CS544 Module 1

1. Module 1 Study Guide and Deliverables

Readings:	Lecture material
Assessments:	Quiz 1 due ...
Assignments:	Assignment 1 due ...

2. Introduction to Statistics

2.1. Statistics

Descriptive Statistics

Methods for organizing and summarizing the data through graphs, charts, tables, averages, measures of variation, and percentiles.

Inferential Statistics

Methods for drawing conclusions and their reliability about a population based on information drawn from a sample of the population.

2.2. Measures of Central Tendency

The descriptive measures that indicate the center of the data set or the most typical value of the data set are known as the measures of central tendency. The common measures are *mean*, *median*, and *mode*.

The *mean* of a numeric data set is the sum of the values divided by the number of the values in the data set. This measure is also referred to as the average.

The *median* of a numeric data set is the number that splits the data between the bottom 50% and the top 50%.

The data is arranged in increasing order. If the number of values is odd, the median is the middle value. If the number of values is even, the median is the mean of the two middle values.

The *mod*e of the data set is the most frequently occurring value in the data set.

Example:

The following data set shows the scores for 10 students in an exam:

$$75, 72, 78, 70, 78, 78, 88, 75, 78, 72$$

The mean of the data is $764/10 = 76.4$.

For calculating the median, arrange the data in increasing order:

$$70, 72, 72, 75, 75, 78, 78, 78, 78, 88$$

Since the data has an even number of values, the median is the mean of the two middle values 75 and 78. Hence, the median is 76.5.

The value 78 is the frequently occurring item and hence is the mode of the data set.

In the above data, the mean and the median are approximately the same. When the data set has extreme values, the mean is sensitive to these extreme values, whereas the median is not. A resistant measure is one that is not sensitive to the extreme values. A trimmed mean can improve the resistance of the mean where a percentage of the smallest and largest values are removed from the data.

For a given sample data of size n, $(x_1, x_2, \ldots, x_n)$, the sample mean is denoted by $\bar{x}$ (read *x bar*):

$$\bar{x} = \frac{\sum x_i}{n}$$

Test Yourself 1.1 (Multipart)

The following are student test scores for an exam 1. 72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94
(Click each step to reveal its solution.)

▶ **Show the R command to compute the mean and what the mean value is.**

```
mean(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94))
```

```
77.9
```

▶ **Show the R command to compute the median and what the median value is.**

```
median(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94))
```

```
83
```

Test Yourself 1.2 (Multipart)

The following are student test scores for an exam 2. 68, 25, 87, 89, 91, 79, 99, 80, 62, 74
(Click each step to reveal its solution.)

▶ **Show the R command to compute the mean and what the mean value is.**

```
mean(c(68, 25, 87, 89, 91, 79, 99, 80, 62, 74))
```

```
75.4
```

▶ **Show the R command to compute the median and what the median value is.**

```
median(c(68, 25, 87, 89, 91, 79, 99, 80, 62, 74))
```

```
79.5
```

2.3. Measures of Variation

The measures of center (*mean*, *median*, and *mode*) allow us to compare two data sets and draw conclusions. However, the two data sets can have similar measures of center but differ significantly. Consider the following scores of a sample of five students in two tests:

Test1	$71, 72, 75, 75, 77$
Test2	$66, 71, 75, 75, 83$

The above two data sets have the same mean (74), median (75), and mode (75). The two data sets however differ, the scores in Test2 vary much more than the scores in Test1.

The most used measures of variation are the range, the standard deviation, and the interquartile range.

The *range* of the data set is the difference between the largest (maximum) and the smallest (minimum) values. For Test1, the range is 6. For Test2, the range is 17. Though the range is easy to compute, it only takes two values into account. The standard deviation takes into account all the values.

The standard deviation is computed using the measure of the deviation from the mean. The variance of a sample is defined as follows, where n is the number of values in the sample and $\bar{x}$ is the mean of the sample.

$$s^2 = \frac{\sum(x_i - \bar{x})^2}{n - 1}$$

The variances of the two test scores are 6 and 39 as shown below.

Test1 ($\bar{x} = 74$)			**Test2 ($\bar{x} = 74$)**		
x	$x - \bar{x}$	$(x - \bar{x})^2$	x	$x - \bar{x}$	$(x - \bar{x})^2$
71	-3	9	66	-8	64
72	-2	4	71	-3	9
75	1	1	75	1	1
75	1	1	75	1	1
77	3	9	83	9	81
$\sum(x - \bar{x})^2$		24	$\sum(x - \bar{x})^2$		156
variance		$\frac{24}{4} = 6$	*variance*		$\frac{156}{4} = 39$

The standard deviation is the square root of the variance. For the two test scores, the standard deviations are 2.45 and 6.24, respectively. On average, the scores of Test1 vary from the mean score of 74 by about 2.45. The scores of Test2 have more variation than the scores of Test1.

An alternate formula for computing the standard deviation of a sample is

$$s = \sqrt{\frac{\sum x_i^2 - \frac{(\sum x_i)^2}{n}}{n-1}}$$

The more variation there is in the data set, the larger the standard deviation. *Chebychev's rule* states that at least 89% of the observations lie within three standard deviations of either side of the mean (the range from $\bar{x} - 3s$ to $\bar{x} + 3s$).

For data that has a bell-shaped distribution, the *empirical rule* states that approximately 99.7% of the observations lie within three standard deviations of either side of the mean, approximately 95% of the observations lie within two standard deviations of either side of the mean, and approximately 68% of the observations lie within one standard deviation of either side of the mean.

Test Yourself 1.3

For the student test scores in exam 1, show the R command to compute the variance and standard deviation.

```
sd(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94))
20.55458
var(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94))
422.4909
```

Test Yourself 1.4

For the student test scores in exam 2, show the R command to compute the variance and standard deviation.

```
sd(c(68, 25, 87, 89, 91, 79, 99, 80, 62, 74))
20.89763
var(c(68, 25, 87, 89, 91, 79, 99, 80, 62, 74))
436.7111
```

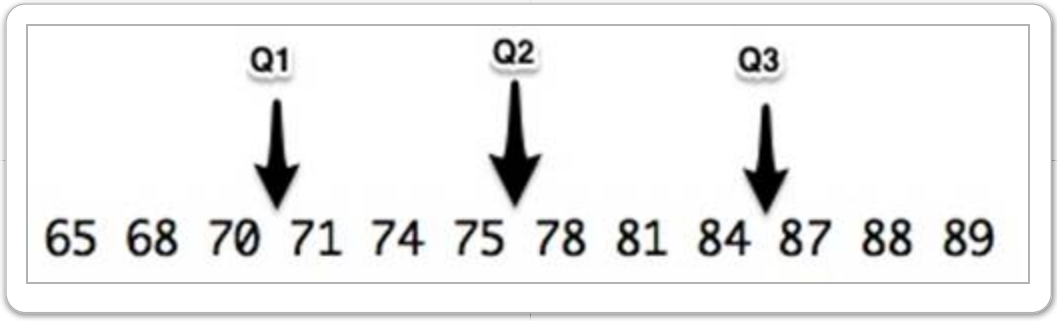
2.4. Five Number Summary

The mean (measure of center) and standard deviation (measure of variation) are sensitive to the extreme values. Measures based on percentiles are resistant measures. Percentiles divide the data set into 100 equal parts. The

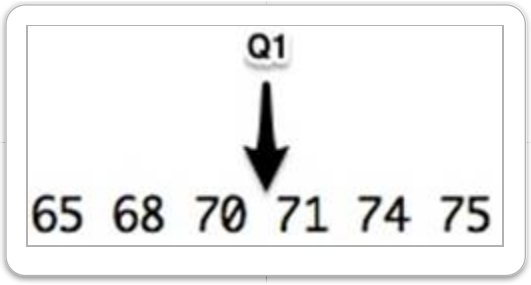
data set has 99 percentiles, $P_1, P_2, \ldots, P_{99}$. The first percentile divides the bottom 1% from the top 99% of the data, the second percentile divides the bottom 2% from the top 98% of the data, etc. The 50th percentile is the median. Other measures used are the *deciles* (dividing the data into 10 equal parts), *quintiles* (dividing the data into 5 equal parts), and *quartiles* (dividing the data into 4 equal parts).

The quartiles are denoted by the three values Q_1, Q_2, and Q_3. The first quartile, Q_1, divides the bottom 25% of the data from the top 75%. The second quartile, Q_2, is the median that divides the bottom 50% from the top 50%. The third quartile, Q_3, divides the bottom 75% from the top 25%.

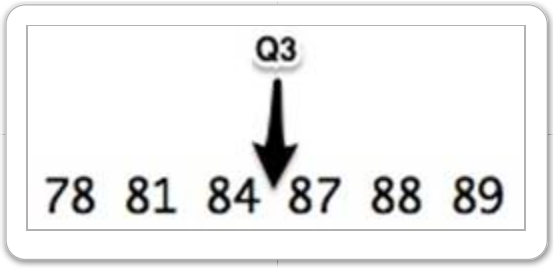
The data set shown below has 12 values. The second quartile (*median)* is the mean of the 6th and 7th values. Hence, $Q_2 = 76.5$.



The bottom half of the data set at or below the median is



The first quartile is the mean of the two values 70 and 71. Hence, $Q_1 = 70.5$. The top half of the data set at or above the median is



The third quartile is the mean of the two values 84 and 87. Hence, $Q_3 = 85.5$.

The *interquartile range*, IQR, is the difference between the third and first quartiles, $IQR = Q_3 - Q_1 = 15$. The IQR shows the variation in the middle 50% of the data.

The five number summary of the data set consists of the *minimum*, Q_1, Q_2, Q_3, and *maximum* values.

For the data set above, the five number summary is $(65, 70.5, 76.5, 85.5, 89)$.

The variation in the first quarter is $Q_1 - \text{ minimum} = 5.5$. The variation in the second quarter is $Q_2 - Q_1 = 6$. The variation in the third quarter is $Q_3 - Q_2 = 9$. The variation in the last quarter is $\text{maximum} - Q_3 = 3.5$.

Test Yourself 1.5

For the student test scores in exam 1, show the R command to compute the five number summary.

```
fivenum(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94))
22.0 74.5 83.0 90.5 96.0
```

Test Yourself 1.6

For the student test scores in exam 2, show the R command to compute the five number summary.

```
fivenum(c(68, 25, 87, 89, 91, 79, 99, 80, 62, 74))
25.0 68.0 79.5 89.0 99.0
```

2.5. Outliers

The lower and upper limits of a data set are $Q_1 - 1.5 \cdot IQR$ and $Q_3 + 1.5 \cdot IQR$. The values that lie below the lower limit or above the upper limit are considered as outliers.

