Module 8: Programming

BIOSTAT 213: Introduction to Computing in the R Software Environment

Monday, November 18, 2019

Dropping and renaming variables

Dropping variables

- frequently asked question: how do you drop a variable in R?

Dropping variables

- one way to drop variables in R is to use indexing and object assignment:

```
> ii<-read.csv('nhanes.csv')
> dim(ii)
[1] 4209  691
> ii<-ii[,names(ii)!='highldl']
> dim(ii)
[1] 4209  690
```

Dropping variables

- or, using `is.element()`:

```
> ii<-read.csv('nhanes.csv')
> dim(ii)
[1] 4209  691
> ii<-ii[,!is.element(names(ii),c('hdlcat','highldl','asthma'))]
> dim(ii)
[1] 4209  688
```

Renaming variables

- how do you rename a variable in R?

Renaming variables

- one way to rename variables in R is to use indexing and object assignment:

```
> table(ii$va)

             No Vigorous activity
            2314              1893
> names(ii)[names(ii)=='va']<-'vigorous.activity'
> table(ii$vigorous.activity)

             No Vigorous activity
            2314              1893
```

Writing functions

Functions

- we've seen many examples over the last few weeks of built-in R functions:

```
> mean(ii$bpxdi1,na.rm=TRUE)
[1] 70.47576
> round(pi,4)
[1] 3.1416
```

- note that often times, functions output some value

Writing functions

- it is possible to write your own functions

Writing functions

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

- `function`: a 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, to save your function
- `nothing`: an arbitrary name for the new function

Writing functions

- the last line of the function should be the value you want the function to return:

```
something<-function(x,y){
  x*y
  x/y
  x-y
  x+y
}
```

```
> something(3,4)
[1] 7
```

Example: absolute values

```
absolute.value<-function(x){
  ifelse(x<0,x*-1,x)
}
```

```
> absolute.value(-4:6)
 [1] 4 3 2 1 0 1 2 3 4 5 6
> abs(-4:6)
 [1] 4 3 2 1 0 1 2 3 4 5 6
```

When to write a function

- it may be tempting to *always* write functions
- but, functions may not always be necessary!
- in general, it is helpful to write a function if you plan on using it over and over again, using different inputs

Example: $y = ax^2 + bx + c$

```
quadratic<-function(x,a,b,c){
  a*x^2+b*x+c
}
```

```
> quadratic(3,2,-3,5)
[1] 14
> quadratic(0,5,5,100)
[1] 100
```

Example: fancy 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: fancy univariate tables

```
> ut(ii$hs)
                Group Count  Percent
1 History of smoking  1687 40.11891
2                  No  2518 59.88109
> ut(ii$military)
                          Group Count   Percent
1 History of military service   275  6.533618
2                           No  3934 93.466382
```

Default argument values

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

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

Example: dichotomous factors

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

Example: dichotomous factors

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

   1    2    7    9 <NA>
1687 2518    1    3    0
> ii$hs.new<-cc(ii$smq020,'Smoking')
> table(ii$hs.new,useNA='always')

Smoking      No    <NA>
   1687    2518       4
```

Isolation of functions

- object assignment within functions will not affect the rest of the R workspace:

```
> x<-3
> change.x<-function(newx){x<-newx}
> change.x(1000)
> x
[1] 3
```

- R isolates the function, to prevent inadvertent modification of objects in the workspace

Isolation of functions

- if you must modify something outside of the function, use <<-:

```
> x<-3
> change.x<-function(newx){x<<-newx}
> change.x(1000)
> x
[1] 1000
```

Isolation of functions

- I suggest using <<- sparingly, as you probably need it less often than you may think
- note, for example, that you can store the output of R functions:

```
> x<-3
> change.x<-function(y){y}
> change.x(1000)
[1] 1000
> x
[1] 3
> x<-change.x(1000)
> x
[1] 1000
```

if()

if()

- the `if()` function provides a way to conditionally evaluate code

if()

