = 1)
[1] 1

> (1 < 2) * (3 < 4) * (5 < 1)       #(T * T * F = 1 * 1 * 0 = 0)
[1] 0

> (1 < 2) == TRUE                   #1 is less than 2 = TRUE, thus TRUE = TRUE
[1] TRUE

> (1 < 2) == 1                      #1 is less than 2 = TRUE = 1, thus 1 = TRUE
[1] TRUE
```

default values for arguments = in R

R employs the use of ***named arguments*** and ***lazy evaluation*** in context. ***Named arguments*** refer to the optional arguments that exist within a given function. ***Lazy evaluation*** refers to how R does not evaluate an argument until/unless the argument is necessary. Therefore, ***named arguments*** may, or may not, be used:

```
read.csv(file, header = TRUE, sep = ",", quote = "\",
dec = ".", fill = TRUE, comment.char = "", ...)
```

The above example of the **read.csv( )** function illustrates the number of **default values** assigned as arguments when the function is called, unless stated otherwise in the code.

return values in R

The return value of a function can be any R object. Values are printed directly to the caller with the **return( )** function. By default, the value of the last executed statement is returned regardless. It is common in practice to avoid the use of the **return( )** function call; there is possibility of lengthening execution time as such. However, for the purpose of tracking function control returning to the caller, the insertion of **return( )** can make the code clearer and maintainable.

Considering the return value can be any R object, **complex objects** are equally applicable:

```
> g                                #print the function g()
function() {
    t <- function(x) return(x^2)
    return(t)
}
> g()                              #print the environment of function g()
function(x) return(x^2)
<environment: 0x0000000005f36210>
```

Functions with **multiple return values** should be placed in a **list**, or other applicable container.

functions are objects in R

Functions in R are referred to as ***first-class objects*** (of class "function"); they can be used like most objects.

```
> g <- function(x) {        #standard function syntax in R
+    return(x+1)
+ }
```

Concretely, the **function( )** is a **function** that serves to create **functions**. The two arguments in the above function are **x**—the formal argument—and the body **return(x + 1)** of class "expression". The illustration above creates a function object which is then assigned to **g**. The brace **"{"** is even a function in itself:

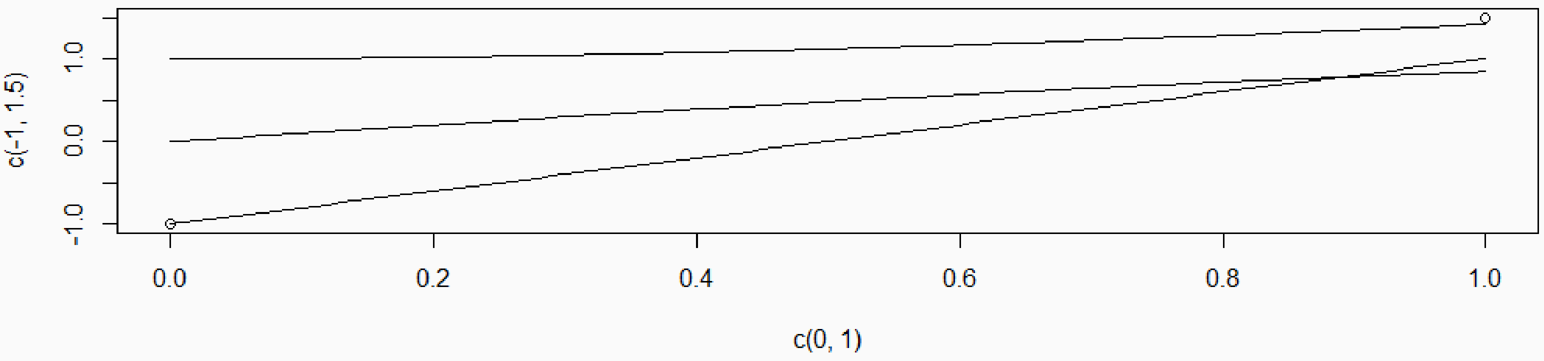
As defined by R – For {, the result of the last expression evaluated. This has the visibility of the last evaluation.

The arguments of the above example can be accessed as follows via the **formals( )** and **body( )** functions.

Because functions are R objects, they can be printed directly to the console by simply calling the function. For longer functions that need examination, the **page( )** function can be used to export the function to text.

Additionally, functions can be assigned to variables and used as arguments in other functions. Functions can be looped through a list of multiple functions equally. Looping through several functions to plot a graph:

