Vector

January 27, 2019

Adding and Deleting Vector Elements

- In R, vectors are stored like arrays in C, contiguously, and thus you cannot insert or delete elements, as you may be used to if you are a Python programmer.
- The size of a vector is determined at its creation
- To add or delete elements into a vector, you need to reassign the vector.

```
#add an element to the middle of a four-element vector
> x <- c(88,5,12,13)

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

- If you know the length of x, so there really is no need to query it.
- However in writing general function code, you often need to know the lengths of vector arguments.

Obtaining the Length of a Vector

The length of a vector might be zero

```
> x <- c()
> x
NULL

> length(x)
[1] 0
```

Declarations

- As with most scripting languages (such as Python and Perl), you do not declare variables in R

```
z <- 3
```

- The above code, with no previous reference to z, is perfectly legal
- However, if you reference specific elements of a vector, you must warn R.

Declarations

- For instance, say we wish y to be a two-component vector with values 5 and 12. The following will NOT work:

```
> y[1] <- 5
> y[2] <- 12
```

Instead, you must create y first,

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

Or

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

Declarations

- The reason we cannot suddenly use an expression like y[2], because R functional language nature.
- The reading and writing of individual vector elements are actually handled by functions.
- If R does not already know that y is a vector, these functions have nothing on which to act.

Declarations

- Just as variables are not declared, they are not constrained in terms of mode.
- The following sequence of events is perfectly

```
> 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 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.

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

- 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

- R is a functional language. Every operator, including + in the following example, is actually a function.

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

- The scalars in R are actually one-element vectors.
- So, we 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
```

Vector Arithmetic

- When we multiply two vectors, the multiplication is done element by element.

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

- The same principle applies to other numeric operators.

```
> 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
```

Vectorized functions in R

```
> 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
- The recycling played a key role here, with the 4 recycled into (4,4,4).

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.
- Format: **vector1[vector2]**

```
> 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

- 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

Pick up all elements of a vector z except for the last.

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

Or

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

Generating Useful Vectors with the : Operator

- The colon operator **:**, which produces a vector consisting of a range of numbers.

```
> 5:8
[1] 5 6 7 8

> 5:1
[1] 5 4 3 2 1
```

- In a loop context

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

Operator precedence issues

```
> i <- 2
```

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

- In the expression 1:i-1, the colon operator takes precedence over the subtraction.
- The expression 1:i is evaluated first, returning 1:2.
- R then subtracts 1 from that expression.
- That means subtracting a one-element vector from a two-element one, which is done via recycling.

```
> 1:(i-1)
[1] 1
```

- In the expression 1:(i-1), the parentheses have higher precedence than the colon.
- 1 is subtracted from i, resulting in 1:1

Generating Vector Sequences with seq()

seq() (sequence) function generates a sequence in arithmetic progression.

```
# yields the vector (3,4,5,6,7,8), with the elements spaced one
    unit apart
> 3:8
[1] 3 4 5 6 7 8

# yields the vector with the elements spaced three unit apart
> seq(from=12,to=30,by=3)
[1] 12 15 18 21 24 27 30
```

seq()

The spacing can be a noninteger 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
```

Or

```
> seq(from=1.1, to=2, by = 0.1)
 [1] 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

The empty-vector problem

```
> x = c()
```

```
> for(i in 1:length(x)){print (i)}
[1] 1
[1] 0
```

- If x is empty, this loop should not have any iterations.
- But it actually has two, since 1:length(x) evaluates to (1,0).
- The following code fix the problem

```
> for (i in seq(x))(print (i))
>
```

- if a vector is not empty,**seq(x)** gives us the same result as **1:length(x)**
- if a vector is empty, **seq(x)** results 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)**, which creates a vector of times*length(x) elements

```
> 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
```

rep()

There is also a named argument *each* of **rep()** function, with interleaves the copies of x.

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

any() and all()

```
 > x <- 1:10
 > x > 8
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE

> any(x > 8)
[1] TRUE
```

When R execute the **any (x $>$ 8)**

- It first evaluates x $>$ 8
- 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.

any() and all()

```
> x <- 1:10
```

```
> any(x > 88)

> all(x > 88)

> all(x > 0)
```

any() and all()

```
> any(x > 88)
[1] FALSE

> all(x > 88)
[1] FALSE

> all(x > 0)
[1] TRUE
```

Strings in R are largely lexicographic

```
> "b" > "v"
[1] FALSE

> "b" > "a"
[1] TRUE
```

String comparison in R

Strings in R are largely lexicographic

```
> "b" > "v"
[1] FALSE

> "b" > "a"
[1] TRUE
```

How about

```
> vals <- c("medium", "low", "high", "low")
> vals[1] > vals[3]
```

Vectorized functions in R

round() function that rounds to the nearest integer to an example vector y:

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

NA and NULL Values in R

The missing data can be represented in R with the value NA.

```
> 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"
```

Filtering

- Filtering allows us to extract a vectors elements that satisfy certain conditions.

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

Question: What is *w*?

Filtering

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

Vector

- We will again define our extraction condition in terms of z
- Then we will use the results to extract from another vector, y, instead of extracting from z

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

Question: What is y[j]?

Vector: filter

```
> 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] 130 5
```

Vector: Filter

- Filtering can also be done with the **subset()** function.
- 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
```

Vector: Filter

- 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
```

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
- The 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

An example:
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.

```
> x <- c(5,2,9,12)
> ifelse(x > 6,2*x,3*x)
```

Question: output?

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

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.

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

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

- For an abalone data set, gender is coded as M, F, or I (for infant).
- We want to recode those characters as 1, 2, or 3.
- The real data set consists of more than 4,000 observations, but for our example, we just use a few, stored in g:

```
> g
[1] "M" "F" "F" "I" "M" "M" "F"

#Nest ifelse()
> ifelse(g == "M",1,ifelse(g == "F",2,3))
```

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

```
> g
[1] "M" "F" "F" "I" "M" "M" "F"

#Nest ifelse()
> ifelse(g == "M",1,ifelse(g == "F",2,3))
[1] 1 2 2 3 1 1 2
```

Extended Example

- Suppose we wish to form subgroups according to gender.
- We could use which() to find the element numbers corresponding to M, F, and I:

```
> g
[1] "M" "F" "F" "I" "M" "M" "F"

> m <- which(g == "M")
> f <- which(g == "F")
> i <- which(g == "I")
```

Extended Example

- Suppose we wish to form subgroups according to gender.
- We could use which() to find the element numbers corresponding to M, F, and I:

```
> g
[1] "M" "F" "F" "I" "M" "M" "F"

> m <- which(g == "M")
> f <- which(g == "F")
> i <- which(g == "I")
> m
[1] 1 5 6
> f
[1] 2 3 7
> i
[1] 4
```

Testing Vector Equality

Suppose we wish to test whether two vectors are equal. The naive approach, using $==$, will work.

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

Testing Vector Equality

- 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
```

Testing Vector Equality

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

```
> identical(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

 #assign name
 > names(x) <- c("a","b","ab")

 #query name
 > names(x)
[1] "a" "b" "ab"

 > x
  a b ab
  1 2 4
```

Vector Element Names

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

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

More on c()

- If the arguments you pass to c() are of differing modes, vectors will be coerced to the highest type of the components in the hierarchy NULL < raw < logical < integer < real < complex < character < list < expression

```
#character > integer
> c(5,2,"abc")
  [1] "5" "2" "abc"

#list > character
> c(5,2,list(a=1,b=4))
  [[1]]
  [1] 5
  [[2]]
  [1] 2
  $a
  [1] 1
  $b
  [1] 4
```