4 Sorting, Printing, and Summarizing Data

4.1 Subsetting Data

Logical vectors can be used to efficiently subset the data. The basic form is:

`data.frame[logical.vector,]`

Only observations satisfying the conditions will be included in the subset. Logical vector can be formed using logical operators. The following table list commonly used logical operators. The left side of the operator is usually a vector, or variable from a `data.frame` object. The right side is an expression that can be evaluated to a constant.

R code	What it computes
!	logical NOT
&	logical AND
|	logical OR
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	logical equals (double=)
!=	not equal
&&	AND with IF
||	OR with IF
xor(x,y)	exclusive OR
is.na(x)	missing value
is.nan(x)	not a number
x %in% y	is in a vector y

Subsetting can be achieved using index vector. The basic form is

`data.frame[index.vector,]`

The index vector takes value from 1 to n, the number of observations in the `data.frame` object. The `which` function can be used to identify observations satisfying certain conditions.

Example You have a database containing information about worm density. A subset of the data appears below. For each field, the data include the observed worm density and environmental conditions:

```
Field.Name Area Slope Vegetation Soil.pH Damp
Worm.density

Nashs.Field 3.6 11 Grassland 4.1 F 4
Silwood.Bottom 5.1 NA Arable 5.2 F 7
Nursery.Field 2.8 3 Grassland 4.3 F 2
Rush.Meadow 2.4 5 Meadow 4.9 T 5
Gunness.Thicket 3.8 0 Scrub 4.2 F 6
Oak.Mead 3.1 2 Grassland 3.9 F 2
Church.Field 3.5 3 Grassland NA NA NA
```

The following code reads the data.

```
setwd("C:/Users/Slava/Dropbox/Documents/Teaching/MEES708XY/y2 Sorting/")
worm <- read.table("./dataRaw/worms.missing.txt", head=TRUE)
```

Suppose a day later you wanted to print a list of just the fields in grassland. You can use logical vectors to do this.

```
(logical.vec <- worm$Vegetation=="Grassland")

##  [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE
## [17] FALSE FALSE  TRUE FALSE

worm[logical.vec,]
```

```
##           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 1        Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
## 3      Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
## 6           Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
## 7       Church.Field  3.5     3  Grassland      NA    NA           NA
## 10     Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
## 12      North.Gravel  3.3     1  Grassland     4.1 FALSE            1
## 13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2
## 14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
## 19        Gravel.Pit   NA     1  Grassland     3.5 FALSE            1

(index.vec <- which(logical.vec))

## [1]  1  3  6  7 10 12 13 14 19

worm[index.vec,]

##           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 1        Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
## 3      Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
## 6           Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
## 7       Church.Field  3.5     3  Grassland      NA    NA           NA
## 10     Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
## 12      North.Gravel  3.3     1  Grassland     4.1 FALSE            1
## 13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2
## 14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
## 19        Gravel.Pit   NA     1  Grassland     3.5 FALSE            1
```

Suppose that you wanted the data from the fields where worm density was higher than the median density and the soil pH was less than 5.2.

```
attach(worm)
mwd <- median(Worm.density,na.rm=T)
combo.vec <- (Worm.density > mwd & Soil.pH < 5.2)
worm[combo.vec,]

##         Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 4      Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 5  Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## NA            <NA>   NA    NA       <NA>      NA    NA           NA
## 10   Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
## 15      Pond.Field  4.1     0     Meadow     5.0  TRUE            6
## 16    Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
## 18      Pound.Hill  4.4     2     Arable     4.5 FALSE            5

detach(worm)
```

Suppose that you wanted to drop all fields in the previous conditions. We can use *negative* subscripts or logical vectors.

```
worm[-which(combo.vec),]
worm[!combo.vec,]
```

