A brief intro to R objects and functions

Aaron Adamack

January 14 2019

## Math in R

Math in R is pretty much the same as what you’ll see in most other computer programs.

Add:

10+17

## [1] 27

Subtact:

10-17

## [1] -7

Multiply:

10\*17

## [1] 170

Divide:

10/17

## [1] 0.5882353

Exponents:

10^17

## [1] 1e+17

If you want to use brackets to control the order of operations:

(5+8)\*(2+3/8)

## [1] 30.875

If you want to leave comments in your code you can use “#” to hide any text following the hash mark on a line.

# 4\*12 # this won't run  
3\*12 # this will run

## [1] 36

Some other math operations that are frequently used:

# the square root of a number  
sqrt(25)

## [1] 5

# the natural log of a number  
log(100)

## [1] 4.60517

# the base 10 log of a number  
log10(100)

## [1] 2

# to get the quotient when dividing a number (note that you may get strange results if you use real numbers rather than intergers)  
35 %/% 3

## [1] 11

# to get remainder when dividing a number (note that you may get strange results if you use real numbers rather than intergers)  
35 %% 3

## [1] 2

There are many other basic math functions for things line sine, cosine, tangent, etc. If you need them, they should be easy to find in any basic R guide.

## Objects

Often you will want to store various types of information when working with data, numbers and files in R. Typically you will use the “<-” function to place information into an object. You’re going to have to come up with a name for the object. There are basically two rules about naming an object: 1.) A name can’t start with a number or symbol 2.) Names can’t include the special symbols ^, !, $, @, +, -, /, ,, or \*

Some examples of good and bad names (ignoring style)

cat<-5

Cat<-10

CAT<-9

C.A.T.<--53

cAt<-55

1cat<-5

## Error: <text>:1:2: unexpected symbol  
## 1: 1cat  
## ^

cat!<-16

## Error: <text>:1:4: unexpected '!'  
## 1: cat!  
## ^

cat-15<-10

## Error in cat - 15 <- 10: could not find function "-<-"

It is worth noting that when R is working with objects or making comparisons between text in various forms that it is case sensitive. Going back to our object examples just above…

cat

## [1] 5

Cat

## [1] 10

CAT

## [1] 9

cAt

## [1] 55

This is particularly important when you get into things like merging objects, subsetting data sets, looking at numbers by a group name, etc.

##Vectors

Often, you will have a bunch of values that you’d like to group together. For example, you might have ran an experiment where you made a measurement every 5 minutes for several hours. You could store these numbers in vectors. In R, you can place values into a vector using the function “c”. As a quick aside, functions are generally followed by round parenthesis which are used to indicate the arguments going into a function.

As a basic example, here is a vector that contains the time (in minutes) of the first half hour of measurements:

c(0,5,10,15,20,25,30)

## [1] 0 5 10 15 20 25 30

# note that you can use spaces if you want...  
c(0, 5, 10, 15, 20, 25, 30)

## [1] 0 5 10 15 20 25 30

It usually isn’t very useful to just place some numbers in a vector as you generally can’t use it for very much. Vectors become a lot more useful if you place them into an object.

# as these are experiment times, let's put the times from above into an object called times  
  
times<-c(0,5,10,15,20,25,30)  
  
# note that unlike above, after you fill the vector with numbers it doesn't repeat the numbers back. That's because when you don't put values into an object, R puts them onto the screen. When you put them into an object, the values are placed into the computer's memory instead.   
  
# to view the values that you placed into the object times, simply type the objects name and run that line   
times

## [1] 0 5 10 15 20 25 30

Typing values into a vector is often an inefficient way of doing things. Imagine if you had to type in experiment measurement times for every 5 minutes for a 4 hour long experiment. It probably wouldn’t be an enjoyable experience. R has a function that makes this easier…

# instead of doing something like  
#times<-c(0,5,10,15,20,25,30,...)  
  
# we can do the following:  
times<-seq(from=0,to=(4\*60),by=5) # 4 hours \* 60 minutes/hour  
  
# which produces the following  
times

## [1] 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80  
## [18] 85 90 95 100 105 110 115 120 125 130 135 140 145 150 155 160 165  
## [35] 170 175 180 185 190 195 200 205 210 215 220 225 230 235 240

# So how would you figure out how the seq function works? To find out how a function works, you access the help file using any of the following commands:  
  
help(seq)

## starting httpd help server ... done

?seq  
  
??seq   
  
# I'd suggest using help or ? in most cases. The ?? command becomes useful if the first two are unsuccessful in finding the helpfile, perhaps because you haven't loaded the correct R package yet or if there are multiple functions with the same name.