```
> x<--3
> if(x<0){x*-1}
[1] 3
```

- the argument to `if()` must be a single logical 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
- the feature is broadly known as control flow
- to access the built-in help file, type `help(Control)` or `?Control`

if()

```
> x<--3
> if(x<0){x*-1}
[1] 3
> x
[1] -3
```

- here, x did not change because we did not include object assignment

if()

```
> x<--3
> if(x<0){x<-x*-1}
> x
[1] 3
```

- here, x changed because we included object assignment

Example: the season

```
> Sys.Date()
[1] "2019-11-18"
> today<-Sys.Date()
> if((today>='2019-06-21')&(today<='2019-09-22')){'Summer'}
> if((today>='2019-09-23')&(today<='2019-12-20')){'Autumn'}
[1] "Autumn"
> if((today>='2019-12-20')&(today<='2020-03-19')){'Winter'}
```

else

```
> x<-100
> if(x<0){sign<-'positive'} else {sign<-'negative'}
> sign
[1] "negative"
```

- the `else` code will be evaluated if and only if the value inside `if()` is `FALSE`
- `else` must appear on the same line as the `TRUE` expression's closing bracket

else

```
x<-100
if(x<0){
  sign<-'positive'
} else {
  sign<-'negative'
}
```

```
> sign
[1] "negative"
```

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

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

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

- this is analogous to the following in R:

```
> mydata[mydata$mpg>25,c('make','mpg')]
```

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

- Stata's `if` allows for data manipulation
- in comparison, my recommendation is that you view R's `if()` as a way to implement decision trees
- there are other features in R that are more useful than `if()` for data manipulation, which we've discussed

Example: hypothesis testing

```
two.sample.test<-function(y,x,non.parametric=FALSE){
  if(!non.parametric){
    t.test(y~x)
  } else {
    wilcox.test(y~x)
  }
}
```

Example: hypothesis testing

```
> two.sample.test(y=ii$bpxdi1,x=ii$gender)

    Welch Two Sample t-test

data:  y by x
t = -7.6465, df = 3958.9, p-value = 2.576e-14
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.698357 -2.188874
sample estimates:
mean in group Female   mean in group Male
            69.03505             71.97867
```

Example: hypothesis testing

```
> two.sample.test(y=ii$bpxdi1,x=ii$gender,non.parametric=TRUE)

    Wilcoxon rank sum test with continuity correction

data:  y by x
W = 1719260, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0
```

The apply functions

tapply()

The apply functions

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

tapply()

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

Example: stratified means

- suppose we want to find the mean body mass index for each gender
- we can repeatedly apply the `mean()` function:

```
> mean(ii$bmxbmi[ii$gender=='Female'])
[1] 30.24384
> mean(ii$bmxbmi[ii$gender=='Male'])
[1] 29.01464
```

Example: stratified means

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

```
> tapply(X=ii$bmxbmi,INDEX=ii$gender,FUN=mean)
  Female     Male
30.24384 29.01464
```

- `X`: the data of interest
- `INDEX`: the stratifying variable
- `FUN`: the function to apply

Example: stratified means

- without argument names:

```
> tapply(ii$bmxbmi,ii$gender,mean)
  Female     Male
30.24384 29.01464
```

Example: stratified means

- mean systolic blood pressure, by gender:

```
> mean(ii$bpxsy1[ii$gender=='Female'])
[1] NA
> mean(ii$bpxsy1[ii$gender=='Male'])
[1] NA
```

- here, we get `NA` because there are missing values in `bpxsy1`

Example: stratified means

- to remove the missing values, we can add the `na.rm` argument:

```
> mean(ii$bpxsy1[ii$gender=='Female'],na.rm=TRUE)
[1] 120.4031
> mean(ii$bpxsy1[ii$gender=='Male'],na.rm=TRUE)
[1] 124.9325
```

Example: stratified means

- we can add pass the `na.rm` argument on to `tapply()`:

