Talk 2 - Control Structures in R

Kevin O'Brien, M.S.
Vision Sciences Laboratory
Fall 2013
University of Georgia
Psychology Department

General Notes/Tips

- This presentation is in black and white to make it more easily printable.
- http://www.statmethods.net is a great resource
- This talk assumes you're using RStudio as a GUI for R.
- Feed R a copy of your data, not the original
- Always think before you code, especially in loops, and start small
- "How can I find where my code is failing" is the first step in finding out why your code failed.
- When you're done with your code, clean everything up and run it again. If it works then, you're set and can repeat your analysis.
- While many computer languages have 0 referencing, R has 1 referencing. If this conflicts with your habits, you're probably at a skill level where you'll be able to readily spot those issues.
- Typing ?[function] (e.g. *?mean()*) pulls up the help page – nicer in RStudio

A Quick Word About Formatting

- #Comments are indicated by an octothorp
- Code in this talk is indicated in blue (should print in gray) and italics
  - *print("This demonstrates the print command");*
- In RStudio's output console, code is blue, output is black, and errors are red.
- It may seem silly to put semicolons at the end of individual lines, but make it a habit. Things fail catastrophically in loops without semicolons.
- The quotation marks that you type into one program may not paste correctly in R: “” vs *""*

Object Types Are Important

- Vectors are one-dimensional arrays
  - *numerals<-c(1,2,3);*
  - *names<-c("one","two","three");*
  - *capitalNames<-c('ONE','TWO','THREE');*
- Data frames are a collection of vectors:
  - *numbers<-data.frame(numerals, names, capitalNames);*
  - *names(numbers)<-c('Numerals','LowerCase','UpperCase');*

Basic Control Structures

- Boolean comparisons
- If() statement (and if()/else() statement)
- For() loop
- While() loop
- Switch Case
- Sapply / Lapply (R specific)
- Sequences and repeat

Boolean Comparisons

- A statement for a boolean comparison has to have a binary state (TRUE or FALSE)
- == (check if equal)
- <, >, <=, >= (less than, greater than, less than OR equal to, greater than OR equal to)
- |, &, ! (or, and, not)
- *isTRUE(1!=1);* (checks if contents are true)

Boolean Comparator Examples

- *1==1;* #True
- *1==0;* #False
- *1!=0;* #True
- *'a'=='a';* #True
- *1<0;* #False
- *1>0;* #True
- *1<=1;* #True
- *1>=1;* #True

- *(TRUE & TRUE);*
- *(TRUE & FALSE);*
- *(TRUE | FALSE);*
- *(TRUE | TRUE);*
- *(FALSE | FALSE);*
- *(!FALSE);*
- *(!TRUE);*

The Humble If() Statement

- “Do this thing in braces if whatever I have in parentheses is true”
- *x<-10;*
- *if(x==10){*
- *print("x is indeed ten");*
- *}*
- If you change the test condition OR the value you're testing so that it evaluates to false, it won't spit out the output from the true condition.
- The if() statement is usually paired with an else().

At long last else

- “if the condition for the if part isn't satisfied, do this thing instead”
- *a<-5;*
- *if(a==5){*
- *print("Yep. It's a five.")*
- *}else{*
- *print("Your contrived demonstration did not satisfy the if() condition");*
- *}*

Well that's pretty boring...

- ...yeah, but it illustrates the mechanics. Let's make a function and cover a new arithmetic operator, because that sounds way less boring (by comparison)!
- %% (modulo / modulus) does integer division and returns the remainder – 10%%3 should return 1, for example
- The code's a bit bulky so it's on the next slide

That next slide I mentioned

```
EvenOrOdd<-function(inputValue){
  if(inputValue%%2 == 0){
    print(paste(inputValue,"is even.",sep=" "));
  }else{
    print(paste(inputValue,"is odd.", sep=" "));
  }
}
EvenOrOdd(5);
EvenOrOdd(6);
```

Other Nifty Tricks With If/Else

- *p<-0;*
- *p<-ifelse(p=="not_potato","potato","not_potato");*
- *z<-0;*
- *if(z==0){*
- * print("Z's zero, so do this thing.");*
- *}else if(z > 0){*
- * print("Z's bigger than zero, so do this thing.");*
- *}else if(z < 0){*
- * print("Z's smaller than zero, so do this thing.");*
- *}else{*
- * print("Wait, if it's not zero, and not larger or smaller...");*
- * print("WHY DID WE EVEN WRITE THIS PART?");*
- *}*
- It's generally good coding practice to ALWAYS have an ELSE, even if it's just empty or returns an error.

WARNING: LOOPS AHEAD

- Loops make it very easy to do repetitive things a tremendous number of times. DO NOT FORGET THAT THEY ARE POWERFUL.
- You can crash R or crash your computer with an infinite loop or a finite loop that uses too much memory.
- Be absolutely certain you know what you're doing if you do file I/O in a loop – you could destroy important stuff outside of R. Seriously.
- Consider yourself warned.

