Chapter 1

Programming Statements

This chapter introduces the main programming commands. These include functions, if-else statements, for-loops, and special procedures for managing the inputs to statistical models.

1.1 Functions

Functions are either built-in or user-defined sets of encapsulated commands which may take any number of arguments. Preface a function with the `function` statement and use the `<-` operator to assign functions to objects in your workspace.

You may use functions to run the same procedure on different objects in your workspace. For example,

```
check <- function(p, q) {
 result <- (p - q)/q
 result
 }
```

is a simple function with arguments `p` and `q` which calculates the difference between the ith elements of the vector `p` and the ith element of the vector `q` as a proportion of the ith element of `q`, and returns the resulting vector. For example, `check(p = 10, q = 2)` returns 4. You may omit the descriptors as long as you keep the arguments in the correct order: `check(10, 2)` also returns 4. You may also use other objects as inputs to the function. If `again = 10` and `really = 2`, then `check(p = again, q = really)` and `check(again, really)` also returns 4.

Because functions run commands as a set, you should make sure that each command in your function works by testing each line of the function at the R prompt.

1.2 If-Statements

Use `if` (and optionally, `else`) to control the flow of R functions. For example, let `x` and `y` be scalar numerical values:

```
if (x == y) {                # If the logical statement in the ()'s is true,
  x <- NA                    #  then `x' is changed to `NA' (missing value).
}
else {                       # The `else' statement tells R what to do if
  x <- x^2                   #  the if-statement is false.
}
```

As with a function, use { and } to define the set of commands associated with each if and else statement. (If you include if statements inside functions, you may have multiple sets of nested curly braces.)

1.3 For-Loops

Use `for` to repeat (loop) operations. Avoiding loops by using matrix or vector commands is usually faster and more elegant, but loops are sometimes necessary to assign values. If you are using a loop to assign values to a data structure, you must first initialize an empty data structure to hold the values you are assigning.

Select a data structure compatible with the type of output your loop will generate. If your loop generates a scalar, store it in a vector (with the ith value in the vector corresponding to the the ith run of the loop). If your loop generates vector output, store them as rows (or columns) in a matrix, where the ith row (or column) corresponds to the ith iteration of the loop. If your output consists of matrices, stack them into an array. For list output (such as regression output) or output that changes dimensions in each iteration, use a list. To initialize these data structures, use:

```
> x <- vector()                          # An empty vector of any length.
> x <- list()                            # An empty list of any length.
```

The `vector()` and `list()` commands create a vector or list of any length, such that assigning `x[5] <- 15` automatically creates a vector with 5 elements, the first four of which are empty values (`NA`). In contrast, the `matrix()` and `array()` commands create data structures that are restricted to their original dimensions.

```
> x <- matrix(nrow = 5, ncol = 2)  # A matrix with 5 rows and 2 columns.
> x <- array(dim = c(5,2,3))       # A 3D array of 3 stacked 5 by 2 matrices.
```

If you attempt to assign a value at $(100, 200, 20)$ to either of these data structures, R will return an error message ("subscript is out of bounds"). R does not automatically extend the dimensions of either a matrix or an array to accommodate additional values.

Example 1: Creating a vector with a logical statement

```
x <- array()             # Initializes an empty data structure.
for (i in 1:10) {        # Loops through every value from 1 to 10, replacing
```

```
  if (is.integer(i/2)) { #  the even values in `x' with i+5.
    x[i] <- i + 5
  }
}                          # Enclose multiple commands in {}.
```

You may use `for()` inside or outside of functions.

Example 2: Creating dummy variables by hand You may also use a loop to create a matrix of dummy variables to append to a data frame. For example, to generate fixed effects for each state, let's say that you have `mydata` which contains `y`, `x1`, `x2`, `x3`, and `state`, with `state` a character variable with 50 unique values. There are three ways to create dummy variables: 1) with a built-in R command; 2) with one loop; or 3) with 2 for loops.

1. R will create dummy variables on the fly from a single variable with distinct values.

   ```
   > z.out <- zelig(y ~ x1 + x2 + x3 + as.factor(state),
                    data = mydata, model = "ls")
   ```

   This method returns $k - 1$ indicators for k states.

