Lab Notes

Chapter 1

OpenIntro Biostatistics

Overview

1. Introduction to Data
   - *OI Biostat* Sections 1.1 - 1.2, 1.4 - 1.6
2. Exploratory Data Analysis: DDS Case Study
   - *OI Biostat* Section 1.7.1
3. Exploratory Data Analysis: Golub Case Study
   - *OI Biostat* Section 1.7.2
4. Exploratory Data Analysis: Arenosa Case Study
   - *OI Biostat* Section 1.7.3

Lab 1 introduces the basic commands for working with data, including those for manipulating dataframes, producing numerical and graphical summaries, and drawing pseudorandom samples. The questions in the last section focus on interpreting summaries and exploring relationships between variables.

Lab 2 provides a walkthrough for conducting an exploratory analysis of a dataset in which confounding may be present. Most of the commands required were introduced previously in Lab 1; this lab more heavily emphasizes interpretation.

Lab 3 demonstrates the use of statistical first principles in conducting a basic analysis of a small microarray dataset. New commands for manipulating matrices are introduced, along with a first look at control structures.

Lab 4 reinforces the commands introduced in Lab 3 and data interpretation skills from Labs 1 and 2, in the context of an RNA sequencing dataset. Both Labs 3 and 4 demonstrate how computing is essential for data analysis; even though the two datsets are relatively small by modern standards, they are already too large to feasibly analyze without statistical computing software.

Lab 1: Introduction to Data

While the following notes are a more systematic introduction to R than the lab exercises, we highly recommend working through the labs before consulting these notes. The labs introduce R functions as needed, and the surrounding exercise structure helps develop intuition for when certain functions are useful. We suggest that these notes are used for review or as a reference.

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. Logical values will be discussed in the Lab 3 notes.

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 6^{th} element of z is created by summing the 6^{th} element of the longer vector (b) with the 1^{st} 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
```

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
```

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

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

```r
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.

```r
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
```

```r
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
```

```r
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
```

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

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

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, rather than the numerical indices. The following code reproduces *OI Biostat* Table 1.6, which shows four rows from the famuss data.[2]

```
#load the data
library(oibiostat)
data("famuss")

#OI Biostat Table 1.6
famuss[c(1, 2, 3, 595), c("sex", "age", "race", "height", "weight",
                          "actn3.r577x", "ndrm.ch")]
```

```
##         sex age      race height weight actn3.r577x ndrm.ch
## 1    Female  27 Caucasian   65.0    199          CC    40.0
## 2      Male  36 Caucasian   71.7    189          CT    25.0
## 3    Female  24 Caucasian   65.0    134          CT    40.0
## 1348 Female  30 Caucasian   64.0    134          CC    43.8
```

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

[2] The last row is labeled 1348 because the famuss data is a subset of a larger dataset; the 595^{th} row in this version of famuss contains the information from participant 1348 in the original dataset.

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.[3]

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

```
## [1] "factor"
```

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

```
## [1] "female" "male"
```

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

```
## [1] "factor"
```

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

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

Creation of factor variables will be discussed in Chapter 6.

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 25^{th} 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.

```
library(oibiostat)
data("frog")
```

[3] In 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.

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

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

```r
#range, minimum, maximum
range(frog$clutch.volume)
```

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

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

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

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

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

```r
#median, quantiles
median(frog$clutch.volume)
```

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

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

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

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

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

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

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

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

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

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

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

