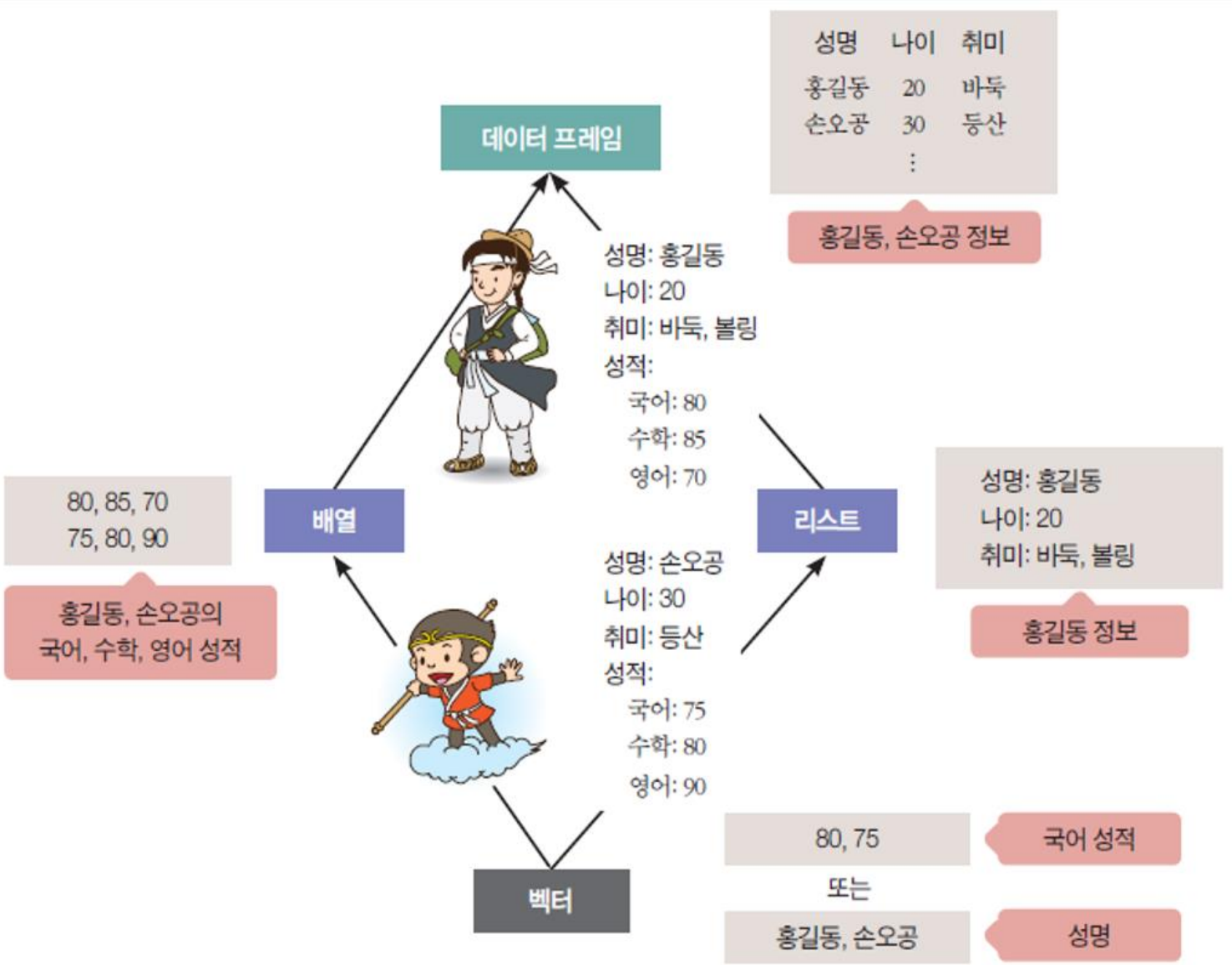
Vectors

Bok, Jong Soon
javaexpert@nate.com
www.javaexpert.info

Data Types in R



Vectors

- The fundamental data type in R.
- Scalar : single-number variables.
- In C-language family.

```
int x;
int y[3];
```

- But in R, numbers are actually considered one-element vectors.
- R variable types are called *modes*.
- All elements in a vector must have the same mode.
- It can be integer, numeric (floating-point number), character (string), logical (Boolean), complex, and so on.

```
1
2  kor <- 85
3  name <- "홍길동"
4  grade <- "B"
5
6  kor
7  name
8  grade
9
```