Test Yourself 1.7

For the student test scores in exam 1, show the R commands to compute the outliers

```
x <- c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94)
f <- fivenum(x)
x[x < f[2] - 1.5 * (f[4] - 4[2])]
x[x > f[4] + 1.5 * (f[4] - f[2])]
22
```

```
Numeric(0)
```

Test Yourself 1.8

For the student test scores in exam 2, show the R commands to compute the outliers

```
x <- c(68, 25, 87, 89, 91, 79, 99, 80, 62, 74)
f <- fivenum(x)
x[x < f[2] - 1.5 * (f[4] - f[2])]
x[x > f[4] + 1.5 * (f[4] - f[2])]
25
Numeric(0)
```

2.6. Descriptive Measures—Population versus Sample

The population mean is the mean of all observations for the entire population. For the variable x and population size N, the mean for a finite population is

$$\mu = \frac{\sum x_i}{N}$$

The sample mean for a sample of size n is

$$\bar{x} = \frac{\sum x_i}{n}$$

There is only one population mean whereas there are many sample means, one for each possible sample of the population.

The population standard deviation is the standard deviation of all observations for the entire population. The defining formula for the standard deviation of a finite population size N is

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

The alternate computing formula for the standard deviation of a finite population size N is

$$\sigma = \sqrt{\frac{\sum x_i^2}{N} - \mu^2}$$

The population variance is σ^2.

The sample standard deviation for a sample of size n is

$$s = \sqrt{\frac{\sum(x_i - \bar{x})^2}{n-1}}$$

2.7. Standardized Variables

A standardized variable has a mean of 0 and a standard deviation of 1. For any given variable x, the standardized version of x is the new variable, z, defined as

$$z = \frac{x - \mu}{\sigma}$$

The mean and the standard deviations used are the sample (or population) mean and standard deviation for the variable x. The computed values are also known as the *z-scores* for the variable x.

For the sample test scores Test1 and Test2, the *z-scores* are computed as follows:

Test1 ***(mean = 74, sd = 2.45)***		**Test2** ***(mean = 74, sd = 6.24)***	
x	z	x	z
71	-1.22	66	-1.28
72	-0.82	71	-0.48
75	0.41	75	0.16
75	0.41	75	0.16
77	1.22	83	1.44

Negative scores indicate that the values are below the mean, whereas positive scores indicate that the values are above the mean. The *z-score* -1.22 (for the value 71 in Test1) indicates that the score is 1.22 standard deviations below the mean. Similarly, the z-score 1.44 (for the value 83 in Test2) indicates that the score is 1.44 standard deviations above the mean. The three standard deviation rule implies that most of the z-scores will be in the range -3 to 3. Values with z-scores less than -3 or greater than 3 are considered as outliers.

Test Yourself 1.9

For the student test scores in exam 1, show the R command to produce the z-scores for all students

```
x <- c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94)
(x - mean(x)) / sd(x)
```

Test Yourself 1.10

For the student test scores in exam 2, show the R command to produce the z-scores for all students

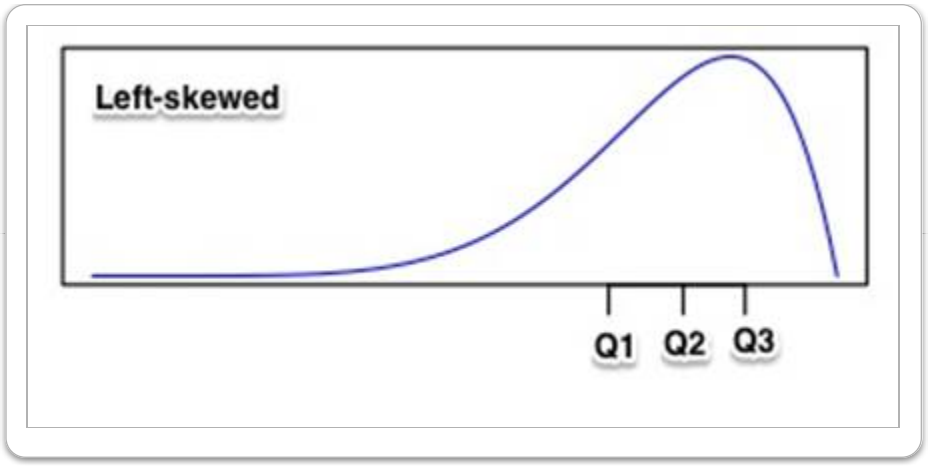
```
x <- c(68, 25, 87, 89, 91, 79, 99, 80, 62, 74)
(x - mean(x)) / sd(x)
```

2.8. Shape of Data

The shape of data is the distribution of data values throughout the range of the data. A distribution is symmetrical when the values below the mean are distributed the same way as the values above the mean. For symmetrical distribution, the mean and the median are the same. If most of the values are in the upper portion of the distribution, the distribution is left-skewed. The presence of fewer numbers of small value pulls the mean towards the left end of the data. The mean is less than the median in this case. On the other hand, if most of the values are in the lower end of the distribution, the distribution is right-skewed. The presence of fewer numbers of larger value pulls the mean towards the right end of the data. The mean is greater than the median in this case.

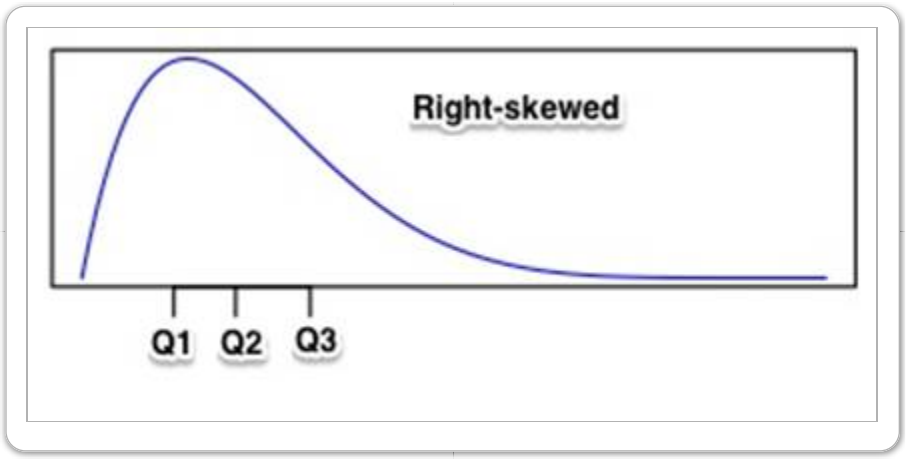
In a left-skewed distribution, the following properties hold:

- The distance (range) from the minimum value to the median is greater than the distance from the median to the maximum value.
- The distance (range) from the minimum value to the first quartile (Q_1) is greater than the distance from the third quartile (Q_3) to the maximum value.
- The distance (range) from the first quartile (Q_1) to the median is greater than the distance from the median to the third quartile (Q_3).

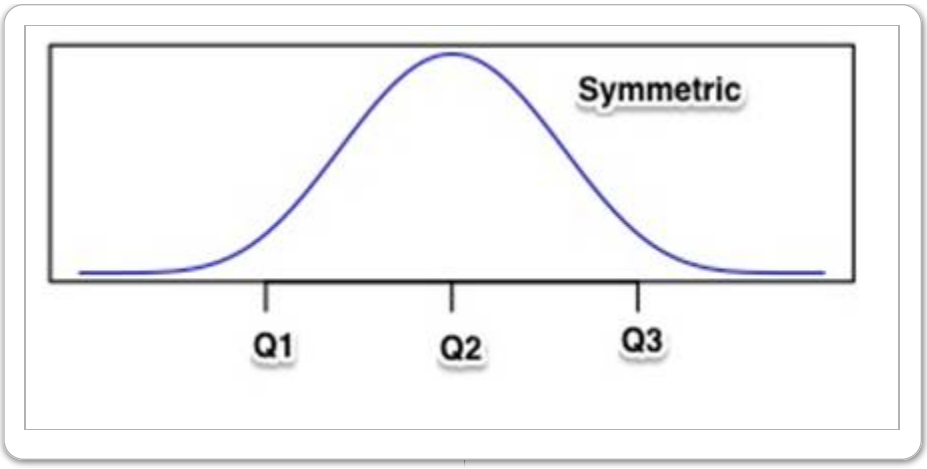


In a right-skewed distribution, the following properties hold:

- The distance (range) from the minimum value to the median is less than the distance from the median to the maximum value.
- The distance (range) from the minimum value to the first quartile (Q_1) is less than the distance from the third quartile (Q_3) to the maximum value.
- The distance (range) from the first quartile (Q_1) to the median is less than the distance from the median to the third quartile (Q_3).



In a symmetric distribution, all the above distances are the same.



3. Basic Concepts of *R* Data Types and Structures

3.1. Identifiers

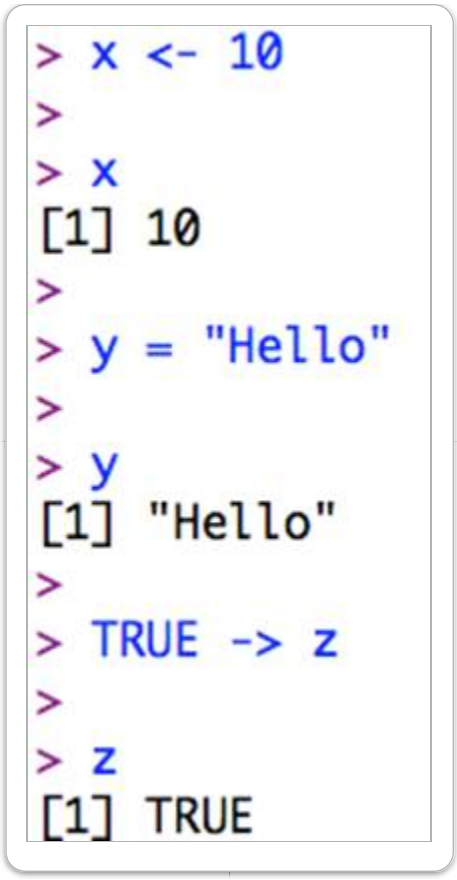
The preferred convention for the object names (variables) in *R* is to have all lowercase letters, with compound words separated by dots.

Examples: salaries, first.names, last.names, etc.

The convention for user-defined function names is to have initial capital letters and no dots (e.g., CalculateMonthlyPremiums, etc.).

3.2. Assignments

The <- operator is preferred when assigning values to objects. The = or -> operators may also be used. The following examples show the three different ways of assigning values to the variables.