Note that there are missing values in the data. Suppose we wanted to drop all missing values. We can use `na.omit` as well as logical vectors.

```
attach(worm)
mwd <- median(Worm.density,na.rm=T)
combo.vec <- (Worm.density > mwd & Soil.pH < 5.2)
na.omit(worm[combo.vec,])
```

```
##         Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 4      Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 5  Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## 10   Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
## 15      Pond.Field  4.1     0     Meadow     5.0  TRUE            6
## 16    Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
## 18      Pound.Hill  4.4     2     Arable     4.5 FALSE            5
```

```
nna.vec <- combo.vec & !is.na(Worm.density)
worm[nna.vec,]
```

```
##         Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 4      Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 5  Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## 10   Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
## 15      Pond.Field  4.1     0     Meadow     5.0  TRUE            6
## 16    Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
## 18      Pound.Hill  4.4     2     Arable     4.5 FALSE            5
```

```
detach(worm)
```

Suppose you were asked to extract a single record for each vegetation type, you can use `unique` to list unique vegetation types and use `!duplicated` to extract a field not duplicated within each vegetation type.

```
unique(worm$Vegetation)
```

```
## [1] Grassland Arable    Meadow    Scrub     Orchard
## Levels: Arable Grassland Meadow Orchard Scrub
```

```
worm[!duplicated(worm$Vegetation),]
```

```
##        Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 1     Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
## 2  Silwood.Bottom  5.1    NA     Arable     5.2 FALSE            7
## 4     Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 5 Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## 9     The.Orchard  1.9     0    Orchard     5.7 FALSE            9
```

4.2 Sorting Data

There are many reasons for sorting your data, to organize data for a report , or to perform stratified analysis. The basic form of sorting is

`data.frame[order(...),]`

The order function takes a list of variables and calculate a *permutation* of row numbers that makes the data in ascending order of the selected variables. Note that the entire `order` function appears before comma, indicating that we want the sorting to apply to all columns (variables) of the data frame.

To sort one variable in descending order, use the `rev` function to reverse the order. More complicated sorting may involve two or more variables. This is achieved by running `order` on multiple variables, separated by comma. R will sort the variables based on the left hand variable, with ties being broken on the second variable, and so on.

Suppose that there are multiple variables, but the variables need to be sorted in opposing directions. For numerical variables, the trick is to put a minus sign in front of the variables.

Example The following data show the average length in feet of selected whales and sharks:

```
beluga   whale 15   dwarf    shark .5    sperm    whale 60
basking  shark 30   humpback   .   50    whale    shark 40
gray     whale 50   blue     whale 100   killer   whale 30
mako     shark 12   whale    shark 40
```

This program reads the data.

```
setwd("C:/Users/Slava/Dropbox/Documents/Teaching/MEES708XY/y2 Sorting/")
tmp <- scan("./dataRaw/Lengths.dat",
        what=list(Name="",Family="",Length=0),
        na.string=".")
tmp2 <- as.data.frame(tmp)
marine <- unique(tmp2)
rm(tmp,tmp2)
```

The scan function reads the raw data from a file. Note the use of `na.string` argument to indicate that "." is a missing value. The `scan` function reads in a `list` object. We can use `as.data.frame` to convert the `list` into a `data.frame`. The `unique` function eliminated a duplicate observation for the whale shark.

The following code order data by `Family`

```
attach(marine)
marine[order(Family),] ## ascending order
```

```
##        Name Family Length
## 2     dwarf  shark    0.5
## 4   basking  shark   30.0
## 6     whale  shark   40.0
## 10     mako  shark   12.0
## 1    beluga  whale   15.0
## 3     sperm  whale   60.0
## 7      gray  whale   50.0
## 8      blue  whale  100.0
## 9    killer  whale   30.0
## 5  humpback   <NA>   50.0
```

```
marine[rev(order(Family)),] ## descending order
```

```
##        Name Family Length
## 5  humpback   <NA>   50.0
## 9    killer  whale   30.0
## 8      blue  whale  100.0
## 7      gray  whale   50.0
## 3     sperm  whale   60.0
## 1    beluga  whale   15.0
## 10     mako  shark   12.0
## 6     whale  shark   40.0
## 4   basking  shark   30.0
## 2     dwarf  shark    0.5
```

