Basics of R

Jyotishka Datta

2020/01/24 (updated: 2023-08-23)

(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
```

```
## Warning in x || y: 'length(x) = 3 > 1' in coercion to 'logical(1)'
```

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

```
x & y
```

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

```
x && y
```

```
## Warning in x && y: 'length(x) = 3 > 1' in coercion to 'logical(1)'
```

```
## Warning in x && y: 'length(x) = 3 > 1' in coercion to 'logical(1)'
```

```
## [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!"
```

```
## Error in if (x >= 2) "Now Here!": the condition has length > 1
```

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
```

Exercise 2

Below is the list of primes between 2 and 100:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97
```

If you were given the vector `x = c(3, 4, 12, 19, 23, 48, 50, 61, 63, 78)`, write out the R code necessary to print only the values of `x` that are *not* prime (without using subsetting or the `%in%` operator).

Your code should use *nested* loops to iterate through the vector of primes and `x`.

Next we will look at functions

source: https://r4ds.hadley.nz/functions.html

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.

When to use ...

Writing a function has four big advantages over using copy-and-paste:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another).
- It makes it easier to reuse work from project-to-project, increasing your productivity over time.

Rule-of-thumb: did you copy-paste a block of code more than twice?

What does this do?

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Error in copy-paste

You might be able to puzzle out that this rescales each column to have a range from 0 to 1.

But did you spot the mistake? I made an error when copying-and-pasting the code for df$b: I forgot to change an a to a b.

Extracting repeated code out into a function is a good idea because it prevents you from making this type of mistake.

What does this do?

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /             #<<
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE)) #<<
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Write a function

To write a function you need to first analyse the code. How many inputs does it have?

```
(df$a - min(df$a, na.rm = TRUE))/(max(df$a, na.rm = TRUE) - min(df$a,
```

```
##  [1] 0.02123465 0.54616475 1.00000000 0.46179244 0.38005883 0.08444326
##  [7] 0.00000000 0.07246454 0.12356484 0.42641593
```

- Why is `TRUE` not an input?
- To make the inputs more clear, it's a good idea to rewrite the code using temporary variables with general names. Here this code only requires a single numeric vector, so I'll call it x:

```
x <- df$a
(x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = T
```

```
##  [1] 0.02123465 0.54616475 1.00000000 0.46179244 0.38005883 0.08444326
##  [7] 0.00000000 0.07246454 0.12356484 0.42641593
```

Redundancy?

There is some duplication in this code. We're computing the range of the data three times, so it makes sense to do it in one step:

```
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])
```

```
##  [1] 0.02123465 0.54616475 1.00000000 0.46179244 0.38005883 0.08444326
##  [7] 0.00000000 0.07246454 0.12356484 0.42641593
```

Pulling out intermediate calculations into named variables is a good practice because it makes it more clear what the code is doing.

Turn this into a function

Now that I've simplified the code, and checked that it still works, I can turn it into a function:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10))
```

```
## [1] 0.0 0.5 1.0
```

Use with earlier example

We can simplify the original example now that we have a function:

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Compare this with the original block of code.

Change easily

Another advantage of functions is that if our requirements change, we only **need to make the change in one place**.

For example, we might discover that some of our variables include infinite values, and `rescale01()` fails:

```
x <- c(1:10, Inf)
rescale01(x)
```

```
##  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Because we've extracted the code into a function, we only need to make the fix in one place:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)
```

```
##  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
##  [8] 0.7777778 0.8888889 1.0000000       Inf
```

Exercise

- Why is `TRUE` not a parameter to `rescale01()`? What would happen if `x` contained a single missing value, and `na.rm` was FALSE?
- In the second variant of `rescale01()`, infinite values are left unchanged. Rewrite `rescale01()` so that `-Inf` is mapped to 0, and `Inf` is mapped to 1.
- Write your own functions to compute the skewness of a numeric vector. Skewness is defined as:

$$Skew(x) = \frac{\frac{1}{n-2}\sum_{i=1}^{n}(x_i - \bar{x})^3}{Var(x)^{3/2}}$$

Now, we will dissect the structure of a function in R

source: https://statprog-s1-2020.github.io/

Functions

- Functions are blocks of code that allows R to be modular and facilitate code reuse.
- An R programmer can define their own functions as follows:

```
function_name <- function([arg1], [arg2], ...){
#function code body
}
```

- The function arguments are optional. Function arguments are the variables passed to the function, used by the function's code to perform calculations. A function can take no arguments.
- A function can also return any R primitive or object using the `return(object)` statement.

Function arguments

Functions have named arguments which potentially have default values.

The *formal arguments* are the arguments included in the function definition

The `formals` function returns a list of all the formal arguments of a function

Not every function call in R makes use of all the formal arguments

Function arguments can be missing or might have default values

Argument matching

R functions arguments can be matched positionally or by name. So the following calls to `sd` are all equivalent.

```
mydata <- rnorm(100)
 sd(mydata)
```

```
## [1] 0.9742524
```

```
 sd(x = mydata)
```

```
## [1] 0.9742524
```

```
 sd(x = mydata, na.rm = FALSE)
```

```
## [1] 0.9742524
```

```
 sd(na.rm = FALSE, x = mydata)
```

```
## [1] 0.9742524
```

```
 sd(na.rm = FALSE, mydata)
```

Examples

- Examples of two different functions

```
# computes the mean of a vector of numbers
mean <- function(a_vector) {
s <- sum(a_vector)
x <- s/length(a_vector)
}
```

```
# checks to see if a string s starts with letter x
startsWith <- function(x, s){
if(x == substr(s, 0, 1){
return(TRUE)
}
return(FALSE)
}
```