3.3. Data Types

The following data types are frequently used in *R*:

- numeric
- integer
- logical
- character
- complex

The default data type for numbers is the *numeric* type. The evaluated numeric expression can be an integer value or a double value.

```
> x <- 7
> y <- x/2
>
> x
[1] 7
> y
[1] 3.5
```

The mode function returns the storage type of these objects as numeric. The typeof function shows that these are treated as double.

```
> mode(x)
[1] "numeric"
> typeof(x)
[1] "double"
>
> mode(y)
[1] "numeric"
> typeof(y)
[1] "double"
```

For creating integer objects, the as.integer function is used to explicitly convert the given data to the integer type.

```
> x <- as.integer(7)
> y <- as.integer(x/2)
>
> x
[1] 7
> y
[1] 3
```

The mode of these objects is still numeric, whereas the typeof function returns the integer type.

```
> mode(x)
[1] "numeric"
> typeof(x)
[1] "integer"
>
> mode(y)
[1] "numeric"
> typeof(y)
[1] "integer"
```

The *logical* constants are TRUE and FALSE. The logical objects are created as a result of comparison (<, <=, >, >=, !=, ==) between objects. The standard logical operators are & (*and*), | (*or*), and ! (*negation*).

```
> a <- 10
> b <- 20
> x <- a >= b
> y <- (a > 5) & (b < 25)
>
> x
[1] FALSE
> y
[1] TRUE
```

The mode and typeof functions for these objects return the type as logical.

```
> mode(x)
[1] "logical"
> typeof(x)
[1] "logical"
>
> mode(y)
[1] "logical"
> typeof(y)
[1] "logical"
```

The logical constant NA represents the missing value when processing data or when converting data.

String values are represented as character objects.

```
> x <- "Hello"
> y <- as.character(123)
>
> x
[1] "Hello"
> y
[1] "123"
```

The mode and typeof functions for these objects return the type as character.

```
> mode(x)
[1] "character"
> typeof(x)
[1] "character"
>
> mode(y)
[1] "character"
> typeof(y)
[1] "character"
```

Strings can be concatenated using the paste function. The function takes a variable number of arguments of any type and returns a string using space as the default separator. The separator may also be explicitly specified using the sep argument.

```
> x <- "Hello"
> y <- 123
>
> paste(x, y)
[1] "Hello 123"
>
> paste(x, y, "Testing", sep = ",")
[1] "Hello,123,Testing"
```

The sprintf function can also be used to return strings using the specified format.

```
> z <- sprintf("You want to say %s on %d", x, y)
> z
[1] "You want to say Hello on 123"
```

The number of characters in the string is obtained using the nchar function.

```
> nchar(z)
[1] 28
```

The substr function is used for extracting portions of the specified string. The start and stop arguments are required. The first character is at position number 1.

```
> substr(z, start = 5, stop = 21)
[1] "want to say Hello"
```

The sub function is used for substituting the first matching string or regular expression (the first argument) with the specified string (the second argument). The new string is returned. The original string is not modified.

```
> sub("Hello", "Bye", z)
[1] "You want to say Bye on 123"
```

The complex numbers are defined in *R* as follows:

```
> x <- 2 + 3i
> y <- 3 - 1i
>
> mode(x)
[1] "complex"
> typeof(x)
[1] "complex"
```

The following shows the addition and multiplication of the complex numbers.

```
> x+y
[1] 5+2i
>
> x*y
[1] 9+7i
```

Test Yourself 1.11

For the student test scores in exam 1, use logical expressions to produce only the test scores that were 90 or above. Show the R command

```
x <- c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94)
x[x > 90]
96 92 94
```

Test Yourself 1.12

For the student test scores in exam 1, use logical expressions to produce only the test scores that are multiples of 4

```
x <- c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94)
x[x %% 4 == 0]
72 96 92
```

Test Yourself 1.13

Using the 'paste' function, convert the set of student test scores from exam 1 into the format "Student#: score" where the # will represent the position of the score in the vector. DO NOT ASSUME THAT YOU KNOW THE NUMBER OF SCORES OR STUDENTS IN THE CLASS. Show the R commands. E.g- Student1: 72, Student2: 22, etc…

```
x <- c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94)
paste0("Student", 1:length(x), ": ", x)
```

3.4. Data Type Conversions

The functions as.numeric, as.character, as.logical, as.integer, and as.complex are used for converting the given data into the required type. If the data cannot be converted, the value NA is returned.

For the logical data objects TRUE and FALSE, the following convert the values to numeric and character types.

```
> x <- TRUE
> y <- FALSE
>
> as.numeric(x)
[1] 1
> as.numeric(y)
[1] 0
>
> as.character(x)
[1] "TRUE"
> as.character(y)
[1] "FALSE"
```

For numeric values, the value 0 is converted to FALSE and any non-zero value is converted to TRUE.

```
> x <- 1
> y <- 0
> z <- 20
```

```
> as.logical(a)
[1] TRUE
> as.logical(b)
[1] FALSE
> as.logical(c)
[1] TRUE
```

The string conversions are straightforward for the numeric data.

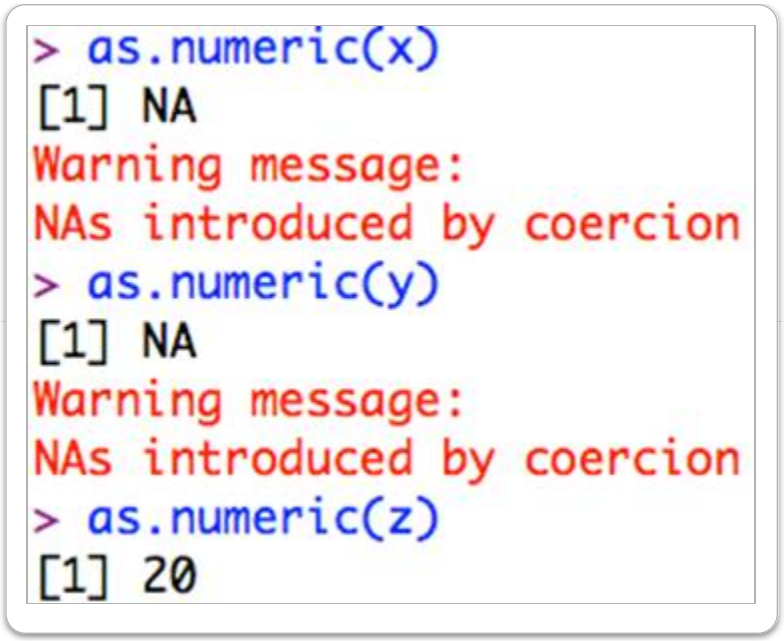
```
> as.character(a)
[1] "1"
> as.character(b)
[1] "0"
> as.character(c)
[1] "20"
```

For character data, the strings "TRUE" and "FALSE" are converted to TRUE and FALSE logical values. Any other string data is returned as NA for logical conversion.

```
> x <- "TRUE"
> y <- "FALSE"
> z <- "20"
```

```
> as.logical(x)
[1] TRUE
> as.logical(y)
[1] FALSE
> as.logical(z)
[1] NA
```

If the string represents numeric data, the conversion succeeds; otherwise NA is returned.



The functions is.numeric, is.character, is.logical, is.integer, and is.complex can be used to check if the given data is of the appropriate type. The is.na is used to check if the given data is a missing value.

```
> is.numeric(10)
[1] TRUE
> is.numeric("ab")
[1] FALSE
>
> is.character("Hello")
[1] TRUE
> is.character(10)
[1] FALSE
>
> is.logical(TRUE)
[1] TRUE
> is.logical("TRUE")
[1] FALSE
>
> is.integer(10)
[1] FALSE
> is.integer(as.integer(10))
[1] TRUE
>
> is.na(NA)
[1] TRUE
> is.na(as.numeric("ab"))
[1] TRUE
Warning message:
NAs introduced by coercion
```

Test Yourself 1.14

Show the R commands to convert a vector of numerics to logicals. Use the student test scores for exam 1 as an example.

```
as.logical(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94))
```

Test Yourself 1.15

Show the R commands to convert a vector of numerics to characters. Use the student test scores for exam 1 as an example.

```
char <- as.character(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94))
```

Test Yourself 1.16

Show the R commands to convert the following vector of characters into numerics:
c('1', '2', '3')

```
as.numeric(c('1', '2', '3')
```

3.5. Data Structures

The following data structures are commonly used in *R*:

- *vector*—a collection of values of the same type
- *factor*—a collection of values from a fixed set of possible values
- *matrix*—a two-dimensional collection of values of the same type
- *list*—a collection of any of the data structures
- *data frame*—a collection of vectors all of the same length

3.6. Vectors

A *vector* in *R* is a collection of values that all have the same data type. If all the elements are numbers, the vector is a numeric vector. If the elements are strings, the vector is a character vector. If the elements are true/false, the vector is a logical vector. The following examples show the assignment of various types of vectors. The function c() takes an arbitrary number of arguments and returns a vector by combining its arguments.

The following example shows a vector of 5 numbers. The default type for numbers is double.

```
> ages <- c(20, 22, 23, 23, 26)
>
> ages
[1] 20 22 23 23 26
>
> mode(ages)
[1] "numeric"
> typeof(ages)
[1] "double"
```

The following example shows explicit double values used for creating the vector.

```
> sizes <- c(12.6, 8.4)
>
> sizes
[1] 12.6  8.4
>
> mode(sizes)
[1] "numeric"
> typeof(sizes)
[1] "double"
```

A numeric vector can be converted to an integer vector as follows. The fraction part, if any, is lost during this process.

```
> x <- as.integer(sizes)
>
> x
[1] 12  8
>
> mode(x)
[1] "numeric"
> typeof(x)
[1] "integer"
```

A vector of character strings is used as shown below.

```
> first.names <- c("Alice", "Bob", "Charlie")
>
> first.names
[1] "Alice"   "Bob"     "Charlie"
>
> mode(first.names)
[1] "character"
> typeof(first.names)
[1] "character"
```

A logical vector with TRUE and FALSE values is constructed as follows.

```
> voted <- c(TRUE, FALSE, TRUE, TRUE)
>
> voted
[1]  TRUE FALSE  TRUE  TRUE
>
> mode(voted)
[1] "logical"
> typeof(voted)
[1] "logical"
```

The data type conversions work with the logical vectors as shown below.

```
> x <- as.numeric(voted)
>
> x
[1] 1 0 1 1
>
> as.logical(x)
[1]  TRUE FALSE  TRUE  TRUE
```

The c() function can be used to combine any number of existing vectors into a new vector. The following scenario shows the resulting vector when the arguments to be combined are of the same type.

```
> c(ages, sizes)
[1] 20.0 22.0 23.0 23.0 26.0 12.6  8.4
```

Since a vector is a collection of all values of the same type, in cases where different types are used during combining, the data is coerced to a common type. The type hierarchy is shown below.

