EEB Open Source for Open Science 2014

Scientific Programming Using R

Thomas Olszewski
Geology and Geophysics

Variables & Assigning Values

Variable = a 'place' in memory to store a value

```
> x <- 6.4
```

value to be stored

assignment operator

variable name (can include any alphanumeric symbol, but must start with a letter)

An *operator* is a symbol that makes the computer *do* something – the assignment operator *assigns* the given value to the declared variable. This is ***NOT*** the same as an equal sign in a mathematical equation (as we will see).

Arithmetic Operators

```
> a+b
> a-b
> a*b
> a/b
> a^b
```

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation

a = 6, *b* = 2:

What does *a*b* equal?

```
> a*b
[1] 12
```

a = 6, *b* = 2, *c* = 1, d = 2:

What does *a/b*(c+d)* equal? 1 or 9? Why?

```
> a/b*(c+d)
```

Order of Operations

(*a* = 6, *b* = 2, *c* = 1, d = 2)
What does *a/b*(c+d)* equal? 9.

```
> a/b*(c+d)
[1] 9
```

The value of depends on the *order of operations*. In most computer languages, including R, the order of arithmetic operations from is:

1) left to right
2) parentheses
3) exponents and roots
4) multiplication and division
5) addition and subtraction

To get an answer of 1, an additional set of parentheses is needed:

```
> a/(b*(c+d))
[1] 1
```

Creating Functions

Functions are used to package a series of commands. A function is a program or script that carries out a particular task. Prepackaged commands in R are examples of functions, but you can also create your own.

function name

assignment operator

parameter – one or more values passed to the function (can be left blank if no parameters

```
> foo <- function(x) {
+ a <- x
+ b <- 3
+ c <- a*b
+ #unread comment
+ return(c)
+ }
```

series of command in squiggly brackets

lines beginning with "#" are ignored – useful for comments and instructions

this command spits out the result of the function

Running Functions

```
> foo
function(x) {
a <- x
b <- 3
c <- a*b
return(c)
}

> foo(2)
[1]  6
```

Typing the name of the function returns its contents (this works for prepackaged functions as well as your own).

Typing the name of the function() and passing it a parameter value results in the commands being carried out.

Editing

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

Functions written elsewhere and saved as text files can be loaded into a workspace using the source() command.

```
> foo2 <- edit(foo)

> X.2 <- edit(X)
```

The edit() command will open any object – a function, a dataframe, a matrix, etc. – and allow you to change it (including prepackaged functions). The new version will be saved to the assigned name.

Creating Vectors

Vector = a series of values

values to be stored

```
> y <- c(2,5,8,11,2)
```

name (and declaration)

assignment operator

a function that combines arguments into a series

```
> y
[1]  2  5  8  11  2
```

Vector Index Numbers

How does one access individual values?

By using *index numbers*.

```
> y
[1]  2  5  8  11  2
```

vector name   square brackets   index

```
> y[2]
[1]  5
```

The critical difference between an index number and the actual value is that an index number refers to a particular slot in a vector (or other object), whereas the value is what is found in that slot.

Individual values can be reassigned:

```
> y[3] <- 28
> y
[1]  2  5  28  11  2
```

Subsets of the vector can be drawn by referring to multiple indices:

```
> y[c(2,4,5)]
[1]  5  11  2

> y[2:4]
[1]  5  28  11
```

":" is an operator that generates a sequence of integers *from:to* with a step size of 1.

```
> 2:5
[1] 2  3  4  5

> 4:2
[1] 4  3  2
```

Creating Matrices

Matrix = a table of values

```
> X <- matrix(1:12,nrow=3,ncol=4)
```

- `X` — name (and declaration)
- `<-` — assignment operator
- `matrix` — function that creates matrices
- `1:12` — values to be stored
- `nrow=3` — number of rows
- `ncol=4` — number of columns

```
> X
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Matrix Index Numbers

How does one access individual values?
Again, by using *index numbers*.

row index

column index

```
> X[2,3]
[1]  8
```

Entire rows or columns can be referenced by leaving the index blank.