9:1 (Top Level)

Console C:/R Home/

```
> name <- "홍길동"
> grade <- "B"
> kor
[1] 85
> name
[1] "홍길동"
> grade
[1] "B"
>
```

Single Element Vector Creation

- Even when you write just one value in R, it becomes a vector of length 1 and belongs to one of the above vector types.

```
Console ~/
> # Atomic vector of type character.
> print("abc")
[1] "abc"
>
> # Atomic vector of type double.
> print(12.5)
[1] 12.5
>
> # Atomic vector of type integer.
> print(63L)
[1] 63
>
> # Atomic vector of type logical.
> print(TRUE)
[1] TRUE
>
> # Atomic vector of type complex.
> print(2 + 3i)
[1] 2+3i
>
> # Atomic vector of type raw.
> print(charToRaw('hello'))
[1] 68 65 6c 6c 6f
```

Multiple Elements Vector Creation

- **Using colon operator(:) with numeric data.**

Console ~/

```
> # Creating a sequence from 5 to 13.
> v <- 5:13
> print(v)
[1]  5  6  7  8  9 10 11 12 13
>
> # Creating a sequence from 6.6 to 12.6
> v <- 6.6 : 12.6
> print(v)
[1]  6.6  7.6  8.6  9.6 10.6 11.6 12.6
>
> # If the final element specified does not belong to the sequence then it is di
scarded.
> v <- 3.8 : 11.4
> print(v)
[1]  3.8  4.8  5.8  6.8  7.8  8.8  9.8 10.8
```

Multiple Elements Vector Creation (Cont.)

- Using `seq()` function.

```
Console ~/
> # Create vector with elements from 5 to 9 incrementing by 0.4.
> print(seq(5, 9, by = 0.4))
 [1] 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0
>
```

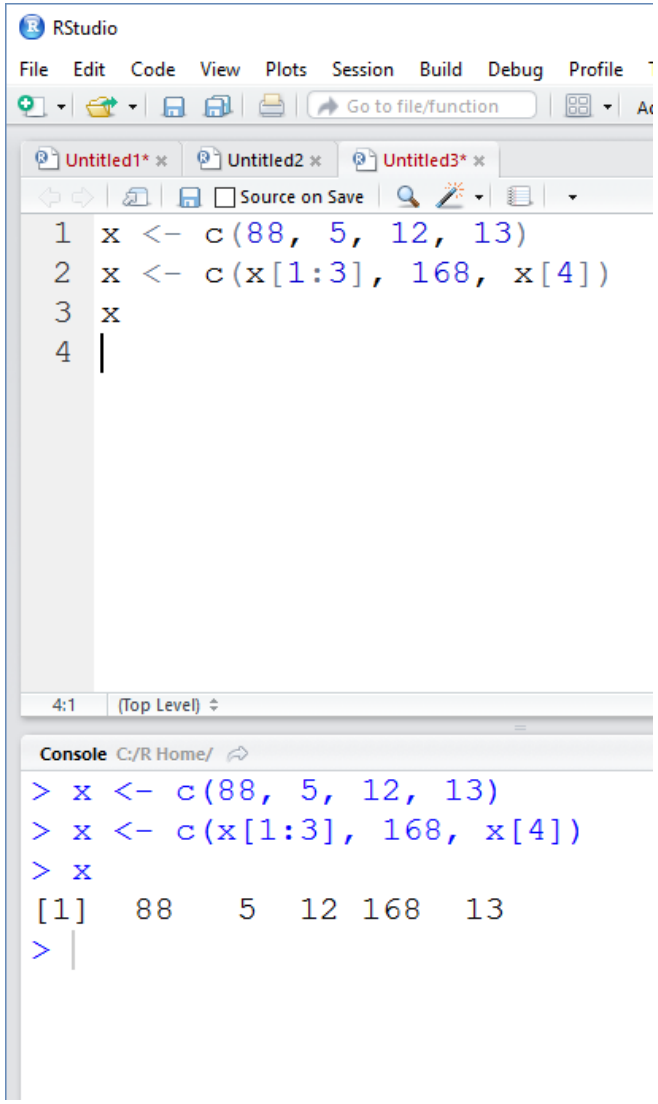
Multiple Elements Vector Creation (Cont.)