logical < integer < double < complex < character

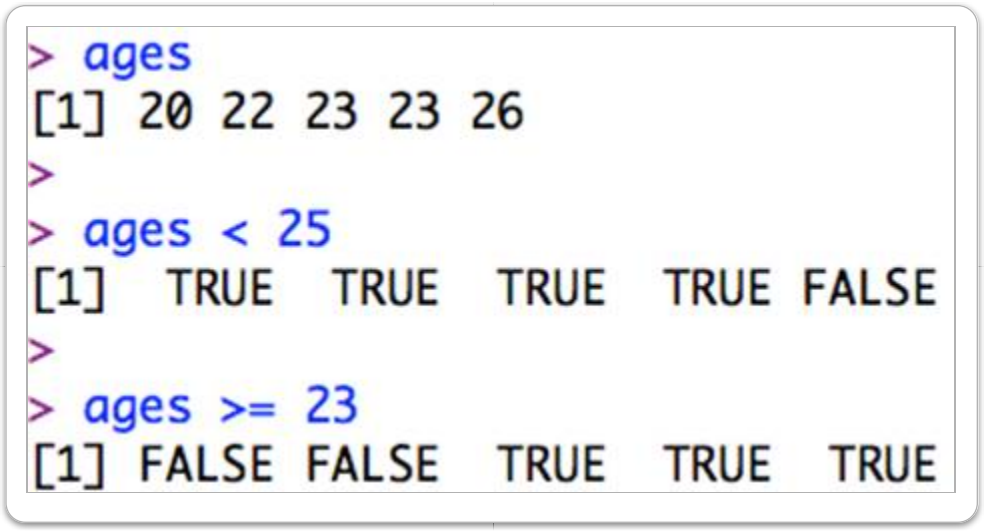
The following example combines a numeric vector and a character vector. The result is a character vector.

```
> x <- c(ages, first.names)
>
> x
[1] "20"      "22"      "23"      "23"
[5] "26"      "Alice"   "Bob"     "Charlie"
>
> as.numeric(x)
[1] 20 22 23 23 26 NA NA NA
Warning message:
NAs introduced by coercion
```

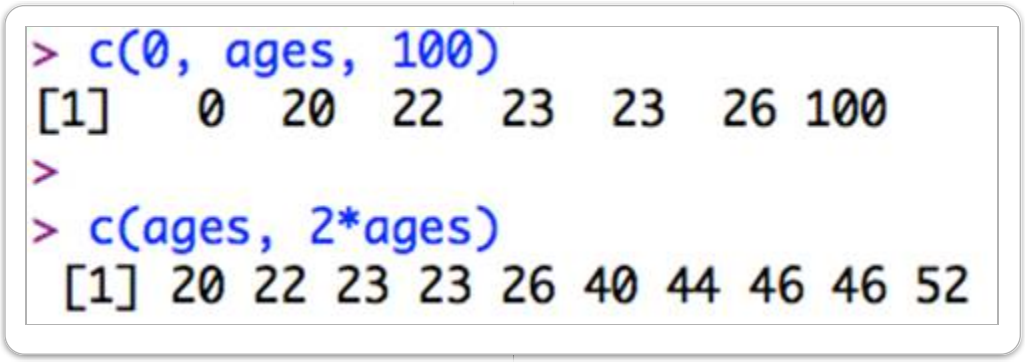
The length function returns the number of values in the given vector. For a numeric vector, the sum function adds all the values. Arithmetic operations can be performed on vectors as shown below.

```
> ages
[1] 20 22 23 23 26
>
> length(ages)
[1] 5
> sum(ages)
[1] 114
>
> ages/2
[1] 10.0 11.0 11.5 11.5 13.0
>
> 2 * ages
[1] 40 44 46 46 52
```

The comparison operations on a vector result in a logical vector of the same length as shown below.



A combination of a vector with basic data values results in a new vector. The basic data values are regarded as a vector of unit length. A vector combining itself and double its values is also shown below.



Arithmetic operations on vectors result in a new vector. If the lengths of the vectors are not equal, then the elements of the shorter vector are recycled to fill the gap. In the following example, the vector y has only two elements. So, the elements are appended to itself to make it a length of four.

```
> x <- c(10, 20, 30, 40)
> y <- c(2, 4)
>
> x*y
[1]  20  80  60 160
> x+y
[1] 12 24 32 44
```

The first.names vector has three strings and the ages vector has five numbers. The paste function concatenates the corresponding values. The first two values of the first.names vector are recycled at the end to make it the same length as the ages vector.

```
> paste(first.names, ages)
[1] "Alice 20"   "Bob 22"     "Charlie 23"
[4] "Alice 23"   "Bob 26"
> paste(first.names, ages, sep=",")
[1] "Alice,20"   "Bob,22"     "Charlie,23"
[4] "Alice,23"   "Bob,26"
```

Test Yourself 1.17

Show the R command to instantiate a vector of numbers from 1 to 10 inclusive

```
seq(1:10)
```

Test Yourself 1.18

Show the R command to instantiate a vector of EVEN numbers from 1 to 20 inclusive

```
seq(from=2, to=20, by=2)
```

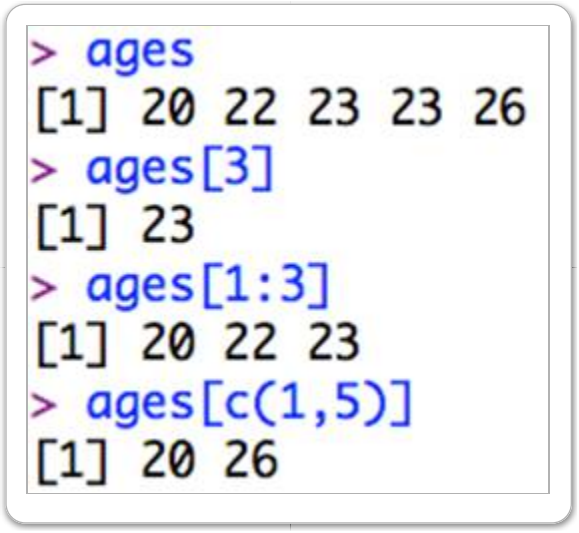
Test Yourself 1.19

Show the R command to instantiate a vector of mixed data types (character, Boolean, numeric). What are the data types in the vector?

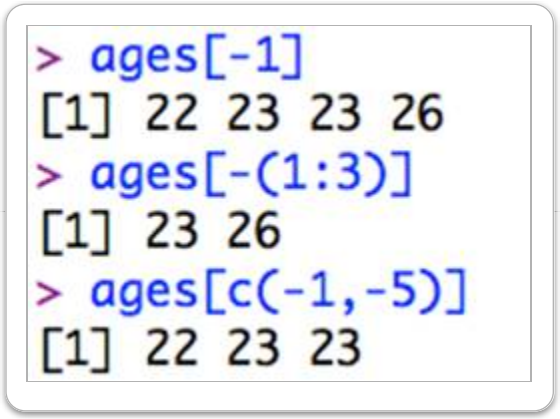
```
x <- c("Hello world", 1234.4, TRUE)
mode(x)
```

3.7. Indexing Vectors

The values of a vector can be indexed one at a time using any of the integer values from 1 to the length of the vector. Multiple values can also be indexed in one step by using a sequence of integers or using a vector of explicit indices.



Using a negative value for the index returns rest of the vector except the value of the specified index. A sequence or a combination of negative values will suppress those elements and return the remaining.



A vector can also be indexed using a logical vector. If the corresponding index is TRUE, the value is included; otherwise, the value is excluded.

```
> ages[c(TRUE, TRUE, TRUE, FALSE, FALSE)]
[1] 20 22 23
```

The following logical vector will return all the odd elements of the vector.

```
> ages[c(TRUE, FALSE)]
[1] 20 23 26
```

The LETTERS (or letters) is a built-in vector in *R* containing the 26 uppercase (or lowercase) letters. The data can be indexed as shown below.

```
> letters[c(1, 26)]
[1] "a" "z"
>
> letters[1:5]
[1] "a" "b" "c" "d" "e"
>
> letters[c(TRUE, FALSE)]
 [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"
```

The function is.vector returns TRUE if the given object is a vector.

```
> is.vector(letters)
[1] TRUE
```

Test Yourself 1.20 (Multipart)

Given the vector below, show the R commands to produce the following results. DO NOT ASSUME

THAT YOU KNOW THE LENGTH OF THE VECTOR.

x <- seq(1:20)

(Click each step to reveal its solution.)

▶ **The third, seventh, and 10th elements of the vector**

```
x[c(3, 7, 10)]
```

▶ **All elements except for the last element**

```
x[-length(x)]
```

▶ **All elements that are divisible by 3**

```
x[x %% 3 == 0]
```

3.8. Sequences

A sequence of values forming a vector of consecutive numbers can be generated using the : operator (*from:to*). If the first value is less than the second, the sequence is generated in increasing order. Otherwise, the sequence is generated in decreasing order. The sequence is generated using the integer type.

```
> 20:25
[1] 20 21 22 23 24 25
>
> 25:20
[1] 25 24 23 22 21 20
>
> mode(20:25)
[1] "numeric"
> typeof(20:25)
[1] "integer"
```

Arithmetic operators can be performed on sequences. The sequence operator has the higher precedence.

```
> 2 * 1:5
[1]  2  4  6  8 10
```

The resulting numeric vectors from arithmetic operations will be of the double type.

```
> x <- 2 * 1:5
> mode(x)
[1] "numeric"
> typeof(x)
[1] "double"
```

The following example shows the precedence of the sequence operator. In the first case, the sequence $1:5$ is generated and the value 1 is subtracted from each member of the sequence. In the second case, the sequence $1:4$ is generated.

```
> n <- 5
>
> 1:n-1
[1] 0 1 2 3 4
>
> 1:(n-1)
[1] 1 2 3 4
```

The generic function for generating sequences is the *seq* function. The optional arguments from and to for the seq function default to 1. If two arguments are specified, the sequence is generated from the first to the second in increments of 1.

```
> seq(0, 5)
[1] 0 1 2 3 4 5
>
> seq(from = 0, to = 5)
[1] 0 1 2 3 4 5
```

If a different step is needed for the sequence, the third argument is specified implicitly or explicitly with the by argument. The length argument may also be used to specify the length of the generated sequence.

```
> seq(2, 10, by = 2)
[1]  2  4  6  8 10
>
> seq(from = 2, to = 10, by = 2)
[1]  2  4  6  8 10
>
> seq(2, by = 2, length = 5)
[1]  2  4  6  8 10
```

If the from value is more than the to value, the sequence is generated in decreasing order using a step value of -1.

```
> seq(10, 2)
[1] 10  9  8  7  6  5  4  3  2
>
> seq(from = 10, to = 2, by = -2)
[1] 10  8  6  4  2
>
> seq(from = 10, by = -2, length = 5)
[1] 10  8  6  4  2
```

All of the above vectors are of the integer type. A vector with the double type of values may also be generated as shown below.

```
> seq(1, 2, by = 0.25)
[1] 1.00 1.25 1.50 1.75 2.00
>
> seq(from = 1.5, to = 4.5, by = 0.5)
[1] 1.5 2.0 2.5 3.0 3.5 4.0 4.5
```

The rep function is used to replicate the given vector. The given vector can be replicated the specified number of times if the second argument is a single number. In the following example, the entire vector x is repeated three times.

```
> x <- c(1,2,3,4,5)
> x
[1] 1 2 3 4 5
>
> rep(x, 3)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
>
> rep(x, times = 3)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

The each parameter allows each value in the vector to be repeated the specified number of times. In the following example, each element of x is repeated three times.