They're actually not that scary

- In most circumstances, you just need to make sure your code runs properly before you put it into a loop. Test the loop with a small amount of data before you let it run on a large amount of data.
- Efficiency increases inside loops are multiplicative, so be mindful of bloated code.
- Be prepared for frustrating errors that will make you feel great to fix.

The For() Loop

- This is the easiest loop to visualize, the hardest loop to break things with, and will cover like 99% of your loop needs.
- For loops require a counter variable and a sequence in R. The next few slides will have several trivial examples before we get into real, useful examples.

For() Loop Baby Steps

- *print(1);*
- *print(2);*
- *print(3);*
- *print(4);*
- *print(5);*
- For something simple like this, a for loop doesn't save us much time, but for something larger, it saves so much time.

```
• for(i in seq(1:5)){
•     print(i);
• }
• for(i in seq(1:100)){
•     print(i);
• }
• for(i in seq(from=0, to=1000,
  by=100)){
•   print(i);
• }
```

Pffft...that still doesn't seem helpful.

- Oh yeah?

```
subNum<-seq(1:1000);
subNum[473]<-4730;
for(i in 1:length(subNum)){
  if(subNum[i]>1000 | subNum[i]<0){
    subNum[i]=NA;
    print(paste(i,"had an error!",sep=" "));
  }
}
```

- Boom! You just changed a value that's impossible to NA so it's flagged properly for your analysis AND had R spit out a message to let you know what value(s) had a problem.

Well, I guess that could be helpful...

- Make this big fake dataset:

```
set<-data.frame();
currentRow = 1;
for(i in 1:10){
   for(j in 1:10){
    for(k in 1:10){
      set[currentRow,1]<-currentRow;
      set[currentRow,2]<-i;
      set[currentRow,3]<-j;
      set[currentRow,4]<-k;
      set[currentRow,5]<-rnorm(1,mean=100,sd=15);
      set[currentRow,6]<-rnorm(1,mean=100,sd=15);
      set[currentRow,7]<-rnorm(1,mean=100,sd=15);
      set[currentRow,8]<-rnorm(1,mean=100,sd=15);
      set[currentRow,9]<-rnorm(1,mean=100,sd=15);
      currentRow=currentRow+1;
    }
  }
}
names(set)<-c("SubjectNo","Cond1","Cond2","Cond3","IQ1","IQ2","IQ3","IQ4","IQ5");
```

...why are we doing this?

```
for(i in 1:nrow(set)){
  set[i,10]<-sum(set[i,5:9])/5;
  if(set[i,10]>105){
    set[i,11]="HIGH";
  }else if(set[i,10]<95){
    set[i,11]="LOW";
  }else{
    set[i,11]="AVG";
  }
}
names(set)[10:11]<-c("Mean","Group");
```

- We can aggregate, encode, replace, and do a lot of other things in for loops that would otherwise be prone to error and highly time consuming.

Switch Case

- Works like if/else but does not perform boolean assessments
- Improved efficiency under some circumstances (not as good as switch case in other languages)
- *demoVariable<-'q';*
- *switch(demoVariable, a="Got a", b="Got b", c="Got c", "Got something else.");*

Sapply / Lapply

- Applies function over specified object or range
- Generally prefer sapply() (neater output)
- *someNumbers<-data.frame(rnorm(1000,0,1),rnorm(1000,6,2),rnorm(1000,12,3.6));*
- *names(someNumbers)<-c("Group1", "Group2", "Group3");*
- *sapply(someNumbers, summary);*
- *lapply(someNumbers, summary);*

Sequences and Repeat

- seq(from, to, by); rep(thingToRepeat, times);
- *seq(from=0, to=1000, by=20);*
- *rep(1,50);*
- *rep(seq(1,5),20);*
- Handy for encoding, generating simulation data, etc.

How do I get my info out?

- (Requires code from slide 21)
- *attach(someNumbers);*
- *output<-t.test(Group1, Group2);*
- *names(output);*
- *tVal<-output[[1]];*
- *tValue<-as.numeric(output[1]);*
- The last line grabs just the numeric value, which is handy
- This is essential for making custom functions, running identical tests on massive data collections, etc.

Saving yourself a lot of copy/paste

- *source(file=file.choose(new = FALSE));*
- *corOut<-all.correlations(someNumbers);*
- Use the first line to add the function in AllCorrelations.R to your script
- Second line runs it and stores the output
- Note that this script does not correct for multiple comparisons

Handy Bonus Trick

- Need to allow the user to interactively select the working directory?
- *library(tcltk);*
- *setwd(tk_choose.dir(default = "", caption = "Select directory"));*