MTH 3220 R Notes 2

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, 6)                           # Create a vector x
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, 6)
y <- c(1, 2, 3)
```

```
xy <- c(x, y)                             # Combine two vectors
xy

## [1] 7 4 6 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, 6)
y <- c(1, 2, 3)
```

```
x + y                                          # Elementwise addition

## [1] 8 6 9
```

- 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                          # Elements of the shorter vector y are recycled

## 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 equl 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 1 Guess what the result of each of the following will be, then check your answers:

a)
```
x <- c(2, 3, 4)
is.vector(x)
```

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

c)
```
x <- 2
is.vector(x)
```

d)
```
is.vector(2)
```

Exercise 2 R operates on vectors element-by-element. Guess what the result of each of the following will be, then check your answers:

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

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

Exercise 3 R operates on vectors element-by-element. 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
```

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

Exercise 4 R recycles elements of the shorter of two vectors when there's a length mismatch. 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
```

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, the vector of the type that's *less flexible* will be ***coerced*** to the type of the one that's *more flexible*. Types from least to most flexible are:

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

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

```
x <- c(4, 7, 5)
y <- c("a", "b")
c(x, y)

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

- R coerces `TRUE` to 1 and `FALSE` to 0 when we combine a `"logical"` vector with a numeric (`"double"` or `"integer"`) one:

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

## [1] 1 0 7 8
```

- In addition, R coerces `TRUE` and `FALSE` to 1 and 0 in arithmetic operations:

```
TRUE + FALSE + TRUE                             # Gets coerced to 1 + 0 + 1

## [1] 2
```

Section 2.3 Exercises

Exercise 5 R coerces to the more flexible type when combining vectors of different types. 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)
```

Exercise 6 R coerces `TRUE` and `FALSE` to 1 and 0 in arithmetic operations. Guess what the result of each of the following will be, then check your answers:

a)
```
TRUE + TRUE + FALSE + TRUE
```

b)
```
TRUE * TRUE
```

c)
```
TRUE * FALSE
```

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*** (position numbers) 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
```

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

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

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

Rearranging Vector Elements

- One way to rearrange (permute) the elements of a vector is to specify the desired rearrangement (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`. Below, only the elements of `x` corresponding to `TRUE` in the `"logical"` vector are returned:

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

## [1] 5 9 8
```

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:

```r
1:10
```

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

produces the same result as:

```r
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`):

```r
rep(1, times = 10)
```

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

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

```r
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`:

```r
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:

```r
rep(c(5, 6, 7), times = c(1, 2, 3))
```

```
## [1] 5 6 6 7 7 7
```

Above, the 5 was repeated once because the first element of `times` is 1, the 6 was repeated twice because the second element of `times` is 2, and the 7 was repeated three times because the third element of `times` is 3.

Section 2.4 Exercises

Exercise 7 Consider the vector

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

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

a) `x[2]`

b) `x[-2]`

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

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

e)
```r
x[c(2, 1, 3, 4, 5)]
```

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

g)
```r
x[c(1, 2)] <- c(40, 50)
x
```

Exercise 8 Consider the vector

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

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 19.
c) Write a command that returns all but the 4th element of `x`.

Exercise 9 Consider the vector

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

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

```
## [1] 17 22 13 23 14
```

Exercise 10 Consider the vector

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

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

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

Exercise 11 Guess what the result of each of the following will be, then check your answers:

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

b)
```r
1:5
```

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

Exercise 12 Consider the vector

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

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

```r
sort(x)
```

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

```r
rev(x)
```

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

```r
rev(sort(x))
```

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

```r
? sort
```

Notice the `sort()` has an optional argument, `decreasing`, whose default value is `FALSE`. Guess what the result of the following command will be, then check your answer:

```r
sort(x, decreasing = TRUE)
```

Exercise 13 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 = 3)`

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

2.5 Computing Summary Statistics

- Several functions take vector arguments and compute summary statistics, among them:

