Basics of R

Jyotishka Datta

2020/01/24 (updated: 2020-09-04)

(Almost) Everything is a Vector

Types of vectors

The fundamental building block of data in R are vectors (collections of related values, objects, other data structures, etc).

R has two fundamental vector classes:

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

Conditionals

Logical (boolean) operations

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

Vectorized?

```
x = c(TRUE,FALSE,TRUE)
y = c(FALSE,TRUE,TRUE)
```

```
  x | y
```

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

```
  x & y
```

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

```
  x || y
```

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

```
  x && y
```

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

Length coercion

```
x = c(TRUE,FALSE,TRUE)
y = c(TRUE)
z = c(FALSE,TRUE)
```

```
  x | y
```

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

```
  x & y
```

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

```
  y | z
```

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

```
  y & z
```

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

```
x | z
```

```
## Warning in x | z: longer object length is not a multiple of shorter object
## length
```

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

Comparisons

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 (for `x`)

Comparisons

```
x = c("A","B","C")
z = c("A")
```

```
  x == z
```

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

```
  x != z
```

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

```
  x > z
```

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

```
  x %in% z
```

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

```
  z %in% x
```

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

Conditional Control Flow

Conditional execution of code blocks is achieved via `if` statements.

*Note that `if` statements are **not** vectorized.*

```
x = c(3,1)

if (3 %in% x)
  "Here!"
```

```
## [1] "Here!"
```

```
if (x >= 2)
  "Now Here!"
```

```
## Warning in if (x >= 2) "Now Here!": the condition has length > 1 and only
## first element will be used

## [1] "Now Here!"
```

Collapsing logical vectors

There are a couple of helper functions for collapsing a logical vector down to a single value: `any`, `all`

```
x = c(3,4)
```

```
  any(x >= 2)
```

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

```
  all(x >= 2)
```

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

```
  !any(x >= 2)
```

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

```
  if (any(x >= 2))
    print("Now There!")
```

```
## [1] "Now There!"
```

Error Checking

`stop` and `stopifnot`

Often we want to validate user input or function arguments - if our assumptions are not met then we often want to report the error and stop execution.

```
ok = FALSE
if (!ok)
  stop("Things are not ok.")
```

```
## Error in eval(expr, envir, enclos): Things are not ok.
```

```
stopifnot(ok)
```

```
## Error: ok is not TRUE
```

Note - an error (like the one generated by `stop`) will prevent an RMarkdown document from compiling unless `error=TRUE` is set for that code block.

Style choices

```
# Do stuff
if (condition_one) {
  ##
  ## Do stuff
  ##
} else if (condition_two) {
  ##
  ## Do other stuff
  ##
} else if (condition_error) {
  stop("Condition error occured")
}
```

Style choices

```
# Do stuff better
if (condition_error) {
  stop("Condition error occured")
}

if (condition_one) {
  ##
  ## Do stuff
  ##
} else if (condition_two) {
  ##
  ## Do other stuff
  ##
}
```

Ultimately, it's subjective !

Loops

`for` loops

Simplest, and most common type of loop in R - given a vector iterate through the elements and evaluate the code block for each.

```
for(x in 1:10)
{
  cat(x^2,"")
}
```

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

```
for(y in list(1:3, LETTERS[1:7], c(TRUE,FALSE)))
{
  cat(length(y),"")
}
```

```
## 3 7 2
```

`while` loops

Repeat until the given condition is **not** met (i.e. evaluates to `FALSE`)

```
i = 1
res = rep(NA,10)

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

res
```

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

`repeat` loops

Repeat until `break`

```
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
```

Special keywords - `break` and `next`

These are special actions that only work *inside* of a loop

- `break` - ends the current *loop* (inner-most)
- `next` - ends the current *iteration*

```
for(i in 1:10) {
    if (i %% 2 == 0)
        break
    cat(i,"")
}
```

```
## 1
```

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

```
## 1 3 5 7 9
```

Some helper functions

Often we want to use a loop across the indexes of an object and not the elements themselves. There are several useful functions to help you do this: `:`, `length`, `seq`, `seq_along`, `seq_len`, etc.

```
4:7
```

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

```
seq(4,7,by=1)
```

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

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

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

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

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

Functions

When to use functions

The goal of a function should be to encapsulate a *small reusable* piece of code.

- Name should make it clear what the function does (think in terms of simple verbs).
- Functionality should be simple enough to be quickly understood.
- The smaller and more modular the code the easier it will be to reuse elsewhere.
- Better to change code in one location than code everywhere.

Next time we will look at functions.

Acknowledgments

Above materials are derived in part from the following sources:

- Colin Rundell's slides.
- Hadley Wickham - Advanced R
- R Language Definition