Programming Structures - Intro

Goals for this session:

* Understand the for-loop structure and how to use it to:

* "Walk" through a vector while accumulating a sum, computing a product, or some other operation.

* "Walk" though a matrix by row, (or column), while accumulating a sum, computing a product or some other arithmetic operation.

* "Walk" through a data frame by row to compute something. Also process the results of a previous "split" operation.

* Understand the if statement and how to branch based on the value of a vector or matrix element.

* Also use the if statement in conjunction with the for loop to do some processing.

Programming Structures - Intro

Some suggestions for the upcoming weeks:

1) Put the statements in the Editor window of RStudio and perfect them there. You can highlight sections of code and hit the "run" button.

2) You will most definitely make mistakes when writing loops. It is guaranteed. Better to get familiar with the most common mistakes ahead of time.

3) Work through the labs. Functions, Week 7, use control statements and loops

4) The next assignment will assume facility with these structures.

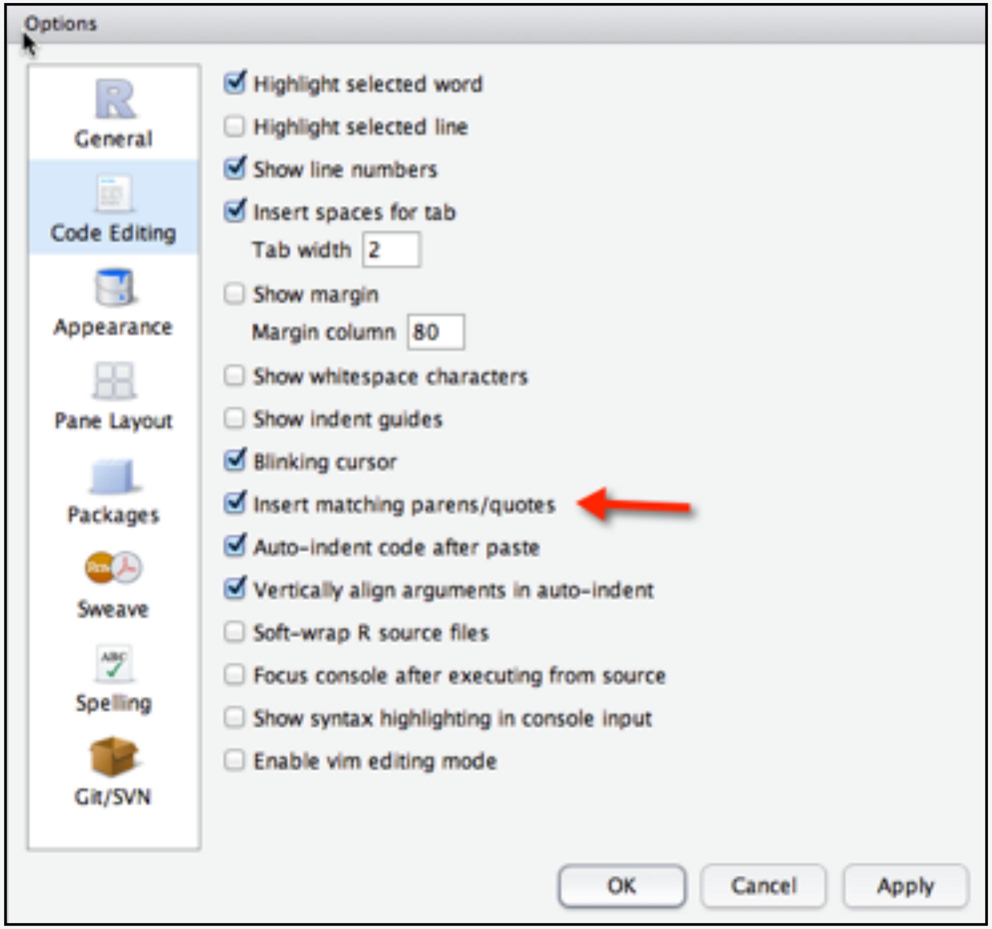
Programming Structures - Intro

```r
# Compute the grades

score = c(74,68,98,90,100,67,59)
for (ii in 1:length(score)) {
  if (score[ii] == 100) {
      grade = "A+"
  } else if (score[ii] >= 90) {
      grade = "A"
  } else if (score[ii] >= 80) {
      grade = "B"
  } else if (score[ii] >= 70) {
      grade = "C"
  } else if (score[ii] >= 60) {
      grade = "D"
  }
  else {
    grade = "F"
  }
  print(grade)
}
```

