UNIT I

INTRODUCTION TO R PROGRAMMING

INTRODUCTION

- R is a programming language and software environment for statistical analysis, graphics representation and reporting.

- The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions.

- R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

Features of R

The following are the important features of R:

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.

- R has an effective data handling and storage facility,

- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.

- R provides a large, coherent and integrated collection of tools for data analysis.

- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

R – Data Types

- In contrast to other programming languages like C and java in R, the variables are not declared as some data type.
- The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable.
- There are many types of R-objects. The frequently used ones are:

✓ Vectors

✓ Lists

✓ Matrices

✓ Arrays

✓ Factors

✓ Data Frames

- The simplest of these objects is the vector object and there are six data types of these atomic vectors, also termed as six classes of vectors.
- The other R-Objects are built upon the atomic vectors.

R Data Structures

Vectors, the R Workhorse

- The vector type is really the heart of R.
- It’s hard to imagine R code, or even an interactive R session, that doesn’t involve vectors.
- The elements of a vector must all have the same mode, or data type.
- You can have a vector consisting of three character strings (of mode character) or three integer elements (of mode integer), but not a vector with one integer element and two character string elements.

Vectors

When you want to create vector with more than one element, you should use **c()** function which means to combine the elements into a vector.

```
# Create a vector.
apple <- c('red','green',"yellow")
print(apple)


# Get the class of the vector.
print(class(apple))
```

When we execute the above code, it produces the following result:

```
[1] "red"    "green"  "yellow"
[1] "character"
```

1. Scalars

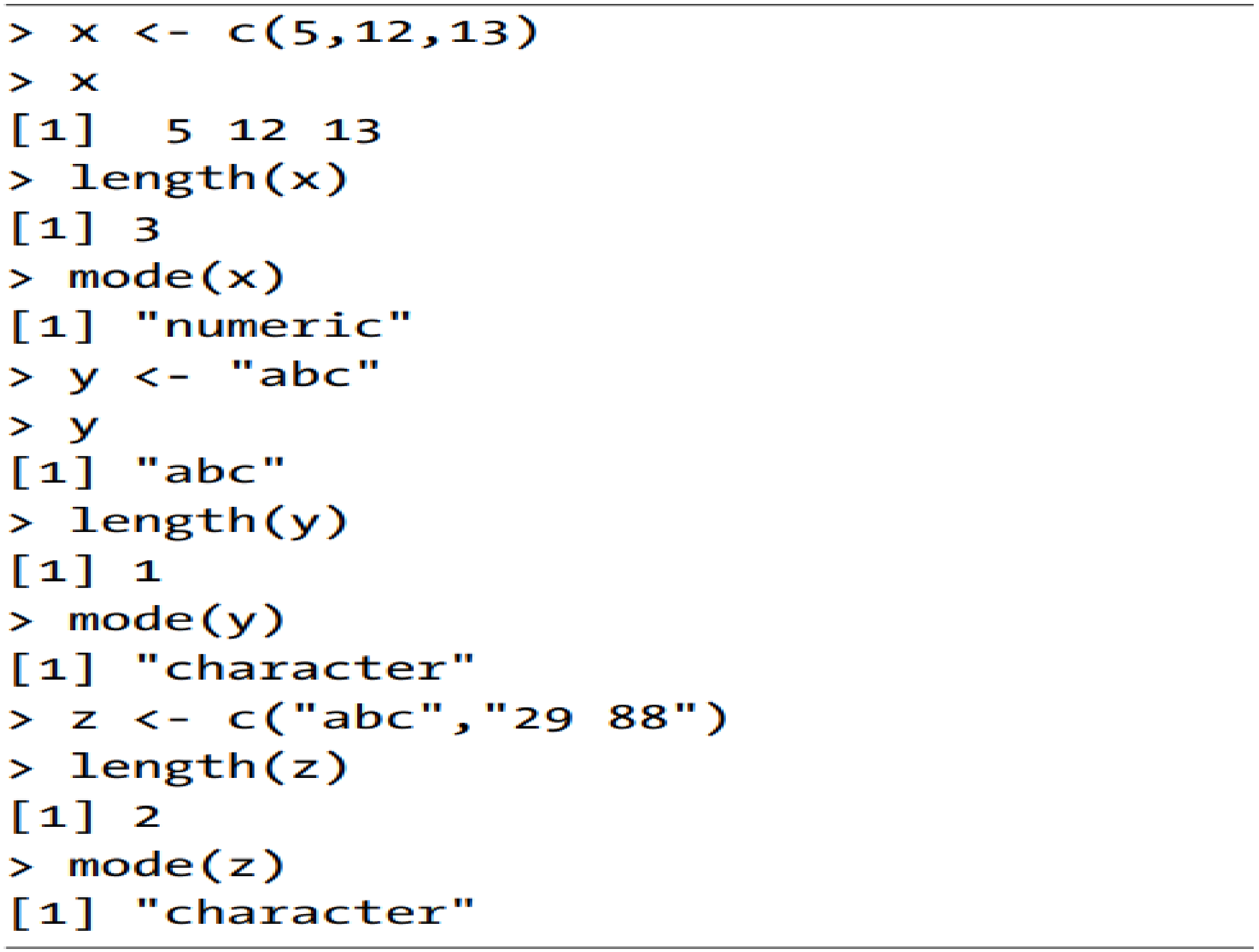
Scalars, or individual numbers, do not really exist in R. As mentioned earlier, what appear to be individual numbers are actually one-element vectors. Consider the following:

```
> x <- 8
> x
[1] 8
```