- Using the `c()` function.
- The non-character values are coerced to character type if one of the elements is a character.

```
Console ~/
> # The logical and numeric values are converted to characters.
> s <- c('apple', 'red', 5, TRUE)
> print(s)
[1] "apple" "red"   "5"     "TRUE"
>
```

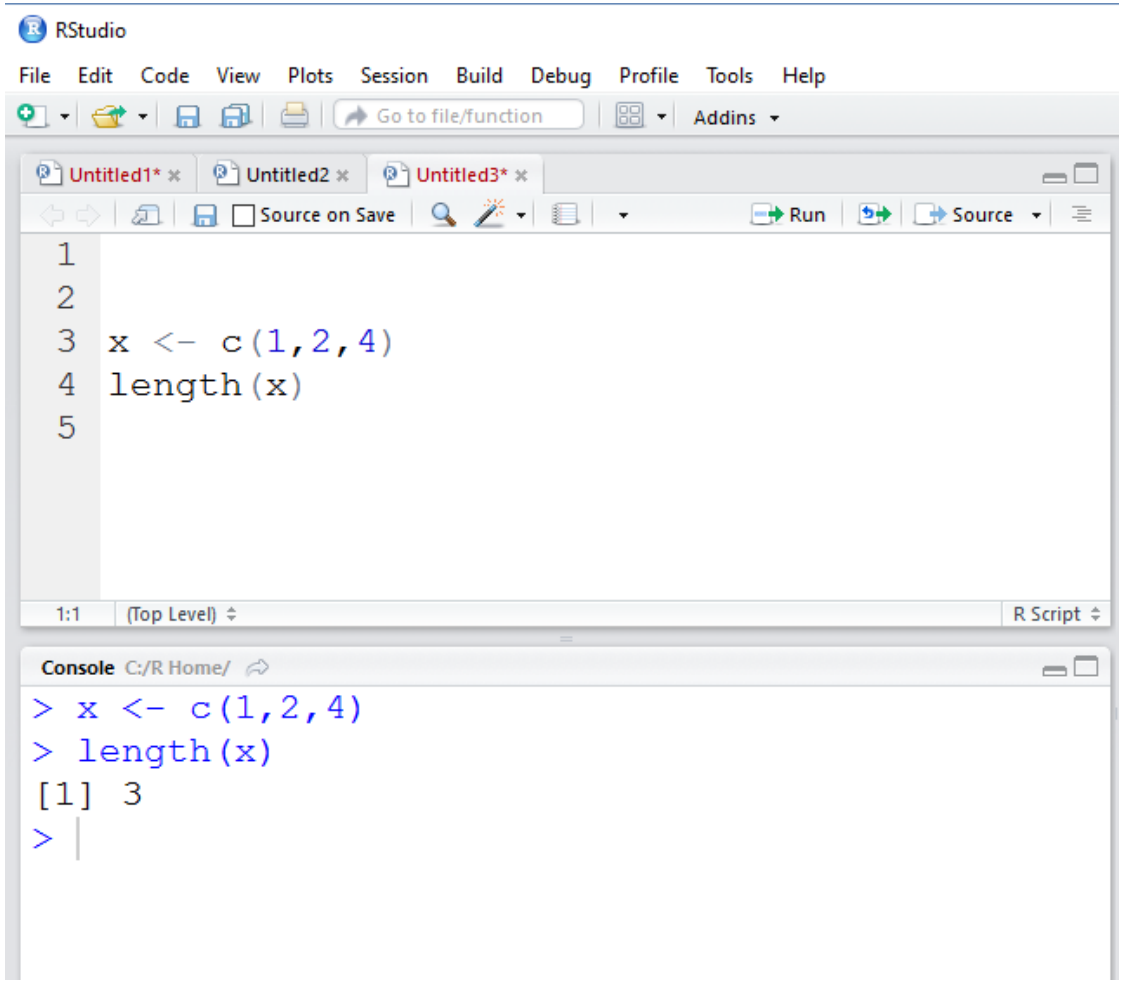
Adding and Deleting Vector Elements

- Vectors are stored like arrays in C.
- But, cannot insert or delete elements.
- The size of a vector is determined at its creation.
- Unlike vector indices in ALGOL-family languages, such as C and Python, vector indices in R begin at 1.