Anatomy of a function

- **Default arguments**
- Function arguments can be assigned default values as follows:

```
sort_vector <- function(a_vector, ascending=TRUE){
# sorting algorithm
}
```

- When calling this function, the arguments given default values do not need to be specified. In this case, the defaults are used.
- Example:

```
sort_vector(a_vector) # returns a_vector in ascending order
sort_vector(a_vector, FALSE) # returns a vector in descending order
```

"Scope" of a variable

- Variable Scoping
- Variables that are bound to an R primitive or object outside a function are called global variables, and are accessible everywhere in an R program.
- Example:

```
x <- 5
test <- function() {
cat(x + 5)
}
test();
```

```
## 10
```

- Here 'x' is a "GLOBAL" variable - accessible from anywhere.

"Scope" (contd.)

- Variables bound inside a function are only accessible within that function. These are called local variables. Example:

```
x <- 10
test <- function() {
x <- 5
cat(x + 20)
}
test() #prints 25
```

```
## 25
```

```
cat(x + 20) #prints 30
```

```
## 30
```

- **Note that the local variable assignment takes precedence inside the function test over the global assignment**.

More about Functions

Function Parts

Functions are defined by two components: the arguments (`formals`) and the code (`body`). Functions are assigned names like any other object in R (using = or <-)

```
gcd = function(long1, lat1, long2, lat2) {
  R = 6371 # Earth mean radius in km

  # distance in km
  acos(sin(lat1)*sin(lat2) + cos(lat1)*cos(lat2) * cos(long2-long1))
}
```

Returning values

There are two approaches to returning values from functions in R - explicit and implicit return values.

Explicit - using one or more `return` statements

```
f = function(x) {
  x + 1
  return(x * x)
}
f(2)
```

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

Implicit - return value of the last expression is returned.

```
g = function(x) {
  x + 1
  x * x
}
g(3)
```

```
## [1] 9
```

Returning multiple values

If we want a function to return more than one value we can group things using either vectors or lists.

```
f = function(x) {
  c(x, x^2, x^3)
}

f(1:2)
```

```
## [1] 1 2 1 4 1 8
```

```
g = function(x) {
  list(x, "hello")
}

g(1:2)
```

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

Argument names

When defining a function we are also implicitly defining names for the arguments, when calling the function we can use these names to pass arguments in a alternative order.

```
f = function(x, y, z) {
  paste0("x=", x, " y=", y, " z=", z)
}
```

```
f(1, 2, 3)
```

```
## [1] "x=1 y=2 z=3"
```

```
f(y=2, 1, 3)
```

```
## [1] "x=1 y=2 z=3"
```

```
f(z=1, x=2, y=3)
```

```
## [1] "x=2 y=3 z=1"
```

```
f(y=2, 1, x=3)
```

```
## [1] "x=3 y=2 z=1"
```

```
f(1, 2, 3, 4)
```

```
## Error in f(1, 2, 3, 4): unused argument (4)
```

Argument defaults

It is also possible to give function arguments default values, so that they don't need to be provided every time the function is called.

```
f = function(x, y=1, z=1) {
  paste0("x=", x, " y=", y, " z=", z)
}
```

```
f(3)
```

```
## [1] "x=3 y=1 z=1"
```

```
f(z=3, x=2)
```

```
## [1] "x=2 y=1 z=3"
```

```
f(x=3)
```

```
## [1] "x=3 y=1 z=1"
```

```
f(y=2, 2)
```

```
## [1] "x=2 y=2 z=1"
```

```
f()
```

```
## Error in paste0("x=", x, " y=", y, " z=", z): argument "x" is missing, wit
```

Scope

R has generous scoping rules, if it can't find a variable in the functions body, it will look for it in the next higher scope, and so on.

```
y = 1
f = function(x) {
  x + y
}

f(3)
```

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

```
y = 1
g = function(x) {
  y = 2
  x + y
}

g(3)
```

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

Additionally, variables defined within a scope only persist for the duration of that scope, and do not overwrite variables at a higher scopes

```
x = 1
y = 1
z = 1
f = function() {
    y = 2
    g = function() {
      z = 3
      return(x + y + z)
    }
    return(g())
}
f()
```

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

```
c(x,y,z)
```

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

Exercise - scope

What is the output of the following code? Explain why.

```
z = 1

f = function(x, y, z) {
  z = x+y

  g = function(m = x, n = y) {
    m/z + n/z
  }

  z * g()
}

f(1, 2, x = 3)
```

Operators as functions

In R, operators are actually a special type of function - using backticks around the operator we can write them as functions.

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

```
typeof(`+`)
```

```
## [1] "builtin"
```

```
x = 4:1
x + 2
```

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

```
`+`(x, 2)
```

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

Getting Help

Prefixing any function name with a `?` will open the related help file for that function.

```
?`+`
?sum
```

For functions not in the base package, you can generally see their implementation by entering the function name without parentheses (or using the `body` function).

```
lm
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## {
##     ret.x <- x
##     ret.y <- y
##     cl <- match.call()
##     mf <- match.call(expand.dots = FALSE)
##     m <- match(c("formula", "data", "subset", "weights", "na.action",
##         "offset"), names(mf), 0L)
```

Less Helpful Examples

```
list
```

```
## function (...)  .Primitive("list")
```

```
`[`
```

```
## .Primitive("[")
```

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

Acknowledgments

Above materials are derived in part from the following sources:

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