There are a few other ways of building vectors…

# if your values are straight forward, you can do something like this to build a vector  
  
simpleseq<-0:250  
simpleseq

## [1] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
## [18] 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33  
## [35] 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  
## [52] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67  
## [69] 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84  
## [86] 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101  
## [103] 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118  
## [120] 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135  
## [137] 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152  
## [154] 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169  
## [171] 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186  
## [188] 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203  
## [205] 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220  
## [222] 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237  
## [239] 238 239 240 241 242 243 244 245 246 247 248 249 250

# alternatively, if you want to create an empty vector (useful if you are going to run a loop and need somewhere to place your results)  
  
emptyvec<-rep(x=NA, length.out=20) # note that NA is often used to represent "no data"  
emptyvec

## [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA

# you may have a situation where you need a repeating sequence of numbers: say 1 to 4, five times:  
  
repeating<-rep(1:4,times=5)  
repeating

## [1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4

#note that if we had used the function argument "length.out" we would produce a different vector:  
  
repeating2<-rep(1:4,length.out=5)  
repeating2

## [1] 1 2 3 4 1

# another scenario is that we might need to repeat each of the values a certain number of times (picture trying to create a results table for an experiment where we measure heights of male and female kids at 5 ages)  
  
repeating3<-rep(c("F","M"),each=5)  
repeating3

## [1] "F" "F" "F" "F" "F" "M" "M" "M" "M" "M"

# sometimes you may not care what the actual values are that are in your dataset or you simply need some random numbers. R has built in functions to generate random numbers from several distributions including the normal, Poisson, Student's t, uniform, binomial and Weibull distributions.   
  
# to build a vector with 100 numbers drawn from a uniform distribution between 0 and 100 you would do the following:  
  
set.seed(1526) # note set.seed sets the starting seed for the pseudo-random number generator making it so we can all get the same set of random numbers.   
uniformdist<-runif(n=100,min=0,max=100) # n is the number of random numbers we want to generate min and max give the range  
  
# Similarly, to draw 100 numbers from a normal distribution with mean 50 and a standard deviation of 25 you would do the following:  
  
set.seed(2851)  
normaldist<-rnorm(n=100, mean=50, sd=25)  
  
# if you want the details on how to draw numbers from other distributions, check out the "See Also" for the help page for runif (accessed by running: help(runif))

##Functions for vectors…

R has a plethora of functions that can be run on vectors. Below I will go through some of the ones that I use most frequently

# often you will want to know how many elements are in a vector (e.g. how long is the vector). To find out, simply use the length function  
length(uniformdist)

## [1] 100

# to find the maximum value of a vector, use the max function  
max(normaldist)

## [1] 111.8279

# to find the minimum value of a vector, use the min function  
min(normaldist)

## [1] -0.9264303

# the mean can be found by using the mean function:  
mean(normaldist)

## [1] 53.99709

# you can get the standard deviation using the sd function:  
sd(normaldist)

## [1] 23.87416

# the median...  
median(normaldist)

## [1] 53.68117

# I'll note that oftentimes you'll run into a vector where some elements are missing. This can cause some of these functions to "blow-up". For example, if I change one of the values in the normaldist vector to missing...  
mean(normaldist)

## [1] 53.99709

storevalue<-normaldist[50]  
normaldist[50]<-NA  
mean(normaldist)

## [1] NA

# to solve this problem, you can tell mean (and many other functions) to ignore missing values. To do this, the mean function can be changed to:  
mean(normaldist,na.rm=TRUE)

## [1] 53.81325

# just restoring the value I changed  
normaldist[50]<-storevalue  
mean(normaldist)

## [1] 53.99709

# I would suggest that you don't do this as standard practice. If you are unexpectedly getting NA results from some of these functions it's a hint that you've made a mistake somewhere along your analysis pathway. By going backwards to see where the NA results start appearing, you can often figure out where things went wrong and the start working on fixing it.   
  