Recall that the [1] here signifies that the following row of numbers begins with element 1 of a vector—in this case, x[1]. So you can see that R was indeed treating x as a vector, albeit a vector with just one element.

2. Character Strings

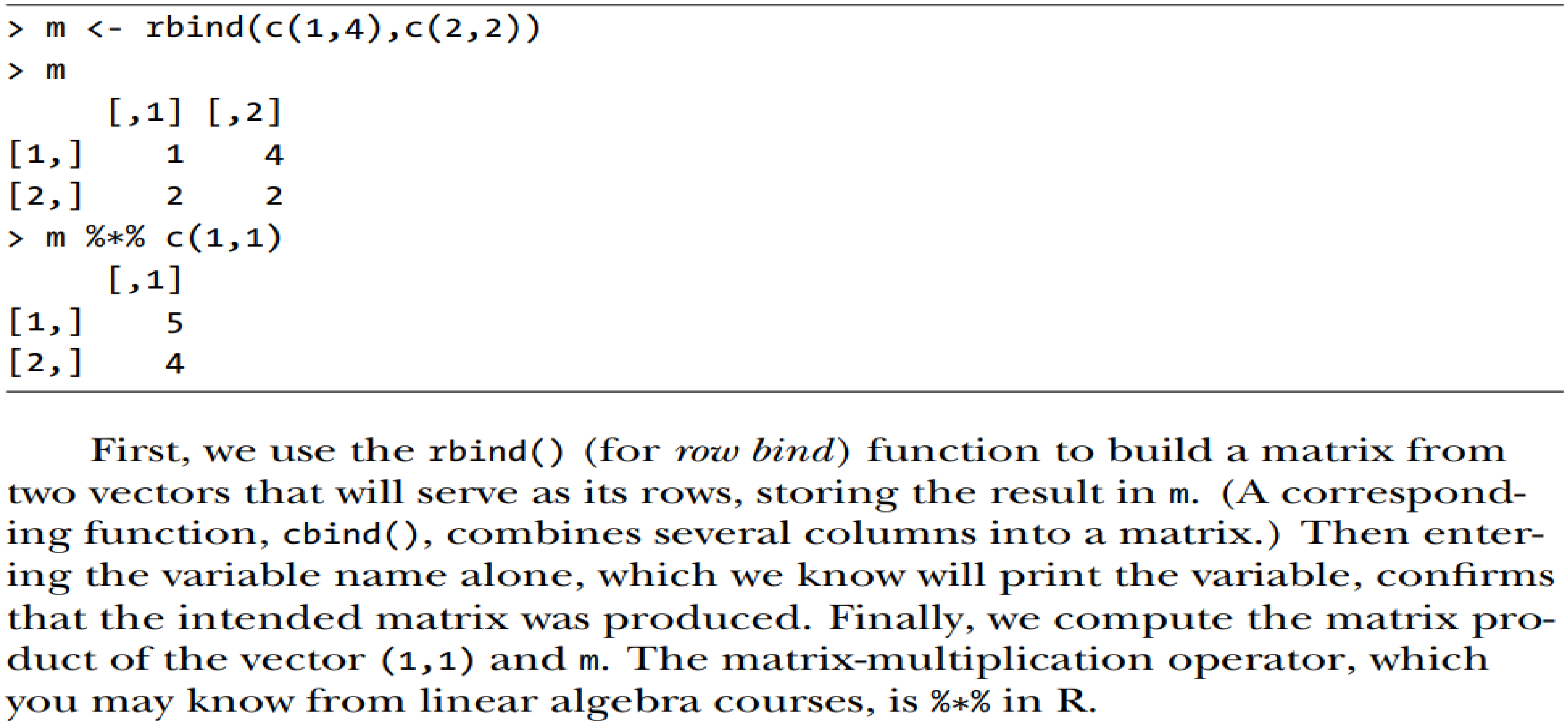
- Character strings are actually single-element vectors of mode character, (rather than mode numeric)



```
> x <- c(5,12,13)
> x
[1]  5 12 13
> length(x)
[1] 3
> mode(x)
[1] "numeric"
> y <- "abc"
> y
[1] "abc"
> length(y)
[1] 1
> mode(y)
[1] "character"
> z <- c("abc","29 88")
> length(z)
[1] 2
> mode(z)
[1] "character"
```