```
> X[2,]
[1]  2  5  8  11

> X[,3]
[1]  7  8  9
```

Note that each of these is a vector.

Matrix Subsets

Subsets can also be referenced and values can be changed:

```
> X[2:3,3:4]
     [,1] [,2]
[1]     8   11
[2]     9   12

> X[c(1,3),c(2,4)]
     [,1] [,2]
[1]     4   10
[2]     6   12
```

```
> X[2,3] <- 36
> X
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5   36   11
[3,]    3    6    9   12
```

<u>Loops</u> are used repeat a series of commands.

index variable or "counter"

```
> A = c("H","A","P","P","Y")
> for (i in 1:5) {
+ print(i*2)
+ print(A[i])
+ }
```

number sequence

squiggly brackets

i is a variable that starts with the first value in the number sequence. Each command after "{" is carried out in succession. Every time the full succession is done (i.e., it hits "}"), *i* goes to the next value in the number sequence and each of the commands is repeated. Commands can change depending on the value of *i*. The loop will continue to repeat until *i* reaches the last value in the number sequence.

Nested Loops

Nested loops are "loops within loops" – they provide a means of working with objects that have multiple indices.

```
> X = matrix(0,nrow=12,ncol=9)
> for (i in 1:12) {
+   for (j in 1:9) {
+     X[i,j] <- i*j
+   }
+ }
```

Indenting the body of a loop is regarded as good programming practice because it is a good way to keep track of the structure of the program.

What happens: first, *i* equals 1; *j* equals 1 and *X*[1,1] is assigned 1*1; next *j* equals 2 and *X*[1,2] is assigned 1*2; *j* equals 3 and *X*[1,3] is assigned 1*3…until *j* equals 9. Now the first sweep through the *i*-loop is done and *i* becomes 2, but the *j*-loop starts again (the previous sweep is done and forgotten), so *j* equals 1 and *X*[2,1] is assigned 2*1; *j* equals 2 and *X*[2,2] is assigned 2*2…etc. Each time the *j*-loop is completed, the *i*-loop steps one value further and the whole set of commands (including the *j*-loop) within the *i*-loop is repeated.

Comparing Values

Comparison Operators

```
> a==b
> a!=b
> a>b
> a<b
> a>=b
> a<=b
```

- a==b - Equal
- a!=b - Not equal
- a>b - Greater than
- a<b - Less than
- a>=b - Greater than *or* equal
- a<=b - Less than *or* equal

```
> 3>2
[1] TRUE

> 3<2
[1] FALSE
```

```
> 3==2
[1] FALSE

> 3!=2
[1] TRUE
```

These operators result in a value of *TRUE* or *FALSE*.

Note that in typical use, an equal sign '=' can mean either assignment of a value OR a logical statement that is either true or false. These two roles have different operators: '<-' and '==,' respectively. In R, '=' is equivalent to assignment, but it is regarded as poor form (except when setting arguments in a function call).

Conditional Statements

Conditional statements are used to compare values.

```
> a <- 3
> b <- 2
> if (a>=b) {
+ print("VICTORY")
+ } else {
+ print("FAILURE")
+ }
[1] "VICTORY"
```

comparison resulting in TRUE or FALSE (a logical value)

squiggly bracket

commands to perform if condition is true

commands to perform if condition is NOT true

The *if-else* framework carries out one series of commands if a condition is TRUE and another if the condition is not TRUE. The () define the condition and the {} define the commands.

Logical Operators (Boolean Algebra)

```
> (a>b) & (c>d)
> (a>b) | (c>d)
> !(a>b)
```

- And: TRUE if both conditions are TRUE
- Or: TRUE if one or other condition is TRUE
- Not: Changes TRUE to FALSE and vice-versa

```
> (3>2) & (5>4)     T
> (3>2) & (5<4)     F
> (3<2) & (5<4)     F

> (3>2) | (5>4)     T
> (3>2) | (5<4)     T
> (3<2) | (5<4)     F
```

```
> (3>2) & !(5>4)     F
> (3>2) & !(5<4)     T
> (3<2) & !(5<4)     F
> !(3<2) & !(5<4)    F
```

These operators compare the truth value of multiple comparisons and result in a value of *TRUE* or *FALSE*.

Logical Values

```
> 3*(4<5)
[1]  3     #3*1
> 3*(4>5)
[1]  0     #3*0
```

In R, logical values (i.e., comparisons resulting in TRUE or FALSE) are assigned numerical values: TRUE = 1 and FALSE = 0, allowing them them to be manipulated as values.

Comparisons can be applied to vectors of values.

```
> x <- c(2,6,3,8,5,3)
> x > 4
[1] FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

Vectors of logical values can be used to subset vectors, etc.

```
> x <- c(2,6,3,8,5,3)
> y <- x > 4
> x[y]
[1] 6 8 5
```