Obtaining the Length of a Vector

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



Matrices and Arrays as Vectors

- Arrays and are actually vectors too.
- Matrices have the number of rows and columns.

```
1  m <- matrix(c(1,2,3,4), nrow=2)
2  m
3  m + 10:13
4
```

4:1 (Top Level)

Console C:/R Home/

```
> m <- matrix(c(1,2,3,4), nrow=2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> m + 10:13
     [,1] [,2]
[1,]   11   15
[2,]   13   17
>
```

Declarations

- Unlike C, Instead array, must create `y` first, for instance this way:

```
> y <- vector(length=2)
> y[1] <- 5
> y[2] <- 12
```

- The following will also work:

```
> y <- c(5,12)
```

Declarations (Cont.)

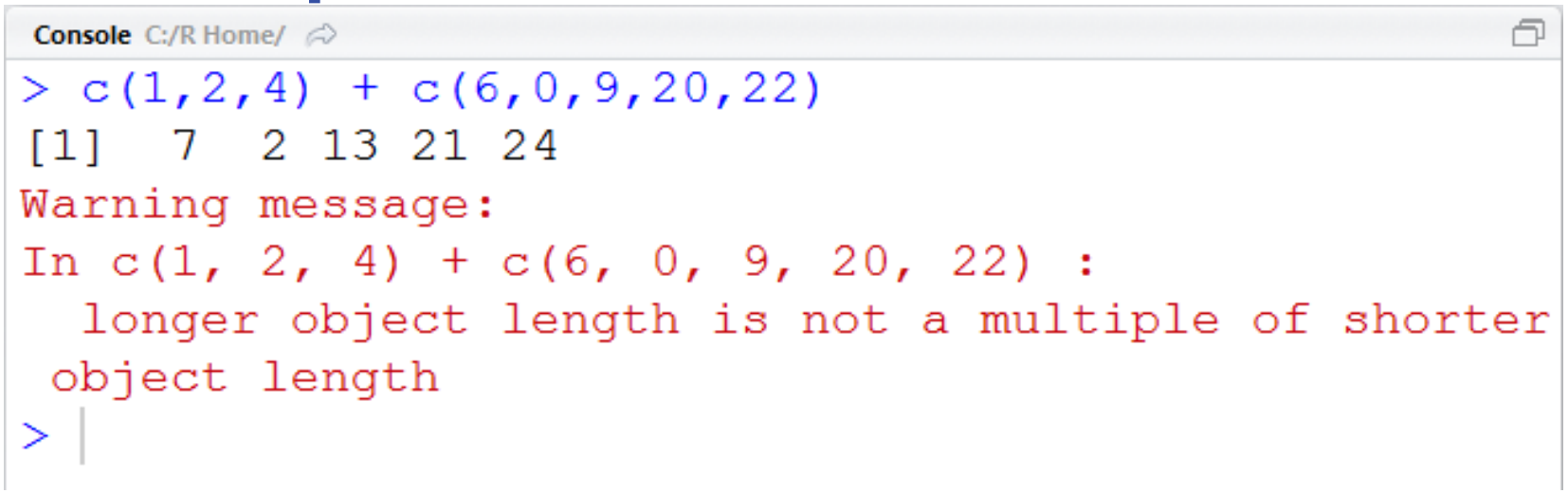
- The following sequence of events is perfectly valid.

```
> x <- c(1,5)
> x
[1] 1 5
> x <- "abc"
```

- First, `x` is associated with a numeric vector, then with a string.

Recycling

- When applying an operation to two vectors.
- 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:



Recycling (Cont.)

- 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)
```

Vector Arithmetic and Logical Operations

- R is a functional language.
- Every operator, including + is actually a function.

```
> 2+3
[1] 5
> "+"(2,3)
[1] 5
```

Vector Arithmetic and Logical Operations (Cont.)

- Can add vectors, and the + operation will be applied element-wise.

```
> x <- c(1, 2, 4)
> x + c(5, 0, -1)
[1] 6 2 3
```

- If you are familiar with linear algebra, you may be surprised at what happens when we multiply two vectors.

```
> x * c(5, 0, -1)
[1] 5 0 -4
```

Vector Arithmetic and Logical Operations (Cont.)

- The same principle applies to other numeric operators.
- Here's an example:

```
> x <- c(1,2,4)
> x / c(5,4,-1)
[1]  0.2  0.5 -4.0
> x %% c(5,4,-1)
[1] 1 2 0
```

Vector Indexing

- One of the most important and frequently used operations in R is that of indexing vectors.
- Form a subvector by picking elements of the given vector for specific indices.