# if you want to order the numbers in your vector, there are at least a couple of options.   
# to just sort the numbers, use sort...  
sort(normaldist, decreasing = FALSE) # numbers sorted in increasing order

## [1] -0.9264303 1.7989654 7.6920140 11.0007620 13.1929871  
## [6] 14.7063483 15.1420474 15.3665612 16.9549046 20.9386146  
## [11] 20.9516374 21.1991956 26.5687756 27.3038881 29.6322860  
## [16] 30.1996253 30.2313136 31.1097462 31.1368118 32.4870698  
## [21] 32.9932958 34.1909996 35.7750262 35.7935768 38.5603706  
## [26] 40.0898246 40.4228540 41.3928951 41.4144082 42.6828333  
## [31] 43.0044182 43.3889239 44.8995551 45.1009473 45.2327090  
## [36] 45.3495038 46.2332691 46.5297536 47.6353144 48.3404713  
## [41] 48.9134070 49.6233012 50.5691095 51.0473180 51.3851639  
## [46] 52.4235611 52.9447824 53.2324236 53.2592412 53.4220617  
## [51] 53.9402693 54.1673112 54.7597759 54.8711741 54.9263167  
## [56] 55.6166600 56.2108414 56.3743118 56.4395433 56.6547406  
## [61] 59.2657582 60.1741199 61.0223365 62.4845789 62.5934706  
## [66] 65.4205304 65.8535476 66.0652713 66.8441539 66.9373366  
## [71] 67.8789765 68.2434783 68.9559228 68.9956160 71.9383452  
## [76] 72.1975034 72.7003327 73.8614206 74.2223608 75.1251620  
## [81] 76.0903847 76.9479162 77.0914169 79.3797609 79.7624343  
## [86] 80.3857815 81.8136356 82.3347046 82.4418694 84.1495018  
## [91] 85.1182857 85.9530114 89.0610827 90.0144599 90.3647350  
## [96] 91.1894095 97.5609287 97.9817591 102.9606334 111.8279245

sort(normaldist, decreasing = TRUE) # numbers sorted in decreasing order

## [1] 111.8279245 102.9606334 97.9817591 97.5609287 91.1894095  
## [6] 90.3647350 90.0144599 89.0610827 85.9530114 85.1182857  
## [11] 84.1495018 82.4418694 82.3347046 81.8136356 80.3857815  
## [16] 79.7624343 79.3797609 77.0914169 76.9479162 76.0903847  
## [21] 75.1251620 74.2223608 73.8614206 72.7003327 72.1975034  
## [26] 71.9383452 68.9956160 68.9559228 68.2434783 67.8789765  
## [31] 66.9373366 66.8441539 66.0652713 65.8535476 65.4205304  
## [36] 62.5934706 62.4845789 61.0223365 60.1741199 59.2657582  
## [41] 56.6547406 56.4395433 56.3743118 56.2108414 55.6166600  
## [46] 54.9263167 54.8711741 54.7597759 54.1673112 53.9402693  
## [51] 53.4220617 53.2592412 53.2324236 52.9447824 52.4235611  
## [56] 51.3851639 51.0473180 50.5691095 49.6233012 48.9134070  
## [61] 48.3404713 47.6353144 46.5297536 46.2332691 45.3495038  
## [66] 45.2327090 45.1009473 44.8995551 43.3889239 43.0044182  
## [71] 42.6828333 41.4144082 41.3928951 40.4228540 40.0898246  
## [76] 38.5603706 35.7935768 35.7750262 34.1909996 32.9932958  
## [81] 32.4870698 31.1368118 31.1097462 30.2313136 30.1996253  
## [86] 29.6322860 27.3038881 26.5687756 21.1991956 20.9516374  
## [91] 20.9386146 16.9549046 15.3665612 15.1420474 14.7063483  
## [96] 13.1929871 11.0007620 7.6920140 1.7989654 -0.9264303