```r
#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
```

```r
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")
```

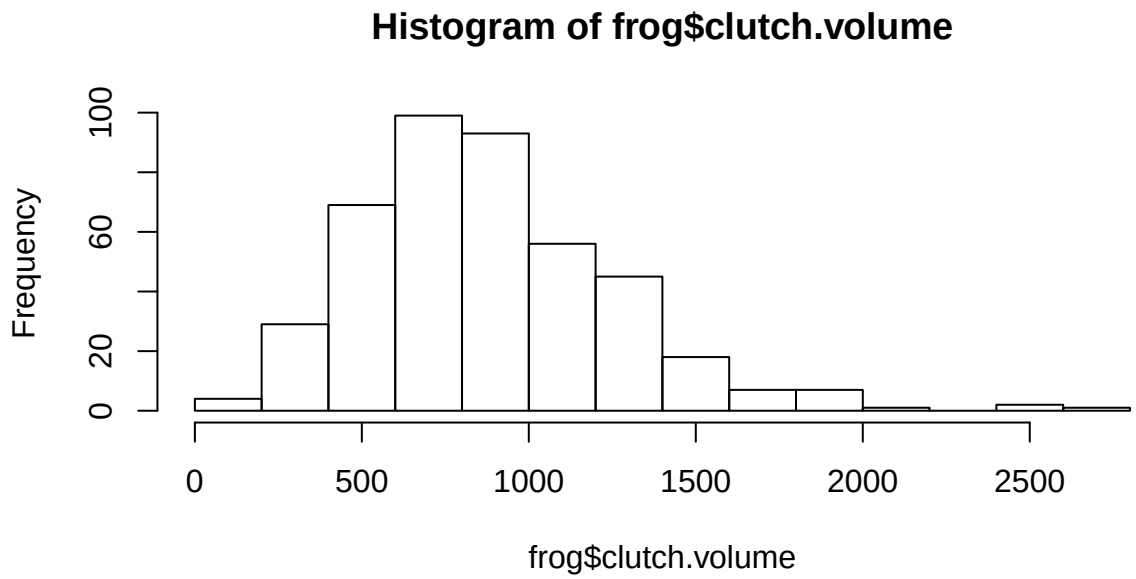
```
## [1] 0.6755435
```

Graphical Summaries

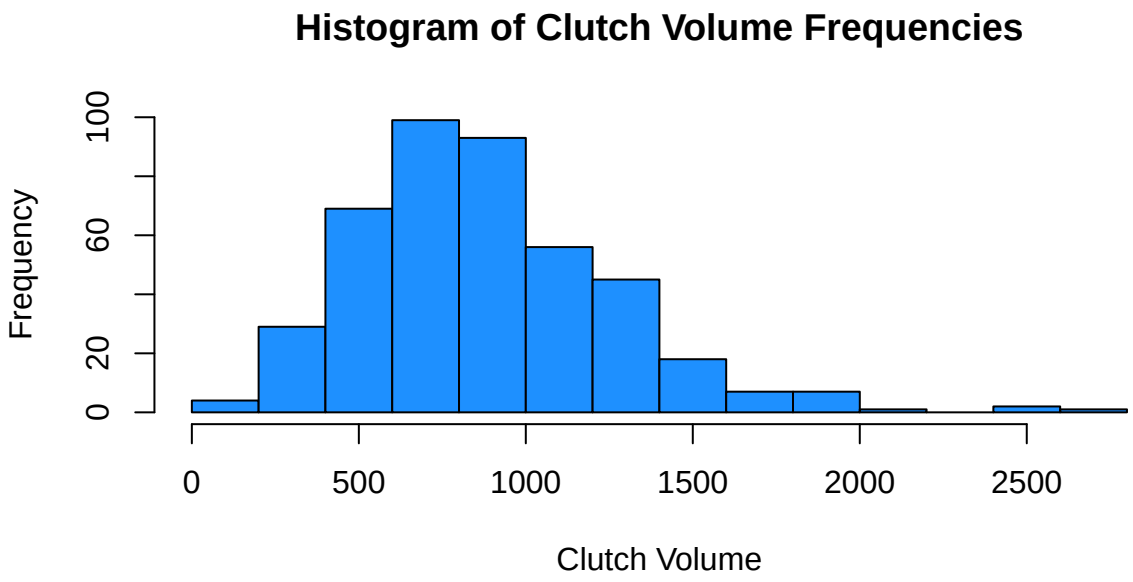
Simple plots can be created with very few specifications, since R will make default choices about marking data points, labeling axes, etc. The full details of optional plotting arguments are explained in the R help files and in various freely accessible resources online. These notes include the minimum information needed to get started with plotting and show some examples of plots that make use of optional arguments.

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)
```

