Module 6: programming

BIOSTAT 213: Introduction to Computing in the R Software Environment

Yea-Hung Chen, PhD, MS

Wednesday, February 20, 2019

if()

if()

```
> x<--1
> if(x<0){x*-1}
[1] 1
```

- the argument to `if()` must be a 1-length logical vector (ie, a single `TRUE`/`FALSE` value)
- the code after `if()` is evaluated if and only if the value inside `if()` is `TRUE`
- the curly brackets are not strictly necessary but are helpful if you have several lines of code
- for the built-in help file, use `help(Control)`

`if()` versus Stata's `if`

- `if()` is not analogous to Stata's `if`
- Stata's `if` is more analogous to indexing (Module 4)
- for example, from Stata's help file on `if`:

```
> list make mpg if mpg>25
```

- this is analogous to the following in R:

```
> make[mpg>25]
```

if()

```
> x<-40
> if(x<0){x*-1}
```

- here, nothing happens because x is not a negative number

if()

```
> x<--1
> if(x<0){x*-1}
[1] 1
> x
[1] -1
```

- note that in the original example, x does not change because we did not include object assignment

if()

```
> x<--1
> if(x<0){x<-x*-1}
> x
[1] 1
```

- adding object assignment here changes the value of `x` if (and only if) `x` is a negative number

else

```
> x<--1
> if(x<0){ x<-x*-1 } else { x<-0 }
> x
[1] 1
> x<-99
> if(x<0){ x<-x*-1 } else { x<-0 }
> x
[1] 0
```

else

- `else` must appear on the same line as the `TRUE` expression or the `TRUE` expression's closing bracket
- for example, the following works:

```
> x<-99
> if(x<0){ x<-x*-1 } else {
+   x<-0
+ }
> x
[1] 0
```

else

- likewise, the following works because `else` is on the same line as the `TRUE` expression's closing bracket:

```
> x<-99
> if(x<0){
+   x<-x*-1
+ } else {
+   x<-0
+ }
> x
[1] 0
```

else

- `else` must appear on the same line as the `FALSE` expression or the `FALSE` expression's opening bracket
- the following does not work:

```
> x<--99
> if(x<0){
+   x<-x*-1
+ } else
+ {
+   x<-0
+ }
> x
[1] 99
```

else

- from the help file:

  Note that it is a common mistake to forget to put braces (`{ .. }`) around your statements, e.g., after `if(..)` or `for(....)`. In particular, you should not have a newline between } and else to avoid a syntax error in entering a `if ... else` construct at the keyboard or via source. For that reason, one (somewhat extreme) attitude of defensive programming is to always use braces, e.g., for `if` clauses.

Multiple lines

```
> x<-2
> if(x<0){
+   x<-x*-1
+ } else
+ {
+   x<-x^2
+   y<-x^3
+   z<-x^4
+ }
> c(x,y,z)
[1]   4  64 256
```

Functions

Functions

- we've seen many examples over the last 5 weeks (!) of built-in R functions:

```
> class(round)
[1] "function"
> class(table)
[1] "function"
```

Functions

- as mentioned in the first week, it is also possible to write new functions

Writing new functions

```
> nothing<-function(){
+   # this function does nothing
+ }
```

- `function`: indicating you are defining a function
- `()`: a space for including arguments to your functions
- `{}`: a space for including the function's R code
- `<-`: object assignment
- `nothing`: an arbitrary name for the new function

Example: absolute values

```
> av<-function(x){
+   if(x<0){x*-1} else {x}
+ }
> av(3.25)
[1] 3.25
> av(-1.72)
[1] 1.72
> av(0)
[1] 0
> av(9/3)
[1] 3
```

Example: absolute values

- unlike `abs()`, our `av()` function only works on single numbers
- that's because `if()` only accepts single `TRUE`/`FALSE` values
- to allow the function to work even if `x` isn't a single number:

```
> av.improved<-function(x){
+   ifelse(x<0,x*-1,x)
+ }
> x<-c(3.25,-1.72,0,9/3)
> av.improved(x)
[1] 3.25 1.72 0.00 3.00
> abs(x)
[1] 3.25 1.72 0.00 3.00
```

When to and when not to write a new function

- it may be tempting to *always* write new functions
- new functions may not always be necessary (many of my R scripts, for example, do not involve new functions)
- in general, it is helpful to write new functions if you plan on using them over and over again

