# Managing Vectors

2022-09-26

## Managing Vectors Course Subsection

### Creating a Vector from Values

```
# Produce a vector with consecutive numbers
x <- 1:5
y <- 6:10
x
```

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

```
y
```

```
## [1]  6  7  8  9 10
```

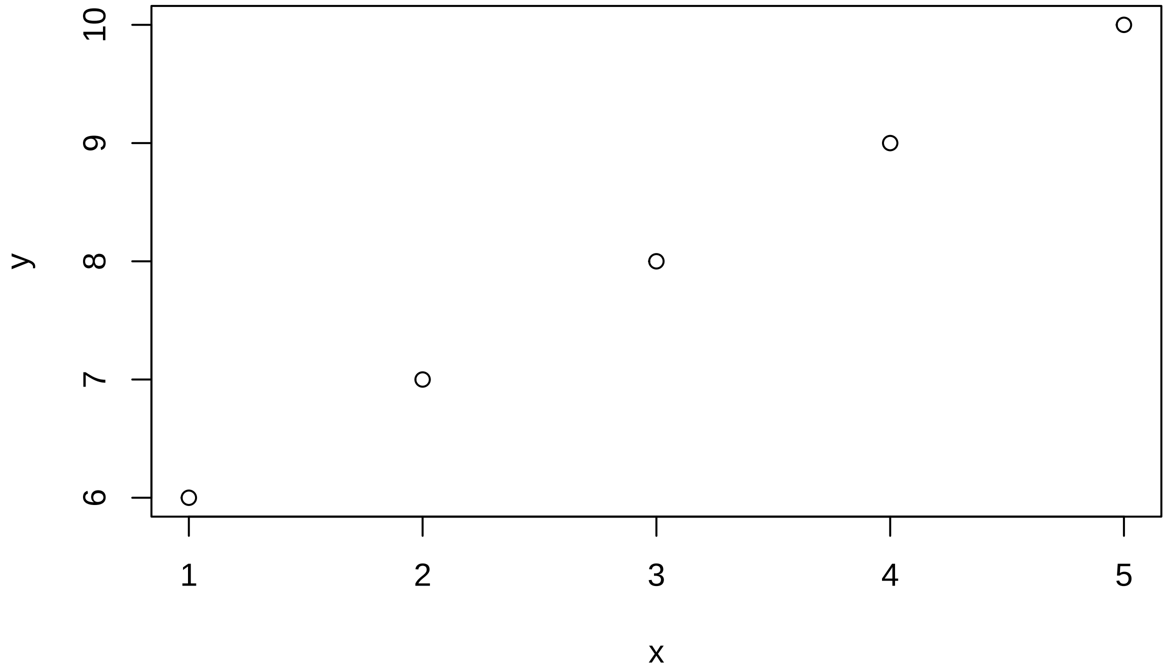
```
is.vector(x)
```

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

```
typeof(y)
```

```
## [1] "integer"
```

```
plot(x, y)
```



- A Vector is a conglomeration of values of the same type.
- Order Matters.
- A number/value is a vector of 1.

```
countries <- c('USA', 'UK')
typeof(countries) # Result: character
```

```
## [1] "character"
```

```
length(countries) # Result: 2
```

```
## [1] 2
```

```
# to check length of a string, use nchar() func
```

Combine function -c() is used when vector is un-ordered.

All elements in a vector must be of the same type. If you want to combine elements of different types using -c(), R will use implicit coercion… changing types automatically to convert them to the same type.

```
mix <- c(1, TRUE, 'way')
mix
```

```
## [1] "1"    "TRUE" "way"
```

```
typeof(mix)
```

```
## [1] "character"
```

Nesting vectors inside of other vectors will flatten them:

```
nesting_doll <- c(1, 2, c(4, 5, c(6, 'seven')))
nesting_doll
```

```
## [1] "1"     "2"     "4"     "5"     "6"     "seven"
```

```
# Note that implicit coercion still applies
```

### Merging Values into a Vector

```


v <- c(2:9)

# Merge values at the start
v <- c(1, v)
v
```

### Merging Values at the start/end of a vector:

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

```
# Merge values at the end
v <- c(v, 10)
v
```

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

```
# Merging multiple values
v1 <- c(1:5)
v2 <- c(6:10)

v3 <- c(v1, v2)
v4 <- c(v2, v1)

v5 <- c(v1, 11:15)

v3
```

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

```
v4
```

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

```
v5
```

```
## [1]  1  2  3  4  5 11 12 13 14 15
```

```
poem <- c('Mary', 'little', 'lamb')

poem <- append(poem, c('had', 'a'), after = 1)
poem
```

#### Merging Values inside a vector is accomplished through the use of the append() func

```
## [1] "Mary"   "had"    "a"      "little" "lamb"
```

Can see a behind the scenes look at append() func implementation:

```
append
```

```
## function (x, values, after = length(x))
## {
##     lengx <- length(x)
##     if (!after)
##         c(values, x)
##     else if (after >= lengx)
##         c(x, values)
##     else c(x[1L:after], values, x[(after + 1L):lengx])
## }
## <bytecode: 0x563ef8c25118>
## <environment: namespace:base>
```

Here we see that the append() is a wrapper around the c() func.

