WEEK 9: ERROR HANDLING

THOMAS ELLIOTT

When writing loops and functions it is a good idea to build in some error handling in case things go a a little wonky. Sometimes, you may want the program to stop if you encounter an error, others you may want the program to ignore the error. There are a variety of ways to handle these situations.

1. BREAK

The first function to consider is `break`. You can use this inside a loop to stop executing the loop. For example:

```
> for(i in c(1,2,3,4,5)) {
+   print(i)
+   if( i == 3 ) break
+ }
[1] 1
[1] 2
[1] 3
```

In the loop above, it should loop over all the values 1 through 5, but because of the last statement once i equals 3, then the loop "breaks" and stops executing. You can use this if, while looping, your program encounters a value it wasn't expecting. Or you could break out of a loop once you've reached the desired outcome, even if the loop isn't finished. You could also combine this with a while loop to have multiple, complex conditions for ending the loop:

```
> set.seed(24601)
> while( TRUE ) {
+   r<-runif(1)
+   print(r)
+   if( r > 0.4 & r < 0.45 ) break
+   if( r > 0.76 & r < 0.8 ) break
+ }
[1] 0.3989171
[1] 0.08880356
[1] 0.7202155
[1] 0.9890866
[1] 0.6820787
[1] 0.3004171
[1] 0.03018914
```

Date: March 9, 2016.

```
[1] 0.1813453
[1] 0.391919
[1] 0.6654068
[1] 0.2623223
[1] 0.6870307
[1] 0.6014073
[1] 0.8931283
[1] 0.3984785
[1] 0.4086484
```

1.1. `next`. A related function is `next` which skips executing the current iteration of the loop and moves to the next:

```
> for(i in c(1,2,3,4,5) ) {
+   if( i == 3 ) next
+   print(i)
+ }
[1] 1
[1] 2
[1] 4
[1] 5
```

One use case for `next` might be when you are iterating over a list, and only want to perform actions on elements of the list that are of a certain type (for example, a vector of numbers) and skip elements that are of a different type. You can test for the type and use next to skip elements you want to ignore:

```
> x<-list(c(1,2,3),c("a","b,","c"),c(4,5,6))
> for(i in x ) {
+   if( class(i) != "numeric" ) next
+   print(mean(i))
+ }
[1] 2
[1] 5
```

In the example above, the list `x` contains three elements. The first and third are numeric vectors while the second is a character vector. The for loop first asks whether the element is numeric and if it isn't, that iteration of the loop is skipped. If it is, the mean of the vector is printed. As an aside, notice that in the loop definition above, i takes on the value of the element of x for the current iteration. In other words, the first time the for loop executes, i will be equal to `c(1,2,3)`.

2. `stop()`

`break` and `next` are used within loops. There are other functions available to use inside your own functions. The first of these is `stop()`, which operates much like break - it will stop executing your function, returning an error message. `stop()` takes as arguments any number of objects which can be coerced into a string. The string versions of these objects will be

concatenated together and printed to the console. `stop()` can be very useful to verify that arguments passed to your function are of the type and shape you are expecting.

```
> mysum<-function(x,y) {
+   if( class(x) != "numeric" | class(y) != "numeric" )
+     stop("x and y must be numeric")
+   if( length(x) != length(y) )
+     stop("x and y must be of the same length")
+   return(x+y)
+ }
> mysum(1,"a")
Error in mysum(1, "a") : x and y must be numeric
> mysum(c(1,2,3),c(1,2))
Error in mysum(c(1, 2, 3), c(1, 2)) : x and y must be of the same length
> mysum(c(1,2,3),c(4,5,6))
[1] 5 7 9
```

In the example above, the function first checks to see that x and y are both numeric. If either of them are not, the function stops and reports an error. Then the function checks to see that x and y are of the same length. If they are not, the function stops and reports an error. Finally, if no errors are reported, the function returns the sum of the two objects.

2.1. `warning()`. Sometimes you might encounter a problem that doesn't warrant stopping the function, but you want to call attention to the problem regardless. You can use `warning()` to achieve this. `warning()` will output a warning to the console, but continue executing the function. In the example below, rather than stopping the function when the length of the two objects are different, the function prints a warning that it will simply repeat the shorter object enough times to make the objects the same length. Since this might be unexpected behavior for someone not paying attention, we want to print the warning explaining what the function is going to do to help the user make sense of the output.

```
> mysum<-function(x,y) {
+   if( class(x) != "numeric" | class(y) != "numeric" )
+     stop("x and y must be numeric")
+   if( length(x) != length(y) ) {
+     warning("x and y are not the same length. The shorter will be repeated as needed t
+     num<-ifelse(length(x) > length(y), length(x), length(y))
+     x<-rep(x,length.out=num)
+     y<-rep(y,length.out=num)
+   }
+   return(x+y)
+ }
> mysum(c(1,2,3),c(4,5))
[1] 5 7 7
Warning message:
In mysum(c(1, 2, 3), c(4, 5)) :
  x and y are not the same length. The shorter will be repeated as
  needed to make them the same length.
```

3. try()

try() works much like capture does in Stata, executing an expression and capturing any error it produces. The program will continue to run, even if the expression in try() produces an error, and try() will return a "try-error" object that contains the text that would be passed to the console so you can parse the error in the program. I often use try() when I'm algorithmically generating QCA results - sometimes the truth table won't reduce and calling the reducing function when there are no rows coded as having the outcome will produce an error. I will call the reducing function in a try function and then check the class of the object to see if it was reduced successfully. You could use a similar process for other types of analysis.

```
> library(QCA)
> data(d.jobsecurity)
> truth<-truthTable(d.jobsecurity,outcome="JSR",
+                   conditions=c("S","C","P"),
+                   sort.by="incl")
> print(truth)

  OUT: outcome value
    n: number of cases in configuration
 incl: sufficiency inclusion score

   S  C  P  OUT n  incl  PRI
6  1  0  1   0  2  0.983 0.971
8  1  1  1   0  3  0.967 0.932
3  0  1  0   0  2  0.877 0.748
7  1  1  0   0  1  0.872 0.370
4  0  1  1   0  3  0.847 0.657
5  1  0  0   0  4  0.630 0.525
1  0  0  0   0  4  0.269 0.000

It seems that all outcome values have been coded to zero.
One suggestion is to lower the inclusion score for the presence of the outcome
The relevant argument is "incl.cut1", which now has a value of 1.

> sol<-try(eqmcc(truth,details=TRUE))
> if( class(sol) == "try-error" ) {
+   print("The truth table couldn't be reduced")
+ } else {
+   print(sol)
+ }
[1] "The truth table couldn't be reduced"
```