Example: univariate tables

```
> ut<-function(xx){
+   tt<-table(xx)
+   data.frame(
+     Group=names(tt),
+     Count=as.numeric(tt),
+     Percent=as.numeric(prop.table(tt))*100
+   )
+ }
```

Example: univariate tables

```
> ut(ii$hs)
    Group Count Percent
1      No  3237 56.1297
2 Smoking  2530 43.8703
> ut(ii$military)
     Group Count   Percent
1 Military   542  9.396671
2       No  5226 90.603329
```

Default argument values

```
> my.power<-function(x,power=2){x^power}
> my.power(3,2)
[1] 9
> my.power(3,3)
[1] 27
> my.power(3)
[1] 9
```

- to include default argument values, use the equal sign (=) when defining the function

Example: dichotomous factors

```
> cc<-function(xx,level1,level2='No',codings=1:2){
+   xx<-factor(xx,codings)
+   levels(xx)<-c(level1,level2)
+   xx
+ }
```

Example: dichotomous factors

- specifying all arguments (and including all argument names):

```
> table(ii$smq020,useNA='always')

   1    2    9 <NA>
2530 3237    2    0
> ii$hs.new<-cc(xx=ii$smq020,level1='Smoking',level2='No',
+               codings=1:2)
> table(ii$hs.new,useNA='always')

Smoking      No    <NA>
   2530    3237       2
```

Example: dichotomous factors

- taking advantage of default argument values:

```
> ii$hs.new<-cc(ii$smq020,'Smoking')
> table(ii$hs.new,useNA='always')

Smoking      No    <NA>
   2530    3237       2
```

Example: dichotomous factors

```
> table(unps$male,useNA='always')

   0    1 <NA>
2471 2541    0
> unps$gender<-cc(unps$male,level1='Female',level2='Male',
+                 codings=0:1)
> table(unps$gender,useNA='always')

Female   Male   <NA>
  2471   2541      0
```

The apply functions

The apply functions

- the apply functions provide a way to repeatedly use (apply) functions
- these include:
  - `apply()` to apply functions to rows or columns of data frames
  - `tapply()` to apply functions to groups (useful for stratified analysis)
  - `sapply()` and ‘lapply() to apply functions to elements in vectors

apply()

apply()

- apply() applies functions to rows or columns of data frames

Example: row means

- diastolic blood pressure in NHANES, using indexing:

```
> head(ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')])
  bpxdi1 bpxdi2 bpxdi3 bpxdi4
1     72     76     74     NA
2     62     80     42     NA
3     90     76     80     NA
4     86     88     86     NA
5     84     82     80     NA
6     80     80     82     NA
```

Example: row means

- how could we find each individual's mean measurement?
- one might try something like this:

```
> ii$diastolic.mean<-with(ii,(bpxdi1+bpxdi2+bpxdi3+bpxdi4)/4)
```

- but this is not quite right: most individuals are missing at least 1 measurement, and we should be adjusting the denominator as needed
- for example, the denominator should be 3, not 4, for individuals with only 3 measurements

Example: row means

- an alternative is to try something like this:

```
> row1<-ii[1,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')]
> row1<-as.numeric(row1)
> mean(row1,na.rm=TRUE)
[1] 74
```

- here, the denominator is appropriate, because we've made use of built-i function `mean()` and its `na.rm` argument
- but, we would need to do this for each row, which would be very tedious

Example: row means

- `apply()` is helpful here:

```
> ii$diastolic.mean<-apply(
+   X=ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')],
+   MARGIN=1,
+   FUN=mean,na.rm=TRUE
+ )
```

- here, `apply()` applies the function `mean` (the `FUN` argument) on the data (the `X` argument), but does this for each row of the data (the `MARGIN` argument)

Example: row means

```
> ii$diastolic.mean<-apply(
+   X=ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')],
+   MARGIN=1,
+   FUN=mean,na.rm=TRUE
+ )
```

- note that `na.rm` is not actually an argument to `tapply()`
- rather, it is an argument to `mean()`
- by including it inside of `apply()` we are actually specifying an argument to `mean()`!

