MTH 3240 R Notes 2

2 Getting Started (Continued)

2.8 A Preview of R Data Structures

- There are five ways to store data sets in R. These differ according to their dimensionality (1D, 2D, or nD) and whether they're *homogeneous* (all contents must be of the same type) or *heterogeneous* (contents can be of different types):
  - **Vectors** (1D, homog.)
  - **Lists** (1D, heterog.)
  - **Matrices** (2D, homog.)
  - **Data Frames** (2D, heterog.)
  - **Arrays** (nD, homog.)

2.8.1 A Preview of Vectors

- ***Vectors*** are created using the "combine" function:

```
c()                    # Combine values to form a vector
```

- Here's an example:

```
num.vec <- c(7, 4, 5)
num.vec

## [1] 7 4 5
```

- Vectors can store any of the *atomic* types. Here's one that stores `"character"` values:

```
char.vec <- c("a", "b", "c")
char.vec

## [1] "a" "b" "c"
```

and here's one that stores `"logical"` values:

```
logic.vec <- c(TRUE, TRUE, FALSE)
logic.vec

## [1]  TRUE  TRUE FALSE
```

- The functions `typeof()`, `is.numeric()`, `is.character()`, and `is.logical()` work on vectors too:

```
typeof(logic.vec)

## [1] "logical"
```

2.8.2 A Preview of Matrices

- ***Matrices*** are like two-dimensional vectors (i.e. they have rows and columns). One way to create a matrix is using:

```
matrix()             # Create a matrix, from a vector, with a speci-
                     # fied number of rows and columns
```

- Here's an example:

```
my.mat <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3)
my.mat

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

Note that by default, R fills the matrix by columns, left to right. The numbers in square brackets `[ ]` identify the row and column.

2.8.3 A Preview of Lists

- The elements of a vector all have to be the same type. ***Lists*** are like vectors, but their elements can be different types. One way to create a list is using:

```
list()               # Create a list from a set of R objects
```

- Here's an example of a list containing three element types, a `"character"` value, numeric one, and a `"logical"` one:

```
my.list <- list("d", 12, TRUE)
my.list

## [[1]]
## [1] "d"
##
## [[2]]
## [1] 12
##
## [[3]]
## [1] TRUE
```

Notice that the way R prints out lists is different from the way it prints out vectors.

- In fact, elements of a list can be *any* R objects, for example here's a list whose two elements are each vectors:

```
my.list <- list(c("a", "b", "c"), c(7, 4, 5))
my.list

## [[1]]
## [1] "a" "b" "c"
##
## [[2]]
## [1] 7 4 5
```

2.8.4 A Preview of Data Frames

- ***Data frames*** are like matrices, but the types of values they contain can differ from one column to the next.
- One way to create a data frame is with the function:

```
data.frame()          # Create a data frame from a set of vectors
                      # (which will form the columns of the data frame)
```

- We can choose names for the columns of a data frame. In the example below, we use `var1` and `var2` for the column names:

```
my.df <- data.frame(var1 = c("a", "b", "c"), var2 = c(7, 4, 5))
my.df

##   var1 var2
## 1    a    7
## 2    b    4
## 3    c    5
```

Notice that the first column is `"character"`, and the second is numeric.

2.8.5 A Preview of Arrays

- An ***array*** is like a matrix, but it can have more than two dimensions (e.g. rows, columns, and layers). We can create an array using:

```
array()          # Create an array, from a vector, with a specified
                 # number of dimensions
```

We won't be using arrays.

Section 2.8 Exercises

Exercise 1 Write a command using `c()` that creates a vector containing the values:

$$3, 7, 2, 8$$

Exercise 2 Type the following command and report the result:

```
x <- matrix(c(2, 4, 6, 8, 10, 12, 14, 16, 18), nrow = 3, ncol = 3)
x
```

Exercise 3 Write a command using `list()` that creates a list containing the following elements:

$$"e", 9, \text{TRUE}$$

Exercise 4 Type the following command and report the result:

```
y <- data.frame(Category = c("A", "A", "B", "B", "C", "C"),
                Value = c(5, 4, 6, 6, 9, 8))
y
```

3 Vectors

3.1 Creating and Examining Vectors

- The following functions will be used to create and examine vectors:

```
c()           # Create a vector of values
length()      # Returns the number of elements in a vector
is.vector()   # Indicates whether or not an object is a vector
```

- Here's an example:

```
x <- c(7, 4, 5)
length(x)

## [1] 3

is.vector(x)

## [1] TRUE
```

- We can also use `c()` to combine two (or more) existing vectors end-to-end to create a new vector:

```
x <- c(7, 4, 5)
y <- c(1, 2, 3)
```

```
my.new.vec <- c(x, y)
```

```
my.new.vec

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

- A (single-valued) variable is actually a one-element vector:

```
x <- 7
is.vector(x)

## [1] TRUE

length(x)

## [1] 1
```

3.2 Vector Arithmetic and Recycling

- When we perform arithmetic operations ('+', '-', '*', '/', and '^') on two vectors, their elements are matched and the operation is performed one pair of elements at a time:

```
x <- c(7, 4, 5)
y <- c(1, 2, 3)
```

```
x + y

## [1] 8 6 8
```

- If the vectors have different lengths, the shorter one is repeated as necessary, i.e. its values are ***recycled***, and R prints a warning message:

```
z <- c(1, 2, 3, 4, 5)
y <- c(1, 2, 3)
```

```
z - y

## Warning in z - y:  longer object length is not a multiple of shorter object
length

## [1] 0 0 0 3 3
```

Above, because `y` is shorter than `z`, the elements of `y` are recycled until the two vectors are of equal length. This is equivalent to subtracting `c(1, 2, 3, 1, 2)` from `z`, i.e.:

```
c(1, 2, 3, 4, 5) - c(1, 2, 3, 1, 2)

## [1] 0 0 0 3 3
```

Section 3.2 Exercises

Exercise 5 Guess what the result of each of the following will be, then check your answers:

a)
```
x <- c(2, 3, 4)
y <- c(6, 7, 8)
x + y
```

b)
```
x <- c(2, 3, 4)
x * x
```

c)
```
x <- c(2, 3, 4)
x^2
```

Exercise 6 Guess what the result of each of the following will be, then check your answers:

a)
```r
x <- c(2, 3, 4)
x + 1
```

b)
```r
x <- c(2, 3, 4)
x * 2
```

Exercise 7 If two vectors have different lengths, the shorter one is repeated as necessary, i.e. its values are ***recycled***. Guess what the result of each of the following will be, then check your answers:

a)
```r
y <- c(6, 7, 8, 9)
z <- c(2, 3)
y + z
```

b)
```r
y <- c(4, 8, 12, 16)
w <- c(2, 4, 6)
y / w
```

Exercise 8 A (single-valued) variable is actually a one-element vector. Guess what the result of each of the following will be, then check your answers:

a)
```r
x <- 2
is.vector(x)
```

b)
```r
is.vector(2)
```

3.3 Vector Coercion

- All elements of a vector must be of the same type, so if you try to combine values of different types, they'll be ***coerced*** to the *most flexible* type. Types from least to most flexible are:

Least Flexible	`"logical"`
↓	`"integer"`
	`"double"`
Most Flexible	`"character"`

- In particular, if we combine numeric values (`"double"`) with a `"character"` value, the numerical values are coerced to `"character"`:

```r
c(4, 7, 5, "a")

## [1] "4" "7" "5" "a"
```

Above, we can tell the numerical values were converted to `"character"`s (as indicated by the quotes around the numbers).

- If we combine `"logical"` values with a numeric (`"double"`) value, `TRUE` is coerced to 1 and `FALSE` to 0:

```r
c(TRUE, FALSE, 7)

## [1] 1 0 7
```

Section 3.3 Exercises

Exercise 9

a) If we combine numeric values with a `"character"` value, the numerical values are coerced to `"character"`. Guess what the result of the following will be, then check your answer:

```r
x <- c(2, 3, "b")
x
```

b) If we combine `"logical"` values with a numeric value, the `"logical"` values (`TRUE` and `FALSE`) are coerced to 0 and 1. Guess what the result of the following will be, then check your answer:

```r
x <- c(FALSE, TRUE, 3)
x
```

3.4 Common Vector Operations

3.4.1 Vector Indexing Using `[ ]`

Accessing Vector Elements

- We access one or more elements of a vector using their ***indices*** in square brackets:

```
[ ]                    # Access vector elements via their indices
```

- For example, typing `x[3]` returns the 3rd element of a vector `x`:

```
x <- c(5, 7, 9, 8, 1)
x[3]

## [1] 9
```

and typing `x[c(3, 4)]` returns the 3rd and 4th elements:

```
x[c(3, 4)]

## [1] 9 8
```