```
> rep(x, each = 3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

If both the times and each arguments are specified, each takes precedence. Each value is first repeated the specified of times, and then the entire result is repeated by the times value.

```
> rep(x, times = 2, each = 3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 1 1 1 2 2 2
[22] 3 3 3 4 4 4 5 5 5
```

If the second argument is a vector, then each element of x is repeated the corresponding number of times as specified in the second argument. A value of 0 does not include the corresponding element in the result.

```
> rep(x, c(1,2,1,2,1))
[1] 1 2 2 3 4 4 5
>
> rep(x, c(0,2,0,3,0))
[1] 2 2 4 4 4
```

The following example shows each element of the given vector repeated that number of times.

```
> rep(x, x)
 [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
```

Test Yourself 1.21 (Multipart)

Using the 'seq' and 'rep' functions, show the R commands to produce the following results. (Click each step to reveal its solution.)

▶ **A vector containing every 5th element from 1 to 100**

```
seq(from=1, to=100, by=5)
```

▶ **A vector containing the sequence (1, 2, 3, 4) 5 times consecutively**

```
rep(c(1, 2, 3, 4), times=5)
```

3.9. Modifying the Vectors

The contents of a vector may be modified one at a time by assigning a new value to the specified index of the vector. Multiple elements of the vector may also be modified at once by assigning a vector of new values to the vector of indices.

```
> x <- 1:5
> x
[1] 1 2 3 4 5
>
> x[1] <- 10
> x
[1] 10  2  3  4  5
>
>
> x[c(2,5)] <- c(20, 50)
> x
[1] 10 20  3  4 50
```

The length of a vector may be affected if a value is assigned to an index that is beyond the current length of the vector. The remaining values are filled with the value NA.

```
> x[8] <- 80
> x
[1] 10 20  3  4 50 NA NA 80
```

Increasing the length of the vector also expands the vector and fills the remaining values with the value NA.

```
> length(x) <- 10
> x
 [1] 10 20  3  4 50 NA NA 80 NA NA
```

Decreasing the length of the vector truncates the vector to the specified length. The remaining values are lost.

```
> length(x) <- 3
> x
[1] 10 20  3
```

Test Yourself 1.22 (Multipart)

Given the vector below, show the R commands to produce the following results.

x <- c(1, 3, 6, 8, 10, 11, 21, 33)

(Click each step to reveal its solution.)

▶ **Change the third element of the vector to a 7**

```
x[3] <- 7
```

▶ **Change the last element of the vector to a 0, assume you do not know the length of the vector**

```
x[length(x)] <- 0
```

▶ **Attempt to change the 10th element to a 5. What happened?**

```
x[10] <- 5
```

The vector got extended to 10 elements with a 5 as the 10th element, elements that were filled in are NA

3.10. Named Vectors

The names function assigns character strings as the names of the elements of the vector. The vector can then be indexed numerically or with the named values. The named vector is created using a vector of values and assigning a vector of strings. When the named vector is displayed, the name and the value of each element are shown column-wise.

```
> ages <- c(20, 22, 23, 23, 26)
>
> names(ages) <- c("Alice", "Bob",
+     "Charlie", "Dave", "Ed")
>
> ages
  Alice     Bob Charlie    Dave      Ed
     20      22      23      23      26
```

One or more values of the vector can then be accessed using the named values.

```
> ages["Bob"]
Bob
 22
>
> ages[c("Alice", "Ed")]
Alice    Ed
   20    26
```

The numeric indices on the named vectors also behave similarly.

```
> ages[2]
Bob
 22
>
> ages[c(1,5)]
Alice    Ed
   20    26
```

The names function returns the names of the values as a character vector.

```
> names(ages)
[1] "Alice"   "Bob"     "Charlie" "Dave"    "Ed"
```

The contents of named vectors, like the contents of regular vectors, may be modified using the name indices or the numeric indices. The names themselves may be modified as shown below.

```
> names(ages)[2] <- "Robert"
>
> ages
  Alice  Robert Charlie    Dave      Ed
     20      22      23      23      26
```

Test Yourself 1.23

Use the names() function to create a named vector which contain the first 3 exam scores for exam 1. Assign them the names "Bob", "Sally", and "Bill".

Recall, the grades from exam 1: 72, 22, 85, 89, 96, 77, 69, 78, 83, 92, 94

```
grades <- c(72, 22, 85)
names(grades) <- c("Bob", "Sally", "Bill")
```

3.11. Scalar as Vector

All scalar values are treated by *R* as vectors of unit length.

```
> x <- 10
> x
[1] 10
> is.vector(x)
[1] TRUE
> length(x)
[1] 1
>
> 10 == c(10)
[1] TRUE
```

The same properties hold for character data and logical data.

```
> y <- "Hello"
> y
[1] "Hello"
>
> z <- TRUE
> z
[1] TRUE
```

3.12. Matrices

A matrix in *R* is a two-dimensional collection of values that all have the same data type. If all the elements are

numbers, the matrix is a numeric matrix. If the elements are strings, the matrix is a character matrix. If the elements are true/false, the matrix is a logical matrix. If all the values of the matrix are available as a vector, the matrix can be constructed using the matrix function as shown below.

```
> data <- c(80, 75, 85, 82,
+           90, 88, 92, 95,
+           81, 78, 84, 87)
>
> scores <- matrix(data,
+   nrow = 3, ncol = 4,
+   byrow = TRUE)
>
> scores
     [,1] [,2] [,3] [,4]
[1,]   80   75   85   82
[2,]   90   88   92   95
[3,]   81   78   84   87
```

In the above example, the vector data has 12 values. A matrix with 3 rows and 4 columns is constructed with the data interpreted row-wise.

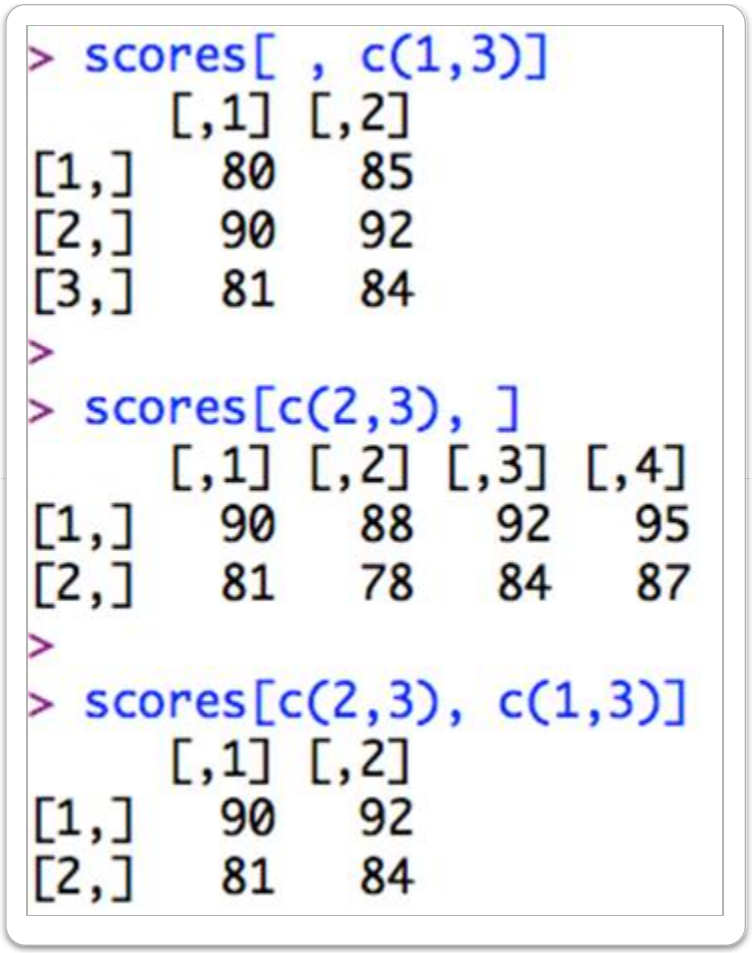
If the data in the vector is available column-wise, the same matrix can be constructed as shown below. The default value for the byrow argument is FALSE.

```
> data <- c(80, 90, 81,
+           75, 88, 78,
+           85, 92, 84,
+           82, 95, 87)
>
> scores <- matrix(data,
+   nrow = 3, ncol = 4)
>
> scores
     [,1] [,2] [,3] [,4]
[1,]   80   75   85   82
[2,]   90   88   92   95
[3,]   81   78   84   87
```

The contents of the matrix can be accessed using the numerical indices; this is shown below. If both the row and column index values are specified, the value at the corresponding position in the matrix is returned. If only the row index is specified, the vector for that row is returned. If only the column index is specified, the vector for that column is returned. The row and column indices start from 1.

```
> scores[1, 2]
[1] 75
>
> scores[1, ]
[1] 80 75 85 82
>
> scores[ ,1]
[1] 80 90 81
```

Multiple columns from the original matrix may be obtained at once by specifying the vector of the required column indices. Similarly, multiple rows may be returned by specifying the appropriate vector of the row indices. Combinations can also be used for both the row and column indices at the same time.