**Performance Implications of one big merge vs many small merges** One big merge is much better unless using a small sample size as the vector needs to be resized with every append.

### Merging Vectors into a Character Vector

Applies to scenarios such as merging strings or an sql query

```
# Wrong way:
#"hello" + "world"
```

Right way: Use paste() function to:

- Merge vectors of length = 1
- Merge vectors of same length > 1
- Merge vectors into one string

```r
# Recap
length("hello") # Vector of length 1
```

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

```r
nchar("hello") # Composed of 5 characters
```

```
## [1] 5
```

```r
c('h', 'e', 'l', 'l', 'o') # Vector of length 5
```

```
## [1] "h" "e" "l" "l" "o"
```

```r
# Merge vectors of length = 1
paste('hello', 'world') # Defaults to incorporating one space between elements
```

```
## [1] "hello world"
```

```r
# Note: One or more objects can be pasted, to then be converted to a character vector (implicit coercio
paste(1, 'two', TRUE)
```

```
## [1] "1 two TRUE"
```

```r
# Test: Replace default separator with an empty space
paste('hello', 'world', sep = '')
```

```
## [1] "helloworld"
```

```r
# Merge vectors of same length > 1
paste(c('name', 'age'), c('John', 5), c('Doe', 'years')) # Returns a character vector of length 2: "nam
```

```
## [1] "name John Doe" "age 5 years"
```

```r
# To merge vectors into one string... use collapse() arg
paste(c('name', 'age'),
      c('John', 5),
      c('Doe', 'years'), collapse = '-') # Returns: "name John Doe-age 5 years"
```

```
## [1] "name John Doe-age 5 years"
```

Also use paste() function and recycling to:

- Merge vectors of different lengths
- Recycling will automatically repeat or ‘recycle’ the shorter vectors to match the length of the largest vector

```
paste(c('name', 'age', 'Name', 'Age'),
      c('John', 5)) # Returns "name John" "age 5"     "Name John" "Age 5" *Equivalent to copying and pa
```

```
## [1] "name John" "age 5"     "Name John" "Age 5"
```

```
paste(c('name', 'age', 'Name', 'Age'),
      c('John', 5,     'John', 5))
```

```
## [1] "name John" "age 5"     "Name John" "Age 5"
```

```
paste(c('name', 'age', 'Name', 'Age', 'Other'),
      c('John', 5,     'John', 5))
```

```
## [1] "name John"  "age 5"      "Name John"  "Age 5"      "Other John"
```

```
# Recycling is applicable to functions other than paste()... for ex:
c(1, 2) + 3 # Returns: [1] 4 5
```

```
## [1] 4 5
```

```
# Which is equivalent to:
c(1, 2) +
c(3, 3) # Returns [1] 4 5
```

```
## [1] 4 5
```

**Merging Vectors into a Matrix** What is a matrix in R?

- A vector with multiple dimensions instead of 1.
- All values in a matrix must be of the same type.

```
first_matrix <- matrix(1, nrow = 2, ncol = 3)
first_matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
```

```
is.matrix(first_matrix) # Returns: TRUE
```

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

```
is.vector(first_matrix) # Returns: FALSE
```

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

```
typeof(first_matrix) # Returns: double
```

```
## [1] "double"
```

```
length(first_matrix) # Returns: 6
```

```
## [1] 6
```

```
larger_matrix <- matrix(1:9, nrow = 3, ncol = 3)
larger_matrix
```

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

```
# Isolate values from a matrix using row and col indicies
second_col <- larger_matrix[,2]
second_col
```

```
## [1] 4 5 6
```

```
is.vector(second_col) # Returns TRUE
```

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

```
third_row <- larger_matrix[3,]
third_row
```

```
## [1] 3 6 9
```

```
is.vector(third_row) # Returns TRUE
```

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

Thus, a matrix is composed of a series of vectors stacked on top of one another, or a series of columns placed side by side. We should logically then be able to build out a matrix by stacking rows or adding columns. To do so, we use cbind()/rbind() functions to merge vectors into a matrix:

```
# Combining vectors of length 1
cbind('hello', 'world')
```

```
##      [,1]    [,2]
## [1,] "hello" "world"
```

```
rbind('hello', 'world')
```

```
##      [,1]
## [1,] "hello"
## [2,] "world"
```

```
# Combining vectors of length > 1
cbind(c(1, 2, 3), c(4, 5, 6)) # Returns: a 3x2 matrix (cbind "sees" two distinct objects)
```

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

```
rbind(c(1, 2, 3), c(4, 5, 6)) # Returns: a 2x3 matrix
```

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

```
#Combining vectors of different lengths - Recycling rule is again applied
cbind(c(1, 2, 3), 5)
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    5
## [3,]    3    5
```

```
rbind(c('a', 'b', 'c', 'd', 'e'), c(1, 2), TRUE)
```

```
## Warning in rbind(c("a", "b", "c", "d", "e"), c(1, 2), TRUE): number of columns
## of result is not a multiple of vector length (arg 2)
```

```
##      [,1]   [,2]   [,3]   [,4]   [,5]
## [1,] "a"    "b"    "c"    "d"    "e"
## [2,] "1"    "2"    "1"    "2"    "1"
## [3,] "TRUE" "TRUE" "TRUE" "TRUE" "TRUE"
```

```
# Note that implicit coercion is applied
```