Replacing Vector Elements

- We can also use the brackets `[ ]` to *replace* specific values in `x`. For example, to replace the third element by 13, type:

```
x[3] <- 13
```

```
x

## [1]  5  7 13  8  1
```

Deleting Vector Elements

- A negative index returns all but that element from the vector. For example to obtain all but the 5th element of `x`, type:

```
x[-5]

## [1]  5  7 13  8
```

If we want to permanently delete the 5th element, we need to overwrite `x` by `x[-5]`:

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

Rearranging Vector Elements

- One way to rearrange (permute) the elements of a vector is to specify the desired permutation in square brackets. For example, consider the vector `y`:

```
y

## [1] 11 18 15
```

If we want its elements in the order 18, 15, 11, we type:

```r
y[c(2, 3, 1)]

## [1] 18 15 11
```

Above, the vector `c(2, 3, 1)` indicates that we want the 2nd element of `y` moved to the first position, the 3rd element to the second position, and the 1st element to the third position.

Other Ways of Rearranging the Elements of a Vector

- Here are some other functions that can be used to rearrange the elements of a vector:

```r
sort()            # Returns the elements of a vector in sorted order
rev()             # Returns the elements of a vector in reverse order
order()           # Returns a vector of indices such that x[order(x)]
                  # returns the vector x in sorted order
```

3.4.2 Introduction to Filtering

- We can use a `"logical"` vector inside square brackets to ***filter*** out certain elements of a vector `x`:

```r
x <- c(5, 7, 9, 1)
x[c(TRUE, FALSE, FALSE, TRUE)]

## [1] 5 1
```

Above, only the elements of `x` corresponding to `TRUE` in the `"logical"` vector are returned.

3.4.3 Creating More Specialized Vectors with `seq()`, `:`, and `rep()`

- The functions and operator below are useful for creating sequences and repeating patterns of values:

```r
seq()          # Create a sequence of values
:              # Create a sequence of integers
rep()          # Create a repeating pattern of values
```

Creating Sequences of Values Using `seq()` and ':'

- `seq()` creates a sequence of values starting at `from` and ending at `to`, with increment `by`:

```r
seq(from = 1, to = 5, by = 0.5)

## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

- The colon operator ':' can be used to create a sequence of consecutive *integers*. For example:

```
1:10

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

produces the same result as:

```
seq(from = 1, to = 10, by = 1)
```

Creating Repeating Patterns of Values Using `rep()`

- `rep()` takes a value (via its first argument `x`) and repeats it a specified number of times (via `times`):

```
rep(1, times = 5)

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

- We can use `rep()` with `"character"` values too:

```
rep("a", times = 5)

## [1] "a" "a" "a" "a" "a"
```

- When the first argument is a vector, `rep()` repeats that vector end-to-end the specified number of `times`:

```
rep(1:3, times = 2)

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

Section 3.4 Exercises

Exercise 10 Consider the vector

```
x <- c(17, 22, 13, 14, 23, 27)
```

Guess what the result of each of the following will be, then check your answers:

a) `x[2]`

b) `x[-2]`

c)
```
x[c(1, 2)]
```

d)
```
x[c(2, 1)]
```

e)
```
x[1] <- 5
x
```

Exercise 11 Consider the vector

```
x <- c(17, 22, 13, 14, 23, 27)
```

a) Write a command using square brackets `[ ]` that returns the 4th element of `x`.

b) Write a command using square brackets `[ ]` and the assignment operator `<-` that replaces the 4th element of `x` with the value 19.

c) Write a command using square brackets `[ ]` that returns all but the 6th element of `x`.

Exercise 12 Consider the vector

```
x <- c(17, 22, 13, 14)
```

The following command will return the elements of `x` in a new order. Guess what order they'll be in, then check your answer.

```
x[c(2, 1, 4, 3)]
```

Exercise 13 Consider the vector

```
x <- c(17, 22, 13, 14, 23, 27)
```

a) Write a command using `sort()` that returns the elements of `x` sorted in ascending order.

b) Write a command using `rev()` that returns the elements of `x` in reverse order.