```
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
cumsum()           # Cumulative sums of the values in a vector
cumprod()          # Cumulative products 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)
```

- For a vector `x` containing the values $x_1, x_2, \ldots, x_n$, `mean(x)` computes the ***sample mean***, $\bar{x}$:

$$\bar{x} \;=\; \frac{1}{n}\sum_{i=1}^{n} x_i$$

- `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***, 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:

$$s \;=\; \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}.$$

Section 2.5 Exercises

Exercise 14 Consider the following data set:

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

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

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

c) Type

```
sort(x)
```

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

d) Typing:

```
y <- x + 10
```

adds 10 to each element of `x`, storing the results in `y`.

How would adding 10 to each value of `x` affect the mean? What about the median? Check your answers.

Exercise 15

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()`.

```
u <- c(5, 6, 7)
v <- c(1, 6, 11)
```

Exercise 16

a) Use `sd()` to compute the standard deviation of `x` from Exercise 14.

b) How would adding 10 to each value of `x` affect the standard deviation? Try it.

2.6 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 2.6 Exercises

Exercise 17 The function `abs()` takes the absolute value of a number, and 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 18 The arithmetic operators '+', '-', '*', '/', and '^' are all *vectorized*. 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:

$$°\mathrm{F} \;=\; \frac{9}{5}\,°\mathrm{C} + 32$$

Guess what the following command does, then check your answer:

```
degreesF <- 9/5 * degreesC + 32
```

Exercise 19

a) The area of a circle is related to its radius by:

$$\text{Area} = \pi \times \text{Radius}^2$$

where $\pi = 3.141593$, which is represented by the constant `pi` in R:

```
pi

## [1] 3.141593
```

The following values represent radii of eight circles:

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

Write a command using the *vectorized* property of the arithmetic operators '`*`' and '`^`' to create a new vector named `areas` containing the areas of the eight circles.

b) The radius of a circle is related to its area by:

$$\text{Radius} = \sqrt{\text{Area}/\pi}$$

The following values represent areas of five circles:

```
areas <- c(9, 11, 15, 13, 10)
```

Write a command using the *vectorized* property of the square root function `sqrt()` and the operator '`/`' that creates a new vector named `radii` containing the radii of the five circles.

2.7 Comparison Operators

- R has several ***comparison operators*** that can be used on (scalar) variables as well as on vectors:

```
<                   # Less than
>                   # Greater than
==                  # Equal to
!=                  # Not equal to
<=                  # Less than or equal to
>=                  # Greater than or equal to
```

- Each one returns a `"logical"` value, `TRUE` or `FALSE`, depending on whether or not the relationship holds. Here are some examples:

```
5 < 6

## [1] TRUE

5 == 6

## [1] FALSE
```

```
5 != 6

## [1] TRUE
```

- They can be used on `"character"` values too:

```
"a" == "b"

## [1] FALSE
```

- When applied to vectors, they operate elementwise and return a `"logical"` vector:

```
x <- c(11, 3, 4, 12, 2)
y <- c(11, 9, 4, 12, 2)
```

```
x == y

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

- They can also be used to compare each element of a vector to a single value. For example (using `x` from above):

```
x > 10

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

- Recall that R coerces `TRUE` and `FALSE` to 1 and 0, respectively, when they're encountered in arithmetic operations. This is very useful, in combination with the function `sum()`, when we want to count how many elements of a vector satisfy a given condition:

```
sum(x > 10)          # Counts how many elements of x are greater than 10.