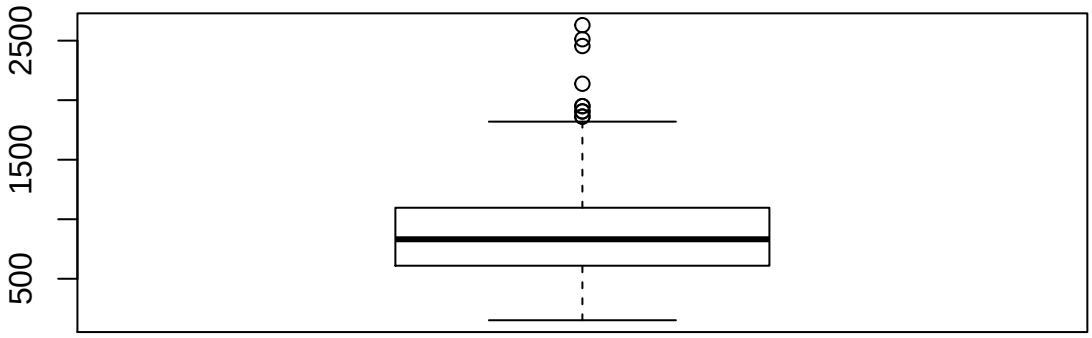



```
#OI Biostat Figure 1.18
hist(frog$clutch.volume, breaks = 14, col = "dodgerblue",
     xlab = "Clutch Volume", ylab = "Frequency", ylim = c(0, 100),
     main = "Histogram of Clutch Volume Frequencies")
```

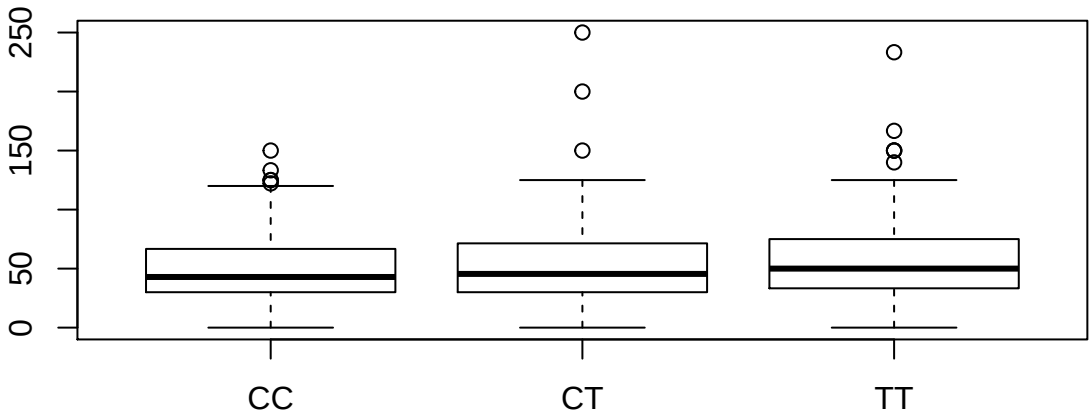


Similarly, to create a boxplot from the data values in a vector `x`, use `boxplot(x)`. The syntax for creating a side-by-side boxplot depends on how the data are organized. To plot the distribution of a variable y separately by the groups defined by a variable x, use `boxplot(y ~ x)`. To simply plot the distributions of variables v and w next to each other, use `boxplot(v, w)`. An example of the second method is shown in Lab 2; refer to the notes for Lab 2 for additional details.

