SIOB 296 Introduction to Programming with R

Eric Archer (eric.archer@noaa.gov)

Week 4: January 29, 2019

reading/writing tables, common functions for data summary and selection

---

Reading and writing text tables (.csv): `write.table`, `read.table`

Data in tabular format, such as matrices or data frames are saved to and read from disk with `write.table` and `read.table` and their wrappers, most commonly `write.csv` and `read.csv`:

```
x <- data.frame(nums = 51:60, lets = letters[1:10])
write.csv(x, file = "test.csv")
rm(list = ls())
df <- read.csv("test.csv")
df
```

```
    X nums lets
1   1   51    a
2   2   52    b
3   3   53    c
4   4   54    d
5   5   55    e
6   6   56    f
7   7   57    g
8   8   58    h
9   9   59    i
10 10   60    j
```

You'll notice that there is a new column, "X" that has the numbers 1-10 in it. This is because by default, `write.csv` writes a file with the rownames in the first column. To change this behavior, set the argument `row.nams = FALSE` in `write.csv`.

```
x <- data.frame(nums = 51:60, lets = letters[1:10])
write.csv(x, file = "test.csv", row.names = FALSE)
rm(list = ls())
df <- read.csv("test.csv")
df
```

```
   nums lets
1    51    a
2    52    b
3    53    c
4    54    d
5    55    e
6    56    f
```

```
7    57    g
8    58    h
9    59    i
10   60    j
```

```
str(df)
```

```
'data.frame':   10 obs. of  2 variables:
 $ nums: int  51 52 53 54 55 56 57 58 59 60
 $ lets: Factor w/ 10 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10
```

Also, notice that the `lets` column is read in as a factor. This is the default behavior of `read.csv` and can be changed with the `stringsAsFactors` argument:

```
df <- read.csv("test.csv", stringsAsFactors = FALSE)
str(df)
```

```
'data.frame':   10 obs. of  2 variables:
 $ nums: int  51 52 53 54 55 56 57 58 59 60
 $ lets: chr  "a" "b" "c" "d" ...
```

Missing data (NAs)

Missing data is denoted in R with `NA` and has to be explicitly tested for and handled specially. To test if values are equal to NA, you can't use `==`, you have to use `is.na()`

```
x <- c(1, NA, 3, 6, NA)
x == NA
```

```
[1] NA NA NA NA NA
```

```
is.na(x)
```

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

Several functions, primarily mathematical summaries, have an `na.rm` argument that will remove `NA`s from the object before acting on the result:

```
x <- sample(c(NA, 1:5), 20, TRUE)
mean(x)
```

```
[1] 2.5
```

```
mean(x, na.rm = TRUE)
```

```
[1] 2.5
```

To remove `NA`s from a vector, use `na.omit()`:

```
x2 <- na.omit(x)
x2
```

```
 [1] 5 1 1 4 4 3 3 1 3 3 2 2 2 1 4 2 3 4 1 1
```

```
str(x2)
```

```
 int [1:20] 5 1 1 4 4 3 3 1 3 3 ...
```

To identify rows in a data frame without `NA`s, use `complete.cases`:

```
mat <- rbind(
  sample(1:5, 8, replace = TRUE),
  sample(c(NA, 1:5), 8, replace = TRUE),
  sample(1:5, 8, replace = TRUE),
  sample(c(NA, 1:5), 8, replace = TRUE)
)
mat
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    5    3    2    4    3    4    3    2
[2,]    4    5    3    1   NA   NA    4   NA
[3,]    1    3    5    4    5    2    3    5
[4,]    4    3    5    2    2    1   NA    5
```

```
i <- complete.cases(mat)
i
```

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

```
mat[i, ]
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    5    3    2    4    3    4    3    2
[2,]    1    3    5    4    5    2    3    5
```

Math summaries

To get the range of a vector of numerics, use `min`, `max`, and `range`:

```
x <- sample(1:100, 10)
x
```

```
 [1] 57 45 64  9 20 76  1 51 55 13
```

```
min(x)
```

```
[1] 1
```

```
max(x)
```

```
[1] 76
```

```
range(x)
```

```
[1]  1 76
```

Sums and products of vectors can be calculated:

```
sum(x)
```

```
[1] 391
```

```
prod(x)
```

```
[1] 8.188977e+13
```

To calculate the difference between values with a given lag, use `diff`:

```
diff(x)
```

```
[1] -12  19 -55  11  56 -75  50   4 -42
```

```
diff(x, lag = 3)
```

```
[1] -48 -25  12  -8  31 -21  12
```