Now sorting is on both `Family` and `Length` in ascending orders:

```
marine[order(Family,Length),]
```

```
##        Name Family Length
## 2     dwarf  shark    0.5
## 10     mako  shark   12.0
## 4   basking  shark   30.0
## 6     whale  shark   40.0
## 1    beluga  whale   15.0
```

```
## 9    killer  whale   30.0
## 7      gray  whale   50.0
## 3     sperm  whale   60.0
## 8      blue  whale  100.0
## 5  humpback   <NA>   50.0
```

To sort `Length` in descending order, we use the minus sign trick:

```
marine[order(Family,-Length),]

##        Name Family Length
## 6     whale  shark   40.0
## 4   basking  shark   30.0
## 10     mako  shark   12.0
## 2     dwarf  shark    0.5
## 8      blue  whale  100.0
## 3     sperm  whale   60.0
## 7      gray  whale   50.0
## 9    killer  whale   30.0
## 1    beluga  whale   15.0
## 5  humpback   <NA>   50.0

detach(marine) ## don't forget this
```

The `order` function rearrange the observations by family in ascending order, with ties broken by average length in ascending order. To list the length in descending order in each family, we use the minus sign before `Length` variable.

4.3 Coding Your Data

If you have coded data, you need to use `factors`. Imagine that you have a survey data and all responses are coded to save disk space and time. For example, the `age` categories teen, adult and senior are coded as numbers 1, 2 and 3. This is convenient for data entry and analysis but bothersome when it comes to interpret the results. A solution is to create a `factor` variable.

`factors` are categorical variables with fixed number of levels. When you create a data frame by reading your data from a file using `read.table`, all character variables will be automatically converted into `factors`. The important function for character factor is `levels`, which can be used to discover the names of the factor levels, to change the ordering of levels from the default alphabetical order, or to combine multiple levels.

Numerical variables can also be coded into categorical variables. The function is `cut`. The general form is `cut(x, breaks, labels)`, where `x` is a numerical vector to be converted, `breaks` is the number of cut points or number of intervals into which `x` is to be cut, and `labels` are the corresponding label of the resulting category.

Example Universe Cars is surveying its customers as to their preferences for car colors. They have information about the customer's age, sex (coded as 1 for male and 2 for female), annual income, and preferred car color (yellows, gray, blue, or white). Here are the data:

```
19 1 14000 Y
45 1 65000 G
72 2 35000 B
31 1 44000 Y
58 2 83000 W
```

The following program reads the data; creates `factor` for age, sex and car color; then prints the data.

```
setwd("C:/Users/Slava/Dropbox/Documents/Teaching/MEES708XY/y2 Sorting/")
tmp <- read.table("./dataRaw/Cars.dat",
 head=F,
 col.names=c("Age","Sex","Income","Color"))
carsurvey<-within(tmp,
```

```
{
  Agegroup <- cut(Age,
         breaks=c(-Inf,20,65,Inf),
         label=c("Teen","Adult","Senior"))
  Gender <- factor(Sex,levels=1:2,
         labels=c("Male","Female"))
  Color.grp <- factor(Color,
         levels=c("B","G","W","Y"),
         labels=c("Sky Blue",
                "Rain Cloud Gray",
                "Moon White",
                "Sunburst Yellow"))
  Color.heat <- Color.grp
  levels(Color.heat) <- list(
         Cold = c("Sky Blue","Rain Cloud Gray"),
         Neutral = c("Moon White"),
         Warm=c("Sunburst Yellow"))

}
)
```

This program creates two numerical based `factors`: `Gender` for the variable `Sex` and `Agegroup` for the variable `Age`. The program creates a character based `factor`: `Color.grp`. The `levels` function was called to combine several levels. The `within` function creates all factors within the data frame.

