DATA ANALYTICS AND R PROGRAMMING
Subject code: 18MIT22C

UNIT-V: Control Structures -Functions- Scoping Rules of R - Loop Functions- Debugging Tool in R- Profiling R Code- Simulation.

Text book:

1. Roger D. Peng, “R Programming for Data Science” Lean Publishing, 2014. (Unit IV & V)

Prepared by Dr.P.Sumathi

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.

For starters, you can just use the **if** statement.

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

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.

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

You can have a series of tests by following the initial **if** with any number of **else if**s.

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

```
}
        ## do something different
```

Here is an 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
```

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

```
if(<con
dition1
>) {

}

if(<con
dition2
>) {

}
```

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 interator 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.)

```
> 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, 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"
```

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"
```

However, I like to use curly braces even for one-line loops, because that way if you decide to expand the loop to multiple lines, you won't be burned because you forgot to add curly braces (and you *will* be burned by this).

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). Be careful with nesting though. 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
+ }
```

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
> 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
```

---

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.

One possible paradigm might be in an iterative algorith where you may be searching for a solution and you don't want to stop until you're close enough to the solution. In this kind of situation, you often don't know in advance how many iterations it's going to take to get "close enough" to the solution.

```
x0 <- 1
tol <- 1e-8

repeat {
        x1 <- computeEstimate()
```

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

Note that the above code will not run if the computeEstimate() function is not defined (I just made it up for the purposes of this demonstration).

The loop agove 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.

The writing of a function allows a developer to create an interface to the code, that is explicitly specified with a set of parameters. This interface provides an abstraction of the code to potential users. This abstraction simplifies the users' lives because it relieves them from having to know every detail of how the code operates. In addition, the creation of an interface allows the developer to communicate to the user the aspects of the code that are important or are most relevant.

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

If you're familiar with 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.

Your First Function

```
> 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*. These are the options that you can specify to the user that the user may explicity set. 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? Also, it would be un-neighborly of you to give your code to someone else and force them to cut-and-paste the code however many times the need to see "Hello, world!".

> In general, if you find yourself doing a lot of cutting and pasting, that's usually a good sign that you might need to write a function.

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 explicity 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.

At this point, we have written a function that

- has one *formal argument* named num with a *default value* of 1. The *formal arguments* are the arguments included in the function definition. The formals() function returns a list of all the formal arguments of a function
- 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

Functions have *named arguments* which can optionally have default values. Because all function arguments have names, they can be specified using their name.

```
> f(num = 2)
Hello, world!
Hello, world!
[1] 28
```

Specifying an argument by its name is sometimes useful if a function has many arguments and it may not always be clear which argument is being specified. Here, our function only has one argument so there's no confusion.

Argument Matching

Calling an R function with arguments can be done in a variety of ways. This may be confusing at first, but it's really handing when doing interactive work at the command line. 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. So in the following call to rnorm()

```
> str(rnorm)
function (n, mean = 0, sd = 1)
> 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.

The following calls to the sd() function (which computes the empirical standard deviation of a vector of numbers) are all equivalent. Note that sd() has two arguments: x indicates the vector of numbers and na.rm is a logical indicating whether missing values should be removed or not.

```
> ## Positional match first argument, default for 'na.rm'
> sd(mydata)
[1] 0.9033251
> ## Specify 'x' argument by name, default for 'na.rm'
> sd(x = mydata)
[1] 0.9033251
> ## Specify both arguments by name
> sd(x = mydata, na.rm = FALSE)
[1] 0.9033251
```

When specifying the function arguments by name, it doesn't matter in what order you specify them. In the example below, we specify the na.rm argument first, followed by x, even though x is the first argument defined in the function definition.

```
> ## Specify both arguments by name
> sd(na.rm = FALSE, x = mydata)
[1] 0.9033251
```

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.

```
> sd(na.rm = FALSE, mydata)
[1] 0.9033251
```

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

Below is the argument list for the lm() function, which fits linear models to a dataset.

```
> 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.

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

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.

Most of the time, named 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 can remember the name of the argument and not its position on the argument list. For example, plotting functions often have a lot of options to allow for customization, but this makes it difficult to remember exactly the position of every argument on the argument list.

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

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Partial matching should be avoided when writing longer code or programs, because it may lead to confusion if someone is reading the code. However, partial matching is very useful when calling functions interactively that have very long argument names.

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

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

}
```

You can check to see whether an R object is NULL with the is.null() function. It is sometimes useful to allow an argument to take the NULL value, which might indicate that the function should take some specific action.

