

Lab 1 Supplement: R Reference Guide

Statistics 139 (special thanks to Julie Vu!)

September 08, 2023

Data Structures

There are several **data types** in R: character, numeric, integer, complex, and logical. The most common data types are character and numeric, where numeric specifically refers to real-valued numbers; characters can be thought of as text labels. Integers and complex numbers can be stored as integer and complex values.

Data structures are different formats for storing data: vectors, matrices, lists, data frames, and factors.

Vectors

A **vector** is a one-dimensional collection of values that are all the same data type; in colloquial language, a vector might simply be referred to as a “list”. For example, a numeric vector is one in which all elements of the vector are real numbers.

The `c()` command concatenates (i.e., combines) elements into a vector. The following code creates a numeric vector containing the values 1, 1.5, 2, 2.5, and 3, which is stored as a variable `x`. The variable `y` is an integer vector containing the integers 5 through 9. The `class()` command identifies the data structure of an object.

```
x = c(1, 1.5, 2, 2.5, 3)
y = 5:9

class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

```
## [1] "integer"
```

When creating a character vector, each element must be contained within quotes to denote it is a character, rather than a variable. Below, the vector `a` is a character vector with the elements `x` and `y`. However, the vector `b` is a numeric vector formed by combining the variables `x` and `y` as defined previously.

```
a = c("x", "y")
a
```

```
## [1] "x" "y"
```

```
b = c(x, y)
b
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 5.0 6.0 7.0 8.0 9.0
```

Operations on vectors are performed element by element; for example, if two vectors are added together, the first element in the resulting vector is the sum of the first elements of the two vectors. If two vectors are not the same length, the shorter vector will be repeated until it is the same length as the longer vector; below, note how the 6th element of **z** is created by summing the 6th element of the longer vector (**b**) with the 1st element of the shorter vector (**y**).

```
x + y
```

```
## [1] 6.0 7.5 9.0 10.5 12.0
```

```
x^2
```

```
## [1] 1.00 2.25 4.00 6.25 9.00
```

```
2*x
```

```
## [1] 2 3 4 5 6
```

```
z = y + b
z
```

```
## [1] 6.0 7.5 9.0 10.5 12.0 10.0 12.0 14.0 16.0 18.0
```

Vectors are indexed, in that each element's position in the vector is identified by a number. To access specific elements in a vector, use square brackets.

```
z[6] #extracts 6th element of z
```

```
## [1] 10
```

```
z[1:3] #extracts elements 1 through 3 of z
```

```
## [1] 6.0 7.5 9.0
```

```
z[c(6, 1:3)] #extracts elements 6 and 1 through 3 of z
```

```
## [1] 10.0 6.0 7.5 9.0
```

```
z[-c(1, 10)] #extracts all elements of z except for elements 1 and 10
```

```
## [1] 7.5 9.0 10.5 12.0 10.0 12.0 14.0 16.0
```

Other useful vector operations exist; a few examples are provided below.

```
sum(x) #sum of all elements in x
```

```
## [1] 10
```

```
prod(x) #product of all elements in x
```

```
## [1] 22.5
```

```
min(x)           #minimum of all elements in x
```

```
## [1] 1
```

```
max(x)           #maximum of all elements in x
```

```
## [1] 3
```

```
length(x)        #length of x; i.e., number of elements in x
```

```
## [1] 5
```

Matrices

A **matrix** is a two-dimensional collection of values that are all the same data type. The dimensions of a matrix are its number of rows and number of columns.

There are various ways to construct a matrix. One is to bind together vectors by either the columns with `cbind()` or by the rows with `rbind()`. Note how the columns in the first matrix are automatically labeled with `x` and `y`, and the same for the rows in the second matrix. Row and column names can also be explicitly specified.