2. Alternatively, you can use a loop to create dummy variables by hand. There are two ways to do this, but both start with the same initial commands. Using vector commands, first create an index of for the states, and initialize a matrix to hold the dummy variables:

   ```
   idx <- sort(unique(mydata$state))
   dummy <- matrix(NA, nrow = nrow(mydata), ncol = length(idx))
   ```

   Now choose between the two methods.

   (a) The first method is computationally inefficient, but more intuitive for users not accustomed to vector operations. The first loop uses `i` as in index to loop through all the rows, and the second loop uses `j` to loop through all 50 values in the vector `idx`, which correspond to columns 1 through 50 in the matrix `dummy`.

   ```
   for (i in 1:nrow(mydata)) {
     for (j in 1:length(idx)) {
       if (mydata$state[i,j] == idx[j]) {
         dummy[i,j] <- 1
       }
       else {
         dummy[i,j] <- 0
       }
     }
   }
   ```

Then add the new matrix of dummy variables to your data frame:

```
names(dummy) <- idx
mydata <- cbind(mydata, dummy)
```

(b) As you become more comfortable with vector operations, you can replace the double loop procedure above with one loop:

```
for (j in 1:length(idx)) {
  dummy[,j] <- as.integer(mydata$state == idx[j])
}
```

The single loop procedure evaluates each element in `idx` against the vector `mydata$state`. This creates a vector of n `TRUE`/`FALSE` observations, which you may transform to `1`'s and `0`'s using `as.integer()`. Assign the resulting vector to the appropriate column in `dummy`. Combine the `dummy` matrix with the data frame as above to complete the procedure.

Example 3: Weighted regression with subsets Selecting the `by` option in `zelig()` partitions the data frame and then automatically loops the specified model through each partition. Suppose that `mydata` is a data frame with variables `y`, `x1`, `x2`, `x3`, and `state`, with `state` a factor variable with 50 unique values. Let's say that you would like to run a weighted regression where each observation is weighted by the inverse of the standard error on `x1`, estimated for that observation's state. In other words, we need to first estimate the model for each of the 50 states, calculate 1 / SE(`x1`$_j$) for each state $j = 1, \ldots, 50$, and then assign these weights to each observation in `mydata`.

- Estimate the model separate for each state using the `by` option in `zelig()`:

  ```
  z.out <- zelig(y ~ x1 + x2 + x3, by = "state", data = mydata, model = "ls")
  ```

  Now `z.out` is a list of 50 regression outputs.

- Extract the standard error on `x1` for each of the state level regressions.

  ```
  se <- array()                           # Initalize the empty data structure.
  for (i in 1:50) {                       # vcov() creates the variance matrix
    se[i] <- sqrt(vcov(z.out[[i]])[2,2]) # Since we have an intercept, the 2nd
  }                                       # diagonal value corresponds to x1.
  ```

- Create the vector of weights.

  ```
  wts <- 1 / se
  ```

  This vector `wts` has 50 values that correspond to the 50 sets of state-level regression output in `z.out`.

- To assign the vector of weights to each observation, we need to match each observation's state designation to the appropriate state. For simplicity, assume that the states are numbered 1 through 50.

```
mydata$w <- NA            # Initalizing the empty variable
for (i in 1:50) {
  mydata$w[mydata$state == i] <- wts[i]
}
```

We use `mydata$state` as the index (inside the square brackets) to assign values to `mydata$w`. Thus, whenever state equals 5 for an observation, the loop assigns the fifth value in the vector `wts` to the variable `w` in `mydata`. If we had 500 observations in `mydata`, we could use this method to match each of the 500 observations to the appropriate `wts`.

If the states are character strings instead of integers, we can use a slightly more complex version

```
mydata$w <- NA
idx <- sort(unique(mydata$state))
for (i in 1:length(idx) {
  mydata$w[mydata$state == idx[i]] <- wts[i]
}
```

- Now we can run our weighted regression:

```
z.wtd <- zelig(y ~ x1 + x2 + x3, weights = w, data = mydata,
               model = "ls")
```