Exercise 14 Consider the vector

```
x <- c(17, 22, 13, 14)
```

Guess what the result of the following will be, then check your answer:

```
x[c(FALSE, FALSE, TRUE, TRUE)]
```

Exercise 15 Guess what the result of each of the following will be, then check your answers:

a)
```
1:5
```

b)
```
6:10
```

Exercise 16 Guess what the result of the following will be, then check your answer:

```
is.vector(1:5)
```

Exercise 17 Guess what the result of the following will be, then check your answer:

```
seq(from = 1, to = 2.5, by = 0.5)
```

Exercise 18 Write a command using the colon operator `:`, that creates the vector:

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

Exercise 19 Write a command using `rep()` that creates the vector:

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

Exercise 20 Write a command using `rep()` that creates the vector:

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

3.5 Computing Summary Statistics

- Several functions take vector arguments and compute summary statistics:

```
length()         # Number of elements in a vector (i.e. sample size)
min(); max()     # Smallest and largest values in a vector
range()          # Range (smallest and largest values) of a vector
sum()            # The sum of the values in a vector
prod()           # The product of the values in a vector
mean()           # The sample mean
median()         # Sample median
```

```
sd(); var()         # Sample standard deviation and variance
mad()               # Median absolute deviation
quantile()          # Sample quantile (percentile)
IQR()               # Interquartile range
summary()           # Five number summary (and sample mean)
```

- `mean(x)` computes the ***sample mean***, or arithmetic average, denoted $\bar{X}$, of the values $X_1, X_2, \ldots, X_n$ contained in a vector `x`:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

- `median(x)` computes the ***sample median***, or "middle value" after sorting the data:

$$\text{Median} = \begin{cases} \text{The middle sorted value if } n \text{ is odd.} \\ \text{The average of the two middle sorted values if } n \text{ is even.} \end{cases}$$

- `var(x)` computes the ***sample variance***, or "average" squared deviation of an X_is away from the mean, denoted S^2:

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2.$$

- `sd(x)` computes the ***sample standard deviation***, S, which is the square root of the variance and represents a typical deviation of an X_i away from the mean:

$$S = \sqrt{S^2} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2}.$$

Section 3.5 Exercises

Exercise 21 Consider the following data set:

```
x <- c(10, 7, 6, 7, 12, 9, 12, 11, 8, 147)
```

a) Use `mean()` to compute the mean.

b) Use `median()` to compute the median.

c) Type

```
sort(x)
```

How far into the sorted data set does the median lie? How far in does the mean lie? Why is the mean so much higher than the median?

d) If we added 10 to each value of `x`, what would happen to the mean? What would happen to the median? Try it:

```
x <- x  + 10
mean(x)
median(x)
```

Exercise 22

a) The standard deviation measures variation in a set of data. What do you think the standard deviation of the following data set will be? Check your answer using `sd()`.

```
u <- c(5, 5, 5, 5, 5, 5)
```

b) Which of the following two data sets do you think will have a larger standard deviation? Check your answer using `sd()`.

```
v <- c(9, 10, 11)
w <- c(1, 10, 19)
```

Exercise 23

a) Use `sd()` to compute the standard deviation of `x` from Exercise 21:

```
x <- c(10, 7, 6, 7, 12, 9, 12, 11, 8, 147)
```

b) What would happen to the standard deviation if we added 10 to each value of `x`? Try it:

```
x <- x  + 10
sd(x)
```

3.6 Vectorized Computations

- We've seen that the operators '`+`', '`-`', '`*`', '`/`', and '`^`' operate elementwise on vectors.
- Many of R's built-in functions operate one element at a time too. For example, watch what happens when we pass a vector to `sqrt()`:

```
x <- c(4, 9, 16, 25)
sqrt(x)
```

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

- Functions that perform calculations on vectors one element at a time are said to be ***vectorized***.

Section 3.6 Exercises

Exercise 24 The function `abs()` takes the absolute value of a number. `abs()` is a *vectorized* function. Guess what the result of the following command will be, then check your answer:

```
x <- c(-1, 3, -4, -2)
abs(x)
```

Exercise 25 Consider the following temperature measurements, in degrees Celsius:

```
degreesC <- c(23, 19, 21, 22, 18, 20, 24, 25)
```

The relation between Celsius (°C) and Fahrenheit (°F) is:

$$^{\circ}\mathrm{F} \;=\; \frac{9}{5} \times {}^{\circ}\mathrm{C} + 32$$

In words, what will the following command do to the Celsius temperatures? Try it.

```
(9/5) * degreesC + 32
```