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

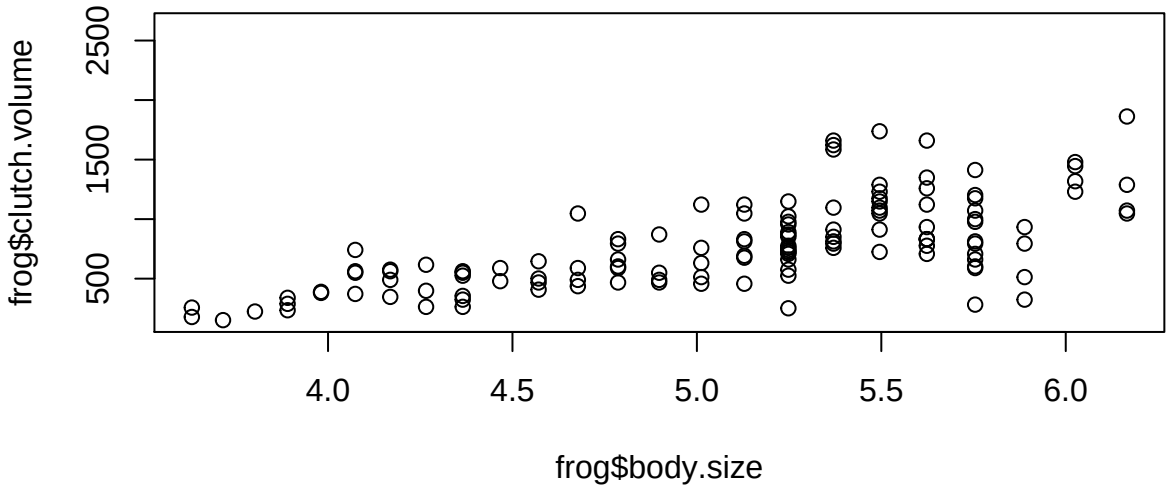


```
#basic boxplot: y ~ x
boxplot(famuss$ndrm.ch ~ famuss$actn3.r577x)
```

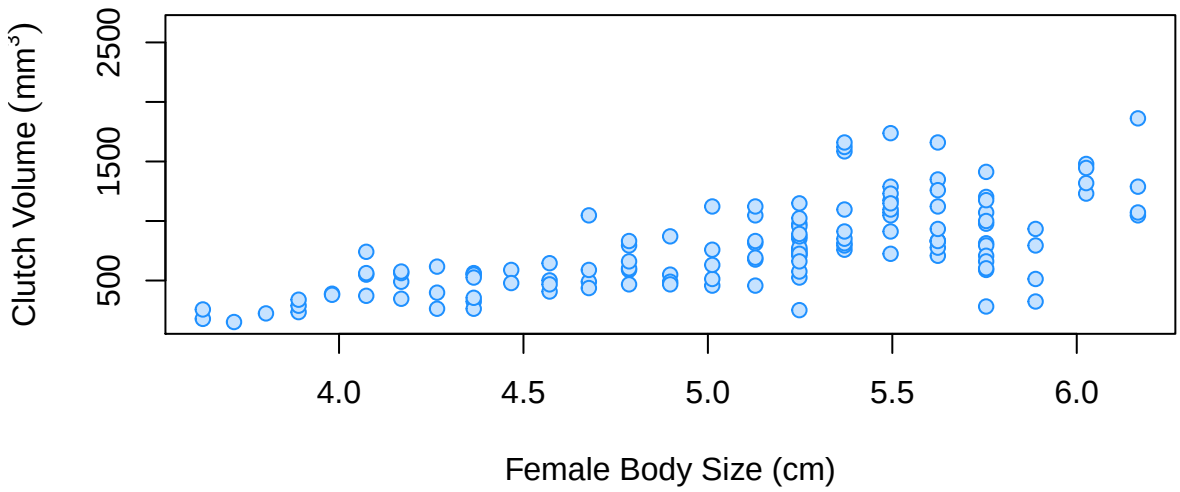


To create a scatterplot of the variable y and the variable x, where y is on the y-axis and x is on the x-axis, use either `plot(y ~ x)` or `plot(x, y)`.

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



```
#OI Biostat Figure 1.26
plot(frog$clutch.volume ~ frog$body.size, col = "dodgerblue",
     bg = "slategray1", pch = 21,
     xlab = "Female Body Size (cm)",
     ylab = expression("Clutch Volume" ~ (mm^3)))
```



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 ≤ and ≥. 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)
```

```
## Female   Male
##    353      0
```

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)
```

```
## Female   Male
##      0    242
```

```
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
```

```
## Female   Male
##      0     82
```

```
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
```

Pseudorandom Sampling

The `sample()` command takes a random sample of a specified size from the elements of a vector, either with or without replacement.

To be able to retrieve a sample that has been generated, it is necessary to associate the sample with a seed (i.e., a tag). Setting the seed allows for the same sample to be generated with a subsequent run of `sample()`.