## [1] 2
```

Section 2.7 Exercises

Exercise 20 Consider the following vector:

```
x <- c(3, 4, 5)
```

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

a) `x > 4`

b) `x >= 4`

c) `> x == 4`

d) `x != 4`

Exercise 21 Consider the following two vectors:

```
x <- c(3, 4, 5)
y <- c(3, 4, 10)
```

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

a) `x == y`

b) `x < y`

c) `x != y`

Exercise 22 Recall that R coerces `TRUE` and `FALSE` to 1 and 0, respectively, when they're encountered in arithmetic operations.

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

```
TRUE + TRUE + FALSE + TRUE
```

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

```
sum(c(TRUE, TRUE, FALSE, TRUE))
```

c) Consider the vector:

```
x <- c(10, 8, -2, 6)
```

and the `"logical"` vector:

```
x > 0

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

Guess what will be returned by the following command, then check your answer:

```
sum(x > 0)
```

Exercise 23 Consider the following two vectors:

```
x <- c(10, 8, -2, -6, -5)
y <- c(8, 8, -2, -7, -5)
```

We want a command that counts how many of the elements of `x` are equal to their corresponding element of `y`. Will the following command do what we want? Check your answer.

```
sum(x == y)
```

2.8 Using `any()`, `all()`, and `which()`

- The following are useful for searching vectors for values that satisfy a given condition:

```
any()         # Do any elements of a vector satisfy a given condition?
              # Returns TRUE or FALSE.
all()         # Do all of the elements of a vector satisfy a given
              # condition?  Returns TRUE or FALSE.
which()       # Returns the indices of the elements of a vector that
              # satisfy a given condition.
which.min()   # Returns the index of the (first) minimum value in a
              # vector.
which.max()   # Like which.min(), but returns the index of the (first)
              # maximum.
```

- `any()` tells us whether *any* values in a vector satisfy a certain condition:

```
x <- c(11, 3, 4, 12, 2)
any(x > 10)

## [1] TRUE
```

- `all()` tells us whether or not *all* of the values in a vector satisfy a condition:

```
all(x > 10)

## [1] FALSE
```

- `which()` determines *which* elements satisfy the condition, and returns their *indices*:

```
which(x > 10)

## [1] 1 4
```

  Thus we see that the 1st and 4th elements of `x` are greater than 10.

- We can use `any()`, `all()`, and `which()` for comparing *two* vectors. For example, consider the vectors:

```
x <- c(11, 3, 4, 12, 2)
y <- c(11, 9, 4, 12, 2)
```

  To find out which of the values in `x` are different from their corresponding value in `y`, we type:

```
which(x != y)

## [1] 2
```

  We see that only the 2nd values of `x` and `y` differ.

- `which.min()` and `which.max()` return the *indices* of the (first occurrences) of the smallest and largest values in a vector. For example:

```
which.max(x)

## [1] 4
```

tells us that the largest value in `x` is the 4th value (12).

- Be aware that if two or more elements are tied as the smallest (or largest) value, `which.min()` (or `which.max()`) only returns the index of the *first* one:

```
x <- c(2, 4, 10, 3, 1, 10)
which.max(x)

## [1] 3
```

Section 2.8 Exercises

Exercise 24 Consider the vector:

```
x <- c(3, 6, 2, 8, 5)
```

Guess what will be returned by of each of the following commands, then check your answers:

a) `any(x == 6)`

b) `all(x == 6)`

c) `which(x == 6)`

d) `which(x != 6)`

Exercise 25 Consider the vector:

```
x <- c(3, 6, 2, 8, 5)
```

a) Write a command that uses `any()` to determine if *any* of the values in `x` are greater than 5.

b) Write a command that uses `all()` to determine if *all* of the values in `x` are greater than 5.

c) Write a command that uses `which()` to determine *which* of the values in `x` are greater than 5.

Exercise 26 Consider the vectors:

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

Guess what will be returned by of each of the following commands, then check your answers:

a) `any(x == y)`

b) `all(x == y)`

c) `which(x == y)`

Exercise 27 Consider the vector:

```
x <- c(3, 6, 2, 8, 5)
```

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

a) `which.min(x)`

b) `which.max(x)`

c) `x[which.min(x)]`

2.9 Filtering

2.9.1 Filtering Using Square Brackets `[ ]`

- ***Filtering*** refers to extracting from a vector those elements for which some condition is met.
- To extract the elements that satisfy some condition, we specify the condition inside square brackets `[ ]`. For example, here's a vector `x`:

  ```
  x <- c(1, 3, 12, 5, 13)
  ```

  To extract the elements of `x` that are greater than 10, type:

  ```
  x[x > 10]                    # Note that x > 10 is a "logical" vector
  ```

  ```
  ## [1] 12 13
  ```

  Note that the expression `x > 10` is actually a `"logical"` vector.