Here is the output.

```
##   Age Sex Income Color Color.heat       Color.grp Gender Agegroup
## 1  19   1  14000     Y       Warm Sunburst Yellow   Male     Teen
## 2  45   1  65000     G       Cold Rain Cloud Gray   Male    Adult
## 3  72   2  35000     B       Cold        Sky Blue Female   Senior
## 4  31   1  44000     Y       Warm Sunburst Yellow   Male    Adult
## 5  58   2  83000     W    Neutral      Moon White Female    Adult
```

By default, factor levels are treated in alphabetical order. If you want to change this (as you might, for instance, in ordering naturally ordered factor levels such as one, two, three etc), type the factor levels in the order that you want them to be used, and provide this vector as the second argument `levels` to the `factor` function.

```
attach(carsurvey)
levels(Color.grp) <- c("B","G","W","Y")
table(Color.grp)

## Color.grp
## B G W Y
## 1 1 1 2

levels(Color.grp) <- c("W","Y","G","B")
table(Color.grp)

## Color.grp
## W Y G B
## 1 1 1 2

detach(carsurvey)
```

4.4 Summarizing Your Data Using `aggregate`

One of the first things people usually want to do with their data, after reading it and making sure it is correct, is look at some simple statistics. Statistics such as the mean value, standard deviation, and minimum and maximum values give

you a feel for your data. These types of information can also alert you to errors in your data (a score of 980 in a basketball game, for example, is suspect). The `aggregate` function provides simple statistics on numeric variables.

Suppose that we have two numeric variables (`y` and `z`) and two categorical variables (`x` and `w`) that we might want to use `aggregate`to summarize functions like mean or variance of `y` and/or `z`. The `aggregate` function has a formula method which allows elegant summaries of four kinds:

one to one `aggregate(y~x, FUN=mean)`

one to many `aggregate(y~x+w,FUN=mean)`

many to one `aggregate(cbind(y,z)~x,FUN=mean)`

many to many `aggregate(cbind(y,z)~x+w,FUN=mean)`

If you specify `FUN` argument as `mean`, `aggregate` will print the means. There are many different statistics you can request with `aggregate` functions. The following is a list of some of the simple statistics.

R code	What it computes
`max`	the maximum value
`min`	the minimum value
`mean`	the mean
`median`	the median
`length`	number of values
`sd`	the standard deviation
`var`	the variance
`sum`	the sum

You can also specify functions that combine several statistics.

R code	What it computes
`range`	the range
`summary`	the Tukey's five number summary (min,Q1,median,Q3,max)

In addition, you can supply your own function to calculate customized statistics. For example, we can calculate the standard error of the mean.

```
aggregate(y~x,FUN=function(x) sd(x,na.rm=T)/sqrt(sum(!is.na(x))))
```

Example A wholesale nursery is selling garden flowers, and they want to summarize their sales figures by month. The data file which follows contains the customer ID, date of sale, and number of petunias, snapdragons, and marigolds sold:

```
756-01  05/04/2008 120  80 110
834-01  05/12/2008  90 160  60
901-02  05/18/2008  50 100  75
834-01  06/01/2008  80  60 100
756-01  06/11/2008 100 160  75
901-02  06/19/2008  60  60  60
756-01  06/25/2008  85 110 100
```