The following code shows samples of size 10 drawn from the integers 1 through 15, either with replacement or without replacement. In the first run, it is possible to obtain a specific number more than once; this is not possible in the other runs. The first two runs are not associated with a seed. The second two are associated with the seed `2018`; running `sample()` with these specifications and seed will always result in the sample 6, 7, 1, 3, 15, 4, 11, 2, 14, 10. Note how the last two runs result in a different sample than the second run.

```
#completely random
sample(1:15, size = 10, replace = TRUE)
```

```
##  [1]  5 14 14 15 12 12 11 14 10 15
```

```r
sample(1:15, size = 10, replace = FALSE)
```

```
##  [1]  5  4 11  6  3  7  9  2 15 14
```

```r
#set.seed
set.seed(2018)
sample(1:15, size = 10, replace = FALSE)
```

```
##  [1]  6  7  1  3 15  4 11  2 14 10
```

```r
set.seed(2018)
sample(1:15, size = 10, replace = FALSE)
```

```
##  [1]  6  7  1  3 15  4 11  2 14 10
```

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.

```r
#load data
data("LEAP")

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

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

```r
#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.

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

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

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

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

Lab 2: DDS Case Study

This lab relies heavily on subsetting and producing graphical and numerical summaries for categorical data. Subsetting was discussed in Lab 1, while tables and barplots are discussed below.

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.

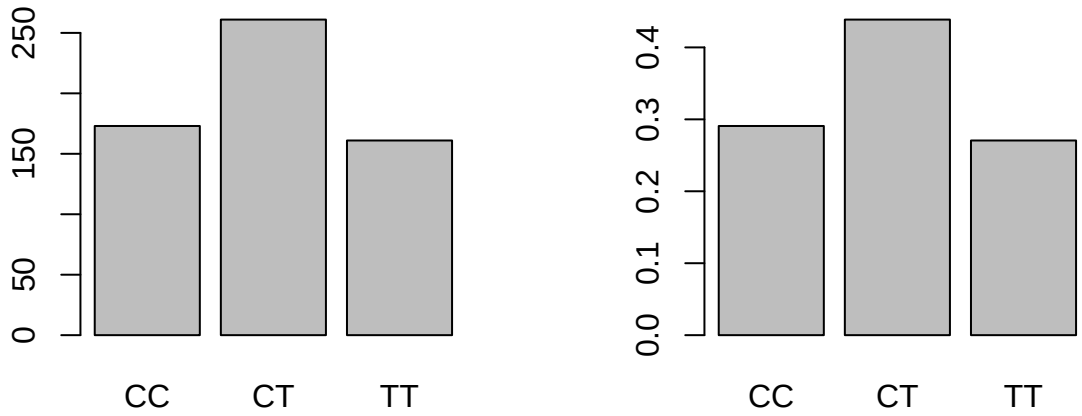
```
table(famuss$actn3.r577x)
```

```
##
##  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.

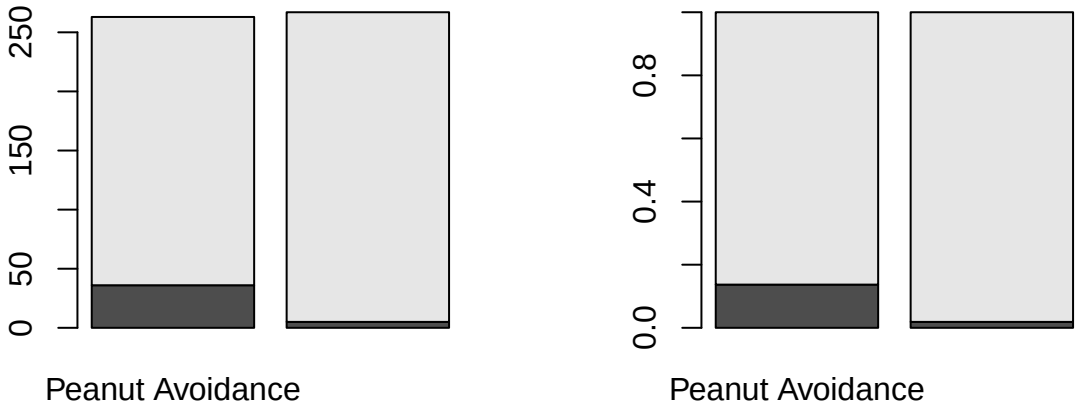
```
addmargins(table(LEAP$overall.V60.outcome, LEAP$treatment.group))
```