```
> tapply(ii$bpxsy1,ii$gender,mean,na.rm=TRUE)
  Female     Male
120.4031 124.9325
```

- to emphasize: `na.rm` is not an argument to `tapply()`
- rather, it is an argument to `mean()`

Example: stratified means

- mean systolic blood pressure, by BMI category:

```
> tapply(ii$bpxsy1,ii$bmicat,mean,na.rm=TRUE)
                Overweight Underweight or normal weight
                  122.9266                    117.7426
             Class I obesity            Class II obesity
                  124.2016                    126.1900
            Class III obesity
                  129.1705
```

Example: stratified means

- equivalently, the long way:

```
> mean(ii$bpxsy1[ii$bmicat=='Underweight or normal weight'],
+      na.rm=TRUE)
[1] 117.7426
> mean(ii$bpxsy1[ii$bmicat=='Overweight'],na.rm=TRUE)
[1] 122.9266
> mean(ii$bpxsy1[ii$bmicat=='Class I obesity'],na.rm=TRUE)
[1] 124.2016
> mean(ii$bpxsy1[ii$bmicat=='Class II obesity'],na.rm=TRUE)
[1] 126.19
> mean(ii$bpxsy1[ii$bmicat=='Class III obesity'],na.rm=TRUE)
[1] 129.1705
```

Example: stratified percentiles

- 80th percentile of systolic blood pressure, by gender:

```
> tapply(ii$bpxsy1,ii$gender,quantile,na.rm=TRUE,probs=0.8)
Female   Male
   132    136
```

- as with previous examples, note that `na.rm` and `probs` are not actually arguments to `tapply()`
- rather, they are arguments to `quantile()`, which we've passed on to `tapply()`!

apply()

apply()

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

Example: row means

- diastolic blood pressure in NHANES:

```
> head(ii[,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')])
  bpxdi1 bpxdi2 bpxdi3 bpxdi4
1     70     64     62     NA
2     88     88     82     NA
3     72     68     70     NA
4     70     54     56     NA
5     70     74     74     NA
6     70     70     72     NA
```

Example: row means

- how could we find each individual's mean measurement?
- we 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

Example: row means

- the following is better, conceptually:

```
> row1<-ii[1,c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')]
> row1<-as.numeric(row1)
> mean(row1,na.rm=TRUE)
[1] 65.33333
```

- here, the denominator is appropriate, because we've made use of built-in function `mean()` and its `na.rm` argument
- but, we've only found the mean for the first individual!
- we need to apply the `mean()` function to each row

Example: row means

- to apply the mean() function to each row, we can use apply():

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

- X: the data frame
- MARGIN: indicates whether to apply the function to the rows or the columns
- FUN: the function to apply

Example: row means

- without argument names:

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

Example: row means

- equivalently, using `rowMeans()`:

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

apply()-like functions

- 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`
  - `colSums()` for column sums, conceptually equivalent to using `apply()` with `MARGIN=2` and `FUN=sum`
- `apply()` is very useful nevertheless!

Example: Biostat 213 grades

- consider the following hypothetical homework scores for this course:

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

Example: Biostat 213 grades

- how can I implement the course's homework policy?
- I can define a function for each student's grade, and apply that function to each student:

```
> drop.lowest<-function(grades){mean(sort(grades)[-1])}
> apply(grades,1,drop.lowest)
[1] 92.33333 91.33333 92.00000 92.66667 93.00000
```

Example: grades

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

```
> apply(grades,1,function(grades){mean(sort(grades)[-1])})
[1] 92.33333 91.33333 92.00000 92.66667 93.00000
```

sapply()

sapply()

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

Example: square roots

```
> sapply(X=1:4,FUN=sqrt)
[1] 1.000000 1.414214 1.732051 2.000000
> sqrt(1:4)
[1] 1.000000 1.414214 1.732051 2.000000
```

- `X`: the vector
- `FUN`: the function to apply
- `sapply()` was not necessary in this example, as shown in the last line

Example: object types

- the object type of each of the first 10 columns of the NHANES data:

```
> sapply(X=1:10,FUN=function(ee){class(ii[,ee])})
 [1] "integer" "integer" "integer" "integer" "integer" "logical"
 [7] "integer" "integer" "integer" "integer"
```

- you could, in theory, do this with `apply()`, but it turns out that this does not work

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

- first, we define a function for the moving average:

```
> ma3<-function(year,y){mean(c(y[year],y[year-1],y[year-2]))}
```

- then, we repeatedly apply the function, beginning with the 3rd year:

```
> sapply(3:nrow(ll),ma3,y=ll$fgp)
[1] 0.4596667 0.4560000 0.4233333 0.4190000 0.4143333 0.4366667
[7] 0.4266667 0.4346667 0.3996667
```

Example: moving averages

- 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 0.8156667
[7] 0.8323333 0.8576667 0.8600000
```

Beyond this course

Other object types

- lists
- matrices and arrays

Regular expressions

- regular expressions for advanced character manipulation: `grep()` and `gsub()`
- you may consider *Handling Strings with R*

for() loops

- for() loops for iteratively running code

dplyr and tidyr

- the dplyr package for data manipulation
- the tidyr package for reshaping data

Statistical/epidemiological analysis

- the epitools package for measures of association
- `glm()` for generalized linear models (including logistic regression)
- the survival package for survival analysis (including Kaplan-Meier estimation and Cox proportional hazards models)
- the lmer package for mixed models
- the geepack package for generalized estimating equations
- the survey package for analysis of surveys
- the rmeta package for meta-analysis

Probability distributions

- various built-in functions exist for probability distributions
- for a list, see `help(Distributions)`
- for example, for normal distributions:
  - `rnorm()` for generating random numbers from normal distributions (useful for simulations or generating fake data)
  - `qnorm()` for finding z-scores corresponding to areas under the curve
  - `pnorm()` for finding areas under the curve

Probability distributions

- for example, for the z-score corresponding to a lower-tail area under the curve of 0.975:

```
> qnorm(0.975)
[1] 1.959964
```

Simulations

- for simulations, be sure to use `set.seed()`
- you may also consider using packages that take advantage of parallel computing, such as doParallel and doRNG

Exercise

Exercise

You may work in groups of 2–3. Use `nhanes.csv` for all of the following problems.

1. Use `tapply()` to find the median systolic blood pressure (`bpxsy1`) in each `bmicat` group. Preferably, first re-order the levels of `bmicat` so they are in scientific order.
2. Use one of the apply functions to find the maximum systolic blood pressure among the 4 possible measurements (`bpxsy1`, `bpxsy2`, `bpxsy3`, and `bpxsy4`) for the first 10 individuals in the data. You should obtain 10 values (each represents the maximum measurement for a unique individual). Hint: you may want to include indexing.

More on the next slide...

Exercise

3. Write a function that implements the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   The symbol $\pm$ means plus or minus. For example, 2 ± 3 means $2 - 3$ and $2 + 3$.

   The function should accept 3 arguments and output 1 value.

   Check your function with a few values of a, b, and c.

More on the next slide...

Exercise

4. Use `tapply()` to find the median systolic blood pressure (`bpxsy1`) in each `bpcat` group. Preferably, first re-order the levels of `bpcat` so they are in scientific order. Note that `bpcat` depends on `bpxsy1` (as well as `bpxdi1`) so you should notice a pattern.
5. Use one of the apply functions to output the number of missing values in each variable.

More on the next slide...

Exercise

6. The `format()` function can be used to format dates. For example:

```
> format(Sys.Date(),'%Y-%m-%d')
[1] "2019-11-18"
```

Write a function that will output a formatted version of `Sys.Date()` (a function that outputs the current date). The function should accept an argument that indicates whether the date should be outputted in a month-day-year format (as in the United States) or a day-month-year format (the convention in most other places). You might consider the list of codes shared in Module 7. Here's an example function:

```
> today()
[1] "11/18/2019"
> today(us=FALSE)
[1] "18 November 2019"
```