The following program reads the data; computes a new variable, `Month`, which is the month of the sale; then summarizes the data by Month using `aggregate`. The `colnames` function is used to name the summary statistics.

```
setwd("C:/Users/Slava/Dropbox/Documents/Teaching/MEES708XY/y2 Sorting/")
sales <- read.table("./dataRaw/Flowers.dat",
        head=F,col.names=c("CustomerID","SaleDate",
        "Petunia","SnapDragon","Marigold"))
sdate <- as.Date(sales$SaleDate,"%m/%d/%Y")
sales$Month <-months(sdate)
attach(sales)
## mean
aggregate(cbind(Petunia,SnapDragon,Marigold)~Month, FUN=mean)
```

```
##   Month Petunia SnapDragon Marigold
## 1  June      81         98       84
## 2   May      87        113       82

## range
aggregate(cbind(Petunia,SnapDragon,Marigold)~Month,FUN=range)

##   Month Petunia.1 Petunia.2 SnapDragon.1 SnapDragon.2 Marigold.1 Marigold.2
## 1  June        60       100           60          160         60        100
## 2   May        50       120           80          160         60        110

## standard error A custom function
aggregate(cbind(Petunia,SnapDragon,Marigold)~Month, FUN=function(x)
  sd(x,na.rm=T)/sqrt(sum(!is.na(x))))

##   Month Petunia SnapDragon Marigold
## 1  June     8.3         24      9.9
## 2   May    20.3         24     14.8

detach(sales)
```

4.5 Counting Your Data With `table`

A frequency table is a simple list of counts answering the question "How many?" When you have counts for one variable, they are called one-way frequencies. When you combine two or more variables, the counts are called two-way, three-way, and so on up to n-way frequencies; or simply cross-tabulations.

The most obvious reason for using `table` is to create tables showing the distribution of categorical data values, but `table` can reveal irregularities in your data. Errors are obvious in a frequency table.

The basic form of `table` is `table(variable combinations)`. To produce one-way frequency table, just list the variable name, `table(var1)`. To produce a cross-tabulation, list variables separated by an comma, `table(var1,var2)`.

The tables of proportions can be calculated using `prop.table`. The margins of a table (the row totals or the column totals) are often useful for calculating proportions instead of counts. Suppose `count <- table(var1,var2)` is the frequency counts, `prop.table(count,1)` calculates the row proportion, `prop.table(count,2)` calculates the column proportion, and `prop.table(count)` calculate the overall proportion. The corresponding sum totals can be found by the `margin.table` function. The `addmargins` function takes a frequency table and returns with it the additional marginal sums.

Example The data were extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (models). For each model, `cyl` denotes the number of cylinders and `gear` denotes the number of forward gears.

```
head(mtcars)

##                   mpg cyl disp  hp drat  wt qsec vs am gear carb
## Mazda RX4          21   6  160 110  3.9 2.6   16  0  1    4    4
## Mazda RX4 Wag      21   6  160 110  3.9 2.9   17  0  1    4    4
## Datsun 710         23   4  108  93  3.8 2.3   19  1  1    4    1
## Hornet 4 Drive     21   6  258 110  3.1 3.2   19  1  0    3    1
## Hornet Sportabout  19   8  360 175  3.1 3.4   17  0  0    3    2
## Valiant            18   6  225 105  2.8 3.5   20  1  0    3    1
```

The following program produces one-way and two-way frequencies:

```
attach(mtcars)
table(cyl)

## cyl
```

```
##  4  6  8
## 11  7 14

(m <- table(cyl, gear))

##    gear
## cyl  3  4  5
##   4  1  8  2
##   6  2  4  1
##   8 12  0  2
```

The output contains two tables. The first is a one-way frequency table for the variable `cyl`. The second table is a two-way cross-tabulation of `cyl` by `gear`. To calculate percentages, use `prob.table`.

```
# Relative percentage
prop.table(m)

##    gear
## cyl     3     4     5
##   4 0.031 0.250 0.062
##   6 0.062 0.125 0.031
##   8 0.375 0.000 0.062

# Row percentage
prop.table(m, 1)

##    gear
## cyl     3     4     5
##   4 0.091 0.727 0.182
##   6 0.286 0.571 0.143
##   8 0.857 0.000 0.143

# Column percentage
prop.table(m, 2)

##    gear
## cyl     3     4     5
##   4 0.067 0.667 0.400
##   6 0.133 0.333 0.200
##   8 0.800 0.000 0.400
```