3. Matrices

- An R matrix corresponds to the mathematical concept of the same name: a rectangular array of numbers. Technically, a matrix is a vector, but with two additional attributes: the number of rows and the number of columns. Here is some sample matrix code:



```
> m <- rbind(c(1,4),c(2,2))
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    2
> m %*% c(1,1)
     [,1]
[1,]    5
[2,]    4
```

First, we use the `rbind()` (for *row bind*) function to build a matrix from two vectors that will serve as its rows, storing the result in `m`. (A corresponding function, `cbind()`, combines several columns into a matrix.) Then entering the variable name alone, which we know will print the variable, confirms that the intended matrix was produced. Finally, we compute the matrix product of the vector `(1,1)` and `m`. The matrix-multiplication operator, which you may know from linear algebra courses, is `%*%` in R.

4. Lists

- Like an R vector, an R list is a container for values, but its contents can be items of different data types.
- List elements are accessed using two-part names, which are indicated with the dollar sign $ in R.

```
> x <- list(u=2, v="abc")
> x
$u
[1] 2
$v
[1] "abc"
> x$u
[1] 2
```

The expression x$u refers to the u component in the list x. The latter contains one other component, denoted by v.

Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
# Create a list.
list1 <- list(c(2,5,3),21.3,sin)


# Print the list.
print(list1)
```

When we execute the above code, it produces the following result:

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


[[2]]
[1] 21.3


[[3]]
function (x)  .Primitive("sin")
```

5. Data Frames

- A data frame in R is a list, with each component of the list being a vector corresponding to a column in our “matrix” of data.
- Indeed, you can create data frames in just this way:

```
> d <- data.frame(list(kids=c("Jack","Jill"),ages=c(12,10)))
> d
   kids  ages
1  Jack   12
2  Jill    10
> d$ages
[1]  12  10
```

6. Classes

- R is an object-oriented language. Objects are instances of classes.
- Classes are a bit more abstract than the data types you’ve met so far.
- Most of R is based on these classes, and they are exceedingly simple.
- Their instances are simply R lists but with an extra attribute: the class name.

VECTORS

Adding and Deleting Vector Elements

- Vectors are stored like arrays in C, contiguously, and thus you cannot insert or delete elements—something you may be used to if you are a Python programmer.
- The size of a vector is determined at its creation, so if you wish to add or delete elements, you'll need to reassign the vector. For example, let's add an element to the middle of a four-element vector:

```
> x <- c(88,5,12,13)
> x <- c(x[1:3],168,x[4])  # insert 168 before the 13
> x
[1]  88   5  12 168  13
```

Obtaining the Length of a Vector

You can obtain the length of a vector by using the `length()` function:

```
> x <- c(1,2,4)
> length(x)
[1] 3
```

Matrices and Arrays as Vectors

```
> m
  [,1] [,2]
[1,]    1    2
[2,]    3    4
> m + 10:13
  [,1] [,2]
[1,]   11   14
[2,]   14   17
```

The 2-by-2 matrix `m` is stored as a four-element vector, column-wise, as (1,3,2,4). We then added (10,11,12,13) to it, yielding (11,14,14,17), but R remembered that we were working with matrices and thus gave the 2-by-2 result you see in the example.