Programming Structures - Intro

Go to Preferences -> Code Editing to turn on "insert matching parens/braces"



Programming Structures - Intro

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if`, `else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an interation of a loop
- `return`: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresisons.

Peng, Roger

Programming Structures - for

This is a looping construct that let's you do some things for a specific number of times. "name" is some index variable that takes on values returned by "expr_1", which is almost always some type of sequence. It could represent the length of a vector or a row of a matrix.

```
for (name in expr_1) {
   expr_2
}

x = 1:3
for (ii in 1:3) {
  print(ii)
}

[1] 1
[1] 2
[1] 3
```

Programming Structures - for with vectors

Better to generalize this - use the length function so it will work with any vector

```
x = 1:3
for (ii in 1:length(x)) {
  print(ii)
}

[1] 1
[1] 2
[1] 3
```

We could go backwards also:

```
x = 1:3
for (ii in length(x):1) {    # We start with the last element number
   print(ii)
}
[1] 3
[1] 2
[1] 1
```

Programming Structures - for with vectors

And we could make the output a little prettier also:

```
x = 10:12
for (ii in 1:length(x)) {
  cat("The value of x element number",ii,"is",x[ii],"\n")
}

The value of x element number 1 is 10
The value of x element number 2 is 11
The value of x element number 3 is 12
```

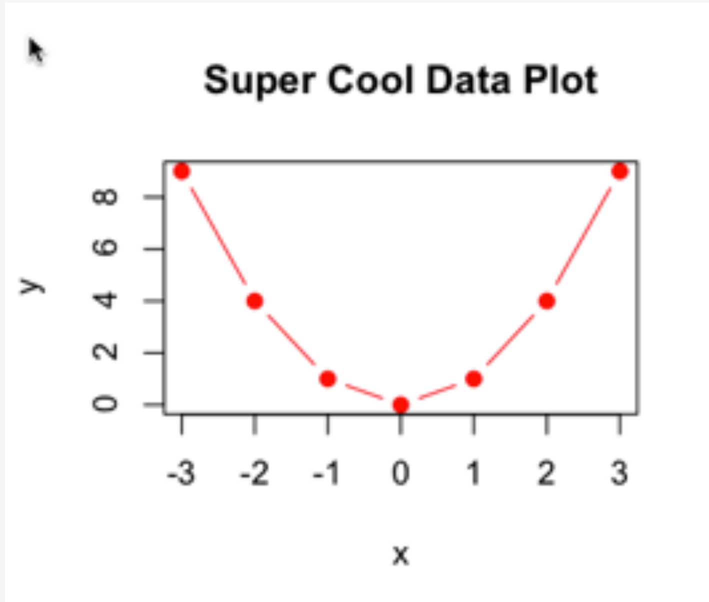
Programming Structures - for with vectors

Let' look at the situation where we have a bunch of x values that we want to provide as input to some function. That is - We want to generate y values for plotting x vs y. Here let's plug the x values into the function x^2. (The resulting plot will be a parabola).

```
y = vector()  # A blank vector
x = -3:3
for (ii in -3:length(x)) {
  y[ii] = x[ii]^2
}

x
[1] -3 -2 -1  0  1  2  3

y
[1]  9  4  1  0  1  4  9
```