# alternatively, if you want to see the ranking of the values, use order  
order(normaldist, decreasing = FALSE) # numbers ranked in increasing order

## [1] 29 18 51 65 73 98 42 80 32 14 17 97 10 25 84 26 78  
## [18] 45 19 68 9 47 11 95 74 22 23 76 70 43 96 41 34 63  
## [35] 83 15 58 82 52 94 100 4 60 33 54 53 99 57 75 2 81  
## [52] 93 89 86 40 16 55 7 5 35 67 64 48 37 21 46 31 20  
## [69] 39 28 59 24 56 62 36 50 30 1 38 3 90 79 8 44 85  
## [86] 87 27 88 72 92 49 12 71 61 77 91 66 6 13 69

order(normaldist, decreasing = TRUE) # numbers ranked in decreasing order

## [1] 69 13 6 66 91 77 61 71 12 49 92 72 88 27 87 85 44  
## [18] 8 79 90 3 38 1 30 50 36 62 56 24 59 28 39 20 31  
## [35] 46 21 37 48 64 67 35 5 7 55 16 40 86 89 93 81 2  
## [52] 75 57 99 53 54 33 60 4 100 94 52 82 58 15 83 63 34  
## [69] 41 96 43 70 76 23 22 74 95 11 47 9 68 19 45 78 26  
## [86] 84 25 10 97 17 14 32 80 42 98 73 65 51 18 29

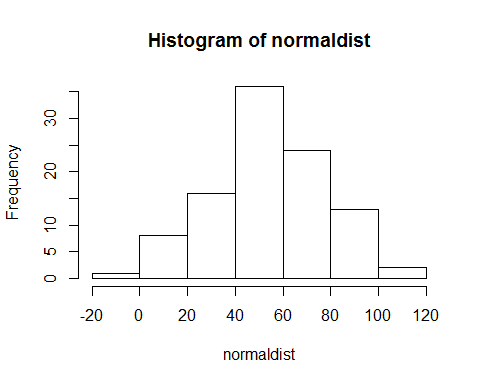
# the rankings from order can be used to produce results similar to sort...  
normaldist[order(normaldist, decreasing = FALSE)]

## [1] -0.9264303 1.7989654 7.6920140 11.0007620 13.1929871  
## [6] 14.7063483 15.1420474 15.3665612 16.9549046 20.9386146  
## [11] 20.9516374 21.1991956 26.5687756 27.3038881 29.6322860  
## [16] 30.1996253 30.2313136 31.1097462 31.1368118 32.4870698  
## [21] 32.9932958 34.1909996 35.7750262 35.7935768 38.5603706  
## [26] 40.0898246 40.4228540 41.3928951 41.4144082 42.6828333  
## [31] 43.0044182 43.3889239 44.8995551 45.1009473 45.2327090  
## [36] 45.3495038 46.2332691 46.5297536 47.6353144 48.3404713  
## [41] 48.9134070 49.6233012 50.5691095 51.0473180 51.3851639  
## [46] 52.4235611 52.9447824 53.2324236 53.2592412 53.4220617  
## [51] 53.9402693 54.1673112 54.7597759 54.8711741 54.9263167  
## [56] 55.6166600 56.2108414 56.3743118 56.4395433 56.6547406  
## [61] 59.2657582 60.1741199 61.0223365 62.4845789 62.5934706  
## [66] 65.4205304 65.8535476 66.0652713 66.8441539 66.9373366  
## [71] 67.8789765 68.2434783 68.9559228 68.9956160 71.9383452  
## [76] 72.1975034 72.7003327 73.8614206 74.2223608 75.1251620  
## [81] 76.0903847 76.9479162 77.0914169 79.3797609 79.7624343  
## [86] 80.3857815 81.8136356 82.3347046 82.4418694 84.1495018  
## [91] 85.1182857 85.9530114 89.0610827 90.0144599 90.3647350  
## [96] 91.1894095 97.5609287 97.9817591 102.9606334 111.8279245

sort(normaldist, decreasing = FALSE)