- **Declarations**

➢ Recycling

When applying an operation to two vectors that requires them to be the same length, R automatically *recycles*, or repeats, the shorter one, until it is long enough to match the longer one. Here is an example:

```
> c(1,2,4) + c(6,0,9,20,22)
[1]  7  2 13 21 24
Warning message:
longer object length
  is not a multiple of shorter object length in: c(1, 2, 4) + c(6,
  0, 9, 20, 22)
```

The shorter vector was recycled, so the operation was taken to be as follows:

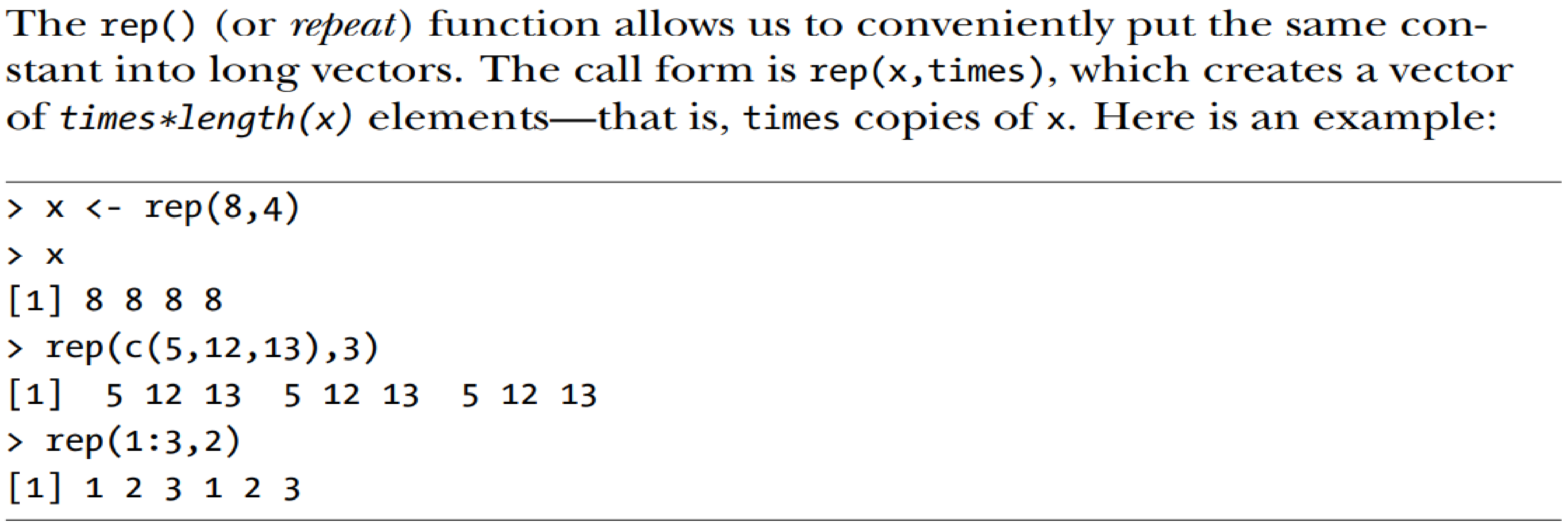
```
> c(1,2,4,1,2) + c(6,0,9,20,22)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> x+c(1,2)
     [,1] [,2]
[1,]    2    6
[2,]    4    6
[3,]    4    8
```

Again, keep in mind that matrices are actually long vectors. Here, `x`, as a 3-by-2 matrix, is also a six-element vector, which in R is stored column by column. In other words, in terms of storage, `x` is the same as `c(1,2,3,4,5,6)`. We added a two-element vector to this six-element one, so our added vector needed to be repeated twice to make six elements. In other words, we were essentially doing this:

```
x + c(1,2,1,2,1,2)
```

Common Vector Operations

- Vector Arithmetic and Logical Operations
- Generating Useful Vectors with the : Operator
- Generating Vector Sequences with seq()
- Repeating Vector Constants with rep()