Lazy Evaluation

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

In this example, the function f() has two arguments: a and b.

```
> 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 an error because the 2 gets positionally matched to a. This behavior can be good or bad. It's common to write a function that doesn't use an argument and not notice it simply because R never throws an error.

This example also shows lazy evaluation at work, but does eventually result in an error.

```
> f <- function(a, b) {
+         print(a)
+         print(b)
+ }
> f(45)
[1] 45
Error in print(b): argument "b" is missing, with no default
```

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) the function had to throw an error.

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.

```
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x7fe7bc5cf988>
<environment: namespace:base>
```

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

```
> 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 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 feature 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*[60] 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 insided 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. That said, you'd have to have a pretty large number of packages attached in order to notice a real difference in performance.

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`

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

lapply()

The `lapply()` function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)
3. and returns a list (the `l` is for "list").

This function takes three arguments: (1) a list `X`; (2) a function (or the name of a function) `FUN`; (3) other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list()`.

The body of the `lapply()` function can be seen here.

```
> lapply
```

```
function (X, FUN, ...)
{
    FUN <- match.fun(FUN)
    if (!is.vector(X) || is.object(X))
        X <- as.list(X)
    .Internal(lapply(X, FUN))
}
<bytecode: 0x7fa339937fc0>
<environment: namespace:base>
```

Note that the actual looping is done internally in C code for efficiency reasons.

It's important to remember that `lapply()` always returns a list, regardless of the class of the input.

Here's an example of applying the `mean()` function to all elements of a list. If the original list has names, the the names will be preserved in the output.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3

$b
[1] 0.1322028
```

Notice that here we are passing the `mean()` function as an argument to the `lapply()` function. Functions in R can be used this way and can be passed back and forth as arguments just like any other object. When you pass a function to another function, you do not need to include the open and closed parentheses `()` like you do when you are *calling* a function.

Here is another example of using `lapply()`.

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.248845

$c
[1] 0.9935285
```

```
$d
[1] 5.051388
```

You can use `lapply()` to evaluate a function multiple times each with a different argument. Below, is an example where I call the `runif()` function (to generate uniformly distributed random variables) four times, each time generating a different number of random numbers.

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.02778712

[[2]]
[1] 0.5273108 0.8803191

[[3]]
[1] 0.37306337 0.04795913 0.13862825

[[4]]
[1] 0.3214921 0.1548316 0.1322282 0.2213059
```

When you pass a function to `lapply()`, `lapply()` takes elements of the list and passes them as the *first argument* of the function you are applying. In the above example, the first argument of `runif()` is `n`, and so the elements of the sequence `1:4` all got passed to the `n` argument of `runif()`.

Functions that you pass to `lapply()` may have other arguments. For example, the `runif()` function has a `min` and `max` argument too. In the example above I used the default values for `min` and `max`. How would you be able to specify different values for that in the context of `lapply()`?

Here is where the `...` argument to `lapply()` comes into play. Any arguments that you place in the `...` argument will get passed down to the function being applied to the elements of the list.

Here, the `min = 0` and `max = 10` arguments are passed down to `runif()` every time it gets called.

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 2.263808

[[2]]
[1] 1.314165 9.815635

[[3]]
[1] 3.270137 5.069395 6.814425

[[4]]
[1] 0.9916910 1.1890256 0.5043966 9.2925392
```

So now, instead of the random numbers being between 0 and 1 (the default), the are all between 0 and 10.

The `lapply()` function and its friends make heavy use of *anonymous* functions. Anonymous functions are like members of Project Mayhem[65]—they have no names. These are functions are generated "on the fly" as you are using `lapply()`. Once the call to `lapply()` is finished, the function disappears and does not appear in the workspace.

Here I am creating a list that contains two matrices.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Suppose I wanted to extract the first column of each matrix in the list. I could write an anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) { elt[,1] })
$a
[1] 1 2

$b
[1] 1 2 3
```

Notice that I put the function() definition right in the call to lapply(). This is perfectly legal and acceptable. You can put an arbitrarily complicated function definition inside lapply(), but if it's going to be more complicated, it's probably a better idea to define the function separately.

For example, I could have done the following.

