factors and tables in R

Factors are data type in R that resembles that of nominal (categorical) variables in statistics. Factors are nonnumeric in nature, even if coded to numerical values in practice. Factors are viewed in R as vectors with some additional information attached; the extra information being a record of distinct values in a vector, referred to as ***levels***. The illustration below displays the results of a factor coercion:

```
> x <- c(5,12,13,12)
> xf <- factor(x)
> xf
[1] 5  12 13 12
Levels: 5 12 13
```

The ***levels*** in the above illustration represent the distinct values in vector **xf—5, 12, 13**.

```
> str(xf)                                  #note the nonnumeric “5” representation
 Factor w/ 3 levels "5","12","13": 1 2 3 2

> unclass(xf)  #note that the core of xf is the level representations of 1, 2, and 3
[1] 1 2 3 2
attr(,"levels")
[1] "5"  "12" "13"
```

In additional to the behavior in the above illustration, the **length( )** of a factor is still represented as the length of the data, opposed to a count of levels, etc.

```
> length(xf)
[1] 4
```

Anticipated levels can be added into the factor’s construction. For example, the current data represents 3 separate levels (categories, labels, classes, etc.) and a fourth is anticipated to exist in a new dataset:

```
> x                                        #four-element vector x
[1]  5 12 13 12

> xf                                       #factor xf with 3 levels (5, 12, 13)
[1] 5  12 13 12
Levels: 5 12 13

> xff <- factor(x, levels = c(5,12,13,88))    #factor xf with 4 levels (5, 12, 13, 88)
> xff
[1] 5  12 13 12
Levels: 5 12 13 88

> xff[2] <- 88                             #example assignment of additional factor
> xff
[1] 5  88 13 12
Levels: 5 12 13 88
```

Conversely, undefined levels within a factor cannot be added if the level within the factor does not exist:

```
> xff[2] <- 28
Warning message:
In `[<-.factor`(`*tmp*`, 2, value = 28) :
  invalid factor level, NA generated
```

factors and functions *f*( ) in R

In the context of factors, the **tapply( )** function is leveraged to apply functions against factor elements.

```
tapply(X, INDEX< FUN = NULL, ..., simplify = TRUE)
```

The call **tapply(x, f, g)** has vector **x**, factor (or list of multiple factors) **f**, and function **g**.

The motivating example illustration is vector **x** of ages of voters and factor **f** of voter party affiliation. Each factor in **f** must consists of equal lengths to **x**. If a component of **f** is a vector, it is coerced into a factor by application of function **as.factor( )** to the vector in **f**.

The operation performed by **tapply( )** temporarily splits **x** into groups, each corresponding to a level (or combination of levels) in factor **f**, finally applying **g( )** to the resulting subvectors of **x** (illustrated below):

```
> ages <- c(25,26,55,37,21,42)
> affils <- c("R","D","D","R","U","D")
> tapply(ages, affils, mean)
 D  R  U
41 31 21
```

The example illustration is expanded to break down groups by both gender and age. When 2 or more factors are applied, each produces a set of groups computed against each other under **AND** logic. The **mean( )** function will then be applied to return the each of the four subgroups inclusive of the operation:

1) Male < 25; **2)** Male > 25; **3)** Female < 25; **4)** Female >25

```
> d <- data.frame(list(gender = c("M","M","F","M","F","F"),
+                      age = c(47,59,21,32,33,24),
+                      income = c(55000,8800,32450,76500,123000,45650)))
> d
  gender age income
1      M  47  55000
2      M  59   8800
3      F  21  32450
4      M  32  76500
5      F  33 123000
6      F  24  45650

> d$over25 <- ifelse(d$age > 25, 1 ,0)
> d
  gender age income over25
1      M  47  55000      1
2      M  59   8800      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0

> tapply(d$income, list(d$gender, d$over25),mean)
      0         1
F 39050 123000.00
M    NA  46766.67
```

The two specified factors (Age < OR > 25 and Gender) each contain two levels. Therefore, **tapply( )** split (partitioned) the data into four groups; each group representative of each possible permutation of gender and age. Lastly, the **mean( )** function was applied to each group individually.

Using the **split( )** function simply forms the vector into groups, opposed to splitting a vector into groups and ***then*** apply a function against each group; seen in the **tapply( )** function.

```
split(x, f, drop = FALSE, ...)
```

The vector **x** and factor **f** serve similar roles in the **split( )** function as they did in the **tapply( )** function. Note that in the **split( )** function, **x** is allowed to be a dataframe; the latter is not true with **tapply( )**.

```
> d
  gender age income over25
1      M  47  55000      1
2      M  59   8800      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0

> split(d$income, list(d$gender, d$over25))  #list output denoted by $
$F.0
[1] 32450 45650

$M.0
numeric(0)

$F.1
[1] 123000

$M.1
[1] 55000  8800 76500
```

Note that the output is a list with components denoted by **$**; the vectors are represented by the combination of factors that make up the results (e.g. $F.0 = Female < 25, $M.1 = Male > 25).