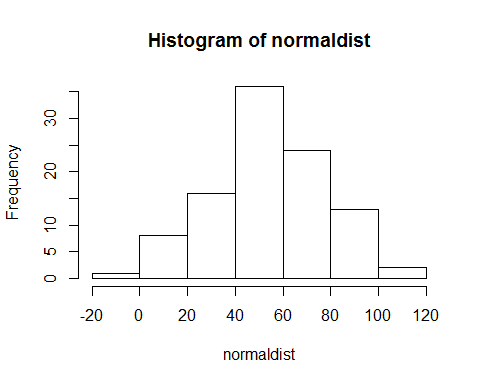
## [1] -0.9264303 1.7989654 7.6920140 11.0007620 13.1929871  
## [6] 14.7063483 15.1420474 15.3665612 16.9549046 20.9386146  
## [11] 20.9516374 21.1991956 26.5687756 27.3038881 29.6322860  
## [16] 30.1996253 30.2313136 31.1097462 31.1368118 32.4870698  
## [21] 32.9932958 34.1909996 35.7750262 35.7935768 38.5603706  
## [26] 40.0898246 40.4228540 41.3928951 41.4144082 42.6828333  
## [31] 43.0044182 43.3889239 44.8995551 45.1009473 45.2327090  
## [36] 45.3495038 46.2332691 46.5297536 47.6353144 48.3404713  
## [41] 48.9134070 49.6233012 50.5691095 51.0473180 51.3851639  
## [46] 52.4235611 52.9447824 53.2324236 53.2592412 53.4220617  
## [51] 53.9402693 54.1673112 54.7597759 54.8711741 54.9263167  
## [56] 55.6166600 56.2108414 56.3743118 56.4395433 56.6547406  
## [61] 59.2657582 60.1741199 61.0223365 62.4845789 62.5934706  
## [66] 65.4205304 65.8535476 66.0652713 66.8441539 66.9373366  
## [71] 67.8789765 68.2434783 68.9559228 68.9956160 71.9383452  
## [76] 72.1975034 72.7003327 73.8614206 74.2223608 75.1251620  
## [81] 76.0903847 76.9479162 77.0914169 79.3797609 79.7624343  
## [86] 80.3857815 81.8136356 82.3347046 82.4418694 84.1495018  
## [91] 85.1182857 85.9530114 89.0610827 90.0144599 90.3647350  
## [96] 91.1894095 97.5609287 97.9817591 102.9606334 111.8279245

# if you want to quickly look at the distribution of the data points, you can get a histogram using the hist function:  
hist(normaldist)

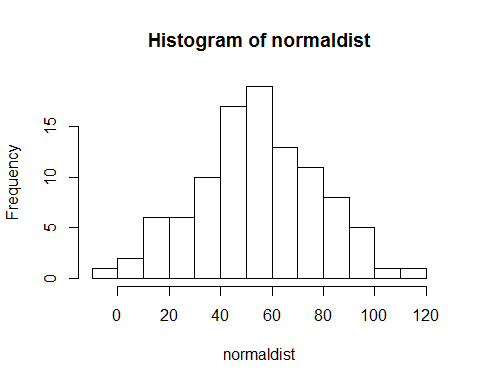


# as with most functions there are a number of options for modifying how a function works. Some examples of how you could change the histogram..

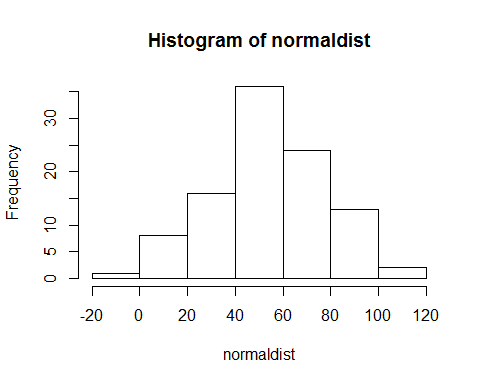
# change the number of breaks...  
  
hist(normaldist)