```
> y <- c(1.2, 3.9, 0.4, 0.12)
> y[c(1,3)] # extract elements 1 and 3 of y
[1] 1.2 0.4
> y[2:3]
[1] 3.9 0.4
> v <- 3:4
> y[v]
[1] 0.40 0.12
```

Vector Indexing (Cont.)

- Note that duplicates are allowed.

```
> x <- c(4, 2, 17, 5)
> y <- x[c(1, 1, 3)]
> y
[1] 4 4 17
```

- Negative subscripts mean that we want to exclude the given elements in our output.

```
> z <- c(5,12,13)
> z[-1]    # exclude element 1
[1] 12 13
> z[-1:-2]  # exclude elements 1 through 2
[1] 13
```

Vector Indexing (Cont.)

- It is often useful to use the `length()` function.
- The following code will do just that:

```
> z <- c(5,12,13)
> z[1:(length(z)-1)]
[1] 5 12
```

- Or more simply:

```
> z[-length(z)]
[1] 5 12
```

- This is more general than using `z[1:2]`.

Generating Useful Vectors with the : Operator

- There are a few R operators that are especially useful for creating vectors.
- Let's start with the colon operator `:`.
- It produces a vector consisting of a range of numbers.

```
> 5:8
[1] 5 6 7 8
> 5:1
[1] 5 4 3 2 1
```

Generating Useful Vectors with the : Operator (Cont.)

- You may recall that it was used earlier in this chapter in a loop context, as follows:

```
for (i in 1:length(x)) {
```

- Beware of operator precedence issues.

```
> i <- 2
> 1:i-1 # this means (1:i) - 1, not 1:(i-1)
 [1] 0 1
> 1:(i-1)
 [1] 1
```

Generating Vector Sequences with seq()

- A generalization of : is the `seq()` (or sequence) function.
- Generates a sequence in arithmetic progression.

```
> seq(from=12,to=30,by=3)
[1] 12 15 18 21 24 27 30
```

- The spacing can be a non-integer value, too, say 0.1.

```
> seq(from=1.1,to=2,length=10)
[1] 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
2.0
```

Generating Vector Sequences with seq() (Cont.)

```
Console C:/R Home/
>
> x <- c(5, 12, 13)
> x
[1]  5 12 13
> seq(x)
[1] 1 2 3
> x <- NULL
> x
NULL
> seq(x)
integer(0)
>
```

- You can see that **`seq(x)`** gives us the same result as **`1:length(x)`** if **`x`** is not empty, but it correctly evaluates to **`NULL`** if **`x`** is empty, resulting in zero iterations in the above loop.

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)`.
- Creates a vector of `times * length(x)` elements
- That is, times copies of x.

```
> 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
```

Repeating Vector Constants with rep() (Cont.)

- There is a named argument each, with very different behavior, which interleaves the copies of **x**.

```
> rep(c(5,12,13),each=2)
[1] 5 5 12 12 13 13
```

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
```

Using all() and any() (Cont.)

- For example, suppose that R executes the following:

```
> any(x > 8)
```

- It first evaluates `x > 8`, yielding this:

```
(FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,TRUE,TRUE)
```

- The `any()` function then reports whether any of those values is `TRUE`.
- The `all()` function works similarly and reports if all of the values are `TRUE`.

Vector In, Vector Out

- You saw examples of vectorized functions earlier in the chapter, with the + and * operators.
- Another example is >.

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

Vector In, Vector Out (Cont.)

- An R function uses *vectorized* operations.
- Here is an example:

```
> w <- function(x) return(x+1)
> w(u)
[1] 6 3 9
```

- Here, `w()` uses `+`, which is *vectorized*, so `w()` is *vectorized* as well.

Vector In, Vector Out (Cont.)

```
> 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
```

- The point is that the `round()` function is applied individually to each element in the vector `y`.

Vector In, Vector Out (Cont.)

- As mentioned earlier, even operators such as + are really functions.
- For example, consider this code:

```
> y <- c(12, 5, 13)
> y + 4
[1] 16 9 17
```

- The reason element-wise addition of 4 works here is that the + is actually a function!

```
> '+'(y, 4)
[1] 16 9 17
```

Vector In, Vector Out (Cont.)