```
> g1 <- function(x) return(sin(x))
> g2 <- function(x) return(sqrt(x^2 + 1))
> g3 <- function(x) return(2*x - 1)
> plot(c(0,1),c(-1,1.5))
> for(f in c(g1, g2, g3)) plot(f, 0, 1, add = TRUE)
```



The **formals( )** and **body( )** functions can equally be used to assign/replace functions (discussed more later).

environment and scope issues in R

Along with **functions** consisting of its arguments and body, the function's environment is also stored; made up of the collection of objects present at creation.

The Top-Level Environment

```
> w <- 12
> f <- function(y) {                        #function f() created at the Top-Level Environment
+     d <- 8                                #aka the interpreter command prompt
+     h <- function() {
+         return(d*(w+y))
+     }
+     return(h())
+ }
> environment(f)
<environment: R_GlobalEnv
```

The top-level environment in R is referred to as **R_GlobalEnv** in the output but **.GlobalEnv** in the code.

```
> ls()                  #lists the objects within the Top-Level environment
[1] "f" "w"

> ls.str()              #lists structure of Top-Level objects with more detail
f : function (y)
w :  num 12
```

Within the **scope hierarchy** of R, the environments to variables are defined similar to that of the C programming language; variable **w** is ***global*** to **f( )**, while variable **d** is ***local*** to **f( )**. The difference in R results from an expanded focus on the ***hierarchy***. Unlike C, R can have functions defined within functions—remembering that functions in R are objects and objects can be defined anywhere in R. Taking the above:

h( ) is ***local*** to **f( )**

d is ***local*** to **f( )**

d is ***global*** to **h( )**

y is ***local*** to **f( )** → arguments are considered ***locals*** in R

w( ) is ***global*** to **f( )**

w( ) is ***global*** to **h( )**

***h( )**'s environment consists of any arguments defined when **h( )** comes into existence; upon assignment

```
> h <- function() {
+     return(d*(w+y))
+ }
```

When **f( )** is called multiple times, **h( )** comes into existence multiple times and is then removed each time **f( )** returns. Therefore, **h( )**'s environment are the objects **d** and **y** created within **f( )**; ***plus*** **f( )**'s environment (**w**). Concretely, if one function is defined within another, the inner function's environment includes the environment of the outer function's, including any ***locals*** created thus far in the out function's environment.

With multiple nested functions, there is a nested sequence of larger and larger environments; the "root" environment consists of the top-level objects.

Applying the latter logic to calling the function **f( )** as follows:

```
> f(2)
[1] 112
```

```
> h
Error: object 'h' not found
```

Calling **f(2)** sets ***local*** **d** to **8** and then calls **h( )** → **d*(w+y)** = **8*(12+2)** resulting in **112**. Note that **w** does not explicitly exist within **h( )** and thus, R ascended the hierarchy (to the Top-Level) to find **w <- 12**.

As discussed above, **h( )** is ***local*** to **f( )** and thus not visible at the Top-Level Environment.

function side-effects in R

A property of functional programming is that functions do not change ***nonlocal*** variables (generally no side-effects). The code within a function has read access to ***nonlocal*** variables (not write access). Assumptions of reassignments are merely copies made during execution of a given function.

```
> w <- 12               #Top-Level Environment variable w remains unchanged as follows
> f
function(y) {
        d <- 8
        w <- w + 1
        y <- y - 2
        print(w)
        h <- function() {
            return(d*(w+y))
        }
        return(h())
}

> t <- 4                #Top-Level Environment variable t remains unchanged as above
> f(t)
[1] 13
[1] 120

> w
[1] 12

> t
[1] 4
```

Concretely as illustrated above, references to the ***local*** variable **w** are sent to the same memory location as the ***global*** variable **w** until the ***local*** variable **w** changes; in this case, a new memory location is used. An exception to the read-only nature of ***globals*** is with the **superassignment** operator (discussed later).

no pointers in R

R does not have ***pointers*** or ***references*** similar to those of programming languages like **C**. In Python, functions are available that directly change the arguments themselves:

```
>>> x = [13, 5, 12]
>>> x.sort()
>>> x
[5, 12, 13]
```

To produce the same result, assigning the sorted values of vector **x** to **x**, a ***reassignment*** is necessary:

```
> x <- c(13, 5, 12)
> sort(x)              #sorting vector x
[1]  5 12 13

> x                    #vector x is not assigned the arguments of sort()
[1] 13  5 12

> x <- sort(x)         #vector x must be reassigned to maintain the latter arguments
> x
[1]  5 12 13
```

The above logic can be applied to functions with multiple arguments equally; although with more complexity in the syntax as the arguments and nature of the function itself, expands.

writing upstairs in R

Although code within a certain level of environmental hierarchy has read access to the variables in levels above, direct write access to upper level variables is not possible with the standard assignment **<-** operator.

In order to write a variable to another in an environmental level higher than the variable of focus, the **superassignment <<-** operator, or the **assign( )** function, must be applied.

Writing to nonlocals with the superassignment operator

```
> two <- function(u) {
+     u <<- 2*u
+     z <- 2*z
+ }
> x <- 1
> z <- 3
> u                                   #u is not assigned in the global environment
Error: object 'u' not found

> two(x)                              #execute function(u) as assigned to variable two
> x                                   #global variable x maintains original assignment
[1] 1

> z                                   #global variable z maintains original assignment
[1] 3

> u                                   #u is superassigned as a top-level variable
[1] 2
```

Although the **superassignment <<-** operator is used to write top-level variables, its function is more discrete. The **<<-** operator performs in upward search in the environment hierarchy, stopping at the first level where a variable of that name is identified; the selected level becomes global in nature.

In the above illustration, **inc( )** is defined within **f( )**. Because there is no assignment to **x** upward in the hierarchy, the **x** within **inc( )** is the one the value is then written to; not **x** at the top-level.

Writing to nonlocals with the assign( ) function

```
> two                         #variation of two( ) above; using assign( ) instead of <<-
function(u) {
    assign("u", 2*u, pos = .GlobalEnv)
    z <- 2*z
}

> two(x)
> x               #x is not within two( ) and maintains its global value
[1] 1

> u               #u is not within two( ) but is superassigned with assign( ) top-level
[1] 2
```

when to use global variables in R

The use of ***global variables*** within the discipline of programming is widely debated. Many experts advocate for the banishment of ***global variables*** under any and all circumstances possible. There are certain circumstances where ***global variables*** provide value in the context of R programming. In the following text, the term ***global variable***, or ***globals*** is used to refer to any variable located higher in the environment hierarchy than the level of the given code within context (focus).

Within the R compiled code and binary constructions, ***globals*** are used widely; both in C code and R routines. For example, the ***superassignment*** **<<-** operator can be found in many R libraries. ***Threaded code*** and ***GPU*** code used for writing fast programs, typically use ***globals*** aggressively in practice; this allows communication avenues between parallel actions/actors.

```
> f
function(lxxyy) {                     #lzzyy is a list containing x and y
    ...
    lxxyy$x <- ...
    lxxyy$y <- ...
    return(lxxyy)
}
> lxy$x <- ...                        #set global variable x
> lxy$y  <- ...                       #set global variable y
> lxy <- f(lxy)
> ... <- lxy$x                        #use new x
> ... <- lxy$y                        #use new y
```

The above code can become unreliable as the variables become more complex, say as list classes. An alternate method of applying ***globals*** within the function are as follows:

```
> f
function() {
    ...
    x <<- ...                         #set global variable x
    y <<- ...                         #set global variable y
}
> x <- ...
> y <- ...
> f()                     #variables x and y are changed within the function execution
> ... <- x                            #use new x
> ... <- y                            #use new y
```

The latter example results in more clear, concise, and manageable code for maintenance and debugging—choosing to use global variables opposed to returning lists. In this theory, the use of ***globals*** become acceptable of they are considered to be **truly global**; being used broadly within the program's environment.

The alter argument to using ***globals*** for simplified code occurs with the cost of simplicity gained; debugging code down the line will become difficult when trying to track the local of ***global*** variable assignments, or reassignments. It is noted, however, that the optimization of text editors allow 'find' functions to locate and identify text within a space (crtl + f); this make sense, considering the original publication calling for abandonment of ***globals*** was written in the 1970s.

An additional argument in the use of ***globals*** is found when certain functions **f( )** are called in multiple independent segments of a program. Each call might require its own value of variables **x** and **y.** The solution would be to set up vectors for **x** and **y** values for each value as a corresponding element in the vector. The ultimate solution, however, loses some of the simplicity gained from the use of ***globals***.

Specifically, in R, a concern with ***globals*** exists at the **Top-Level Environment**. Code using ***globals*** runs the risk of overwriting unrelated variables consisting of the same name, or naming convention. A solution to protecting the integrity of ***globals*** at the **Top-Level** would be to employ application-specific assignments:

The below (left) → **is replaced by the below (right)**

```
> sim <<- list()
```

```
> assign("simenv", new.env(), envir = .GlobalEnv)
```

The above creates a new environment to capture ***globals*** at the top-level; accessed with **get( )** or **assign( )**:

The below (left) → **is replaced by the below (right)**

```
> if(is.null(sim$evnts)) {
+     sim$evnts <<- newevnt
+ }
```

```
> if(is.null(get("evnts", envir = simenv))) {
+     assign("evnts", newevnt, envir = simenv)
+ }
```

The above illustrations loses simplicity for the sake of ***global*** variable integrity, but still remains more manageable than other circumstances (lists of lists of lists) and mitigates unintended reassignments.

closures {⚙} in R

R ***closures*** consist of a function's arguments and body together with its **environment** at the time of a call. Including the **environment** is exploiting a type of programming with a feature **also known as** a ***closure***. A ***closure*** consists of a function that sets up a local variable and creates *another* function to access the variable.

```
> counter
function() {
    ctr <- 0
    f <- function() {
        ctr <<- ctr + 1
        cat("this count currently has value", ctr, "\n")
    }
    return(f)
}
```

Illustrating the operation of a single function in multiple programming environments as follows:

```
> c1 <- counter()                         #assigning counter() to variable c1
> c2 <- counter()                         #assigning counter() to variable c2
> c1                                      #c1 calls counter() in its own environment
function() {
        ctr <<- ctr + 1
        cat("this count currently has value", ctr, "\n")
    }
<environment: 0x0000000003cfbb30>
> c2                                      #c2 calls counter() in its own environment
function() {
        ctr <<- ctr + 1
        cat("this count currently has value", ctr, "\n")
    }
<environment: 0x00000000035ed130>
> c1()                                    #call to counter with c1, registers “1”
this count currently has value 1
> c1()                                    #call to counter with c1, registers “2”
this count currently has value 2
> c2()                                    #call to counter with c2, registers “1”
this count currently has value 1
> c2()                                    #call to counter with c2, registers “2”
this count currently has value 2
> c2()                                    #call to counter with c2, registers “3”
this count currently has value 3
> c1()                               #note counter() operates in separate environments
this count currently has value 3
```

recursion in R

A ***recursive*** function in R is one that calls itself; the intuition behind ***recursion*** is relatively simple:

To solve a problem of type **X** by writing a **recursive** function **f( ):**

… Break the original problem of type **X** into one or more smaller problems of type **X**.
… Within function **f( )**, call function **f( )** on each smaller problem segment.
… Within function **f( )**, converge the results of each call to **f( )**; solving the original problem.

An famously illustrative example of ***recursion*** is found in the **Towers of Hanoi Problem**.

Illustrative recursion through a Quicksort implementation

Quicksort is an algorithm used to sort a vector of numbers from smallest to largest. The implementation in R can be explained as with the vector (5,4,12,13,3,8,88):

All elements (4,12,13,3,8,88) are compared to element 5 and two subvectors are formed

… Subvector1 → all elements < 5 → (4,3)
… Subvector2 → all elements >= 5 → (12,13,8,88)
… The function is then called upon the subvectors, returning (3,4) and (8,12,13,88)
… The returns are stringed together with element 5, resulting in (3,4,5,8,12,13,88)

The following example is for illustrative purposes, considering R's **sort( )** function is compiled in C (faster):