```
matrix.a = cbind(x, y)
matrix.a
```

```
##      x y
## [1,] 1.0 5
## [2,] 1.5 6
## [3,] 2.0 7
## [4,] 2.5 8
## [5,] 3.0 9
```

```
matrix.b = rbind(x, y)
matrix.b
```

```
##  [,1] [,2] [,3] [,4] [,5]
## x    1  1.5    2  2.5    3
## y    5  6.0    7  8.0    9
```

```
rownames(matrix.a) = c("A", "B", "C", "D", "E")
matrix.a
```

```
##      x y
## A 1.0 5
## B 1.5 6
## C 2.0 7
## D 2.5 8
## E 3.0 9
```

```
colnames(matrix.b) = c("A", "B", "C", "D", "E")
matrix.b
```

```
##  A   B C   D E
```

```
## x 1 1.5 2 2.5 3
## y 5 6.0 7 8.0 9
```

To create a matrix with the `matrix()` command, specify a vector containing the elements of the matrix, the number of rows, and the number of columns. By default, the matrix is filled by column, with the first column being filled top to bottom before moving to the next column. To fill the matrix by row, with the first row being filled left to right before moving down to the next row, specify `byrow = TRUE`.

```
matrix.c = matrix(1:9, nrow = 3, ncol = 3, byrow = FALSE)
matrix.c
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
matrix.d = matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
matrix.d
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Operations on matrices are also performed element by element.¹

```
matrix.c + matrix.d
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    6   10   14
## [3,]   10   14   18
```

```
matrix.c*matrix.d
```

```
##      [,1] [,2] [,3]
## [1,]    1    8   21
## [2,]    8   25   48
## [3,]   21   48   81
```

```
matrix.c^2
```

```
##      [,1] [,2] [,3]
## [1,]    1   16   49
## [2,]    4   25   64
## [3,]    9   36   81
```

```
2*matrix.c
```

```
##      [,1] [,2] [,3]
## [1,]    2    8   14
## [2,]    4   10   16
```

¹Matrix multiplication, in the matrix algebra sense, must be specified with the operator `%*%`.

```
## [3,]    6   12   18
```

As with vectors, specific elements of a matrix are indexed and can be accessed using square bracket notation.

```
matrix.c[1, 2]      #extracts the element in row 1, column 2 of matrix.c
```

```
## [1] 4
```

```
matrix.c[1, 1:2]    #extracts the elements in row 1, columns 1 and 2 of matrix.c
```

```
## [1] 1 4
```

```
matrix.c[1, ]      #extracts all elements in row 1 of matrix.c
```

```
## [1] 1 4 7
```

```
matrix.c[, 1]      #extracts all elements in column 1 of matrix.c
```

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

```
matrix.c[, -1]     #extracts all elements in all columns of matrix.c except col 1
```

```
##      [,1] [,2]
```

```
## [1,]    4    7
```

```
## [2,]    5    8
```

```
## [3,]    6    9
```

Data Frames

A **list** is a one-dimensional collection of values that are different data types; i.e., a vector that contains more than one data type. This is a useful format for outputting function results; lists will be seen later on beginning in Unit 4. In this course, creating lists will not be necessary.

```
list(a, b)
```

```
## [[1]]
```

```
## [1] "x" "y"
```

```
##
```

```
## [[2]]
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 5.0 6.0 7.0 8.0 9.0
```

Just as a matrix is the two-dimensional analog of a vector, a **data frame** is the two-dimensional analog of a list. A data frame contains vectors of the same length, where the vectors can be of different types. This is often the most convenient format for storing a statistical dataset.

The following code creates a small data frame called `patient.info` from four vectors: `id`, `weight`, `gender`, and `age`.

```
id = c("A", "B", "C", "D", "E")
weight = c(210, 140, 120, 180, 160)
gender = c("male", "female", "female", "male", "male")
age = c(22, 45, 35, 50, 70)
```

```
patient.info = data.frame(id, weight, gender, age)
patient.info
```

```
##   id weight gender age
## 1  A    210   male  22
## 2  B    140 female  45
## 3  C    120 female  35
## 4  D    180   male  50
## 5  E    160   male  70
```

Square bracket notation applies to data frames just as for matrices. With data frames, a specific column can be accessed using the dollar sign symbol with the column name.

```
patient.info[, 2]           #elements in second column of patient.info
```

```
## [1] 210 140 120 180 160
```

```
patient.info[1, 2]         #first element of second column of patient.info
```

```
## [1] 210
```

```
patient.info$weight        #elements in weight column of patient.info
```

```
## [1] 210 140 120 180 160
```

```
patient.info$weight[1]     #first element in weight column of patient.info
```

```
## [1] 210
```

Square bracket notation can also be used with row and column names (specified in quotation marks), rather than the numerical indices.