Example: row means

- the same example, without argument names:

```
> ii$diastolic.mean<-apply(
+   ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')],
+   1,mean,na.rm=TRUE
+ )
```

- note that you must keep the `na.rm` name because it is an argument to `mean()` not an argument to `apply()`

Example: column means

- similarly, if we wanted to find the mean of each measurement:

```
> apply(
+   ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')],
+   2,mean,na.rm=TRUE
+ )
  bpxdi1   bpxdi2   bpxdi3   bpxdi4
69.91235 69.39600 69.25495 72.11168
```

- notice that we're using `MARGIN=2` instead of `MARGIN=1` to indicate that we want to apply `mean()` to each column

Example: row means, revisited

- it turns out there is a function called `rowMeans()`, for row means:

```
> ii$diastolic.mean.2<-rowMeans(
+   ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')],
+   na.rm=TRUE
+ )
```

Example: row means, revisited

- this is equivalent to what we did using `apply()`:

```
> table(ii$diastolic.mean==ii$diastolic.mean.2)

TRUE
5392
```

Example: column means, revisited

- likewise, there is a function called `colMeans()`, for column means:

```
> colMeans(
+   ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')],
+   na.rm=TRUE
+ )
  bpxdi1   bpxdi2   bpxdi3   bpxdi4
69.91235 69.39600 69.25495 72.11168
```

- this is equivalent to what we did using `apply()` (several slides back)

apply()-like functions

- in total, there are 4 functions that can be thought of as special cases of `apply()`:
  - `rowMeans()` for row means, conceptually equivalent to using `apply()` with `MARGIN=1` and `FUN=mean`
  - `colMeans()` for column means, conceptually equivalent to using `apply()` with `MARGIN=2` and `FUN=mean`
  - `rowSums()` for row sums, conceptually equivalent to using `apply()` with `MARGIN=1` and `FUN=sum`
  - `colMSums()` for column sums, conceptually equivalent to using `apply()` with `MARGIN=2` and `FUN=sum`

apply()

- apply() is very useful nevertheless: suppose, for example, we want to compute the median of the diastolic blood pressures!
- I often use apply() to apply functions I've written

Example: row medians

- using `apply()` to compute row medians:

```
> ii$diastolic.median<-apply(
+   ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')],
+   1,median,na.rm=TRUE
+ )
```

Example: grades

- consider the following hypothetical homework grades:

```
> grades
  hw1 hw2 hw3 hw4 hw5
1  92  94  89  91  81
2  90  94  88  90  85
3  87  96  89  91  81
4  92  94  86  92  89
5  93  95  86  91  95
```

Example: grades

- what if I have a policy that allows students to drop their lowest score?
- I can use `apply()` to find each student's mean score, after dropping each student's lowest score:

```
> drop.lowest<-function(grades){mean(sort(grades)[-1])}
> apply(grades,1,drop.lowest)
[1] 91.50 90.50 90.75 91.75 93.50
```

- here, I'm applying a custom function, `drop.lowest()` to each row

Example: grades

- equivalently, defining the function within `apply()`:

```
> apply(grades,1,function(grades){mean(sort(grades)[-1])})
[1] 91.50 90.50 90.75 91.75 93.50
```

tapply()

tapply()

- `tapply()` applies functions to groups
- it is useful for stratified analysis

Example: group means

```
> tapply(X=unps$assetscore,INDEX=unps$gender,FUN=mean)
     Female        Male
-0.04351296 -0.12354920
```

- here, `tapply()` applies the function `mean` (the `FUN` argument) on `assetscore` (the `X` argument), but does this for each value of `gender` (the `INDEX` argument)
- if we think of this as stratified analysis, `INDEX` specifies the groups

Example: group means

- the same example, without argument names:

```
> tapply(unps$assetscore,unps$gender,mean)
    Female       Male
-0.04351296 -0.12354920
```

Example: group means

- using `tapply()` here is conceptually equivalent to using indexing over and over again, but less tedious:

```
> mean(unps$assetscore[unps$gender=='Female'])
[1] -0.04351296
> mean(unps$assetscore[unps$gender=='Male'])
[1] -0.1235492
> tapply(unps$assetscore,unps$gender,mean)
     Female        Male
-0.04351296 -0.12354920
```

Example: group means

- a similar example from NHANES:

```
> tapply(ii$bmxbmi,ii$gender,mean)
Female   Male
    NA     NA
```

- here, we get `NA` because there are missing values in `bmxbmi`

Example: group means

- to remove the missing values using `mean()` we would add the `na.rm` argument:

```
> mean(ii$bmxbmi[ii$gender=='Female'],na.rm=TRUE)
[1] 29.65621
> mean(ii$bmxbmi[ii$gender=='Male'],na.rm=TRUE)
[1] 28.4981
```

Example: group means

- likewise, we can add the argument to `tapply()`:

```
> tapply(ii$bmxbmi,ii$gender,mean,na.rm=TRUE)
  Female     Male
29.65621 28.49810
```

- note that `na.rm` is not actually an argument to `tapply()`
- rather, it is an argument to `mean()`
- by including it inside of `tapply()` we are actually specifying an argument to `mean()`!