Other numeric summaries such as the median, mean, variance, and standard deviation are available:

```
median(x)
```

```
[1] 48
```

```
mean(x)
```

```
[1] 39.1
```

```
var(x)
```

```
[1] 681.6556
```

```
sd(x)
```

```
[1] 26.10853
```

Any set of quantiles can be calculated with the `quantiles` function:

```
x <- sample(1:1000, 100)
quantile(x, probs = c(0.025, 0.05, 1/3, 0.5, 0.99))
```

```
    2.5%        5% 33.33333%       50%       99%
   40.35     74.55    310.00    436.00    981.07
```

Discrete values

The function `unique()` will list the unique values in a vector in the order it finds them:

```
x <- sample(letters, 10, replace = TRUE)
x
```

```
 [1] "p" "l" "m" "w" "r" "u" "f" "v" "a" "g"
```

```
unique(x)
```

```
 [1] "p" "l" "m" "w" "r" "u" "f" "v" "a" "g"
```

The function `duplicated()` will identify those elements in a vector that occur at an earlier position:

```
duplicated(x)
```

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

```
# the negation of duplicated is the same as unique
x[!duplicated(x)]
```

```
 [1] "p" "l" "m" "w" "r" "u" "f" "v" "a" "g"
```

```
unique(x)
```

```
 [1] "p" "l" "m" "w" "r" "u" "f" "v" "a" "g"
```

To calculate the frequency of values in a vector (the number of occurences), use `table()`:

```
x <- sample(letters, 20, replace = TRUE)
table(x)
```

```
x
b c e h i k l n p q r s v y z
2 2 2 1 1 2 1 1 1 1 1 1 2 1 1
```

`table` can be used for cross-tabulation as well - counting frequency of occurence of a combination of categories

```
months <- sample(month.abb, 100, replace = TRUE)
sex <- sample(c("m", "f"), 100, replace = TRUE)
freq <- table(sex, months)
freq
```

```
   months
sex Apr Aug Dec Feb Jan Jul Jun Mar May Nov Oct Sep
  f   7   6   6   4   5   6   5   2   6   3   4   7
  m   2   2   3   2   2   2   2   6   2   6   2   8
```

The values in a table can be accessed like a vector or matrix

```
freq["m", ]
```

```
Apr Aug Dec Feb Jan Jul Jun Mar May Nov Oct Sep
  2   2   3   2   2   2   2   6   2   6   2   8
```

```
freq["f", c("Jun", "Jul", "Aug")]
```

```
Jun Jul Aug
  5   6   6
```

Data selection and manipulation

To identify values of one vector that are within another one, use `%in%`:

```
letters %in% c("a", "f", "G", "b")
```

```
 [1]  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE
```

There are three other functions to examine set membership between vectors. The first, `union`, returns a vector that is a combination of the unique values in the two input vectors:

```
union(c("z", "a", "c", "j"), c("a", "j", "a", "e", "d"))
```

```
[1] "z" "a" "c" "j" "e" "d"
```

The next one, `intersect`, returns a vector that is just the unique values that the two vectors have in common:

```
intersect(c("z", "a", "c", "j"), c("a", "j", "a", "e", "d"))
```

```
[1] "a" "j"
```

Finally, `setdiff` returns a vector of all values in the first vector that do not occur in the second vector:

```
setdiff(c("z", "a", "c", "j"), c("a", "j", "a", "e", "d"))
```

```
[1] "z" "c"
```

To identify values of a logical vector that are `TRUE`, use `which`:

```
x <- sample(1:100, 20)
x
```

```
 [1]  2 98 72 24 33 70 89 75 19 48 77 52 58  4 59 53  3 26 63 44
```

```
which(x < 50)
```

```
[1]  1  4  5  9 10 14 17 18 20
```

To identify the minimum and maximum values, use `which.min` and `which.max`:

```
which.min(x)
```

```
[1] 1
```

```
which.max(x)
```

```
[1] 2
```

Vectors can be reversed with `rev`:

```
x <- sample(1:5, 10, replace = T)
x
```

```
 [1] 1 4 3 1 2 3 5 5 2 5
```

```
rev(x)
```

```
 [1] 5 2 5 5 3 2 1 3 4 1
```

and sorted with `sort`:

```
sort(x)
```

```
 [1] 1 1 2 2 3 3 4 5 5 5
```

```
# in decreasing order
sort(x, decreasing = TRUE)
```

```
 [1] 5 5 5 4 3 3 2 2 1 1
```