The `rep()` (or *repeat*) function allows us to conveniently put the same constant into long vectors. The call form is `rep(x,times)`, which creates a vector of *times*length(x)* elements—that is, `times` copies of `x`. Here is an example:

```
> x <- rep(8,4)
> x
[1] 8 8 8 8
> rep(c(5,12,13),3)
[1]  5 12 13  5 12 13  5 12 13
> rep(1:3,2)
[1] 1 2 3 1 2 3
```

Using all() and any()

The any() and all() functions are handy shortcuts. They report whether any or all of their arguments are TRUE.

```
> x <- 1:10
> any(x > 8)
[1] TRUE
> any(x > 88)
[1] FALSE
> all(x > 88)
[1] FALSE
> all(x > 0)
[1] TRUE
```

Vectorized Operations

- Suppose we have a function f() that we wish to apply to all elements of a vector x. In many cases, we can accomplish this by simply calling f() on x itself.

- This can really simplify our code and, moreover, give us a dramatic performance increase of hundredsfold or more.

- One of the most effective ways to achieve speed in R code is to use operations that are vectorized, meaning that a function applied to a vector is actually applied individually to each element.

➢ Vector In, Vector Out

```
> u <- c(5,2,8)
> v <- c(1,3,9)
> u > v
[1]  TRUE FALSE FALSE
```

Here, the > function was applied to u[1] and v[1], resulting in TRUE, then to u[2] and v[2], resulting in FALSE, and so on.

```
> sqrt(1:9)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
[9] 3.000000
```

```
> y <- c(1.2,3.9,0.4)
> z <- round(y)
> z
[1] 1 4 0
```

➢ Vector In, Matrix Out

The vectorized functions we've been working with so far have scalar return values. Calling sqrt() on a number gives us a number. If we apply this function to an eight-element vector, we get eight numbers, thus another eight-element vector, as output.

But what if our function itself is vector-valued, as z12() is here:

```
z12 <- function(z) return(c(z,z^2))
```

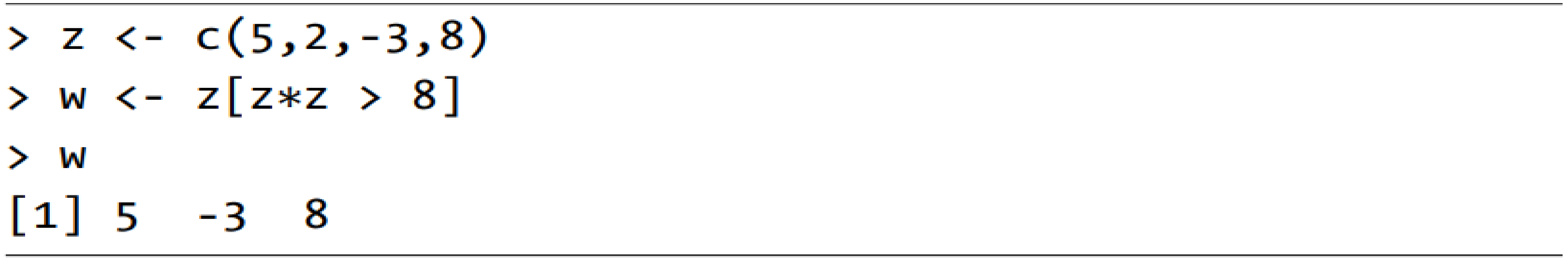
Applying z12() to 5, say, gives us the two-element vector (5,25). If we apply this function to an eight-element vector, it produces 16 numbers:

```
x <- 1:8
> z12(x)
 [1]  1  2  3  4  5  6  7  8  1  4  9 16 25 36 49 64
```

Filtering

- Another feature reflecting the functional language nature of R is filtering. This allows us to extract a vector’s elements that satisfy certain conditions.
- Filtering is one of the most common operations in R, as statistical analyses often focus on data that satisfies conditions of interest.

Generating Filtering Indices



```
> z <- c(5,2,-3,8)
> w <- z[z*z > 8]
> w
[1] 5  -3  8
```

```
> z <- c(5,2,-3,8)
> z
[1]  5  2 -3  8
> z*z > 8
[1]  TRUE FALSE  TRUE  TRUE
```

Evaluation of the expression `z*z > 8` gives us a vector of Boolean values! It's very important that you understand exactly how this comes about.