It is also easy to calculate marginal distributions from the two way frequency table using `margin.table` or `addmargins`.

```
# Margin containing column sums
addmargins(m,1)

##      gear
## cyl    3  4  5
##   4    1  8  2
##   6    2  4  1
##   8   12  0  2
##   Sum 15 12  5

# Margin containing row sums
addmargins(m,2)

##    gear
```

```
## cyl  3  4  5 Sum
##   4  1  8  2  11
##   6  2  4  1   7
##   8 12  0  2  14

# Both Margins
addmargins(m)

##      gear
## cyl    3  4  5 Sum
##   4    1  8  2  11
##   6    2  4  1   7
##   8   12  0  2  14
##   Sum 15 12  5  32

detach(mtcars)
```

4.6 Creating Summary Reports with `ftable`

The `ftable` function can be used to produce frequency counts using a readable flat format. `ftable` is very flexible. we will illustrate it with the contingency count object.

The basic form is `ftable(count,row.vars,col.vars)`, here `count` is an frequency counts calculated using the `table` or `xtab` function. The `table` function has been described in the previous section. The `xtabs` function is similar to `table`, but requires each rows of the data to contain *frequency* counts instead of individuals. The arguments `row.vars` and `col.vars` denote the names of variables to be used for the rows or columns of the flat contingency tables.

Example Here are the data about national parks and monuments. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.

```
Dinosaur              NM West 2  6
Ellis Island          NM East 1  0
Everglades            NP East 5  2
Grand Canyon          NP West 5  3
Great Smoky Mountains NP East 3 10
Hawaii Volcanoes      NP West 2  2
Lava Beds             NM West 1  1
Statue of Liberty     NM East 1  0
Theodore Roosevelt    NP      2  2
Yellowstone           NP West 9 11
Yosemite              NP West 2 13
```

The following program reads the data. Since `Museums` and `Campings` denote the *frequency* counts, we use `xtab` to create a contingency table by region and type. The code contains two `ftable`. In the first, `Region` and `Type` are defined as the row variables. In the second, `Region` is still a group variable, but `Type` is an across variable.

```
setwd("C:/Users/Slava/Dropbox/Documents/Teaching/MEES708XY/y2 Sorting/")
natparks <- read.fwf("./dataRaw/Parks.dat",
        widths=c(22,2,5,3,3),
        col.names=c("Name","Type",
                "Region","Museums","Camping")
        )
count <- xtabs(cbind(Museums,Camping)~Region+Type,
        data=natparks)
ftable(count,row.vars=c("Region","Type"))

##              Museums Camping
```

```
## Region Type
##        NM          0       0
##        NP          2       2
##  East  NM          2       0
##        NP          8      12
##  West  NM          3       7
##        NP         18      29

ftable(count,row.vars="Region")

##        Type      NM              NP
##             Museums Camping Museums Camping
## Region
##                   0       0       2       2
##  East             2       0       8      12
##  West             3       7      18      29
```

4.7 Adding Statistics To Summary Table Using `tapply`

The function `tapply` is an important function in R for generating summary tables. It is called `tapply` because it applies a named function across specified margins (factor levels) to create a table. `tapply` was the underlying work horse function for `aggregate` described in the previous section.

Suppose you have a numeric variable `y` and three categorical variables `x_1`,. . .,`x_3`, and you want to apply a function `f` to `y` for each unique combinations of `x_1`,. . .,`x_3`, the basic form is

```
tapply(y,list(x1,x2,x3),f)
```

The output is a three dimensional *array*. The first dimension corresponds to variable `x_1`, the second to `x_2` and so on. The values are `f` applied to `y` to unique combinations of factors.