The default in R when creating data frames is to convert any character vectors to factors. A **factor** is a specific data structure for values that come from a fixed set of possible values, and is ideal for storing categorical data. When `patient.info` was created, the character vector `gender` was converted into a factor variable with two levels, `female` and `male`. To prevent a character vector from being converted into a factor, use `I()` to indicate that the object should remain as-is.²

```
class(patient.info$gender)
```

```
## [1] "character"
```

```
levels(patient.info$gender)
```

```
## NULL
```

```
class(patient.info$id)
```

```
## [1] "character"
```

```
patient.info = data.frame(I(id), weight, gender, age) #rebuild patient.info
class(patient.info$id)                               #id remains a character vector
```

²In the *RStudio* interface, the data types of objects are visible in the Environment tab. To view the data types of objects within a data frame, select the blue arrow to the left of the name of the data frame.

```
## [1] "AsIs"
```

Creation of factor variables are not discussed here.

Some useful commands for viewing features of a dataframe are shown below.

```
nrow(patient.info)  #number of rows in patient.info
```

```
## [1] 5
```

```
ncol(patient.info)  #number of columns in patient.info
```

```
## [1] 4
```

Numerical and Graphical Summaries

Numerical Summaries

The following code demonstrates several functions that produce numerical summaries.

The `summary()` command applied to a (non-character) vector returns the minimum, first quartile, median, mean, third quartile, and maximum values of the vector. If there are any missing values, which R denotes as the value NA, the number of missing values is also reported.

The `quantile()` command returns quantiles corresponding to the specified probabilities. To produce the first quartile (i.e., the 25th percentile), specify `probs = 0.25`; several probabilities can also be specified.

Functions like `mean()`, `median()`, `IQR()`, etc. will either return NA or an error message if the vector contains missing values. To specify that the missing values should be omitted before the summary is computed, specify `na.rm = TRUE`. This is shown below with the `body.size` variable.

```
frog=read.csv("data/frog.csv")
```

```
summary(frog$clutch.volume)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    151.4   609.6   831.8   882.5  1096.5  2630.3
```

```
#range, minimum, maximum
```

```
range(frog$clutch.volume)
```

```
## [1] 151.3561 2630.2680
```

```
min(frog$clutch.volume)
```

```
## [1] 151.3561
```

```
max(frog$clutch.volume)
```

```
## [1] 2630.268
```

```
#median, quantiles
```

```
median(frog$clutch.volume)
```

```
## [1] 831.7638
```

```
quantile(frog$clutch.volume, probs = 0.25)
```

```
##      25%  
## 609.5773
```

```
quantile(frog$clutch.volume, probs = c(0.25, 0.50, 0.75))
```

```
##      25%      50%      75%  
## 609.5773 831.7638 1096.4782
```

```
#measures of spread  
IQR(frog$clutch.volume)
```

```
## [1] 486.9009
```

```
var(frog$clutch.volume)
```

```
## [1] 143680.9
```

```
sd(frog$clutch.volume)
```

```
## [1] 379.0527
```

```
#effect of missing values  
summary(frog$body.size)
```

```
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.   NA's  
##   3.631  4.677   5.248   5.096  5.623   6.166    302
```

```
mean(frog$body.size)
```

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

```
mean(frog$body.size, na.rm = TRUE)
```

```
## [1] 5.096367
```

The correlation between two variables x and y can be calculated using `cor(x, y)`. To calculate a correlation after dropping missing values, use the argument `use = "complete.obs"`.

```
cor(frog$body.size, frog$clutch.volume, use = "complete.obs")
```