First, in the expression `z*z > 8`, note that *everything* is a vector or vector operator:

- Since `z` is a vector, that means `z*z` will also be a vector (of the same length as `z`).
- Due to recycling, the number 8 (or vector of length 1) becomes the vector (8,8,8,8) here.
- The operator `>`, like `+`, is actually a function.

➢ Filtering with the subset() Function

Filtering can also be done with the `subset()` function. When applied to vectors, the difference between using this function and ordinary filtering lies in the manner in which NA values are handled.

```
> x <- c(6,1:3,NA,12)
> x
[1]  6  1  2  3 NA 12
> x[x > 5]
[1]  6 NA 12
> subset(x,x > 5)
[1]  6 12
```

When we did ordinary filtering in the previous section, R basically said, "Well, `x[5]` is unknown, so it's also unknown whether its square is greater than 5." But you may not want NAs in your results. When you wish to exclude NA values, using `subset()` saves you the trouble of removing the NA values yourself.

➢ The Selection Function which()

As you've seen, filtering consists of extracting elements of a vector z that satisfy a certain condition. In some cases, though, we may just want to find the positions within z at which the condition occurs. We can do this using which(), as follows:

```
> z <- c(5,2,-3,8)
> which(z*z > 8)
[1] 1 3 4
```

The result says that elements 1, 3, and 4 of z have squares greater than 8.

As with filtering, it is important to understand exactly what occurred in the preceding code. The expression

```
z*z > 8
```

is evaluated to (TRUE,FALSE,TRUE,TRUE). The which() function then simply reports which elements of the latter expression are TRUE.

A Vectorized if-then-else: The ifelse() Function

In addition to the usual if-then-else construct found in most languages, R also includes a vectorized version, the ifelse() function. The form is as follows:

```
ifelse(b,u,v)
```

where b is a Boolean vector, and u and v are vectors.

The return value is itself a vector; element i is u[i] if b[i] is true, or v[i] if b[i] is false. The concept is pretty abstract, so let's go right to an example:

```
> x <- 1:10
> y <- ifelse(x %% 2 == 0,5,12)  # %% is the mod operator
> y
 [1] 12  5 12  5 12  5 12  5 12  5
```

Here, we wish to produce a vector in which there is a 5 wherever x is even or a 12 wherever x is odd. So, the actual argument corresponding to the formal argument b is (F,T,F,T,F,T,F,T,F,T). The second actual argument, 5, corresponding to u, is treated as (5,5,...) (ten 5s) by recycling. The third argument, 12, is also recycled, to (12,12,...).

Here is another example:

```
> x <- c(5,2,9,12)
> ifelse(x > 6,2*x,3*x)
[1] 15  6 18 24
```

We return a vector consisting of the elements of `x`, either multiplied by 2 or 3, depending on whether the element is greater than 6.

Again, it helps to think through what is really occurring here. The expression `x > 6` is a vector of Booleans. If the i^{th} component is true, then the i^{th} element of the return value will be set to the i^{th} element of `2*x`; otherwise, it will be set to `3*x[i]`, and so on.

The advantage of `ifelse()` over the standard if-then-else construct is that it is vectorized, thus potentially much faster.

Testing Vector Equality

Suppose we wish to test whether two vectors are equal. The naive approach, using ==, won't work.

```
> x <- 1:3
> y <- c(1,3,4)
> x == y
[1]  TRUE FALSE FALSE
```

In fact, == is a vectorized function. The expression x == y applies the function ==() to the elements of x and y. yielding a vector of Boolean values.

What can be done instead? One option is to work with the vectorized nature of ==, applying the function all():

```
> x <- 1:3
> y <- c(1,3,4)
> x == y
[1]  TRUE FALSE FALSE
> all(x == y)
[1] FALSE
```

Vector Element Names

We can assign or query vector element names via the names() function:

```
> x <- c(1,2,4)
> names(x)
NULL
> names(x) <- c("a","b","ab")
> names(x)
[1] "a"  "b"  "ab"
> x
 a  b ab
 1  2  4
```

We can remove the names from a vector by assigning NULL:

```
> names(x) <- NULL
> x
[1] 1 2 4
```

We can even reference elements of the vector by name:

```
> x <- c(1,2,4)
> names(x) <- c("a","b","ab")
> x["b"]
b
2
```