Fundamentals of R

Programming for Statistical Science

Shawn Santo

Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Companion videos

- RStudio Tour
- Vectors
- Operators, vectorization, and length coercion
- Control flow
- Error action
- Loops

Videos were created for STA 323 & 523 - Summer 2020

Additional resources

- Google's R Style Guide
- Hadley's R Style Guide
- Sections 3.1 – 3.2 Advanced R
- Chapter 5 Advanced R

Vectors

Vectors

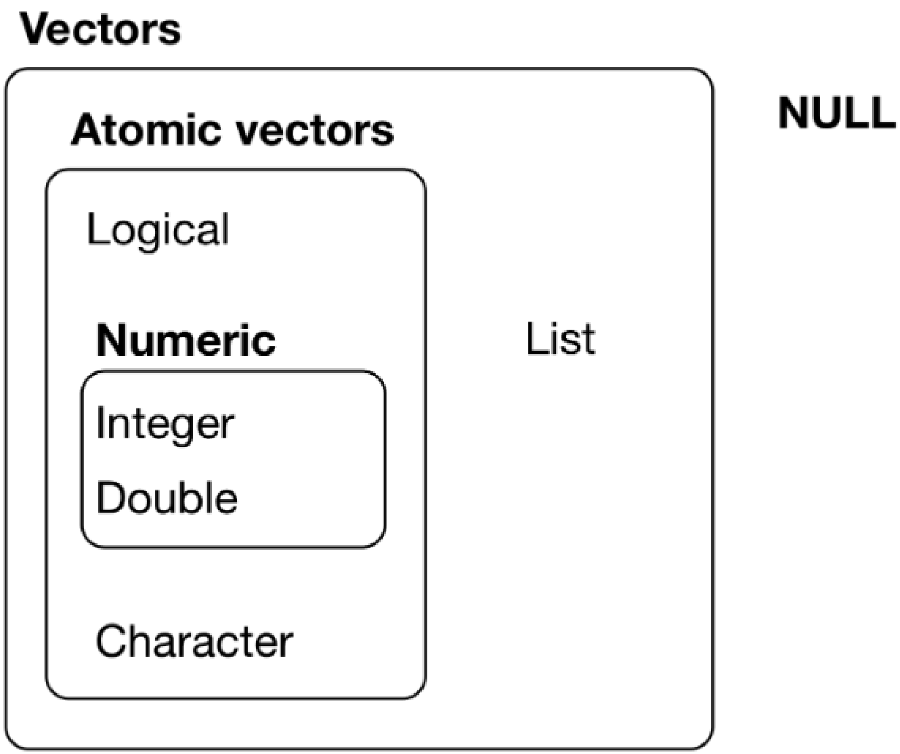
The fundamental building block of data in R is a vector (collections of related values, objects, other data structures, etc).

R has two types of vectors:

- **atomic** vectors
  - homogeneous collections of the *same* type (e.g. all logical values, all numbers, or all character strings).
- **generic** vectors
  - heterogeneous collections of *any* type of R object, even other lists (meaning they can have a hierarchical/tree-like structure).

I will use the term component or element when referring to a value inside a vector.

Vector interrelationships



Source: https://r4ds.had.co.nz/vectors.html

Atomic vectors

R has six atomic vector types:

`logical`, `integer`, `double`, `character`, `complex`, `raw`

In this course we will mostly work with the first four. You will rarely work with the last two types - complex and raw.

```
x <- c(T, F, TRUE, FALSE)
typeof(x)
```

```
#> [1] "logical"
```

```
y <- c("a", "few", "more", "slides")
typeof(y)
```

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

Coercion hierarchy

If you try to combine components of different types into a single atomic vector, R will try to coerce all elements so they can be represented as the simplest type.

$$\texttt{character} \rightarrow \texttt{double} \rightarrow \texttt{integer} \rightarrow \texttt{logical}$$

```
x <- c(T, 5, F, 0, 1)
y <- c("a", 1, T)
z <- c(3.0, 4L, 0L)
```

```
x
```

```
#> [1] 1 5 0 0 1
```

```
typeof(x)
```

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

```
y
```

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

```
typeof(y)
```

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

```
z
```

```
#> [1] 3 4 0
```

```
typeof(z)
```

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

Concatenation

One way to construct atomic vectors is with function `c()`.

```
c(1, 0, 1, 1, 6)
```

```
#> [1] 1 0 1 1 6
```

```
c(c(3, 4), c(10, TRUE))
```

```
#> [1]  3  4 10  1
```

```
c(pi)
```

```
#> [1] 3.141593
```

Operators, vectorization, and length coercion

Logical (Boolean) operators

Operator	Operation	Vectorized?
`x	y`	or	Yes
`x & y`	and	Yes
`!x`	not	Yes
`x || y`	or	No
`x && y`	and	No
`xor(x,y)`	exclusive or	Yes

What do we mean if we say a function or operation is vectorized?