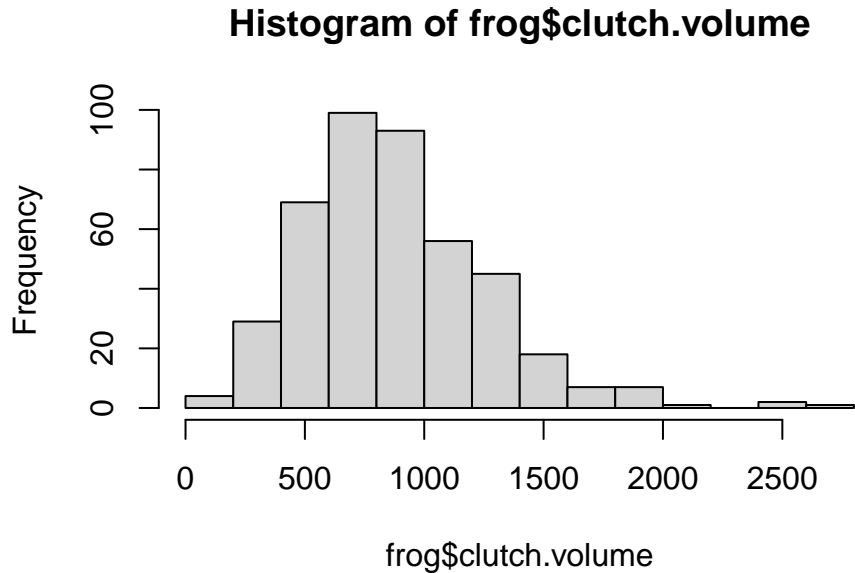
```
## [1] 0.6755435
```

Graphical Summaries

Simple plots can be created with very few specifications, since R will make default choices about marking data points, labeling axes, etc. The full details of optional plotting arguments are explained in the R help files and in various freely accessible resources online.

To create a histogram from the data values in a vector x , use `hist(x)`. The basic plot will label the x -axis with the name of the vector, x . The optional arguments allow for control over how many bins are used, the range of values for the y -axis, the axis labels, plot title, etc.

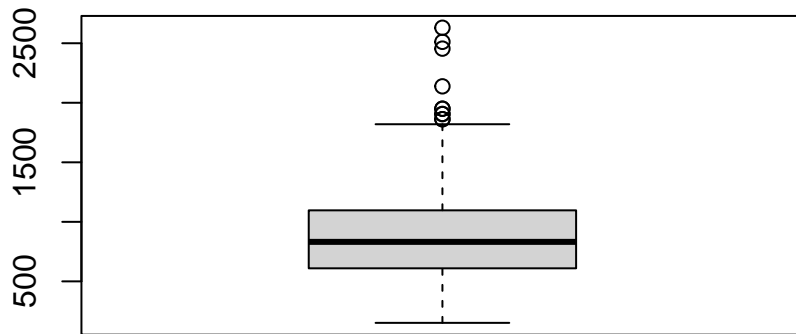

```
#basic histogram
hist(frog$clutch.volume)
```



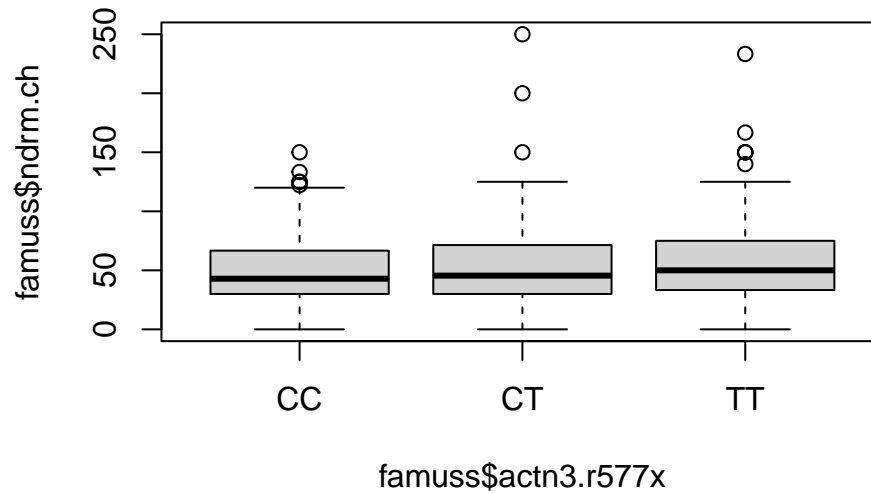
Similarly, to create a boxplot from the data values in a vector `x`, use `boxplot(x)`. The syntax for creating a side-by-side boxplot depends on how the data are organized. To plot the distribution of a variable `y` separately by the groups defined by a variable `x`, use `boxplot(y ~ x)`. To simply plot the distributions of variables `v` and `w` next to each other, use `boxplot(v, w)`.

```
#load famuss data
famuss = read.csv("data/famuss.csv")

#basic boxplot: single variable
boxplot(frog$clutch.volume)
```