```
> f <- function(elt) {
+         elt[, 1]
+ }
> lapply(x, f)
$a
[1] 1 2

$b
[1] 1 2 3
```

Now the function is no longer anonymous; it's name is **f**. Whether you use an anonymous function or you define a function first depends on your context. If you think the function **f** is something you're going to need a lot in other parts of your code, you might want to define it separately. But if you're just going to use it for this call to lapply(), then it's probably simpler to use an anonymous function.

sapply()

The sapply() function behaves similarly to lapply(); the only real difference is in the return value. sapply() will try to simplify the result of lapply() if possible. Essentially, sapply() calls lapply() on its input and then applies the following algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

Here's the result of calling lapply().

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] -0.251483

$c
[1] 1.481246

$d
[1] 4.968715
```

Notice that lapply() returns a list (as usual), but that each element of the list has length 1.

Here's the result of calling sapply() on the same list.

```
> sapply(x, mean)
        a         b         c         d
2.500000 -0.251483  1.481246  4.968715
```

Because the result of lapply() was a list where each element had length 1, sapply() collapsed the output into a numeric vector, which is often more useful than a list.

split()

The split() function takes a vector or other objects and splits it into groups determined by a factor or list of factors.

The arguments to split() are

```
> str(split)
function (x, f, drop = FALSE, ...)
```

where

- x is a vector (or list) or data frame

- `f` is a factor (or coerced to one) or a list of factors
- `drop` indicates whether empty factors levels should be dropped

The combination of `split()` and a function like `lapply()` or `sapply()` is a common paradigm in R. The basic idea is that you can take a data structure, split it into subsets defined by another variable, and apply a function over those subsets. The results of applying tha function over the subsets are then collated and returned as an object. This sequence of operations is sometimes referred to as "map-reduce" in other contexts.

Here we simulate some data and split it according to a factor variable.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$`1`
 [1]  0.3981302 -0.4075286  1.3242586 -0.7012317 -0.5806143 -1.0010722
 [7] -0.6681786  0.9451850  0.4337021  1.0051592

$`2`
 [1] 0.34822440 0.94893818 0.64667919 0.03527777 0.59644846 0.41531800
 [7] 0.07689704 0.52804888 0.96233331 0.70874005

$`3`
 [1]  1.13444766  1.76559900  1.95513668  0.94943430  0.69418458
 [6]  1.89367370 -0.04729815  2.97133739  0.61636789  2.65414530
```

A common idiom is `split` followed by an `lapply`.

```
> lapply(split(x, f), mean)
$`1`
[1] 0.07478098

$`2`
[1] 0.5266905

$`3`
[1] 1.458703
```

tapply

`tapply()` is used to apply a function over subsets of a vector. It can be thought of as a combination of `split()` and `sapply()` for vectors only. I've been told that the "t" in `tapply()` refers to "table", but that is unconfirmed.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

The arguments to tapply() are as follows:

- X is a vector

- INDEX is a factor or a list of factors (or else they are coerced to factors)
- FUN is a function to be applied
- … contains other arguments to be passed FUN
- simplify, should we simplify the result?

Given a vector of numbers, one simple operation is to take group means.

```
> ## Simulate some data
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> ## Define some groups with a factor variable
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
        1         2         3
0.1896235 0.5336667 0.9568236
```

We can also take the group means without simplifying the result, which will give us a list. For functions that return a single value, usually, this is not what we want, but it can be done.

```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.1896235

$`2`
[1] 0.5336667

$`3`
[1] 0.9568236
```

We can also apply functions that return more than a single value. In this case, tapply() will not simplify the result and will return a list. Here's an example of finding the range of each sub-group.

```
> tapply(x, f, range)
$`1`
[1] -1.869789  1.497041

$`2`
[1] 0.09515213 0.86723879

$`3`
[1] -0.5690822  2.3644349
```

apply()

The apply() function is used to a evaluate a function (often an anonymous one) over the margins of an array. It is most often used to apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). However, it can be used with general arrays, for example, to take the average of an array of matrices. Using apply() is not really faster than writing a loop, but it works in one line and is highly compact.

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

The arguments to apply() are

- X is an array
- MARGIN is an integer vector indicating which margins should be "retained".
- FUN is a function to be applied
- ... is for other arguments to be passed to FUN