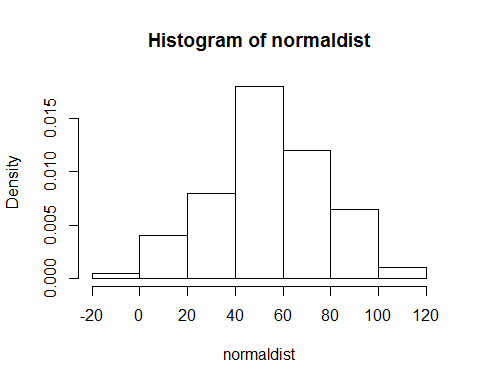
hist(normaldist,breaks = 10)



hist(normaldist,breaks=5)



# instead of showing counts, you could show probability densities...  
hist(normaldist,breaks=8,freq=FALSE)



##Indexxing

I briefly introduced indexxing before. This is basically a tool for selecting a subset of a vector (or a data.frame). You can use it to access a single vector element like I did before or you can use it to select a selection of vector elements. A **key detail** to note is that while functions use round brackets (), for indexxing you need to use square brackets[]!!!

normaldist[100]

## [1] 48.91341

To select a range of values or to select a mix of values

# select the first 5 values of the normaldist distribution  
normaldist[1:5]

## [1] 73.86142 53.42206 75.12516 49.62330 56.43954

# Alternatively, grab the first 5 values, the last 5 values, and observations 42, 64, and 89...  
normaldist[c(1:5,42,64,89,96:100)]

## [1] 73.86142 53.42206 75.12516 49.62330 56.43954 15.14205 60.17412  
## [8] 54.75978 43.00442 21.19920 14.70635 52.94478 48.91341

You can also use logical operations to select values from a vector…

unif2<-runif(20,0,30)  
unif2

## [1] 28.398627 2.353312 22.962016 19.033400 15.079502 28.961555 24.923837  
## [8] 5.430055 18.591092 9.777309 17.754721 21.781073 5.409856 5.734562  
## [15] 5.437632 19.683748 22.762829 10.947057 2.198773 22.165548

# only get values greater than 10...  
unif2[unif2>10]

## [1] 28.39863 22.96202 19.03340 15.07950 28.96155 24.92384 18.59109  
## [8] 17.75472 21.78107 19.68375 22.76283 10.94706 22.16555

# only get values less than 10...  
unif2[unif2<10]

## [1] 2.353312 5.430055 9.777309 5.409856 5.734562 5.437632 2.198773

# only get values less than 5 or more than 15...  
unif2[unif2<5 | unif2>15]

## [1] 28.398627 2.353312 22.962016 19.033400 15.079502 28.961555 24.923837  
## [8] 18.591092 17.754721 21.781073 19.683748 22.762829 2.198773 22.165548

# Note that | is an "OR" operator. That means if logical argument 1 is TRUE OR if logical argument 2 is TRUE then the final outcome is TRUE  
  
# only get values that are greater than 5 or less than 15...  
unif2[unif2>5 & unif2<15]

## [1] 5.430055 9.777309 5.409856 5.734562 5.437632 10.947057

# Note that & is an "AND" operator. That means if logical argument 1 is TRUE AND if logical argument 2 is TRUE then the final outcome is TRUE. If either of the arguments is FALSE then the final outcome is FALSE.   
  
# Switching to a vector of integers...  
pois1<-rpois(20,10)  
pois1

## [1] 14 9 4 7 8 10 9 16 14 12 12 10 10 5 12 10 15 11 11 7

# when working with integer values (and characters and factors) you can use the equal option. So, to get all of the 6s in the vector you would do the following:  
pois1[pois1==6]

## integer(0)

# I would recommend against using the == fuction for real values (e.g. 6.5251, 6.00000000000001, 4.25, etc.) as the results can be a bit unpredictable due to how the computer stores these types of values.   
  
test1<-c(6.111,6,6.000000000001,4.25,6.001)  
test1[test1==6]

## [1] 6

# If you have a situation where you need to identify vector elements that are "very close" to a value, but the vector elements are real numbers, there are <at least> a couple of ways you can do it...  
  