Boolean examples

```
x <- c(T, F, T, T)
y <- c(F, F, T, F)
```

```
!x
```

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

```
x | y
```

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

```
x || y
```

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

```
x & y
```

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

```
x && y
```

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

```
xor(x, y)
```

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

Comparison operators

Operator	Comparison	Vectorized?
`x < y`	less than	Yes
`x > y`	greater than	Yes
`x <= y`	less than or equal to	Yes
`x >= y`	greater than or equal to	Yes
`x != y`	not equal to	Yes
`x == y`	equal to	Yes
`x %in% y`	contains	Yes (over `x`)

Comparison examples

```r
x <- c(4, 10, -5)
y <- c(0, 51, 9 / 5)
z <- c("four", "for", "4")
```

```r
x > y
```

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

```r
x != y
```

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

```r
x == z
```

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

```r
x %in% z
```

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

What else is vectorized?

- Most of the mathematical operators
- Many functions in `base` R and created by user's in packages

```
a <- c(0, -3, sqrt(75))
b <- c(1, 3, 2)
```

```
a + b
```

```
#> [1]  1.00000  0.00000 10.66025
```

```
a ^ b
```

```
#> [1]   0 -27  75
```

```
rnorm(n = 3, mean = a, sd = b)
```

```
#> [1] -0.6483697  1.6219890  6.7336622
```

```
exp(a / b)
```

```
#> [1]  1.0000000  0.3678794 75.9539335
```

Length coercion (vector recycling)

The shorter of two atomic vectors in an operation is recycled until it is the same length as the longer atomic vector.

```
x <- c(2, 4, 6)
y <- c(1, 1, 1, 2, 2)
```

```
x > y
```

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

```
x == y
```

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

```
10 / x
```

```
#> [1] 5.000000 2.500000 1.666667
```

Control flow

Conditional control flow

Conditional (choice) control flow is governed by `if` and `switch()`.

```
if (condition) {
  # code to run
  # when condition is
  # TRUE
}
```

```
if (TRUE) {
  print("The condition must have b
}
```

`if` examples

```r
if (1 > 0) {
  print("Yes, 1 is greater than 0.")
}
```

```
#> [1] "Yes, 1 is greater than 0."
```

```r
x <- c(1, 2, 3, 4)
if (3 %in% x) {
  print("Yes, 3 is in x.")
}
```

```
#> [1] "Yes, 3 is in x."
```

```r
if (-6) {
  print("Other types are coerced to logical if possible.")
}
```

```
#> [1] "Other types are coerced to logical if possible."
```

More `if` examples

```r
if (c(F, T, T)) {
  print("How many logical values can if handle?")
}
```

```
#> Warning in if (c(F, T, T)) {: the condition has length > 1 and only the first
#> element will be used
```

```r
x <- c(1, 2, 3, 4)
if (x %in% 3) {
  print("This works?")
}
```

```r
if (c(1, 0, 1)) {
  print("Other types are coerced to logical if possible.")
}
```

```
#> [1] "Other types are coerced to logical if possible."
```

I suppressed warnings in the last two examples.

`if` is not vectorized

To remedy this potential problem of a non-vectorized `if`, you can

1. try to collapse a logical vector of length greater than 1 to a logical vector of length 1 with functions
    - `any()`
    - `all()`
2. use a vectorized conditional function such as `ifelse()` or `dplyr::case_when()`.

Functions `any()` and `all()`

```
x <- c(-5, 0, 5, 10, 15)
any(x >= 5)
```

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

```
all(x >= 5)
```

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

Functions `any()` and `all()` require a logical vector as input.

Vectorized `if`

```r
z <- c(-4:-1, 1:3)
z
```

```
#> [1] -4 -3 -2 -1  1  2  3
```

```r
ifelse(test = z < 0, yes = "neg", no = "pos")
```

```
#> [1] "neg" "neg" "neg" "neg" "pos" "pos" "pos"
```

```r
set.seed(532)
x <- rnorm(n = 4, mean = 0, sd = 1)
x
```

```
#> [1]  3.105059 -1.329432 -1.466140 -0.345289
```

```r
ifelse(test = abs(x) > 3, yes = "outlier", no = "no outlier")
```

```
#> [1] "outlier"    "no outlier" "no outlier" "no outlier"
```

Nested conditionals

```r
if (condition_one) {
  ##
  ## Code to run
  ##
} else if (condition_two) {
  ##
  ## Code to run
  ##
} else {
  ##
  ## Code to run
  ##
}
```

```r
x <-  0
if (x < 0) {
  "Negative"
} else if (x > 0) {
  "Positive"
} else {
  "Zero"
}
```

```
#> [1] "Zero"
```

Error action

Execute error action

Functions `stop()` and `stopifnot()` execute an error action. These are useful if you want to validate inputs or function arguments.

```
x <- -1
if (x < 0) {
  stop("Negative numbers not allowed!")
}
```

```
#> Error in eval(expr, envir, enclos): Negative numbers not allowed!
```

```
x <- c(3, 9, 28)
stopifnot(any(x >= 0), all(x %% 3 == 0))
```

```
#> Error: all(x%%3 == 0) is not TRUE
```

If any of the expressions in function `stopifnot()` are not `TRUE`, then function `stop()` is called and an error message is shown.