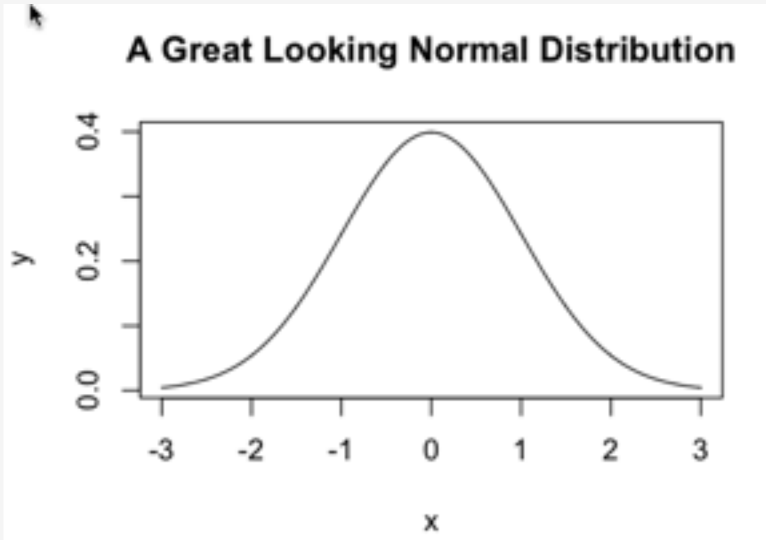

```
plot(x,y,main="Super Cool Data Plot",type="b",pch=19,col="red")
```

Programming Structures - for with vectors

```
y = vector()
x = seq(-3,3,by=0.005)      # seq let's us specify and increment
for (ii in -3:length(x)) {
  y[ii] = dnorm(x[ii])
}

length(x)
[1] 1201

plot(x,y,main="A Great Looking Normal Distribution",type="l")
```



Programming Structures - for with vectors

We frequently use for-loops to do some arithmetic - like add things up.

```
x = 1:3
mysum = 0
for (ii in 1:length(x)) {
  mysum = mysum + x[ii]
}

mysum
[1] 6

Here is what is happening as the loop is executing:

mysum = 0
for (ii in 1:length(x)) {
  mysum = mysum + x[ii]
  cat("ii is currently",ii,"and mysum is",mysum,"\n")
}

ii is currently 1 and mysum is 1
ii is currently 2 and mysum is 3
ii is currently 3 and mysum is 6
```

Programming Structures - for with vectors

Now - we would usually use vector arithmetic to do this but there are situations when a for loop will be fine also.

```
mysum = 0
for (ii in 1:length(x)) {
  mysum = mysum + x[ii]
}
avg = mysum / length(x)

cat("The average of this vector is:", avg ,"\n")
[1] "The average of this vector is: 2"


all.equal(avg, mean(x))
[1] TRUE
```

Programming Structures - for with vectors

```
This will work on large vectors too

x = rnorm(1000,20,4)  # 1,000 random elements from a N(20,4)

mysum = 0
for (ii in 1:length(x)) {
  mysum = mysum + x[ii]
}
avg = mysum / length(x)
cat("The average of this vector is:",avg,"\n")

[1] "The average of this vector is: 20.1320691898645"

We could clean up the output a little bit

cat("The average of this vector is:",round(avg,2),"\n")
[1] "The average of this vector is: 20.13"
```

Programming Structures - for with vectors

Here is an example where we compute the product of all vector elements (note there is an R function for this called 'prod')

```
x = 1:6
myprods = 1
for (ii in 1:length(x)) {
  myprods = myprods * x[ii]
}

myprods
[1] 720

prod(x)
[1] 720

all.equal(myprods, prod(x))
[1] TRUE
```

Programming Structures - for with vectors

Given a vector find the smallest value without using the "min" function:

```
set.seed(188)
x = rnorm(1000)  # 1,000 random elements from a N(20,4)

mymin = x[1] # Set the min to the first element of x. Unless we are
             # very lucky then this will change as we walk through
             # the vector

for (ii in 1:length(x)) {
  if (x[ii] < mymin) {
     mymin = x[ii]
  }
}

mymin
[1] -3.422185

min(mymin)       # The internal R function matches what we got
[1] -3.422185
```

Programming Structures - for with dataframes

We can loop through data frames also. Consider the following data frame that contains the number of hours worked that week for an employee. There is also the hourly wage information so we can compute the weekly pay.