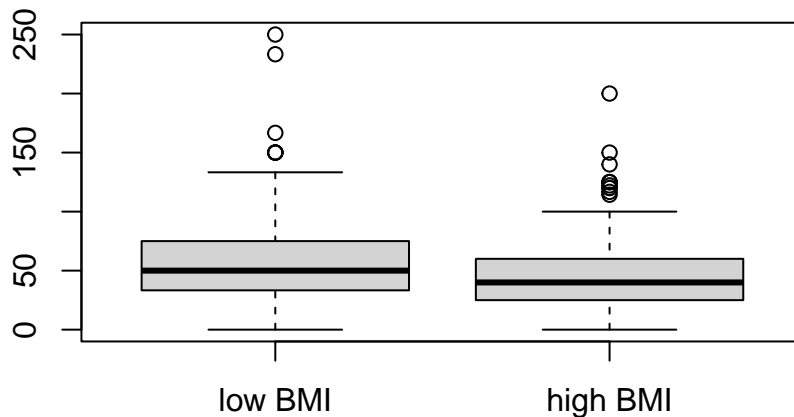


```
#basic boxplot: y ~ x
boxplot(famuss$ndrm.ch ~ famuss$actn3.r577x)
```



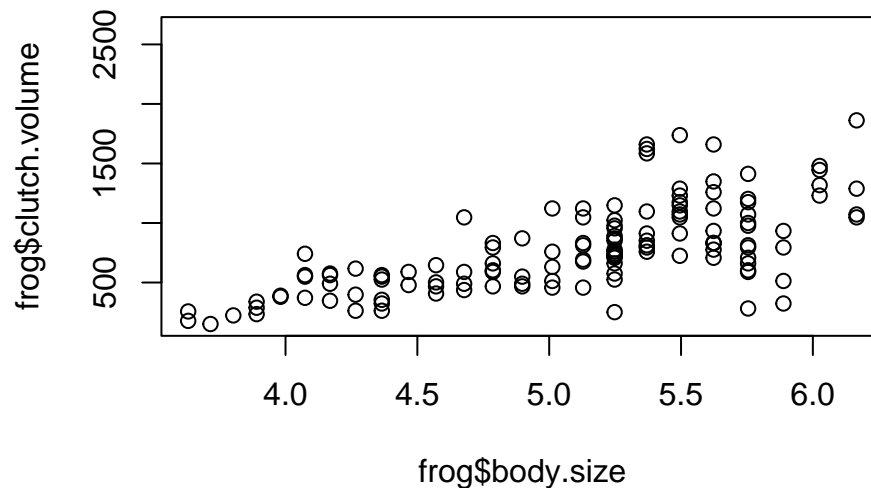
To plot the distributions of two variables v and w in a side-by-side boxplot, use `boxplot(v, w)`. The boxplots can be labeled using the optional argument `names`. The following code produces a side-by-side boxplot of change in non-dominant arm strength for participants with BMI less than 25 and with BMI greater than or equal to 25.

```
#basic boxplot: v, w
boxplot(famuss$ndrm.ch[famuss$bmi < 25], famuss$ndrm.ch[famuss$bmi >= 25],
        names = c("low BMI", "high BMI"))
```



To create a scatterplot of the variable y and the variable x , where y is on the y -axis and x is on the x -axis, use either `plot(y ~ x)` or `plot(x, y)`.

```
#basic scatterplot
plot(frog$clutch.volume ~ frog$body.size)
```



Subsetting Data

The `subset()` command is one way to create subsets of vectors, matrices, or dataframes that meet specific conditions. The conditions are specified using logical operators that express “less than”, “greater than or equal to”, “exactly equal to”, etc.

The following code creates subsets from the `famuss` data that contain only the rows (i.e., participants) that are older than 27, 27 or younger, or 27 years old. The `<=` and `>=` denote \leq and \geq . Note that two equal signs (`==`) are required to specify exactly equal to.