However, sort can't be applied to a matrix or data.frame to sort the rows. For that, you need `order`. `order` returns a vector of indices in the order they should be as if they were sorted:

```
x <- data.frame(
  v1 = sample(letters, 20, replace = TRUE),
  v2 = sample(letters, 20, replace = TRUE),
  v3 = sample(letters, 20, replace = TRUE)
)
x
```

```
   v1 v2 v3
1   k  n  o
2   o  p  l
3   b  w  l
4   g  u  r
5   l  m  p
6   k  s  t
7   x  n  m
8   o  x  y
9   n  i  w
10  l  j  s
```

```
11  r  h  d
12  v  g  c
13  k  x  c
14  y  s  x
15  i  j  g
16  b  g  h
17  d  n  p
18  k  r  j
19  w  t  w
20  r  f  r
```

```
x.ord <- order(x$v1)
x[x.ord, ]
```

```
   v1 v2 v3
3   b  w  l
16  b  g  h
17  d  n  p
4   g  u  r
15  i  j  g
1   k  n  o
6   k  s  t
13  k  x  c
18  k  r  j
5   l  m  p
10  l  j  s
9   n  i  w
2   o  p  l
8   o  x  y
11  r  h  d
20  r  f  r
12  v  g  c
19  w  t  w
7   x  n  m
14  y  s  x
```

```
# also in decreasing order
x[order(x$v1, decreasing = TRUE), ]
```

```
   v1 v2 v3
14  y  s  x
7   x  n  m
19  w  t  w
12  v  g  c
11  r  h  d
20  r  f  r
2   o  p  l
8   o  x  y
9   n  i  w
5   l  m  p
10  l  j  s
1   k  n  o
6   k  s  t
13  k  x  c
18  k  r  j
```

```
15  i  j  g
4   g  u  r
17  d  n  p
3   b  w  l
16  b  g  h
```

`order` can take several vectors to do hierarchical sorting.

```
i <- order(x$v2, x$v1, x$v3)
i
```

```
 [1] 20 16 12 11  9 15 10  5 17  1  7  2 18  6 14 19  4  3 13  8
```

```
x[i, ]
```

```
   v1 v2 v3
20  r  f  r
16  b  g  h
12  v  g  c
11  r  h  d
9   n  i  w
15  i  j  g
10  l  j  s
5   l  m  p
17  d  n  p
1   k  n  o
7   x  n  m
2   o  p  l
18  k  r  j
6   k  s  t
14  y  s  x
19  w  t  w
4   g  u  r
3   b  w  l
13  k  x  c
8   o  x  y
```

Binning values

To create bins of a continuous variable, the `cut` function is very handy. It has several arguments that regulate how the binning is to be done that are worth examining:

```
y <- c(4, 5, 6, 10, 11, 30, 49, 50, 51)

# We want the following bins : 5 > y <= 10, 10 > y <= 30, 30 > y <= 50
y.cut <- cut(y, breaks = c(5, 10, 30, 50))
y.cut
```

```
[1] <NA>    <NA>    (5,10]  (5,10]  (10,30] (10,30] (30,50] (30,50] <NA>
Levels: (5,10] (10,30] (30,50]
```

```
str(y.cut)
```

```
 Factor w/ 3 levels "(5,10]","(10,30]",..: NA NA 1 1 2 2 3 3 NA
```

A factor is created that replaces the values with the selected bins. The bins labels use the parentheses ("`(`" and "`)`") to denote that the value is not included in the bin, while the brackets ("`[`" and "`]`") denote that the value is included. Let's change the binning, so that 5 (the lowest bin value) is included, using `include.lowest = TRUE`:

```
# Bins : 5 >= y <= 10, 10 > y <= 30, 30 > y <= 50
cut(y, breaks = c(5, 10, 30, 50), include.lowest = TRUE)
```

```
[1] <NA>    [5,10]  [5,10]  [5,10]  (10,30] (10,30] (30,50] (30,50] <NA>
Levels: [5,10] (10,30] (30,50]
```

By including the argument `right = FALSE`, the default binning is flipped so that the lowest value is included, but the highest is not:

```
# Bins : 5 >= y < 10, 10 >= y < 30, 30 >= y < 50
cut(y, breaks = c(5, 10, 30, 50), right = FALSE)
```

```
[1] <NA>    [5,10)  [5,10)  [10,30) [10,30) [30,50) [30,50) <NA>    <NA>
Levels: [5,10) [10,30) [30,50)
```

Including both `include.lowest` and `right` causes all bin values to be included:

```
# Bins : 5 >= y < 10, 10 >= y < 30, 30 >= y <= 50
cut(y, breaks = c(5, 10, 30, 50), include.lowest = TRUE, right = FALSE)
```

```
[1] <NA>    [5,10)  [5,10)  [10,30) [10,30) [30,50] [30,50] [30,50] <NA>
Levels: [5,10) [10,30) [30,50]
```