```
> apply(x, 1, sum)    ## Take the mean of each row
 [1] -0.48483448  5.33222301 -3.33862932 -1.39998450  2.37859098
 [6]  0.01082604 -6.29457190 -0.26287700  0.71133578 -3.38125293
[11] -4.67522818  3.01900232 -2.39466347 -2.16004389  5.33063755
[16] -2.92024635  3.52026401 -1.84880901 -4.10213912  5.30667310
```

Note that in both calls to apply(), the return value was a vector of numbers.

You've probably noticed that the second argument is either a 1 or a 2, depending on whether we want row statistics or column statistics. What exactly *is* the second argument to apply()?

The MARGIN argument essentially indicates to apply() which dimension of the array you want to preserve or retain. So when taking the mean of each column, I specify

```
> apply(x, 2, mean)
```

because I want to collapse the first dimension (the rows) by taking the mean and I want to preserve the number of columns. Similarly, when I want the row sums, I run

```
> apply(x, 1, mean)
```

because I want to collapse the columns (the second dimension) and preserve the number of rows (the first dimension).

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
          [,1]       [,2]
[1,] 0.1681387 -0.1039673
[2,] 0.3519741 -0.4029737
```

In the call to apply() here, I indicated via the MARGIN argument that I wanted to preserve the first and second dimensions and to collapse the third dimension by taking the mean.

There is a faster way to do this specific operation via the colMeans() function.

```
> rowMeans(a, dims = 2)     ## Faster
          [,1]       [,2]
[1,] 0.1681387 -0.1039673
[2,] 0.3519741 -0.4029737
```

In this situation, I might argue that the use of rowMeans() is less readable, but it is substantially faster with large arrays.

Debugging

- message: A generic notification/diagnostic message produced by the message() function; execution of the function continues
- warning: An indication that something is wrong but not necessarily fatal; execution of the function continues. Warnings are generated by the warning() function
- error: An indication that a fatal problem has occurred and execution of the function stops. Errors are produced by the stop() function.
- condition: A generic concept for indicating that something unexpected has occurred; programmers can create their own custom conditions if they want.

Here is an example of a warning that you might receive in the course of using R.

```
> log(-1)
Warning in log(-1): NaNs produced
[1] NaN
```

This warning lets you know that taking the log of a negative number results in a NaN value because you can't take the log of negative numbers. Nevertheless, R doesn't give an error, because it has a useful value that it can return, the NaN value. The warning is just there to let you know that something unexpected happen. Depending on what you are programming, you may have intentionally taken the log of a negative number in order to move on to another section of code.

Here is another function that is designed to print a message to the console depending on the nature of its input.

```
> printmessage <- function(x) {
+         if(x > 0)
+                 print("x is greater than zero")
+         else
+                 print("x is less than or equal to zero")
+         invisible(x)
+ }
```

This function is simple—it prints a message telling you whether x is greater than zero or less than or equal to zero. It also returns its input *invisibly*, which is a common practice with "print" functions. Returning an object invisibly means that the return value does not get auto-printed when the function is called.

Take a hard look at the function above and see if you can identify any bugs or problems.

We can execute the function as follows.

```
> printmessage(1)
[1] "x is greater than zero"
```

The function seems to work fine at this point. No errors, warnings, or messages.

```
> printmessage(NA)
Error in if (x > 0) print("x is greater than zero") else print("x is less than o\
r equal to zero"): missing value where TRUE/FALSE needed
```

What happened?

Well, the first thing the function does is test if x > 0. But you can't do that test if x is a NA or NaN value. R doesn't know what to do in this case so it stops with a fatal error.

We can fix this problem by anticipating the possibility of NA values and checking to see if the input is NA with the is.na() function.

```
> printmessage2 <- function(x) {
+         if(is.na(x))
+                 print("x is a missing value!")
+         else if(x > 0)
+                 print("x is greater than zero")
+         else
+                 print("x is less than or equal to zero")
+         invisible(x)
+ }

> printmessage2(NA)
[1] "x is a missing value!"
```

And all is fine.

Now what about the following situation.

```
> x <- log(c(-1, 2))
Warning in log(c(-1, 2)): NaNs produced
> printmessage2(x)
Warning in if (is.na(x)) print("x is a missing value!") else if (x > 0)
print("x is greater than zero") else print("x is less than or equal to
zero"): the condition has length > 1 and only the first element will be
used
[1] "x is a missing value!"
```

Now what?? Why are we getting this warning? The warning says "the condition has length > 1 and only the first element will be used".