To figure an employee's pay check we multiply total work Hours by the Wage:

```
tt
      Hours Wage
Frank    40 30.0
John     42 31.5
Lisa     38 26.5

for (row in 1:nrow(tt)) {
     pay.check = tt[row,'Hours'] * tt[row,'Wage']
     cat(rownames(tt)[row]," gets $",pay.check," this week \n")
}


Frank  gets $ 1200  this week
John  gets $ 1323  this week
Lisa  gets $ 1007  this week
```

Programming Structures - for with dataframes

We can loop through data frames also. Let's see if we can compute the mean of the MPG for all cars. Note that we use the nrow function to get the number of rows to loop over.

```
mpgsum = 0
for (row in 1:nrow(mtcars)) {
  mpgsum = mpgsum + mtcars[row,"mpg"]
}

mpgmean = mpgsum/nrow(mtcars)   # Divide the sum by the # of records

cat("Mean MPG for all cars is:",mpgmean,"\n")

Mean MPG for all cars is: 20.09062

all.equal(mpgmean,mean(mtcars$mpg))
[1] TRUE
```

Programming Structures - for with dataframes

Remember the split command ? We can work with the output of that also. Relative to mtcars we let's split up the data frame by cylinder number, which is (4,6, or 8).

```
mysplits = split(mtcars, mtcars$cyl)

str(mysplits, max.level=1)
List of 3
 $ 4:'data.frame':    11 obs. of  11 variables:
 $ 6:'data.frame':    7 obs. of  11 variables:
 $ 8:'data.frame':    14 obs. of  11 variables:
```

We get back a list that contains 3 elements each of which has a data frame corresponding to the number of cylinders. If we wanted to we could summarize each of these data frame elements using a for loop

Programming Structures - for with dataframes

```
mysplits
$`4`
              mpg cyl  disp hp drat    wt  qsec vs am gear carb
Merc 240D     24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
Merc 230      22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2
Toyota Corona 21.5   4 120.1 97 3.70 2.465 20.01  1  0    3    1
..
..

$`6`
               mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
..
..

$`8`
                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
..
..
```

Programming Structures - for with dataframes

```
mysplit = split(mtcars,mtcars$cyl)

for (ii in 1:length(mysplit)) {
   print(nrow(mysplit[[ii]]))
}

[1] 11
[1] 7
[1] 14

# This is equivalent to

lapply(mysplit, nrow)
$`4`
[1] 11

$`6`
[1] 7

$`8`
[1] 14
```

Programming Structures - for with dataframes

```
mysplit = split(mtcars,mtcars$cyl)

for (ii in 1:length(mysplit)) {
   splitname = names(mysplit[ii])
   cat("mean for",splitname,"cylinders is",mean(mysplit[[ii]]$mpg),"\n")
}
mean for 4 cylinders is 26.66364
mean for 6 cylinders is 19.74286
mean for 8 cylinders is 15.1

# This is basically equivalent to

lapply(mysplit, function(x) mean(x$mpg))
$`4`
[1] 26.66364

$`6`
[1] 19.74286

$`8`
[1] 15.1
```

Programming Structures - for with dataframes

What about looping over each split and pulling out only those cars with an manual transmission ? (am == 1)

```
data(mtcars)

mysplit = split(mtcars,mtcars$cyl)

mylist = list() # Setup a blank list to contain the subset results

for (ii in 1:length(mysplit)) {
  mylist[[ii]] = subset(mysplit[[ii]], am == 1)
}

mylist

# Equivalent to:

lapply(mysplit, subset, am == 1)
```

Programming Structures - for with dataframes

What about looping over each split and sampling two records from each group ?

```
for (ii in 1:length(mysplits)) {
    recs = sample(1:nrow(mysplits[[ii]]),2,F)
    print(mysplits[[ii]][recs,])
}
             mpg cyl disp hp drat    wt  qsec vs am gear carb
Honda Civic 30.4   4 75.7 52 4.93 1.615 18.52  1  1    4    2
Fiat 128    32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1

              mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4

                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
```