The `subset()` command can also be used on character vectors like `sex`.

```
famuss.older = subset(famuss, famuss$age > 27)
range(famuss.older$age)
```

```
## [1] 28 40
```

```
famuss.younger = subset(famuss, famuss$age <= 27)
range(famuss.younger$age)
```

```
## [1] 17 27
```

```
famuss.27 = subset(famuss, famuss$age == 27)
range(famuss.27$age)
```

```
## [1] 27 27
```

```
famuss.females = subset(famuss, famuss$sex == "Female")
summary(famuss.females$sex)
```

```
##      Length      Class      Mode
##      353 character character
```

Subsetting can also be done through bracket notation and logical operators. The following code creates a subset of males, a subset of males over 25, and a subset of individuals who are either male or over 25. The logical “and” is denoted by `&` and the logical “or” is denoted by `|`. Note that in the last subset, males of any age are included, while females must be older than 25.

```
famuss.males = famuss[famuss$sex == "Male", ]
summary(famuss.males$sex)

##      Length      Class      Mode 
##      242 character character 

famuss.males.and.over25 = famuss[famuss$sex == "Male" & famuss$age > 25, ]
range(famuss.males.and.over25$age); summary(famuss.males.and.over25$sex)

## [1] 26 40

##      Length      Class      Mode 
##      82 character character 

famuss.males.or.over25 = famuss[famuss$sex == "Male" | famuss$age > 25, ]
females = famuss.males.or.over25[famuss.males.or.over25$sex == "Female", ]
range(females$age)

## [1] 26 40
```

Tables

The `table()` command produces a 2×2 contingency table of counts at each combination of factor levels for two specified variables. Running `table(x, y)` results in a table where `x` specifies the rows and `y` specifies the columns. The `addmargins()` command prints the row and column sums on the sides of the tables.

```
#load data
LEAP = read.csv("data/LEAP.csv")

#simple table
table(LEAP$treatment.group, LEAP$overall.V60.outcome)

##
##              FAIL OFC PASS OFC
## Peanut Avoidance      36    227
## Peanut Consumption      5    262

#OI Biostat Table 1.2
addmargins(table(LEAP$treatment.group, LEAP$overall.V60.outcome))

##
##              FAIL OFC PASS OFC Sum
## Peanut Avoidance      36    227 263
## Peanut Consumption      5    262 267
## Sum                   41    489 530
```

Using `table()` on a single factor produces the counts in each level; using `summary()` on a factor has the same result.

```
table(LEAP$treatment.group)
```

```
##
```

```
## Peanut Avoidance Peanut Consumption
##                263                267
```

```
summary(LEAP$treatment.group)
```

```
##      Length      Class      Mode
##      530 character character
```

Barplots

The command `prop.table()` converts a table of counts to a table of proportions. To produce a bar plot, apply `barplot()` to the table. A bar plot of counts can also be produced by directly applying `plot()` to a categorical variable; this method is shown in the lab solutions.

```
table(famuss$actn3.r577x)
```

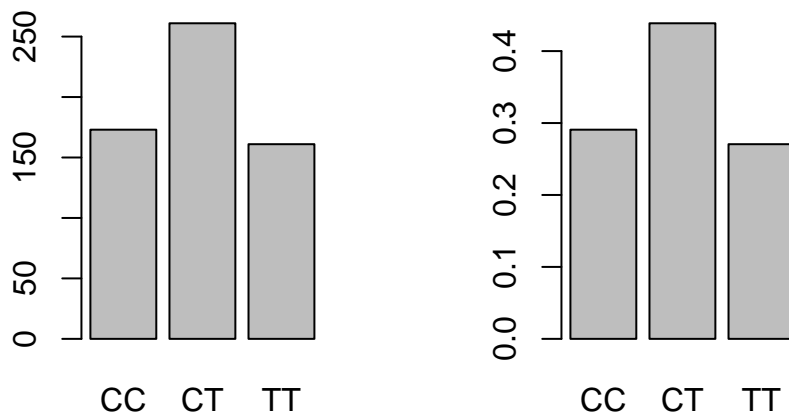
```
##
## CC CT TT
## 173 261 161
```

```
prop.table(table(famuss$actn3.r577x))
```

```
##
##      CC      CT      TT
## 0.2907563 0.4386555 0.2705882
```

```
#OI Biostat Figure 1.25
```

```
par(mfrow = c(1, 2))
barplot(table(famuss$actn3.r577x))
barplot(prop.table(table(famuss$actn3.r577x)))
```



A segmented bar plot is produced from applying `barplot()` to a contingency table of counts. A standardized segmented bar plot is produced from applying `barplot()` to a contingency table of proportions. When using `prop.table()` on a contingency table, specify 2 to calculate column proportions; specify 1 to calculate row proportions.

```
addmargins(table(LEAP$overall.V60.outcome, LEAP$treatment.group))
```

```
##
## Peanut Avoidance Peanut Consumption Sum
```