Example: group percentiles

- suppose we want to find the 80th percentile of body mass index, by gender
- we could do this "manually" using indexing:

```
> quantile(ii$bmxbmi[ii$gender=='Female'],na.rm=TRUE,probs=0.8)
 80%
35.4
> quantile(ii$bmxbmi[ii$gender=='Male'],na.rm=TRUE,probs=0.8)
 80%
32.4
```

Example: group percentiles

- a more efficient solution is to use `tapply()`:

```
> tapply(ii$bmxbmi,ii$gender,quantile,na.rm=TRUE,probs=0.8)
Female   Male
  35.4   32.4
```

- here, we've added the two `quantile()` arguments (`na.rm` and `probs`) from the previous slide

Example: group means

- the efficiency of `tapply()` is even more apparent when we have more than 2 groups:

```
> tapply(ii$bmxbmi,ii$bpcat,mean,na.rm=TRUE)
            Normal           Elevated            Stage 1
          27.67927           29.47767           30.41979
           Stage 2 Hypertensive crisis
          29.88089           26.13673
```

Example: group means

- for comparison, here's how we would find the means "manually":

```
> mean(ii$bmxbmi[ii$bpcat=='Normal'],na.rm=TRUE)
[1] 27.67927
> mean(ii$bmxbmi[ii$bpcat=='Elevated'],na.rm=TRUE)
[1] 29.47767
> mean(ii$bmxbmi[ii$bpcat=='Stage 1'],na.rm=TRUE)
[1] 30.41979
> mean(ii$bmxbmi[ii$bpcat=='Stage 2'],na.rm=TRUE)
[1] 29.88089
> mean(ii$bmxbmi[ii$bpcat=='Hypertensive crisis'],na.rm=TRUE)
[1] 26.13673
```

sapply()

sapply()

- `sapply()` applies functions to elements in vectors

Example: absolute values, revisited

- our original version of `av()` only works on single numbers:

```
> av<-function(x){
+   if(x<0){x*-1} else {x}
+ }
```

Example: absolute values, revisited

- a solution around this problem (besides the revision to `av()` presented earlier) is to use `sapply()`:

```
> x<-c(3.25,-1.72,0,9/3)
> sapply(x,av)
[1] 3.25 1.72 0.00 3.00
```

Example: absolute values, revisited

- sapply() is only "necessary"" with `av()` because `av()` only works on single numbers (because we didn't do a good job defining the function!)
- for example, with `av.improved()`, we can simply do this, avoiding `sapply()` altogether:

```
> av.improved(x)
[1] 3.25 1.72 0.00 3.00
```

- nevertheless, hopefully, the example with the original version of `av()` serves as a useful illustrative demonstration

Example: moving averages

- the Kevin Love data:

```
> ll[1:4,1:11]
   season age  tm  lg pos  g gs   mp  fg  fga   fgp
1 2008-09  20 MIN NBA   C 81 37 25.3 3.9  8.5 0.459
2 2009-10  21 MIN NBA  PF 60 22 28.6 4.9 10.8 0.450
3 2010-11  22 MIN NBA  PF 73 73 35.8 6.6 14.1 0.470
4 2011-12  23 MIN NBA  PF 55 55 39.0 8.6 19.3 0.448
```

- suppose we want to examine a 3-year moving average of Kevin Love's field goal percent (`fgp`)

Example: moving averages

- we can find the moving averages via `sapply()`:

```
> ma3<-function(year,y){
+   mean(c(y[year],y[year-1],y[year-2]))
+ }
> sapply(3:nrow(ll),ma3,y=ll$fgp)
[1] 0.4596667 0.4560000 0.4233333 0.4190000 0.4143333
[6] 0.4366667 0.4266667 0.4346667 0.3996667
```

- here, we are applying a custom function to Love's 3rd through most recent NBA seasons

Example: moving averages

- because `ma3()` accepts any `y`, we can examine other measures of Love's performance
- for example, here are 3-year moving averages of Love's free-throw percent:

```
> sapply(3:nrow(ll),ma3,y=ll$ftp)
[1] 0.8180000 0.8296667 0.7926667 0.7830000 0.7763333
[6] 0.8156667 0.8323333 0.8576667 0.8600000
```

Example: moving averages

- there *are* packages that implement moving averages, but I think it can be useful and instructive to implement things "manually" where possible