2.9.2 Replacing Elements that Satisfy Some Condition

- We can also use square brackets to *replace* elements that satisfy some condition. For example, using the vector `x` from above, if we want to replace all of the values in `x` that are greater than 10 by, say, 11, we type:

  ```
  x[x > 10] <- 11              # Note that x > 10 is a "logical" vector
  x

  ## [1]  1  3 11  5 11
  ```

2.9.3 Using Values in One Vector to Determine Which Elements Of Another Vector Should Be Extracted

- Sometimes we need to extract from one vector the elements for which the values in another vector satisfy some condition. For example, suppose we have heights (inches) and weights (lbs) of 12 people:

```
ht <- c(69, 71, 67, 66, 72, 71, 61, 65, 73, 70, 68, 74)
wt <- c(175, 170, 210, 190, 195, 165, 163, 172, 158, 191, 213, 215)
```

  To find the weights of the people who are 72 inches tall or taller, we type:

```
wt[ht >= 72]                # Note that ht >= 72 is a "logical" vector

## [1] 195 158 215
```

- This method is useful when we have a `"character"` vector indicating group membership, and we want to extract from another vector the individuals belonging to a particular group. For example, consider this data set:

Gender	Age
f	33
m	35
f	29
m	34
m	37
f	36
f	35
f	40
m	43
f	38
f	40
m	44

  After creating the vectors:

```
Gender <- c("f", "m", "f", "m", "m", "f", "f", "f", "m", "f", "f", "m")
Age <- c(33, 35, 29, 34, 37, 36, 35, 40, 43, 38, 40, 44)
```

  we can extract the ages of just the females by typing:

```
Age[Gender == "f"]          # Note that Gender == "f" is a "logical" vector

## [1] 33 29 36 35 40 38 40
```

2.9.4 Filtering Using `subset()`

- Another way to extract from a vector the elements that satisfy some condition is to use the function:

```
subset()          # Extract a subset of vector elements that satisfy a
                  # given condition
```

- `subset()` takes arguments `x`, a vector, and `subset`, a `"logical"` vector specifying the condition to be met by the elements extracted from `x`. For example, using the `Age` and `Gender` vectors created above, to (again) extract the `Age`s corresponding to females, we can type:

```
subset(x = Age, subset = Gender == "f")

## [1] 33 29 36 35 40 38 40
```

Section 2.9 Exercises

Exercise 28 Consider the vector:

```
x <- c(53, 42, 68, 71, 84, 64, 95)
```

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

```
x[x > 70]
```

b) Guess what values will be contained in `x` after executing the following commands, then check your answer:

```
x[x > 70] <- 0
x
```

Exercise 29 Consider this data set, showing the genders, ages, and systolic blood pressures for 12 people:

Gender	Age	Blood Pressure
f	33	118
m	35	115
f	29	110
m	34	117
m	37	112
f	36	119
f	35	114
f	40	121
m	43	123
f	38	117
f	40	120
m	44	121

Here are the same data as three vectors:

```
Gender <- c("f", "m", "f", "m", "m", "f", "f", "f", "m", "f", "f", "m")
Age <- c(33, 35, 29, 34, 37, 36, 35, 40, 43, 38, 40, 44)
BP <- c(118, 115, 110, 117, 112, 119, 114, 121, 123, 117, 120, 121)
```

a) In words, what does the following command do?

```
BP[Gender == "m"]
```

b) In words, what does the following command do?

```
mean(BP[Gender == "m"])
```

c) In words, what does the following command do?

```
Age[BP > 117]
```

2.10 Vector Element Names

- We can assign names to vector elements (or get the names if they already exist) using:

```
names()          # Get or set the names of a vector (or other object)
```