```
##
##            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
```

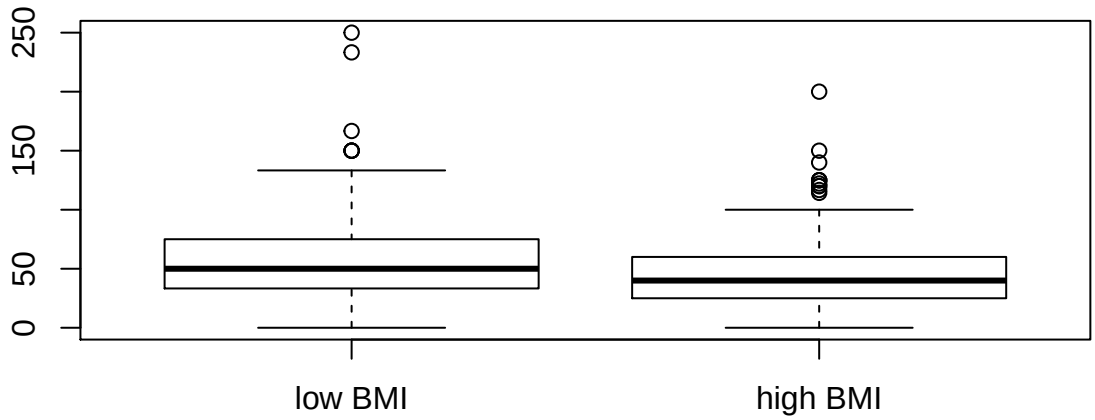
```
#OI Biostat Figure 1.3
par(mfrow = c(1, 2))
barplot(table(LEAP$overall.V60.outcome, LEAP$treatment.group))
barplot(prop.table(table(LEAP$overall.V60.outcome, LEAP$treatment.group), 2))
```



Boxplots, Again

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.

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



Lab 3: Golub Case Study

The analysis of the Golub data shown in the text uses a small version of the complete dataset (`golub.small`) that contains only the data for 10 patients and 10 genes. For the conceptual details behind the analysis approach, refer to the text. The lab shows the computational details for working with the dataset in R.

Part 1, identifying informative genes, mostly relies on functions and ideas that have been introduced previously, such as producing summaries and taking pseudorandom samples. Part 2, predicting leukemia type, requires the use of loops and conditional statements. All code containing loops and conditional statements are provided in the lab; at this stage, the goal is to begin understanding the logic behind control structure rather than the precise syntax. A more formal introduction to loops and conditional statements is provided in Chapter 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, as shown in the subsetting section of the Lab 1 notes. 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 apply() Function

To use the apply() function to a matrix, specify the function and whether it should be applied to the rows or the columns. The function can be a pre-defined R function like mean(), range(), summary(), class(), etc. or a user-defined function. R also has specific functions for calculating the marginal sums and means of a matrix: rowSums(), colSums(), rowMeans(), and colMeans().

```
m = matrix(1:9, nrow = 3)
m
```

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

```
apply(m, 1, sum)   #calculate row sums
```

```
## [1] 12 15 18
```

```
apply(m, 2, sum)   #calculate column sums
```

```
## [1]  6 15 24
```

```
rowSums(m)
```

```
## [1] 12 15 18
```

```
colSums(m)
```

```
## [1]  6 15 24
```

```
rowMeans(m)
```

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

```
colMeans(m)
```

```
## [1] 2 5 8
```

Working with Vector Indices

The order() function applied to a vector returns the indices of the vector in sorted order; order is either ascending or descending, as specified by the argument decreasing. For example, in the following code, the first value of indices is 4 because the 4^{th} element of scores has the lowest value.