Test Yourself 1.24

Use the following exam scores to create a 2 x 5 matrix. Show the R commands for accessing the element at first row, third column. Show the R commands for accessing the fourth column

72, 22, 85, 89, 96, 77, 69, 78, 83, 92

```
x <- matrix(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92), nrow=2, ncol=5)
x[1, 3]
x[ , 4]
```

3.13. Named Matrices

The row and column dimensions may be named using the dimnames function. The function takes a list with the first argument being the vector for the row names and the second argument being the vector for the column names.

```
> dimnames(scores) <- list(
+     c("Alice", "Bob", "Charlie"),
+     c("Quiz1", "Quiz2", "Quiz3", "Quiz4"))
>
> scores
        Quiz1 Quiz2 Quiz3 Quiz4
Alice      80    75    85    82
Bob        90    88    92    95
Charlie    81    78    84    87
```

The matrix can now be indexed using the row and column names. Specifying both the values returns the corresponding entry from the matrix. Specifying only the row name returns the entire row as a named vector. Similarly, specifying only the column name will return the corresponding column as a named vector.

```
> scores["Alice", "Quiz2"]
[1] 75
>
> scores["Alice", ]
Quiz1 Quiz2 Quiz3 Quiz4
   80    75    85    82
>
> scores[ ,"Quiz1"]
  Alice     Bob Charlie
     80      90      81
```

Combinations of rows and/or columns can be accessed using the appropriate vector of row names or column names.

```
> scores[ , c("Quiz1", "Quiz3")]
        Quiz1 Quiz3
Alice      80    85
Bob        90    92
Charlie    81    84
>
> scores[c("Alice", "Charlie"), ]
        Quiz1 Quiz2 Quiz3 Quiz4
Alice      80    75    85    82
Charlie    81    78    84    87
>
> scores[c("Alice", "Charlie"),
+         c("Quiz1", "Quiz3")]
        Quiz1 Quiz3
Alice      80    85
Charlie    81    84
```

The row names and column names for a named matrix can be accessed as shown below.

```
> rownames(scores)
[1] "Alice"   "Bob"     "Charlie"
>
> colnames(scores)
[1] "Quiz1" "Quiz2" "Quiz3" "Quiz4"
```

The is.matrix function returns TRUE if the object is a matrix. The dim function returns a vector of the dimensions of the matrix. The nrow and ncol functions return the corresponding dimension values individually.

```
> is.matrix(scores)
[1] TRUE
>
> dim(scores)
[1] 3 4
>
> nrow(scores)
[1] 3
>
> ncol(scores)
[1] 4
```

The as.vector function converts the matrix into a vector. The data is combined column-wise.

```
> as.vector(scores)
 [1] 80 90 81 75 88 78 85 92 84 82 95 87
```

Test Yourself 1.25

Use dimnames and some of the previously explored commands to name the columns "Test 1" - "Test 5", and the rows "Bob" and "Sally"

```
x <- matrix(c(72, 22, 85, 89, 96, 77, 69, 78, 83, 92), nrow=2, ncol=5)
row_names <- c("Bob", "Sally")
col_names <- paste("Test", seq(1:5))
dimnames(x) <- list(row_names, col_names)
```

3.14. Modifying Matrix Entries

A single entry in the matrix may be modified by assigning a new value to the indexed row and column.

```
> scores[1, 1] <- 100
> scores
        Quiz1 Quiz2 Quiz3 Quiz4
Alice     100    75    85    82
Bob        90    88    92    95
Charlie    81    78    84    87
```

An entire row or column may also be modified, either assigning a single value or a vector of values, by indexing the row or column. In the following example, the first scenario assigns the value 90 to all the entries in the first row. In the second scenario, the vector of specified values is assigned to the named row. In the third scenario, as only two values are specified in the vector, the entries are recycled and the entire row is modified.

```
> scores[1, ] <- 90
> scores
        Quiz1 Quiz2 Quiz3 Quiz4
Alice      90    90    90    90
Bob        90    88    92    95
Charlie    81    78    84    87
>
> scores["Alice", ] <- c(91, 92, 93, 94)
> scores
        Quiz1 Quiz2 Quiz3 Quiz4
Alice      91    92    93    94
Bob        90    88    92    95
Charlie    81    78    84    87
>
> scores["Alice", ] <- c(80, 90)
> scores
        Quiz1 Quiz2 Quiz3 Quiz4
Alice      80    90    80    90
Bob        90    88    92    95
Charlie    81    78    84    87
```

The following scenario modifies all the entries with a single value.

```
> scores[,] <- 100
> scores
        Quiz1 Quiz2 Quiz3 Quiz4
Alice     100   100   100   100
Bob       100   100   100   100
Charlie   100   100   100   100
```

Test Yourself 1.26

Change Bob's 3rd test score to a 100.

Change all of Sally's scores to 80.

Show the R commands when accessing by index value, and by name.

```
x["Bob", "Test 3"] <- 100
X["Sally", ] <- rep(80, 5)
x[1, 3] <- 100
x[2, ] <- rep(80, 5)
```

3.15. Data Frames

A data frame is a collection of vectors. All the vectors in the data frame have the same length. Each vector could be of a different type. So, a data frame is similar to a matrix, except that the type of each column could be different. The following table shows the information about top five athletes, their sport, their salaries, and their endorsements (in millions of dollars) as of the year 2014.

	Name	Salary	Endorsements	Sport
1	Mayweather	105.0	0	Boxing
2	Ronaldo	52.0	28	Soccer
3	James	19.3	53	Basketball
4	Messi	41.7	23	Soccer
5	Bryant	30.5	31	Basketball

http://www.forbes.com/athletes/list/#tab:overall

A data frame for the above athletes' data can be constructed as follows. The individual columns are first created as vectors.

```
> athlete.names <- c("Mayweather", "Ronaldo",
+                    "James", "Messi", "Bryant")
> athlete.sport <- c("Boxing", "Soccer",
+                   "Basketball", "Soccer", "Basketball")
> athlete.salary <- c(105, 52, 19.3, 41.7, 30.5)
> athlete.endorsements <- c(0, 28, 53, 23, 31)
```

The data.frame function creates the two-dimensional data table using the arguments provided for the function.

```
> athlete.info <- data.frame(
+                   athlete.names, athlete.salary,
+                   athlete.endorsements, athlete.sport)
```

By default, the names of the argument objects are used as the column names for the data frame.

```
> athlete.info
  athlete.names athlete.salary athlete.endorsements athlete.sport
1    Mayweather          105.0                    0        Boxing
2       Ronaldo           52.0                   28        Soccer
3         James           19.3                   53    Basketball
4         Messi           41.7                   23        Soccer
5        Bryant           30.5                   31    Basketball
```

Explicit names for the columns can also be specified while creating the data frame.

```
> athlete.info <- data.frame(
+       Name = athlete.names,
+       Salary = athlete.salary,
+       Endorsements =  athlete.endorsements,
+       Sport = athlete.sport)
```

The column names of the data frame for the above example are shown below.

```
> athlete.info
        Name Salary Endorsements      Sport
1 Mayweather  105.0             0     Boxing
2    Ronaldo   52.0            28     Soccer
3      James   19.3            53 Basketball
4      Messi   41.7            23     Soccer
5     Bryant   30.5            31 Basketball
```

The colnames function returns the column names for the given data frame.

```
> colnames(athlete.info)
[1] "Name"         "Salary"       "Endorsements" "Sport"
```

The column names may also be modified later by assigning a new vector of strings to the colnames function.

```
> x <- data.frame(c(1,2), c(3,4))
> x
  c.1..2. c.3..4.
1       1       3
2       2       4
>
> colnames(x) <- c("Col1", "Col2")
> x
  Col1 Col2
1    1    3
2    2    4
```

The number of rows and the number of columns of the data frame can be obtained through the dim function or individually through the nrow and ncol functions.

```
> dim(athlete.info)
[1] 5 4
>
> nrow(athlete.info)
[1] 5
>
> ncol(athlete.info)
[1] 4
```

The dimnames function returns a list, the first component being a vector of row names and the second component being a vector of column names.

```
> dimnames(athlete.info)
[[1]]
[1] "1" "2" "3" "4" "5"

[[2]]
[1] "Name"          "Salary"        "Endorsements" "Sport"
```

The default rows names are the strings “1”, “2”, …

```
> rownames(athlete.info)
[1] "1" "2" "3" "4" "5"
```

The names function also returns the names of the columns as a vector.

```
> names(athlete.info)
[1] "Name"          "Salary"        "Endorsements" "Sport"
```

Test Yourself 1.27

Convert the following table into a data frame and print the dimensions. Show the R commands, make sure you name the columns appropriately

Person	Height	Age	Weight	IQ
John	173	25	200	95
Peter	175	26	185	75
Greg	195	32	191	65
James	165	28	160	150
Matthew	152	15	140	135
Peter	145	12	130	100

▶ Show Hint

```
df <- data.frame(
+     c('John', 'Peter', 'Greg', 'James', 'Matthew', 'Peter'),
+     c(173, 175, 195, 165, 152, 145),
+     c(25, 26, 32, 28, 15, 12),
+     c(200, 185, 191, 160, 140, 130),
+     c(95, 75, 65, 150, 135, 100)
colnames(df) <- ('Person', 'Height', 'Age', 'Weight', 'IQ')
dim(df)
```

3.16. Accessing Data Frame Data

The column data of the data frame may be accessed using the numeric index or the name of the column enclosed in [[]]. The character data is returned as a factored vector. The column data can also be accessed using the $ notation.

```
> athlete.info[[1]]
[1] Mayweather Ronaldo    James      Messi      Bryant
Levels: Bryant James Mayweather Messi Ronaldo
>
> athlete.info[["Salary"]]
[1] 105.0  52.0  19.3  41.7  30.5
>
> athlete.info$Sport
[1] Boxing     Soccer     Basketball Soccer     Basketball
Levels: Basketball Boxing Soccer
```

An alternative way to access the column data is to use the matrix notation, omitting the row information and only specifying the column index or name.

```
> athlete.info[ ,1]
[1] Mayweather Ronaldo    James      Messi      Bryant
Levels: Bryant James Mayweather Messi Ronaldo
>
> athlete.info[ ,"Salary"]
[1] 105.0  52.0  19.3  41.7  30.5
```

The summary function produces different output on the column data. For character data and logical data, the frequency for each of the levels is returned as a named vector. For numeric data, the mean and the five number summary (min, first quartile, median, second quartile, and max) values are returned.

```
> summary(athlete.info$Sport)
Basketball     Boxing     Soccer
         2          1          2
>
> summary(athlete.info$Salary)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   19.3    30.5    41.7    49.7    52.0   105.0
```

Attaching the data frame object to the *R* search path makes it convenient to work with the column data.

```
> attach(athlete.info)
>
> Salary
[1] 105.0  52.0  19.3  41.7  30.5
>
> summary(Sport)
Basketball     Boxing     Soccer
         2          1          2
>
> summary(Salary)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   19.3    30.5    41.7    49.7    52.0   105.0
>
> detach(athlete.info)
```

Individual values from the data frame may be accessed in different ways as shown below.

```
> athlete.info[3, 2]
[1] 19.3
> athlete.info[3,"Salary"]
[1] 19.3
> athlete.info[athlete.info$Name == "James","Salary"]
[1] 19.3
>
> attach(athlete.info)
> athlete.info[Name == "James","Salary"]
[1] 19.3
> detach(athlete.info)
```

Test Yourself 1.28

Using the previous Data Frame, access James' Height, Age, and IQ. Using the $ notation, print all Weights. Show the R commands

```
df[4, 2]
```

```
df[4, 3]
df[4, 4]
df$Weight
```