Programming Structures - for with dataframes

What about looping over each split and sampling two records from each group ?

```
lapply(mysplit, function(x) {
                      recs = sample(1:nrow(x),2,F)
                      return(x[recs,])
                  })
$`4`
             mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Volvo 142E  21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2

$`6`
          mpg cyl  disp  hp drat   wt  qsec vs am gear carb
Merc 280 19.2   6 167.6 123 3.92 3.44 18.30  1  0    4    4
Valiant  18.1   6 225.0 105 2.76 3.46 20.22  1  0    3    1

$`8`
                  mpg cyl disp  hp drat    wt  qsec vs am gear carb
Duster 360       14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
Pontiac Firebird 19.2   8  400 175 3.08 3.845 17.05  0  0    3    2
```

Programming Structures - for with dataframes

Let's say we want to plot MPG vs. Weight for each cylinder group. Check it out:

```
mysplits = split(mtcars, mtcars$cyl)

par(mfrow=c(1,3))    # This relates to plotting

for (ii in 1:length(mysplits)) {
  hold = mysplits[[ii]]
  plot(hold$wt, hold$mpg, pch = 18, main=paste("MPG vs. Weight for",
       names(mysplits[ii]), "cyl",sep=" "))
}

NOTE:
names(mysplits[1])
[1] "4"
names(mysplits[2])
[1] "6"
names(mysplits[3])
[1] "8"
```

Programming Structures - for with dataframes

Technically we don't need to create the "hold" variable. It just makes it easier to see what is going on. But it isn't necessary:

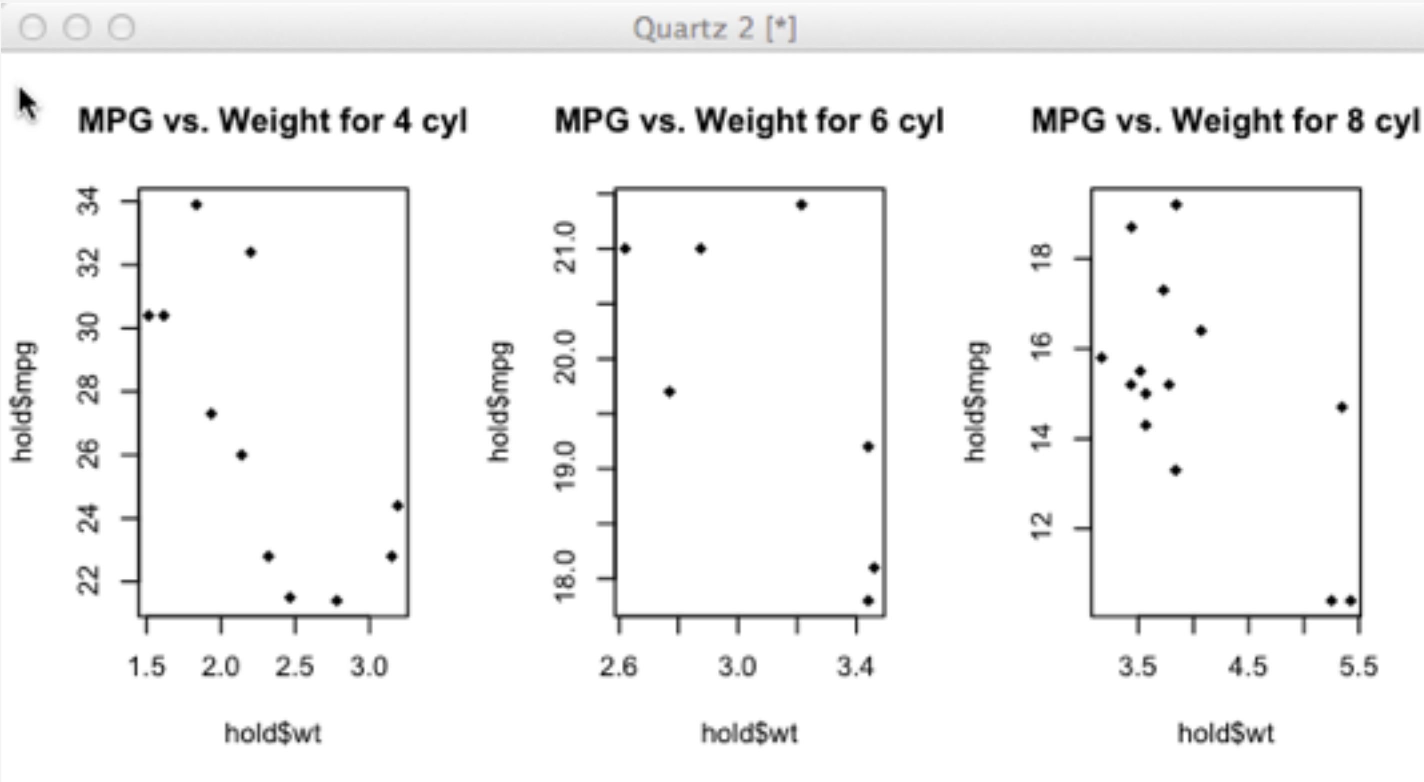
```
mysplits = split(mtcars, mtcars$cyl)

par(mfrow=c(1,3))    # This relates to plotting

for (ii in 1:length(mysplits)) {
  plot(mysplits[[ii]]$wt, mysplits[[ii]]$mpg, pch = 18,
       main=paste("MPG vs. Weight for",
       names(mysplits[ii]), "cyl",sep=" "))
}

NOTE:
names(mysplits[1])
[1] "4"
names(mysplits[2])
[1] "6"
names(mysplits[3])
[1] "8"
```