The *array* generated by `tapply` can be reported using `ftable`. This is especially useful for reporting high dimensional tables. The basic form is

```
ftable(array,row.vars,col.vars)
```

where `array` is the output from a `tapply` function. Similar to the above, the arguments `row.vars` and `col.vars` denote the names of variables to be used for the rows or columns of the output tables.

If you want to export the tabular output into Excel or Latex, the `format.ftable` function can be used to convert the results into a `data.frame`, which can be easily exported as a `csv` file.

Example Here again are the data about national parks and monuments. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds. We want the sample mean of the variable `Museums` by `Region` and `Type`.

```
Dinosaur              NM West 2  6
Ellis Island          NM East 1  0
Everglades            NP East 5  2
Grand Canyon          NP West 5  3
Great Smoky Mountains NP East 3 10
Hawaii Volcanoes      NP West 2  2
Lava Beds             NM West 1  1
Statue of Liberty     NM East 1  0
Theodore Roosevelt    NP      2  2
Yellowstone           NP West 9 11
Yosemite              NP West 2 13
```

The following program contains `tapply` statements that request the same mean.

```
setwd("C:/Users/Slava/Dropbox/Documents/Teaching/MEES708XY/y2 Sorting/")
natparks <- read.fwf("./dataRaw/Parks.dat",
        widths=c(22,2,5,3,3),
        col.names=c("Name","Type",
                "Region","Museums","Camping")
        )
attach(natparks)
ftable(tapply(Museums,list(Region,Type),mean))

##         NM  NP
##
##         NA 2.0
##  East  1.0 4.0
##  West  1.5 4.5
```

The following program requested sample size and sample standard deviation in addition to the sample mean. In order to report the results, we first create an additional dimension in the array to hold the summary statistics information, with the dimension appropriately named.

```
## by Region and Type
t1 <- tapply(Museums,list(Region,Type),length) # sample size
t2 <- tapply(Museums,list(Region,Type),mean) # mean
t3 <- tapply(Museums,list(Region,Type),sd) # standard deviation
detach(natparks)

## create a 3-D array
## dimension 1: Region
## dimension 2: Type
## dimension 3: summary statitics (n,mean,sd)
dimnamelist <- c(dimnames(t1), list(c("N","Mean","Std Dev"))) ## dimension names
dimvec <- c(dim(t1),3) ## dimension sizes
a3 <- array(c(t1,t2,t3),dim=dimvec, dimnames=dimnamelist) ## create this array
 ## note the last dimension stat changes the slowest
names(attr(a3,"dimnames")) <-  c("Region","Type","stats") ## 3 dimension titles
(t1 <- ftable(a3)) ## call ftable to visualize the results

##             stats    N Mean Std Dev
## Region Type
##        NM           NA   NA      NA
##        NP         1.00 2.00      NA
##  East  NM         2.00 1.00    0.00
##        NP         2.00 4.00    1.41
##  West  NM         2.00 1.50    0.71
##        NP         4.00 4.50    3.32
```

We need the `format.ftable` function to export the table to a `csv` file, which can be easily incorporated into a report.

```
write.csv(format(t1,method="compact",quote=F), file="table1.csv",row.names=F)
```

Region	Type	N	Mean	Std Dev
	NM	NA	NA	NA
	NP	1.00	2.00	NA
East	NM	2.00	1.00	0.00
	NP	2.00	4.00	1.41
West	NM	2.00	1.50	0.71
	NP	4.00	4.50	3.32

4.8 Exercises

The `birthwt` data set gives characteristics on 189 randomly chosen births at a US hospital, with the main interest being in low birth weight `low`.

```
library(MASS)
data(birthwt)
```

1. Code the age of mother in years `age` as less than 20, 20-35 or greater than 35.
2. Count the number of normal or low birth weight births according to `race`, coded age, smoking status `smoke`, and history of hypertension `ht`.
3. Calculate the percent of each race, age group, smoking status and hypertension according to low or normal birth respectively.
4. Report the data using a flat table, with `low` as the column variable and the others as row variables.