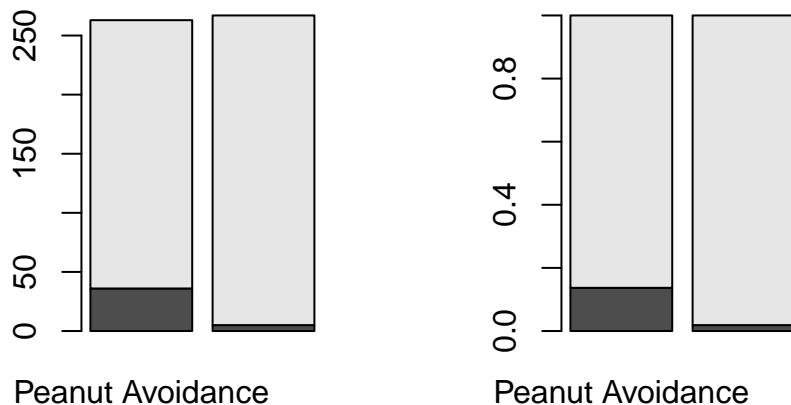
```
##   FAIL OFC           36           5  41
##   PASS OFC          227          262 489
##   Sum              263          267 530
```

```
prop.table(table(LEAP$overall.V60.outcome, LEAP$treatment.group), 2)
```

```
##
##           Peanut Avoidance Peanut Consumption
##   FAIL OFC      0.13688213      0.01872659
##   PASS OFC      0.86311787      0.98127341
```

#OI Biostat Figure 1.3

```
par(mfrow = c(1, 2))
barplot(table(LEAP$overall.V60.outcome, LEAP$treatment.group))
barplot(prop.table(table(LEAP$overall.V60.outcome, LEAP$treatment.group), 2))
```



Logical Values

A **logical** vector contains elements that can take on either of two pre-defined values: **TRUE** or **FALSE**. Logical vectors are typically created from the use of logical operators. The following code repeats the creation of `famuss.males`, but explicitly shows how the condition `famuss$sex == "Male"` is a logical vector specifying which rows to extract from `famuss`. R extracts the rows with indices that have value **TRUE** in the logical vector; i.e., rows for which the variable `sex` has value "Male".

```
male.rows = (famuss$sex == "Male")
head(male.rows)
```

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

```
class(male.rows)
```

```
## [1] "logical"
```

```
famuss.males = famuss[male.rows, ]           #vector stored as male.rows
famuss.males = famuss[famuss$sex == "Male", ] #syntax used in Lab 1 Notes
```

In various programming contexts, it can sometimes be useful to express a logical vector as a numeric vector. R always associates the logical value **TRUE** with the numeric value 1, and the logical value **FALSE** with the numeric value 0. This idea will be revisited in later chapters.

```
male.rows.numeric = as.numeric(famuss$sex == "Male")
head(male.rows.numeric)
```

```
## [1] 0 1 0 0 0 0
```

```
class(male.rows.numeric)
```

```
## [1] "numeric"
```

The `tapply()` Function

The `tapply()` function allows a specific function to be applied to a matrix; the function can be a predefined R function like `mean()` or a user-defined function. The power of `tapply()` is that it allows for a vector to be split into groups, with the function applied to each group.

The function has the generic structure

```
tapply(y, x, FUN)
```

where `y` is the vector of data, `x` is the grouping variable, and `FUN` is the function of interest.

The following code shows how to calculate the mean change in non-dominant arm strength for each genotype group in the FAMuSS data.

```
#load the data
```

```
famuss = read.csv("data/famuss.csv")
```

```
tapply(famuss$ndrm.ch, famuss$actn3.r577x, mean)
```

```
##          CC          CT          TT
## 48.89422 53.24904 58.08385
```

```
# alternatively the by command can do the same calculations
```

```
by(famuss$ndrm.ch, famuss$actn3.r577x, mean)
```

```
## famuss$actn3.r577x: CC
```

```
## [1] 48.89422
```

```
## -----
```

```
## famuss$actn3.r577x: CT
```

```
## [1] 53.24904
```

```
## -----
```

```
## famuss$actn3.r577x: TT
```

```
## [1] 58.08385
```