```
> qs
function(x) {
    if(length(x) <= 1) return(x)
    pivot <- x[1]
    therest <- x[-1]
    sv1 <- therest[therest < pivot]
    sv2 <- therest[therest >= pivot]
    sv1 <- qs(sv1)
    sv2 <- qs(sv2)
    return(c(sv1, pivot, sv2))
}
> x <- c(5,4,12,13,3,8,88)
> qs(x)
[1]  3  4  5  8 12 13 88
```

Noting the ***termination condition***:

```
    if(length(x) <= 1) return(x)
```

Without the above constraint, R would endlessly call upon itself with empty vectors (until R interpreter fails).

Two potential reservations about ***recursion*** in R:

… ***Recursion*** can be fairly abstract when implemented
… ***Recursion*** can be memory-intensive when operating on larger problems

replacement functions in R

```
1    > x <- c(1,2,4)
2    > names(x)
3    NULL
4
5    > names(x) <- c("a","b","ab")
6    > names(x)
7    [1] "a"  "b"  "ab"
8
9    > x
10    a  b ab
11    1  2  4
```

Recalling an example introduced earlier:

Focusing on **line 5**, appears rather dormant. However, note the that a value is assigned to the result of a function call.

The availability of such an action in R is due to R's ***replacement function*** property. Below is the actual result of executing the script in **line 5**:

```
> x <- "names<-"(x, value = c("a","b","ab"))
```

Concretely, the call is function **names<-( )**.

Any assignment statement where the left side is ***not just*** an identifier (variable name) is considered a ***replacement function***. Concretely, when R is fed the following syntax, note the behavior:

The below (left)	→	**is computed by R as below (right)**
`> g(u) <- v`		`> u <- "g<-"(u, value = v)`

Note that function **g(u)** has to be defined in the current R environment prior to the call being successful.

Another example of ***replacement functions*** can be found through **subscripting** operations (also functions):

```
1    > x <- c(8,88,5,12,13)
2    > x
3    [1]  8 88  5 12 13
4
5    > x[3]
6    [1] 5
7
8    > "["(x,3)
9    [1] 5
10
11   > x <- "[<-"(x, 2:3, value = 99:100)
12   > x
13   [1]   8  99 100  12  13
```

As illustrated before, the call in **line 11** is performing the backend calculations in R when the code is executed: `> x[2:3] <- 99:100` which is ultimately verified as follows:

```
> x <- c(8,88,5,12,13)
> x[2:3]  <-  99:100
> x
[1]   8  99 100  12  13
```

tools for function composition in R

Functions can be written directly in the terminal session console (not advised for longer functions):

```
> g <- function() {
+     return(x+1)
+ }
```

Text editors (like Notepad) can be used and directly from the R Console through the **source( )** function:

```
>source("xyz.R")
```

Another option for quick edits to functions is through the **edit( )** function:

```
>f1 <- edit(f1)
```

The **edit( )** function can be used to edit **data structures** equally.

creating binary operations ⊕ in R

Similar to creating functions, ***binary operations*** can also be created in R:

```
> "%a2b%" <- function(a,b) return(a+2*b)
> 3 %a2b% 5
[1] 13
```

anonymous functions in R

In R, the purpose of the **function( )** function is to create functions. Considering the following code:

```
> inc <- function(x) return(x+1)
> inc
function(x) return(x+1)
```

The above illustration instructs R to create a function that adds 1 to its argument and then assigns that function to variable **inc**. However, the assignment is not always taken; it is available to use the function object created by the call to **function( )** without naming the object. Functions in the context are referred to as ***anonymous*** since they are unnamed. The proceeding example illustrates the latter:

```
> z <- matrix(1:6, nrow = 3)          #create matrix z
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> f <- function(x) x/c(2,8)           #create function(x) assigned to f
> y <- apply(z,1,f)                   #apply function f to matric z as variable y
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

The following bypasses the **assignment** to **f** by using an ***anonymous*** function without the call to **apply( )**:

```
> y <- apply(z,1,function(x) x/c(2,8))
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

The third **formal argument** to **apply( )** is required to be a function. This is represented in the above illustration considering that the return value of **function( )** is a function. Sometimes, it is more clear to write the code using ***anonymous functions*** opposed to defining functions ***externally***. However, more complex functions would be less like to benefit from the above application.