The problem here is that I passed printmessage2() a vector x that was of length 2 rather then length 1. Inside the body of printmessage2() the expression is.na(x) returns a vector that is tested in the **if** statement. However, **if** cannot take vector arguments so you get a warning. The fundamental problem here is that printmessage2() is not *vectorized*.

We can solve this problem two ways. One is by simply not allowing vector arguments. The other way is to vectorize the printmessage2() function to allow it to take vector arguments.

For the first way, we simply need to check the length of the input.

```
> printmessage3 <- function(x) {
+         if(length(x) > 1L)
+                 stop("'x' has length > 1")
+         if(is.na(x))
+                 print("x is a missing value!")
+         else if(x > 0)
+                 print("x is greater than zero")
+         else
+                 print("x is less than or equal to zero")
+         invisible(x)
+ }
```

Now when we pass printmessage3() a vector we should get an error.

```
> printmessage3(1:2)
Error in printmessage3(1:2): 'x' has length > 1
```

Vectorizing the function can be accomplished easily with the Vectorize() function.

```
> printmessage4 <- Vectorize(printmessage2)
> out <- printmessage4(c(-1, 2))
[1] "x is less than or equal to zero"
[1] "x is greater than zero"
```

Profiling R Code

```
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
   user  system elapsed
  0.004   0.002   0.431
```

```
## Elapsed time < user time
> hilbert <- function(n) {
+         i <- 1:n
+         1 / outer(i - 1, i, "+")
+ }
> x <- hilbert(1000)
> system.time(svd(x))
   user  system elapsed
  1.035   0.255   0.462
```

In this case I ran a singular value decomposition on the matrix in x, which is a common linear algebra procedure. Because my computer is able to split the work across multiple processors, the elapsed time is about half the user time.

Timing Longer Expressions

You can time longer expressions by wrapping them in curly braces within the call to system.time().

```
> system.time({
+         n <- 1000
+         r <- numeric(n)
+         for(i in 1:n) {
+                 x <- rnorm(n)
```

```
+                 r[i] <- mean(x)
+         }
+ })
   user  system elapsed
  0.086   0.001   0.088
```

If your expression is getting pretty long (more than 2 or 3 lines), it might be better to either break it into smaller pieces or to use the profiler. The problem is that if the expression is too long, you won't be able to identify which part of the code is causing the bottleneck.

```
> Rprof()       ## Turn on the profiler
```

The profiler can be turned off by passing NULL to Rprof().

```
> Rprof(NULL)       ## Turn off the profiler
```

The raw output from the profiler looks something like this. Here I'm calling the lm() function on some data with the profiler running.

The "by.self" output corrects for this discrepancy.

Now you can see that only about 4% of the runtime is spent in the actual lm() function, whereas over 40% of the time is spent in lm.fit(). In this case, this is no surprise since the lm.fit() function is the function that actually fits the linear model.

You can see that a reasonable amount of time is spent in functions not necessarily associated with linear modeling (i.e. as.list.data.frame, [.data.frame). This is because the lm() function does a bit of pre-processing and checking before it actually fits the model. This is common with modeling functions—the preprocessing and checking is useful to see if there are any errors. But those two functions take up over 1.5 seconds of runtime. What if you want to fit this model 10,000 times? You're going to be spending a lot of time in preprocessing and checking.

The final bit of output that summaryRprof() provides is the sampling interval and the total runtime.

```
$sample.interval
[1] 0.02

$sampling.time
[1] 7.41
```

Simulation

Simulation is an important (and big) topic for both statistics and for a variety of other areas where there is a need to introduce randomness. Sometimes you want to implement a statistical procedure that requires random number generation or samplie (i.e. Markov chain Monte Carlo, the bootstrap, random forests, bagging) and sometimes you want to simulate a system and random number

generators can be used to model random inputs.

R comes with a set of pseudo-random number generators that allow you to simulate from well-known probability distributions like the Normal, Poisson, and binomial. Some example functions for probability distributions in R

- rnorm: generate random Normal variates with a given mean and standard deviation
- dnorm: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- pnorm: evaluate the cumulative distribution function for a Normal distribution
- rpois: generate random Poisson variates with a given rate

For each probability distribution there are typically four functions available that start with a "r", "d", "p", and "q". The "r" function is the one that actually simulates randon numbers from that distribution. The other functions are prefixed with a

- d for density
- r for random number generation
- p for cumulative distribution
- q for quantile function (inverse cumulative distribution)

If you're only interested in simulating random numbers, then you will likely only need the "r" functions and not the others. However, if you intend to simulate from arbitrary probability distributions using something like rejection sampling, then you will need the other functions too.