Programming Structures - for with dataframes



Programming Structures - for with dataframes

Our loop indices do not have to be numeric. Consider the mtcars data frame. We can use the names of the columns to loop through things:

```
for (colname in names(mtcars)) {
  cat("column ", colname,
      " of mtcars has a class of ",class(mtcars[,colname]),"\n")
}

column  mpg  of mtcars has a class of  numeric
column  cyl  of mtcars has a class of  numeric
column  disp  of mtcars has a class of  numeric
column  hp  of mtcars has a class of  numeric
column  drat  of mtcars has a class of  numeric
column  wt  of mtcars has a class of  numeric
column  qsec  of mtcars has a class of  numeric
column  vs  of mtcars has a class of  numeric
column  am  of mtcars has a class of  numeric
column  gear  of mtcars has a class of  numeric
column  carb  of mtcars has a class of  numeric

This is the equivalent to the sapply(mtcars, class) command
```

Programming Structures - for with matrices

```
set.seed(123)

mymat = matrix(round(rnorm(6),2),3,2)

      [,1] [,2]
[1,] -0.56 0.07
[2,] -0.23 0.13
[3,]  1.56 1.72


for (row in 1:nrow(mymat)) {
 for (col in 1:ncol(mymat)) {
  cat("The value at row",row ,"and column", col,"is",mymat[row,col],"\n")
  }
}

The value at row 1 and column 1 is -0.56
The value at row 1 and column 2 is 0.07
The value at row 2 and column 1 is -0.23
The value at row 2 and column 2 is 0.13
The value at row 3 and column 1 is 1.56
The value at row 3 and column 2 is 1.72
```

Programming Structures - for with matrices

Let's say we wanted to sum all the rows:

```
      [,1] [,2]
[1,] -0.56 0.07
[2,] -0.23 0.13
[3,]  1.56 1.72

rowtotal = 0                  # initialize a variable to hold row total
for (row in 1:nrow(mymat)) {
  for (col in 1:ncol(mymat)) {
    rowtotal = rowtotal + mymat[row, col]
  }
  print(rowtotal)
  rowtotal = 0
}

[1] -0.49
[1] -0.1
[1] 3.28

                    # same values as:
apply(mymat,1,sum)
[1] -0.49 -0.10  3.28
```