- For example, consider again the vector of peoples' heights (in inches):

```
ht <- c(69, 71, 66, 67, 63, 70, 64, 68)
```

To assign the name of the person to each height, we type:

```
names(ht) <- c("Joe", "Jon", "Ann", "Tom", "Kim", "Abe", "Mia", "Ken")
```

Now we get:

```
ht

## Joe Jon Ann Tom Kim Abe Mia Ken
##  69  71  66  67  63  70  64  68
```

Note that `names()` serves as a so-called ***replacement function*** when it's used on the left side of the assignment operator `<-` as above.

- To look at the (existing) names of a vector's elements, use `names()` as follows:

```
names(ht)

## [1] "Joe" "Jon" "Ann" "Tom" "Kim" "Abe" "Mia" "Ken"
```

Section 2.10 Exercises

Exercise 30 Consider the following vector:

```
x <- c(2, 4, 6)
names(x) <- c("a", "b", "c")
x

## a b c
## 2 4 6
```

If a vector's elements are named, we can use the names as indices in square brackets `[ ]`. Guess what the result of each of the following will be, then check your answers.

a) `x["c"]`

b) `x[c("a", "b")]`

2.11 `NA` Values

2.11.1 Introduction

- Data sets often contain *missing values*, for example when a survey question was left unanswered or a laboratory measurement failed due to equipment malfunction.
- In R, missing values are represented by `NA`, which stands for ”not available”:

```
NA                  # Indicates a missing value ("not available")
```

- We can test for a missing value using `is.na()`:

```
is.na()             # Returns TRUE or FALSE depending on whether or
                    # not a value is NA
```

- When applied to a vector, `is.na()` returns a `"logical"` vector whose elements are `TRUE` or `FALSE` depending on whether the corresponding value in the original vector is `NA`:

```
x <- c(2.1, 4.1, NA, 4.4, 3.7)
is.na(x)

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

- To decide *which* values in a vector are missing, we can type:

```
which(is.na(x))

## [1] 3
```

2.11.2 Computing Summary Statistics from Data with `NA`s

- Most of the R functions for computing summary statistics return `NA` when passed a data set that contains `NA`s:

```
x <- c(2.1, 4.1, NA, 4.4, 3.7)
mean(x)

## [1] NA
```

- Some of them have an optional argument `na.rm`, though, that can be used to remove the `NA`s before carrying out the calculations:

```
mean(x, na.rm = TRUE)

## [1] 3.575
```

2.11.3 Removing or Replacing `NA`s

- To obtain the non-`NA` values from a vector, we make use of an operator, `!`, that stands for ”not”:

```
# The ! operator stands for "not", and !is.na(x) is a "logical" vector:
x[!is.na(x)]

## [1] 2.1 4.1 4.4 3.7
```

- To replace the `NA`s in a vector by some value, like 0, we use:

```
x[is.na(x)] <- 0
x

## [1] 2.1 4.1 0.0 4.4 3.7
```

- Another way to remove `NA`s from a vector is to use the function:

```
na.omit()              # Removes NAs from a vector
```

- For example:

```
x <- c(2.1, 4.1, NA, 4.4, 3.7)
as.vector(na.omit(x))

## [1] 2.1 4.1 4.4 3.7
```

  Above, `as.vector()` was used to convert the object that's returned by `na.omit()` to a vector.

Section 2.11 Exercises

Exercise 31 Consider the vector:

```
x <- c(1, 2, NA)
```

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

a) `is.na(x)`

b) `mean(x)`

c) `mean(x, na.rm = TRUE)`

d) `x[!is.na(x)]`

e)
```
x[is.na(x)] <- 0
x
```

f) `as.vector(na.omit(x))`