The example below determines the indices of which vector elements correspond to Male, Female, or Infants:

```
> g <- c("M","F","F","I","M","M","F")
> split(1:7,g)
$F
[1] 2 3 7

$I
[1] 4

$M
[1] 1 5 6
```

The **by( )** function operates similar to that of **tapply( )**, which is internally called within **by( )**. The difference is how the **by( )** function is applied to **objects**, rather than vectors. This is useful for performing certain operations against dataframes and matrices.

```
split(x, f, drop = FALSE, ...)
```

Calls to the **by( )** function are similar to those of the **tapply( )** function. The first argument **x** specifies the data, the second argument **f** specifies the grouping factor(s), and the third argument specifies the function to be applied to each factor group individually. Just as **tapply( )** forms groups of indices of a vector corresponding to each factor level, **by( )** finds groups of row numbers of a dataframe, creating sub dataframes for each corresponding factor level. The specified function is then called against each of the identified factor groups defined in the problem.

working with tables in R

Given the following example:

```
> u <- c(22,8,33,6,8,29,-2)
> fl <- list(c(5,12,13,12,13,5,13),c("a","bc","a","a","bc","a","a"))
> tapply(u,fl,length)
   a bc
5  2 NA
12 1  1
13 2  1
```

Again, **tapply( )** partitions vector **u** into subvectors and applies the **length( )** function to each subvector. Note that the latter operation is independent of the contents of vector **u**; the focus is strictly on the factors. The resulting ***contingency table*** illustrates the frequency of factors occurring in the data. Note the occurrence of **NA** representing that **5** occurred with **"bc"** in no observations from the function. The problem with the above illustration is the misrepresentation of **0** occurrences as an **NA** value; use **table( )** instead:

```
> table(fl)    #applying the table( ) function to appropriately represent all cases
    fl.2
fl.1 a bc
  5  2  0
  12 1  1
  13 2  1
```

The first argument in the **table( )** function is either a factor or a list of factors (two factors in above).

Typically, a dataframe serves as the **table( )** argument. Consider the following example of polling-data:

```
> ct <- data.frame("Vote.for.X" = c("Yes", "Yes", "No", "Not Sure", "No"),
+                  "Voted.for.X.Last.Time" = c("Yes", "No", "No", "Yes", "No"))
> ct
  Vote.for.X Voted.for.X.Last.Time
1        Yes                   Yes
2        Yes                    No
3         No                    No
4   Not Sure                   Yes
5         No                    No
```

The **table( )** function computes the ***contingency table*** for the above dataset.

```
> cttab <- table(ct)
> cttab
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
```

One-dimensional counts are equally available; counts on a single factor:

```
> table(c(5,12,13,12,8,5))

 5  8 12 13
 2  1  2  1
```

matrix/array-like operations on tables in R

Considering that nonmathematical operators can be used on dataframes, the same applies to tables—*the cell counts component of a table object is an array*. Table cell counts can be accessed with matrix notation:

```
> class(cttab)
[1] "table"

> cttab[1,1]
[1] 2

> cttab[1,]
 No Yes
  2   0
```

Below illustrates multiplicative properties of table operations by converting cell counts to proportions:

```
> cttab/5
          Voted.for.X.Last.Time
Vote.for.X  No Yes
  No        0.4 0.0
  Not Sure 0.0 0.2
  Yes       0.2 0.2
```

The ***marginal values*** (statistical method are the values of a variable when the variable itself is held constant while other variables are summed) are computed with the **apply( )** function below:

```
> apply(cttab, 1, sum)
      No Not Sure      Yes
       2        1        2
```

The above illustration computes that marginal values of the polling data (1+1=2, 1+0=1, 1+1=2).

The ***marginal values*** both dimensions can be computed simultaneously with the **addmargins( )** function.

```
> addmargins(cttab)
          Voted.for.X.Last.Time
Vote.for.X No Yes Sum
  No        2   0   2
  Not Sure  0   1   1
  Yes       1   1   2
  Sum       3   2   5
```

The dimension names of a table can be accessed through the **dimnames( )** function as follows:

```
> dimnames(cttab)
$Vote.for.X
[1] "No"       "Not Sure" "Yes"

$Voted.for.X.Last.Time
[1] "No"  "Yes"
```

other factor- and table-related functions in R

the **aggregate( )** function calls **tapply( )** once for each variable in a group.

```
aggregate(x, ...)
```

The **cut( )** function is a common way to generate factors (particularly with tables) by providing data vector **x** and a set of bins defined by vector **b**. The functions determines the bins each element of **x** falls into.

```
cut(x, b, labels = FALSE, ...)
```