Programming Structures - for with matrices

Let's create a new matrix based on the old one. Here we subtract each element from the mean of its respective column:

```
set.seed(123)

mymat = matrix(round(rnorm(6),2),3,2)

newmat = matrix(rep(0,6),3,2) # Setup a new mat of the same size

for (col in 1:ncol(mymat)) {
  for (row in 1:nrow(mymat)) {
    newmat[row,col] = mymat[row,col] - mean(mymat[,col])
  }
}

newmat
           [,1]  [,2]
[1,] -0.8166667 -0.57
[2,] -0.4866667 -0.51
[3,]  1.3033333  1.08
```

Programming Structures - for with matrices

Let's create a new matrix based on the old one. Here we subtract each element from the mean of its respective column:

```
newmat
           [,1]  [,2]
[1,] -0.8166667 -0.57
[2,] -0.4866667 -0.51
[3,]  1.3033333  1.08


# This matches this command:

apply(mymat, 2, function(x) x - mean(x))
           [,1]  [,2]
[1,] -0.8166667 -0.57
[2,] -0.4866667 -0.51
[3,]  1.3033333  1.08
```

Programming Structures - if

This is an easy structure. It tests for a conditions and, based on that, executes a specific block of code.

```
if (logical_expression) {
   do something
   ...
}

if (logical_expression) {
  do something
  ..
} else {
  do something else
  ...
}
```

Programming Structures - if

```
x = 3

x
[1] 3

if (is.numeric(x)) {
   print("x is a number")
}

[1] "x is a number"

if (x != 3) {
     print("x is not equal to 3")
} else {
    print("guess what ? x is in fact equal to 3")
}
[1] "guess what ? x is in fact equal to 3"
```

Programming Structures - if

This is an easy structure. It tests for a conditions and, based on that, executes a specific block of code.

```
some.num = 3


if (some.num < 3) {          # A more involved if statement
     print("Less than 3")
} else if (some.num > 3) {
     print("Greater than 3")
} else {
     print("Must be equal to 3")
}
[1] "Must be equal to 3"
```

Programming Structures - error checking

Let’s extend the example just a little bit to make sure that x and y are numeric as it doesn’t make sense to check their identity unless the are both numbers. This is called "error checking" and we do it a lot when writing functions.

```
x=4 ; y=5

if (!is.numeric(x) | !is.numeric(y)) {
   stop("I need numeric values to do this")
} else {
   if (x == y) {
       print("Equal")
   } else {
      print("Not equal")
   }
}

[1] "Not equal"
```

Programming Structures - error checking

Let's extend the example just a little bit to make sure that x and y are numeric as it doesn't make sense to check their identity unless the are both numbers. This is called "error checking" and we do it a lot when writing functions.

```
x=4 ; y="5"

if (!is.numeric(x) | !is.numeric(y)) {
   stop("I need numeric values to do this")
} else {
   if (x == y) {
       print("Equal")
   } else {
      print("Not equal")
   }
}

Error: I need numeric values to do this
```

Programming Structures - for and if

So if statements are usually part of some other structure - like within a for-loop:

```
score = c(74,68,98,90,100,67,59)

for (ii in 1:length(score)) {
  if (score[ii] == 100) {
      grade = "A+"
  } else if (score[ii] >= 90) {
      grade = "A"
  } else if (score[ii] >= 80) {
      grade = "B"
  } else if (score[ii] >= 70) {
      grade = "C"
  } else if (score[ii] >= 60) {
      grade = "D"
  }
  else {
    grade = "F"
  }
  print(grade)
}
[1] "C"
[1] "D"
[1] "A"
[1] "A"
[1] "A+"
[1] "D"
[1] "F"
```