Exercises

1. What does each of the following return? Run the code to check your answer.

```
if (1 == "1") "coercion  works" else "no coercion "

ifelse(5 > c(1, 10, 2), "hello", "olleh")
```

2. Consider two vectors, `x` and `y`, each of length one. Write a set of conditionals that satisfy the following.

   - If `x` is positive and `y` is negative or `y` is positive and `x` is negative, print "knits".
   - If `x` divided by `y` is positive, print "stink".
   - Stop execution if `x` or `y` are zero.

   Test your code with various `x` and `y` values. Where did you place the stop execution code?

Loops

Loop types

R supports three types of loops: `for`, `while`, and `repeat`.

```
for (item in vector) {
  ##
  ## Iterate this code
  ##
}
```

```
while (we_have_a_true_condition) {
  ##
  ## Iterate this code
  ##
}
```

```
repeat {
  ##
  ## Iterate this code
  ##
}
```

In the `repeat` loop we will need a `break` statement to end iteration.

`for` loop

A `for` loop allows you to iterate code over items in a vector.

```
k <- 0
for (i in c(2, 4, 6, 8)) {
  print(i ^ 2)
  k <- k + i ^ 2
}
```

```
#> [1] 4
#> [1] 16
#> [1] 36
#> [1] 64
```

```
k
```

```
#> [1] 120
```

```
for (i in c(2, 4, 6, 8)) {
  i ^ 2
}
```

Automatic printing is turned off inside loops.

`while` loop

A `while` loop will iterate code until a given condition is `FALSE`.

```
i <-  1
res <- rep(0, 10)

i
```

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

```
res
```

```
#>  [1] 0 0 0 0 0 0 0 0 0 0
```

```
while (i <= 10) {
  res[i] <- i ^ 2
  i <- i + 1
}

res
```

```
#>  [1]   1   4   9  16  25  36  49  64  81 100
```

`repeat` loop

A `repeat` loop will iterate code until a `break` statement is executed.

```
i <- 1
res <- rep(NA, 10)

repeat {
  res[i] <- i ^ 2
  i <- i + 1
  if (i > 10) {break}
}

res
```

```
#>  [1]   1   4   9  16  25  36  49  64  81 100
```

Loop keywords: `next` and `break`

- `next` exits the current iteration and advances the looping index
- `break` exits the loop
- Both `break` and `next` apply only to the innermost of nested loops.

```r
for (i in 1:10) {
  if (i %% 2 == 0) {next}

  print(paste("Number ", i, " is odd."))

  if (i %% 7 == 0) {break}
}
```

```
#> [1] "Number  1  is odd."
#> [1] "Number  3  is odd."
#> [1] "Number  5  is odd."
#> [1] "Number  7  is odd."
```

Ancillary loop functions

You may want to loop over indices of an object as opposed to the object's values. To do this, consider using one of `length()`, `seq()`, `seq_along()`, and `seq_len()`.

```
4:7
```

```
#> [1] 4 5 6 7
```

```
length(4:7)
```

```
#> [1] 4
```

```
seq(4, 7)
```

```
#> [1] 4 5 6 7
```

```
seq_along(4:7)
```

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

```
seq_len(length(4:7))
```

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

```
seq(4, 7, by = 2)
```

```
#> [1] 4 6
```

Iterating over `seq_along(x)` is a better option than `1:length(x)`.

Loop tips

1. Preallocate your output object when possible.
2. Don't use a `while` or `repeat` loop if a `for` loop is possible.
3. Don't use any type of loop if vectorization is possible.

Slow...

```
a <- c()
for (i in seq_len(10)) {
  a <- c(a, i ^ 3)
}
```

Faster...

```
a <- numeric(10)
for (i in seq_len(10)) {
  a[i] <- i ^ 3
}
```

Even faster...

```
(1:10) ^ 3
```

Exercises

1. Consider the vector `x` below.

   ```
   x <- c(3, 4, 12, 19, 23, 49, 100, 63, 70)
   ```

   Write R code that prints the perfect squares in `x`.

2. Consider `z <- c(-1, .5, 0, .5, 1)`. Write R code that prints the smallest non-negative integer k satisfying the inequality

   $$|cos(k) - z| < 0.001$$

   for each component of `z`.

References

1. Grolemund, G., & Wickham, H. (2019). R for Data Science. https://r4ds.had.co.nz/
2. Wickham, H. (2019). Advanced R. https://adv-r.hadley.nz/