# Method 1  
threshold<-0.01 # basically, if the number is within 0.01 of 6 then it is close enough for us  
test1[test1>(6-threshold) & test1<(6+threshold)]

## [1] 6.000 6.000 6.001

# Method 2  
threshold<-0.01  
test1[abs(test1-6)<threshold]

## [1] 6.000 6.000 6.001

Many times, it is actually more important to know which elements meet the requirements of a logical argument than what their actual values are. For example, we really don’t need to know that the values of test1 equal 6 when we are searching for all values of test1 that equal 6. We already know that. What we actually want to know is which elements are equal to 6. To find those elements, we use the which function.

equals6<-which(abs(test1-6)<threshold)  
test1

## [1] 6.111 6.000 6.000 4.250 6.001

equals6

## [1] 2 3 5

test1[equals6]

## [1] 6.000 6.000 6.001

The final function I’ll introduce you to is the table function. This function is a quick way of getting a summary count of the number of values in a category. It generally works best when you’re looking at two or more columns of data, but you can run it on a single column of data if you want.

# example with a vector of integers  
table(pois1)

## pois1  
## 4 5 7 8 9 10 11 12 14 15 16   
## 1 1 2 1 2 4 2 3 2 1 1

# example with a vector of reals  
table(normaldist)

## normaldist  
## -0.926430284806557 1.79896543640841 7.69201397053228   
## 1 1 1   
## 11.000762003624 13.1929871076812 14.7063483420737   
## 1 1 1   
## 15.1420473699602 15.3665612220591 16.9549045906601   
## 1 1 1   
## 20.9386145945163 20.9516374138916 21.1991955553029   
## 1 1 1   
## 26.5687756030368 27.3038880994755 29.632286041537   
## 1 1 1   
## 30.1996252819162 30.2313135592845 31.1097462424678   
## 1 1 1   
## 31.1368118421999 32.4870697692152 32.9932958179715   
## 1 1 1   
## 34.1909995538151 35.7750261668776 35.7935768169932   
## 1 1 1   
## 38.560370621796 40.0898246418362 40.4228540392589   
## 1 1 1   
## 41.392895077425 41.4144082206127 42.6828333034015   
## 1 1 1   
## 43.0044181539558 43.3889238758463 44.8995550965594   
## 1 1 1   
## 45.1009472995233 45.2327089955699 45.3495037799592   
## 1 1 1   
## 46.2332690690193 46.5297535972216 47.6353143672991   
## 1 1 1   
## 48.3404713434934 48.9134070426906 49.6233012474711   
## 1 1 1   
## 50.5691095229793 51.0473180311634 51.3851639295691   
## 1 1 1   
## 52.4235611311329 52.9447823733163 53.2324235689839   
## 1 1 1   
## 53.2592411975663 53.4220616569402 53.9402693431873   
## 1 1 1   
## 54.1673111754585 54.7597759390691 54.8711740579208   
## 1 1 1   
## 54.9263167268971 55.6166599776277 56.2108413849739   
## 1 1 1   
## 56.3743118341743 56.4395432577072 56.6547405842904   
## 1 1 1   
## 59.2657581697729 60.1741199315699 61.0223364619484   
## 1 1 1   
## 62.4845788747403 62.5934705574972 65.420530434761   
## 1 1 1   
## 65.8535476261738 66.0652712847386 66.8441539476275   
## 1 1 1   
## 66.9373365875589 67.8789765015406 68.2434783005223   
## 1 1 1   
## 68.9559227544179 68.9956160277502 71.9383451533261   
## 1 1 1   
## 72.1975034341638 72.7003327050558 73.8614205563819   
## 1 1 1   
## 74.2223608036835 75.1251619505398 76.0903847316085   
## 1 1 1   
## 76.9479161747025 77.0914169449833 79.3797609286111   
## 1 1 1   
## 79.7624343193494 80.3857814568153 81.813635577989   
## 1 1 1   
## 82.3347046003372 82.4418694206914 84.1495017779167   
## 1 1 1   
## 85.1182857220971 85.9530114319653 89.0610827461747   
## 1 1 1   
## 90.0144598613937 90.3647350019334 91.1894095371345   
## 1 1 1   
## 97.5609287208506 97.981759118151 102.960633439519   
## 1 1 1   
## 111.827924478812   
## 1