Programming Structures - for and if

So if statements are usually part of some other structure - like within a for-loop:

```
set.seed(123)
x = round(runif(10,1,20))
[1]  6 16  9 18 19  2 11 18 11 10

for (ii in 1:length(x)) {
    if (x[ii] %% 2 == 0) {
        print(TRUE)
    }
    else {
        print(FALSE)
    }
}

[1] TRUE
[1] TRUE
[1] FALSE
[1] TRUE
[1] FALSE
[1] TRUE
[1] FALSE
[1] TRUE
[1] FALSE
[1] TRUE
```

Programming Structures - for and if

We can mimic the bracket notation approach here:

```
set.seed(123)
x = round(runif(10,1,20))
 [1]  7 13 16  8 13 19  9 11 13 11

logvec = vector()             # Setup an empty vector
for (ii in 1:length(x)) {
    if (x[ii] %% 2 == 0) {
        logvec[ii] = TRUE
    }
    else {
        logvec[ii] = FALSE
    }
}
logvec

logvec
[1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE


x[logvec]
[1]  6 16 18  2 18 10
```

Programming Structures - for and if

One can easily "break" out of a for loop based on some condition. Normally you should clean your data before processing it but perhaps you thought you did. Let's say that you are processing elements of a vector and if you encounter a value of NA then you want to stop the for loop.

```
my.vec = c(1,2,3,NA,5,6,7,8,9,10)

for (ii in 1:length(my.vec)) {
    if (is.na(my.vec[ii])) {
       break
    }
    cat("element is ",ii,"\n")
}

element is  1
element is  2
element is  3
```

Programming Structures - for and if

Here we want to "catch" the missing value and then "skip over it". To do this we would use the "next" statement.

```
my.vec = c(1,2,3,NA,5,6,7,8,9,10)


for (ii in 1:length(my.vec)) {
    if (is.na(my.vec[ii])) {
       next
    }
    cat("element is ",ii,"\n")
}

element is  2
element is  3
element is  5
element is  6
element is  7
element is  8
element is  9
element is  10
```

Programming Structures - for and if

You will see for-loops that contain other programming constructs such as if/else statements.

```
x    |  <= 0     > 0
-----------------------
f(x) |  x^2      x^3

set.seed(123)
myvec = round(rnorm(10,1,2),2)
[1] -0.12  0.54  4.12  1.14  1.26  4.43  1.92 -1.53 -0.37  0.11

for (ii in 1:length(myvec)) {
   if (myvec[ii] <= 0) {
     myvec[ii] = myvec[ii]^2
   } else {
     myvec[ii] = myvec[ii]^3
   }
}

myvec
[1]  0.014400  0.157464 69.934528  1.481544  2.000376 86.938307  7.077888
[8]  2.340900  0.136900  0.001331
```

Programming Structures - for and if

Here is an example that will be useful when processing things like genetic sequences. Let's say we have a string of text we wish to "encode" by changing all vowels to something else. This isn't a tough code to break but let's see what is involved. In our code:

We'll change a to s,
e to t,
i to u,
o to v,
u to w

So a string like :

sequence = "Hello my name is Ed. Happy to meet you"

would come out like:

```
"Htllv my nsmt us td. Hsppy tv mttt yvw"
```

Programming Structures - for and if

```
sequence = "Hello my name is Ed. Happy to meet you"

seq = unlist(strsplit(sequence,""))

[1] "H" "e" "l" "l" "o" " " "m" "y" " " "n" "a" "m" "e" " " "i" "s" " " "E"
"d" [20] "." " " "H" "a" "p" "p" "y" " " "t" "o" " " "m" "e" "e" "t" " " "y"
"o" "u"

sequence = "Hello my name is Ed. Happy to meet you"
seq = unlist(strsplit(sequence,""))
for (ii in 1:length(seq)) {
         if ( seq[ii] == "a") {
               seq[ii] = "s"
         } else if (seq[ii] == "e") {
               seq[ii] = "t"
         ..
         ..
         and so on
}
# Use the "paste collapse" trick to get the character vector back into a
# string

"Htllv my nsmt us Ed. Hsppy tv mttt yvw"
```