MTH 2520 R Notes 2

1 Getting Started (Continued)

1.3 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.)

1.3.1 A Preview of Vectors

- ***Vectors*** (sometimes called ***atomic 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"`

```
> 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"
```

1.3.2 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.

1.3.3 A Preview of Lists

- The elements of an atomic vector all have to be the same type. ***Lists*** are vectors whose 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:

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

[[1]]
[1] "d"

[[2]]
[1] 12
```

```
[[3]]
[1] TRUE
```

- In fact, elements of a list can be *any* R objects, for example vectors:

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

[[1]]
[1] "a" "b" "c"

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

- Lists are sometimes called ***recursive vectors*** because the elements of a list can be other lists:

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

1.3.4 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
```

1.3.5 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
```

Section 1.3 Exercises

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

$$3, 7, 2, 8$$

Exercise 2 Write a command using `matrix()` that creates the following matrix:

$$\begin{pmatrix} 2 & 8 & 14 \\ 4 & 10 & 16 \\ 6 & 12 & 18 \end{pmatrix}$$

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

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

Exercise 4 Write a command using `data.frame()` that creates a data frame containing the following data set:

Category	Value
A	5
A	4
B	6
B	6
C	9
C	8

2 Vectors

2.1 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 (scalar) variable is actually a one-element vector:

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

[1] TRUE

> length(x)

[1] 1
```

2.2 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

[1] 0 0 0 3 3

Warning message:
In z - y :  longer object length is not a multiple of shorter object length
```

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 2.2 Exercises

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

a)
```
> x <- c(2, 3, 4, 5)
> y <- c(6, 7, 8, 9)
> c(x, y)
```

b)
```
> x <- c(2, 3, 4, 5)
> y <- c(6, 7, 8, 9)
> x + y
```

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

a)
```
> x <- c(2, 3, 4, 5)
> x + 1
```

b)
```
> x <- c(2, 3, 4, 5)
> x * 2
```

c)
```
> x <- c(2, 3, 4, 5)
> x^2
```

Exercise 7 Guess what the result of each of the following will be, then check your answers:

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

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

Exercise 8 In R, single-valued variables and constants are vectors of length one. Guess what the result of each of the following will be, then check your answers:

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

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

2.3 Vector Coercion

- All elements of a vector must be of the same type, so if you try to combine vectors 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 a numeric vector (`"double"`) with a `"character"` vector, the numerical values are coerced to `"character"`:

```
> num.vec <- c(4, 7, 5)
> char.vec <- c("a", "b")
> c(num.vec, char.vec)

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

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

```
> c(TRUE, FALSE, 7, 8)

[1] 1 0 7 8
```

Section 2.3 Exercises

Exercise 9 Guess what the result of each of the following will be, then check your answers:

a)
```
> x <- c(2, 3)
> y <- c("a", "b")
> c(x, y)
```

b)
```
> x <- c(2, 3)
> y <- c(FALSE, TRUE)
> c(x, y)
```

c)
```
> x <- c("a", "b")
> y <- c(FALSE, TRUE)
> c(x, y)
```

2.4 Common Vector Operations

2.4.1 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`:

```
> 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:

```
> 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:

```
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
```

2.4.2 Introduction to Filtering

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

```
> x <- c(5, 7, 9, 8, 1)
> x[c(TRUE, FALSE, TRUE, TRUE, FALSE)]

[1] 5 9 8
```

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

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

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

```
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`:

```
> 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 = 10)

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

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

```
> rep("a", times = 10)

 [1] "a" "a" "a" "a" "a" "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 = 4)
```

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

- When both arguments are vectors (of equal length), the second vector indicates the number of `times` to repeat *each* element of the first vector. For example:

```
> rep(c(5, 6, 8), times = c(1, 2, 3))
```

```
[1] 5 6 6 8 8 8
```

Above, the first element of `c(5, 6, 8)` was repeated once (because the first element of `times` is 1), the second element two times (because the second element of `times` is 2), and the third element three times (because the third element of `times` is 3).

Section 2.4 Exercises

Exercise 10 Consider the vector

```
> x <- c(7, 6, 4, 2, 3, 5)
```

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(1, 2)]`

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

f)
```
> x[1] <- 5
> x
```

g)
```
> x[c(1, 2)] <- c(8, 9)
> x
```

Exercise 11 Consider the vector

```
> x <- c(7, 6, 4, 2, 3, 5)
```

a) Write a command that returns the 4th element of `x`.

b) Write a command that replaces the 4th element of `x` with the value 1.

c) Write a command that returns all but the 6th element of `x`.

Exercise 12 Consider the vector

```
> x <- c(7, 6, 4, 2)
```

Write a command using square brackets, `[ ]`, that returns the elements of `x` in the following order:

```
[1] 6 7 4 2
```

Exercise 13 Consider the vector

```
> x <- c(7, 6, 4, 2, 3, 5)
```

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.

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

```
> rev(sort(x))
```

d) Look at the help file for `sort()` by typing

```
> ? sort
```

Notice the `sort()` has an optional argument, `decreasing`, whose default value is `FALSE`. Write a command using `sort()` that returns the elements of `x` in *descending* order by setting `decreasing = TRUE`.

Exercise 14 Consider the vector

```
> x <- c(7, 6, 4, 2, 3, 5)
```

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

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

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

a) `> 1:5`

b) `> 6:10`

c) `> 5:1`

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

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

Exercise 17 Open the help file for operator precedence by typing:

```
> ? Syntax
```

After checking which operator, `-` or `:`, has higher precedence, guess what the result of each the following will be, then check your answers:

a) `> -3:6`

b) `> -(3:6)`

c) `> -3:-6`

Exercise 18 Guess what the result of each of the following will be, then check your answers:

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

b) `> seq(from = 2.5, to = 1, by = -0.5)  # Note that 'by' is negative`

Exercise 19 Write a command, using the colon operator `:` and `c()`, that creates the vector:

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

Exercise 20 Guess what the result of each of the following will be, then check your answers:

a) `> rep(3, times = 4)`

b) `> rep(1:3, times = 4)`

c) `> rep(1:3, times = 1:3)`