Console C:/R Home/

```
>
> f <- function(x, c){
+     return ((x + c) ^ 2)
+ }
> f
function(x, c){
    return ((x + c) ^ 2)
}
> f(1:3, 0)
[1] 1 4 9
> f(1:3, 1)
[1]  4  9 16
>
>
> f(1:3, 1:3)
[1]  4 16 36
>
```

Vector In, Matrix Out

- 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
```

Vector In, Matrix Out (Cont.)

- It might be more natural to have these arranged as an 8-by-2 matrix, which we can do with the matrix function:

```
> matrix(z12(x), ncol=2)
     [,1] [,2]
[1,]    1    1
[2,]    2    4
[3,]    3    9
[4,]    4   16
[5,]    5   25
[6,]    6   36
[7,]    7   49
[8,]    8   64
```

Using NA

- In statistical data sets, often encounter missing data, which we represent in R with the value **NA**.

```
> x <- c(88, NA, 12, 168, 13)
> x
[1] 88 NA 12 168 13
> mean(x)
[1] NA
> mean(x, na.rm=T)
[1] 70.25
> x <- c(88, NULL, 12, 168, 13)
 > # R automatically skipped over the NULL value
> mean(x)
[1] 70.25
```

Using NA (Cont.)

- There are multiple **NA** values, one for each mode:

Console C:/R Home/

```
>
> x <- c(5, NA, 12)
> mode(x[1])
[1] "numeric"
> mode(x[2])
[1] "numeric"
> y <- c("abc", "def", NA)
> mode(y[2])
[1] "character"
> mode(y[3])
[1] "character"
>
```

Using NULL

- One use of `NULL` is to build up vectors in loops.
- Each iteration adds another element to the vector.
- In this simple example, we build up a vector of even numbers:

```
# build up a vector of the even numbers in 1:10
> z <- NULL
> for (i in 1:10) if (i %% 2 == 0) z <- c(z,i)
> z
[1] 2 4 6 8 10
```

Using NULL (Cont.)

- If we were to use **NA** instead of **NULL** in the preceding example, we would pick up an unwanted **NA**:

```
> z <- NA
> for (i in 1:10) if (i %%2 == 0) z <-
c(z,i)
> z
 [1] NA 2 4 6 8 10
```

Using NULL (Cont.)

- `NULL` values really are counted as nonexistent, as you can see here:

```
> u <- NULL
> length(u)
[1] 0
> v <- NA
> length(v)
[1] 1
```

- `NULL` is a special R object with no mode.

Generating Filtering Indices

- Let's start with a simple example:

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

Generating Filtering Indices (Cont.)

- Let's look at it done piece by piece:

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

Generating Filtering Indices (Cont.)

- The operator >, like +, is actually a function.
- Let's look at an example of that last point:

```
> ">"(2, 1)
[1] TRUE
> ">"(2, 5)
[1] FALSE
```

Generating Filtering Indices (Cont.)

- Thus, the following:

```
z*z > 8
```

- is really this:

```
">"(z*z, 8)
```

- The resulting Boolean values are used to cull out the desired elements of z:

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

Generating Filtering Indices (Cont.)

- **Let's look at this code.**

```
> z <- c(5, 2, -3, 8)
> j <- z*z > 8
> j
 [1]  TRUE FALSE  TRUE
TRUE
> y <- c(1,2,30,5)
> y[j]
 [1] 1 30 5
```

- Or, more compactly, we could write the following:

```
> z <- c(5,2,-3,8)
> y <- c(1,2,30,5)
> y[z*z > 8]
 [1] 1 30 5
```

Filtering 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
```

The Selection Function which()

- 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
```

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

- 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`.

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

- 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
```

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

- Here is another example:

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

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
```

Testing Vector Equality (Cont.)

- Just like almost anything else in R, == is a function.

```
> "=="(3,2)
[1] FALSE
> i <- 2
> "=="(i,2)
[1] TRUE
```

Testing Vector Equality (Cont.)

- 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
```

Testing Vector Equality (Cont.)

- Or even better, we can simply use the identical function, like this:

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

Testing Vector Equality (Cont.)

- Consider this little R session:

```
> x <- 1:2
> y <- c(1,2)
> x
[1] 1 2
> y
[1] 1 2
> identical(x,y)
[1] FALSE
> typeof(x)
[1] "integer"
> typeof(y)
[1] "double"
```

So, `:` produces integers while `c()` produces floating-point numbers.

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
```

Vector Element Names (Cont.)

- 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
```