Probably the most common probability distribution to work with the is the Normal distribution (also known as the Gaussian). Working with the Normal distributions requires using these four functions

---

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Here we simulate standard Normal random numbers with mean 0 and standard deviation 1.

```
> ## Simulate standard Normal random numbers
> x <- rnorm(10)
> x
 [1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513
 [6]  0.38979430 -1.20807618 -0.36367602 -1.62667268 -0.25647839
```

We can modify the default parameters to simulate numbers with mean 20 and standard deviation 2.

```
> x <- rnorm(10, 20, 2)
> x
 [1] 22.20356 21.51156 19.52353 21.97489 21.48278 20.17869 18.09011
```

```
 [8] 19.60970 21.85104 20.96596
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.09   19.75   21.22   20.74   21.77   22.20
```

If you wanted to know what was the probability of a random Normal variable of being less than, say, 2, you could use the pnorm() function to do that calculation.

```
> pnorm(2)
[1] 0.9772499
```

You never know when that calculation will come in handy.

Setting the random number seed

When simulating any random numbers it is essential to set the *random number seed*. Setting the random number seed with set.seed() ensures reproducibility of the sequence of random numbers.

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

Note that if I call rnorm() again I will of course get a different set of 5 random numbers.

```
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
```

If I want to reproduce the original set of random numbers, I can just reset the seed with set.seed().

```
> set.seed(1)
> rnorm(5)      ## Same as before
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

In general, you should **always set the random number seed when conducting a simulation!** Otherwise, you will not be able to reconstruct the exact numbers that you produced in an analysis.

It is possible to generate random numbers from other probability distributions like the Poisson. The Poisson distribution is commonly used to model data that come in the form of counts.

```
> rpois(10, 1)     ## Counts with a mean of 1
 [1] 0 0 1 1 2 1 1 4 1 2
> rpois(10, 2)     ## Counts with a mean of 2
 [1] 4 1 2 0 1 1 0 1 4 1
> rpois(10, 20)    ## Counts with a mean of 20
 [1] 19 19 24 23 22 24 23 20 11 22
```

Simulating a Linear Model

```
> ## Always set your seed!
> set.seed(20)
>
> ## Simulate predictor variable
> x <- rnorm(100)
>
> ## Simulate the error term
> e <- rnorm(100, 0, 2)
>
> ## Compute the outcome via the model
> y <- 0.5 + 2 * x + e
> summary(y)
    Min.  1st Qu.   Median     Mean 3rd Qu.     Max.
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
```

What if we wanted to simulate a predictor variable x that is binary instead of having a Normal distribution. We can use the rbinom() function to simulate binary random variables.

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> str(x)       ## 'x' is now 0s and 1s
 int [1:100] 1 0 0 1 0 0 0 0 1 0 ...

> set.seed(1)
>
> ## Simulate the predictor variable as before
> x <- rnorm(100)
```

Now we need to compute the log mean of the model and then exponentiate it to get the mean to pass to rpois().

```
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    1.00    1.00    1.55    2.00    6.00
> plot(x, y)
```

We can build arbitrarily complex models like this by simulating more predictors or making transformations of those predictors (e.g. squaring, log transformations, etc.).

Random Sampling

The sample() function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions of numbers.

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
>
> ## Doesn't have to be numbers
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
>
> ## Do a random permutation
> sample(1:10)
 [1]  4  7 10  6  9  2  8  3  1  5
> sample(1:10)
 [1]  2  3  4  1  9  5 10  8  6  7
>
> ## Sample  w/replacement
>sample(1:10, replace = TRUE)
 [1] 2 9 7 8 2 8 5 9 7 8
```

To sample more complicated things, such as rows from a data frame or a list, you can sample the indices into an object rather than the elements of the object itself.

Here's how you can sample rows from a data frame.

```
> library(datasets)
> data(airquality)
> head(airquality)
```

Now we just need to create the index vector indexing the rows of the data frame and sample directly from that index vector.

```
> set.seed(20)
>
> ## Create index vector
> idx <- seq_len(nrow(airquality))
>
> ## Sample from the index vector
> samp <- sample(idx, 6)
> airquality[samp, ]
```

Other more complex objects can be sampled in this way, as long as there's a way to index the sub-elements of the object.