3.17. Slicing Data Frame Columns

The columns of the data frame may be sliced to produce new data frames. For single column slices, the [] notation is used with the numeric index or the column name.

```
> athlete.info[1]
        Name
1 Mayweather
2    Ronaldo
3      James
4      Messi
5     Bryant
>
> athlete.info["Sport"]
       Sport
1     Boxing
2     Soccer
3 Basketball
4     Soccer
5 Basketball
```

Multiple columns may be sliced to get new data frames by specifying a vector of column indices or names.

```
> athlete.info[c(1,4)]
        Name       Sport
1 Mayweather      Boxing
2    Ronaldo      Soccer
3      James Basketball
4      Messi      Soccer
5     Bryant Basketball
>
> athlete.info[c("Name", "Sport")]
        Name       Sport
1 Mayweather      Boxing
2    Ronaldo      Soccer
3      James Basketball
4      Messi      Soccer
5     Bryant Basketball
```

Test Yourself 1.29

Slice the data frame to print all of the IQ information using index value. Slice the data frame to print all the weights using column name. Slice the data frame to get all the names, heights, and weights. Show the R commands

```
df[ , 5]
df[ , 4]
df[ , c(1, 2, 4)]
```

3.18. Naming Data Frame Rows

When the data frame is created, the default names for the rows are the strings “1”, “2”, … The rownames function can be used to explicitly assign names for the rows.

```
> rownames(athlete.info)
[1] "1" "2" "3" "4" "5"
```

```
> rownames(athlete.info) <-
+     c("First", "Second", "Third",
+       "Fourth", "Fifth")
```

When displaying the data frame, the row names are used to identify the rows.

```
> athlete.info
              Name Salary Endorsements       Sport
First  Mayweather  105.0             0      Boxing
Second    Ronaldo   52.0            28      Soccer
Third       James   19.3            53 Basketball
Fourth      Messi   41.7            23      Soccer
Fifth      Bryant   30.5            31 Basketball
```

```
> rownames(athlete.info)
[1] "First"  "Second" "Third"  "Fourth" "Fifth"
```

3.19. Slicing Data Frame Rows

The rows of the data frame may be sliced to produce new data frames. For single row slices, the [] notation is used with the numeric row index or the explicit row name.

```
> athlete.info[2, ]
           Name Salary Endorsements  Sport
Second Ronaldo     52            28 Soccer
>
> athlete.info["Second", ]
           Name Salary Endorsements  Sport
Second Ronaldo     52            28 Soccer
```

Multiple rows may be sliced to get new data frames by specifying a vector of row indices or row names.

```
> athlete.info[c(1, 3), ]
             Name Salary Endorsements       Sport
First Mayweather  105.0             0      Boxing
Third      James   19.3            53 Basketball
>
> athlete.info[c("First", "Third"), ]
             Name Salary Endorsements       Sport
First Mayweather  105.0             0      Boxing
Third      James   19.3            53 Basketball
```

Logical indexing can also be used to select the desired rows. If a vector of TRUE/FALSE values is provided for the row index, only the rows corresponding to the TRUE index values are returned.

```
> athlete.info[c(FALSE, TRUE, FALSE, TRUE, FALSE), ]
     Name Salary Endorsements  Sport
2 Ronaldo   52.0            28 Soccer
4   Messi   41.7            23 Soccer
```

If the TRUE values are based on a logical expression, the expression can be used for the row index. The following code shows the selection of all athletes who play Soccer.

```
> athlete.info$Sport == "Soccer"
[1] FALSE  TRUE FALSE  TRUE FALSE
```

```
> athlete.info[athlete.info$Sport == "Soccer", ]
          Name Salary Endorsements  Sport
Second Ronaldo   52.0           28 Soccer
Fourth   Messi   41.7           23 Soccer
```

Test Yourself 1.30

Slice the data frame to print all of Matthew's info using index value. Use logical indexing to slice only the rows of people who weigh less than 170. Show the R commands

```
df[5, ]
df[df$Weight < 170, ]
```

3.20. Subset of a Data Frame

The subset function can be used with a data frame to return a new data frame that meets the specified condition. The following examples show the filtering of the data based on one or more columns.

```
> subset(athlete.info, Sport == "Soccer")
          Name Salary Endorsements  Sport
Second Ronaldo   52.0           28 Soccer
Fourth   Messi   41.7           23 Soccer
>
> subset(athlete.info,
+   Sport == "Soccer" & Salary > 50)
          Name Salary Endorsements  Sport
Second Ronaldo     52           28 Soccer
```

In the above scenarios, all the columns are included in the returned data frame. The required columns can also be selected explicitly.

```
> subset(athlete.info, Sport == "Soccer",
+    select = c(Name, Salary))
           Name Salary
Second Ronaldo   52.0
Fourth   Messi   41.7
```

Test Yourself 1.31

Use subset() on the data frame to select only the rows with age under 25, weight under 190, and IQ over 90. Without using subset, select the same rows using slicing. Show the R commands

```
subset(df, Age < 25 & Weight < 190 & IQ > 90)
df[df$Age < 25 & df$Weight < 190 & df$IQ > 90, ]
```

3.21. Modifying a Data Frame

A new column can be added to the data frame using a vector of explicit values or as a result of a calculation using the existing columns.

```
> data.orig <- athlete.info
>
> athlete.info$Pay <-
+    athlete.info$Salary + athlete.info$Endorsements
>
> athlete.info
             Name Salary Endorsements       Sport   Pay
First  Mayweather  105.0             0      Boxing 105.0
Second    Ronaldo   52.0            28      Soccer  80.0
Third       James   19.3            53  Basketball  72.3
Fourth      Messi   41.7            23      Soccer  64.7
Fifth      Bryant   30.5            31  Basketball  61.5
```

The Pay column is added to the data frame. The data frame is modified. If a copy of the data frame was made before the modification, the copy is not affected.

```
> data.orig
             Name Salary Endorsements       Sport
First  Mayweather  105.0             0      Boxing
Second    Ronaldo   52.0            28      Soccer
Third       James   19.3            53  Basketball
Fourth      Messi   41.7            23      Soccer
Fifth      Bryant   30.5            31  Basketball
```

The individual entries in the data frame can be modified by accessing the target position using the row and column index or the row and column name. The Salary and Endorsements values are modified for the first row in the following example.

```
> athlete.info[1,2] <- 0
> athlete.info["First", "Endorsements"] <- 100
> athlete.info
             Name Salary Endorsements      Sport   Pay
First  Mayweather    0.0           100     Boxing 105.0
Second    Ronaldo   52.0            28     Soccer  80.0
Third       James   19.3            53 Basketball  72.3
Fourth      Messi   41.7            23     Soccer  64.7
Fifth      Bryant   30.5            31 Basketball  61.5
```

Assigning a single value to a column modifies all the values for the column.

```
> athlete.info$Salary <- 50
> athlete.info
             Name Salary Endorsements      Sport   Pay
First  Mayweather     50           100     Boxing 105.0
Second    Ronaldo     50            28     Soccer  80.0
Third       James     50            53 Basketball  72.3
Fourth      Messi     50            23     Soccer  64.7
Fifth      Bryant     50            31 Basketball  61.5
```

Explicit values specified through a vector can be assigned to a column.

```
> athlete.info$Salary <- c(10, 20, 30, 40, 50)
> athlete.info
             Name Salary Endorsements      Sport   Pay
First  Mayweather     10           100     Boxing 105.0
Second    Ronaldo     20            28     Soccer  80.0
Third       James     30            53 Basketball  72.3
Fourth      Messi     40            23     Soccer  64.7
Fifth      Bryant     50            31 Basketball  61.5
```

Once the modifications are done, the Pay column has to be recomputed to reflect the correct values as the sum of the Salary and Endorsements columns.

```
> athlete.info$Pay <-
+    athlete.info$Salary + athlete.info$Endorsements
>
> athlete.info
             Name Salary Endorsements       Sport Pay
First  Mayweather     10          100      Boxing 110
Second    Ronaldo     20           28      Soccer  48
Third       James     30           53 Basketball  83
Fourth      Messi     40           23      Soccer  63
Fifth      Bryant     50           31 Basketball  81
```

A column can be removed entirely from the data frame by assigning the value NULL for that column.

```
> athlete.info$Pay <- NULL
>
> athlete.info
             Name Salary Endorsements       Sport
First  Mayweather     10          100      Boxing
Second    Ronaldo     20           28      Soccer
Third       James     30           53 Basketball
Fourth      Messi     40           23      Soccer
Fifth      Bryant     50           31 Basketball
```

Test Yourself 1.32

John went on a diet and lost 50 pounds, modify John's weight in the data frame.

Matthew hit his growth spurt and grew 10 cm. Modify Matthew's weight.

Add a new column named "Density" and fill it values by taking the weight and dividing it by the height.

Show the R commands

```
df[1, 4] <- df[1, 4] - 50
df[5, 2 <- df[5, 2] + 10
```

3.22. Factors

Factors are variables that have a limited number of different values. Factors are usually used for categorical data. The data can be a numeric vector or a character vector. The factor function creates levels for each of the distinct values. The factors are stored internally as integer values and displayed using a corresponding set of character values.

The following example shows the data collected about whether five individuals voted or not. The vector is character data. The factor function converts the data using the distinct levels present in the data. By default, the levels are not ordered and are displayed in the alphabetical order.

```
> voted <- c("yes", "no", "yes", "yes", "no")
> voted
[1] "yes" "no"  "yes" "yes" "no"
>
> f1 <- factor(voted)
> f1
[1] yes no  yes yes no
Levels: no yes
>
> levels(f1)
[1] "no"  "yes"
```

The following shows that the given data is a character vector, but the factored result is not. The is.factor returns true for the factored result.

```
> is.character(voted)
[1] TRUE
>
> is.character(f1)
[1] FALSE
>
> is.factor(f1)
[1] TRUE
```

Since the input data is character type, the numeric conversion produces NA values for each. However, the numeric conversion of the factored data uses the integers. The first level is 1, the second level is 2, …

```
> as.numeric(voted)
[1] NA NA NA NA NA
Warning message:
NAs introduced by coercion
>
> as.numeric(f1)
[1] 2 1 2 2 1
```

The factored data can be assigned explicit labels as shown below. The no and yes levels are mapped to the levels bad and good respectively.

```
> f2 <- factor(voted, labels=c("bad", "good"))
> f2
[1] good bad  good good bad
Levels: bad good
>
> levels(f2)
[1] "bad"  "good"
```

Mapping the factors using a different order for the labels produces a different result, which is shown below.

```
> f3 <- factor(voted, labels=c("good", "bad"))
> f3
[1] bad  good bad  bad  good
Levels: good bad
>
> levels(f3)
[1] "good" "bad"
```

The summary function shows the frequencies of each distinct value for the factored data.

```
> summary(voted)
   Length     Class      Mode
        5 character character
>
> summary(f1)
 no yes
  2   3
>
> summary(f2)
 bad good
   2    3
>
> summary(f3)
good  bad
   2    3
```