```
scores = c(67, 88, 75, 52, 93)                  #vector of scores
indices = order(scores, decreasing = FALSE)     #sorted in ascending order
indices
```

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

```
scores[indices]                                 #ordered vector of scores
```

```
## [1] 52 67 75 88 93
```

The `which()` function returns the indices of a logical vector that have value `TRUE`. When applied to a non-character vector, `which.min()` and `which.max()` return the indices of the minimum and maximum value, respectively. When applied to a logical vector, `which.min()` returns the index of the first `FALSE` and `which.max()` returns the index of the first `TRUE`.

```
passing = (scores >= 60)
passing
```

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

```
which(passing)
```

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

```
which.min(scores)
```

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

```
which.max(scores)
```

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

Lab 4: Arenosa Case Study

This lab integrates the statistical concepts covered in Chapter 1 (numerical and graphical summary measures, transformations, identifying outliers) with the computing techniques introduced in earlier labs.

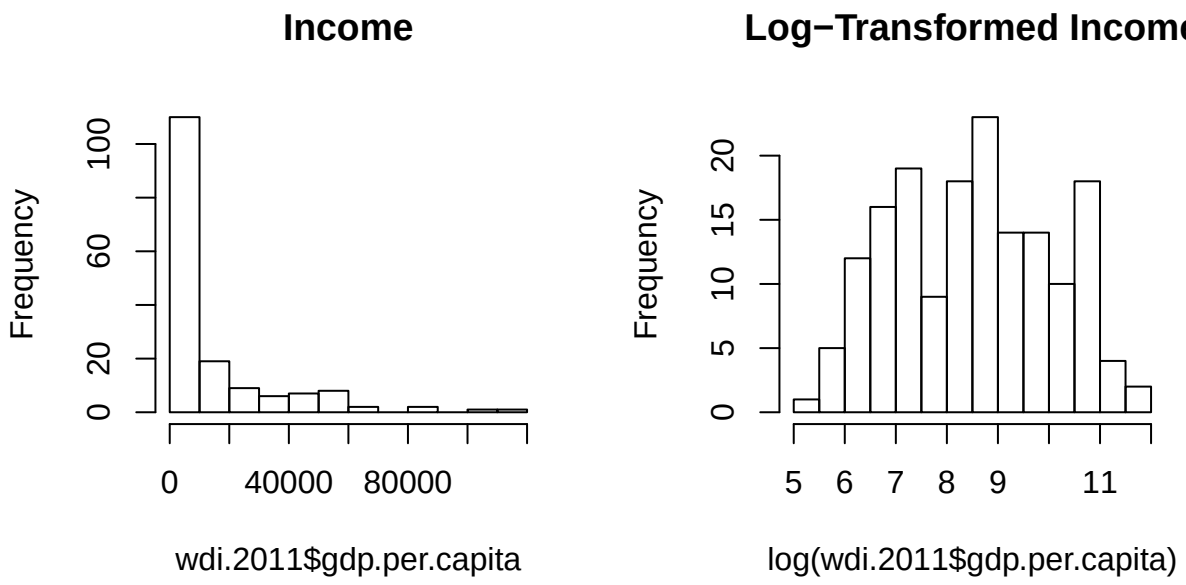
Logarithms and Exponentials

The `log()` function computes natural logarithms by default, and the base can be specified with the optional argument base. The `log10()` function computes base 10 logarithms and the `log2()` function computes base 2 logarithms.

The `exp()` function computes the exponential function.

```
#load the data
data("wdi.2011")

#OI Biostat Figure 1.22
par(mfrow = c(1, 2))
hist(wdi.2011$gdp.per.capita, main = "Income")
hist(log(wdi.2011$gdp.per.capita), main = "Log-Transformed Income")
```



Set Operations

Set operations are useful for comparing vectors. The `union()` function returns all elements in both vectors, without repeating the elements common to both. The `intersect()` function returns the elements common to both vectors.

The `setdiff(v, w)` function is order-specific, returning the elements in v that are not in w.

The `%in%` operator can be used to produce a logical vector indicating matches between two sets.

```
v = c(1, 2, 3, 4, 5)
w = c(4, 5, 6, 7, 8)

union(v, w)
```

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

```
intersect(v, w)
```

```
## [1] 4 5
```

```
setdiff(v, w)
```

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

```
setdiff(w, v)
```

```
## [1] 6 7 8
```

```
v %in% w
```

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

```
w %in% v
```

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