If the input data is a logical vector, the factored data will have two levels as shown below.

```
> voted <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
> voted
[1]  TRUE FALSE  TRUE  TRUE FALSE
>
> f1 <- factor(voted)
> f1
[1] TRUE  FALSE TRUE  TRUE  FALSE
Levels: FALSE TRUE
>
> levels(f1)
[1] "FALSE" "TRUE"
```

The following shows that the given data is a logical vector, but the factored result is not. The is.factor returns true for the factored result.

```
> is.logical(voted)
[1] TRUE
>
> is.logical(f1)
[1] FALSE
>
> is.factor(f1)
[1] TRUE
```

If the input data is a numeric vector with zeros and ones, the factored data will have the levels shown below.

```
> voted <- c(1, 0, 1, 1, 0)
> voted
[1] 1 0 1 1 0
>
> f1 <- factor(voted)
> f1
[1] 1 0 1 1 0
Levels: 0 1
>
> levels(f1)
[1] "0" "1"
```

The following shows that the given data is a numeric vector, but the factored result is not. The is.factor returns TRUE for the factored result.

```
> is.numeric(voted)
[1] TRUE
>
> is.numeric(f1)
[1] FALSE
>
> is.factor(f1)
[1] TRUE
```

In the above scenarios, there is no ordering between the levels. The factored data cannot be compared if one level is less than the other.

```
> f1[1] < f1[2]
[1] NA
Warning message:
In Ops.factor(f1[1], f1[2]) : < not meaningful for factors
```

In cases where ordering exists among the different levels, the ordered function factors the input data with the levels ordered by alphabetical order as default. The factor function with the ordered argument as TRUE also produces the same result.

```
> skills <- c("advanced", "novice", "novice",
+     "intermediate", "advanced")
>
> skills
[1] "advanced"     "novice"       "novice"       "intermediate" "advanced"
>
> o1 <- ordered(skills)
> o1
[1] advanced     novice       novice       intermediate advanced
Levels: advanced < intermediate < novice
>
> o1 <- factor(skills, ordered = TRUE)
> o1
[1] advanced     novice       novice       intermediate advanced
Levels: advanced < intermediate < novice
>
> levels(o1)
[1] "advanced"     "intermediate" "novice"
```

The is.ordered and is.factor functions return TRUE for the ordered factored result.

```
> is.factor(o1)
[1] TRUE
>
> is.ordered(o1)
[1] TRUE
```

In the default ordering, the levels are ordered by alphabetical order. If that is not the case, the levels argument is specified with the correct ordering for the levels.

```
> o2 <- ordered(skills,
+   levels=c("novice", "intermediate", "advanced"))
> o2
[1] advanced     novice       novice
[4] intermediate advanced
Levels: novice < intermediate < advanced
>
> levels(o2)
[1] "novice"       "intermediate" "advanced"
```

The ordered data can now be compared against each other.

```
> o2[1] > o2[2]
[1] TRUE
> o2[1] < o2[4]
[1] FALSE
```

Test Yourself 1.33 (Multipart)

Use the factor() function to show the levels of the following vector:

c("High", "Medium", "Low", "Medium", "Low")

(Click each step to reveal its solution.)

▶ Show the R commands.

```
x <- c("High", "Medium", "Low", "Medium", "Low")
f <- factor(x)
f
levels(f)
```

▶ Order the factor levels as Low < Medium < High. Show the R commands.

```
x <- c("High", "Medium", "Low", "Medium", "Low")
o <- factor(x, levels = c("Low", "Medium", "High"))
# or
o <- ordered(x, levels = c("Low", "Medium", "High"))
```

- o

3.23. Lists

A list is an ordered collection of components. Each component could be a vector, a primitive (a vector of unit length), or any other data structure.

The following example constructs a list from a string, a numeric vector, a character vector, and a number. When a list is displayed, the individual component indices are printed using the [[]] notation followed by the contents of that component.

```
> num.data  <- c(3, 5, 7)
> char.data <- c("a1", "b2", "c3", "d4")
>
> list.data <- list("Hello", num.data, char.data, 20)
>
> list.data
[[1]]
[1] "Hello"

[[2]]
[1] 3 5 7

[[3]]
[1] "a1" "b2" "c3" "d4"

[[4]]
[1] 20
```

The components of a list are accessed (sliced) using the [] notation. The index can be a single value or a vector of indices. The values are returned as a list.

```
> list.data[2]
[[1]]
[1] 3 5 7

>
> list.data[c(2, 3)]
[[1]]
[1] 3 5 7

[[2]]
[1] "a1" "b2" "c3" "d4"
```

The contents of the list can be accessed using the [[]] notation. The index used returns the corresponding content at that index.

```
> list.data[[2]]
[1] 3 5 7
```

Using a vector for the [[]] index recursively accesses the subcomponents of that component.

```
> list.data[[c(2,1)]]
[1] 3
> list.data[[c(2,3)]]
[1] 7
```

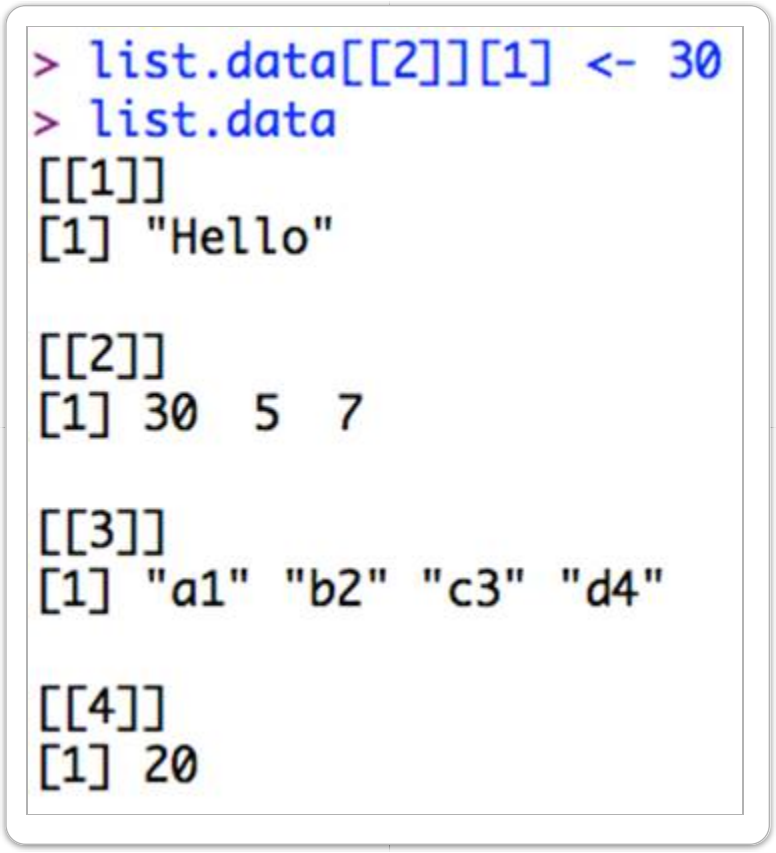
Test Yourself 1.34

Create a list holding both Bob and Sally's test scores as individual vectors. Show the R Commands.

```
list_scores <- list(x[1, ], x[2, ])
```

3.24. Modifying Lists

A list can be modified by explicitly assigning a value to the index of the subcomponent. In the following example, the first value of the second component of the list is modified.



A subcomponent of the list can be replaced with new data as shown below. The second component is replaced with a new vector.

```
> list.data[[2]] <- 1:10
> list.data
[[1]]
[1] "Hello"

[[2]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[3]]
[1] "a1" "b2" "c3" "d4"

[[4]]
[1] 20
```

The original vector from which the list is constructed is not affected by the above modifications as the list uses a copy of the data.

```
> num.data
[1] 3 5 7
```

The following example copies the subcomponent of the list first and then modifies the list. The original data that was copied remains unaffected.

```
> x <- list.data[[2]]
>
> x
 [1]  1  2  3  4  5  6  7  8  9 10
>
> list.data[[2]] <- 10:1
>
> list.data
[[1]]
[1] "Hello"

[[2]]
 [1] 10  9  8  7  6  5  4  3  2  1

[[3]]
[1] "a1" "b2" "c3" "d4"

[[4]]
[1] 20


>
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

Test Yourself 1.35

Change Sally's 3rd test score to a 100. Show the R commands.

```
list_scores[[2]][3] <- 100
```

3.25. Named Lists

The components of the list can be named and referenced using the names instead of the indices. In the following example, the two components of the list are named as teams and players, respectively. Displaying the list now lists the names with the $ prefix followed by the contents for each component of the list.

```
> team.names  <- c("Patriots", "Red Sox")
> player.names <- c("Brady", "Federer", "Pele")
>
> favorites <- list(teams = team.names,
+                   players = player.names)
>
> favorites
$teams
[1] "Patriots" "Red Sox"

$players
[1] "Brady"   "Federer" "Pele"
```

The contents of the list components can now be accessed using the numeric index or the named index. The third option shows the list accessed without the brackets, but with the $ notation. The teams and players can be accessed as shown below.

```
> favorites[[1]]
[1] "Patriots" "Red Sox"
>
> favorites[["teams"]]
[1] "Patriots" "Red Sox"
>
> favorites$teams
[1] "Patriots" "Red Sox"
```

```
> favorites[[2]]
[1] "Brady"   "Federer" "Pele"
>
> favorites[["players"]]
[1] "Brady"   "Federer" "Pele"
>
> favorites$players
[1] "Brady"   "Federer" "Pele"
```

The attach function adds the specified object to the search path of *R*. The components can now be accessed directly as shown below.

```
> attach(favorites)
>
> teams
[1] "Patriots" "Red Sox"
>
> players
[1] "Brady"   "Federer" "Pele"
>
> detach(favorites)
```

The above access works only when the names are not previously assigned to other data.

The contents of the list can also be modified using the named access as shown below.

```
> favorites$teams[2] <- "Yankees"
>
> favorites
$teams
[1] "Patriots" "Yankees"

$players
[1] "Brady"    "Federer" "Pele"
```

The is.list function returns TRUE if the object is a list. Note that the components of the list are returned as lists too.

```
> is.list(list.data)
[1] TRUE
>
> is.list(list.data[2])
[1] TRUE
>
> is.list(favorites)
[1] TRUE
>
> is.list(favorites$players)
[1] FALSE
>
> is.list(favorites["players"])
[1] TRUE
```

Test Yourself 1.36

It is sad that we don't have student name labels for these two vectors. We can name the elements of list when we instantiate it. Create a named list of Bob and Sally's test scores, show the R commands.

```
list_scores <- list(Bob=x[1, ], Sally= x[2, ])
```