

R Programming for Data Analysis

Topics

R Foundations

Conditional statements and loops

Functions

Functionals

Case Study 1: String parsing

Map functions in the purrr package

Case Study 2: Data modeling using the broom package

Overview of R and RStudio

https://github.com/suvampaul88/R_class

R Foundations

R's Data Structures

- Atomic vectors
- Lists
- Matrix
- Data frames
- Arrays

Vectors

The basic data structure in R is the vector

Every vector has three properties: type, length, attributes

```
x <- 1:5
```

```
typeof(x)
```

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

```
length(x)
```

```
#[1] 5
```

```
attributes(x)
```

```
#NULL
```

Atomic Vectors

There are four common types of atomic vectors

- logical
- integer
- double
- character

Atomic Vectors

Atomic vectors are created with `c()`

```
dbl_var <- c(1, 2.5, 4.5)
```

```
int_var <- c(1L, 6L, 10L)
```

```
log_var <- c(FALSE, TRUE, T, F)
```

```
chr_var <- c("strings", "there", "are", ":yoda")
```

Atomic Vectors

Atomic Vectors are always flat, even if you nest `c()`'s.

These two vectors are the same:

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

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

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

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

Coercion

What happens when you try to combine different atomic vectors with `c()`?

```
a <- c(1,2,3)
```

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

```
a <- c(a, "hi")
```

```
[1] "1" "2" "3" "hi"
```

Order of coercion:

logical -> integer -> numeric -> character

Missing Values

NA: logical constant of length 1 which contains a missing value indicator

```
x1 <- c(1, 4, 3, NA, 7)
```

```
x2 <- c("a", "B", NA, "NA")
```

```
is.na(x1)
```

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

```
is.na(x2)
```

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

Can you add these vectors?

```
b <- c(1,2,3,4)
```

```
d <- c(2,3,4,5)
```

```
b1 <- c(1,2,NA,4)
```

```
d1 <- c(2,3,4,5)
```

```
b2 <- c(T,T,F,F)
```

```
d2 <- c(2,3,4,5)
```

```
b3 <- c(3,4,6)
```

```
d3 <- c(3,4,5,6,7)
```

Lists

Lists are a special type of vector that can contain elements of any type

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
```

```
str(x)
```

```
#List of 4
```

```
#$ : int [1:3] 1 2 3
```

```
#$ : chr "a"
```

```
#$ : logi [1:3] TRUE FALSE TRUE
```

```
#$ : num [1:2] 2.3 5.9
```

Matrices and Arrays

A `dim()` attribute to an atomic vector makes it a multi-dimensional array.

A matrix is a special case of the array because it has two dimensions.

```
a <- matrix(1:6, ncol = 3, nrow = 2)
```

or

```
a <- 1:6
```

```
dim(a) <- c(2,3)
```

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

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

```
#[2,]  2   4   6
```

Matrices and Arrays

```
length(a) #[1] 6
```

```
nrow(a) #[1] 2
```

```
ncol(a) #[1] 3
```

```
rownames(a) <- c("A", "B")
```

```
colnames(a) <- c("a", "b", "c")
```

```
# a b c
```

```
#A 1 3 5
```

```
#B 2 4 6
```


Matrices and Arrays

```
b <- array(1:12, c(2,3,2))
```

```
#, , 1
```

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

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

```
#[2,]  2   4   6
```

```
#, , 2
```

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

```
#[1,]  7   9  11
```

```
#[2,]  8  10  12
```

Data frames

Most common way of store data in R.

A data frame is a list of equal-length vectors.

It has the properties of both the matrix and the list.

Data frames

```
df_1 <- data.frame(x = 1:3, y = c("a", "b", "c"))
```

```
str(df_1)
```

```
#'data.frame': 3 obs. of 2 variables:
```

```
# $ x: int 1 2 3
```

```
# $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Data frames

```
df_1 <- data.frame(x = 1:3, y = c("a", "b", "c"),
```

```
stringsAsFactors = FALSE)
```

```
str(df_1)
```

```
#'data.frame': 3 obs. of 2 variables:
```

```
# $ x: int 1 2 3
```

```
# $ y: chr "a" "b" "c"
```

Summary

R's data structures can be organized by dimensionality and whether they hold same types or different types of data.

	Same types	Different types
1d	Atomic vector	List
2d	Matrix	Data Frame
nd	Array	

Subsetting

R has powerful subsetting operators

[, [[, and \$

Subsetting - Vectors

```
x <- c(2.1, 4.2, 3.3, 5.3)
```

Positive integers

```
x[c(3,1)] #[1] 3.3 2.1
```

Negative integers

```
x[-c(3,1)] #[1] 4.2 5.3
```

Logical vectors

```
x[c(TRUE, TRUE, FALSE, FALSE)] #[1] 2.1 4.2
```

Nothing

```
x[] #[1] 2.1 4.2 3.3 5.3
```

Zero

```
x[0] #numeric(0)
```

Subsetting - Matrices

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

```
colnames(a) <- c("A", "B", "C")
```

```
#      A B C
```

```
#[1,] 1 4 7
```

```
#[2,] 2 5 8
```

```
#[3,] 3 6 9
```



```
a[1:2,]
```

```
#      A B C
```

```
#[1,] 1 4 7
```

```
#[2,] 2 5 8
```

```
a[c(T,F,T), c("B", "A")]
```

```
#      B A
```

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

```
#[2,] 6 3
```

Subsetting - Data frames

```
df_2 <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
```

```
#   x y z
```

```
#1 1 3 a
```

```
#2 2 2 b
```

```
#3 3 1 c
```

Subset with a single vector

```
df_2[c("x","z")]
```

```
#   x z
```

```
#1 1 a
```

```
#2 2 b
```

```
#3 3 c
```

Subset with two vectors

```
df_2[, c("x", "z")]
```

```
#   x z
```

```
#1 1 a
```

```
#2 2 b
```

```
#3 3 c
```

Like a list - if you subset with a single vector

```
str(df_2["x"])
```

```
#'data.frame': 3 obs. of 1 variable:
```

```
# $ x: int 1 2 3
```

Like a matrix - if you subset with two vectors (simplifies by default)

```
str(df_2[, "x"])
```

```
# int [1:3] 1 2 3
```

Subsetting Operators

There are two other types of subsetting operators: `[[` and `$`.

`[[` returns a single value, used with lists

“If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6.”

- @RLangTip

Because data frames are lists of (equal length) vectors, you can use `[[` to extract a column from data frames

```
mtcars[[1]], mtcars[["cyl"]]
```

Simplifying vs. Preserving Subsetting

Simplifying subsets returns the simplest possible data structure that can represent the output

Preserving subsets keeps the structure of the output the same as input

	Simplifying	Preserving
Vectors	<code>x[[1]]</code>	<code>x[1:4]</code>
Matrices/ Data Frames	<code>x[,1]</code>	<code>x[, 1, drop = F]</code>
Lists	<code>x[[1]]</code> <code>x\$name</code>	<code>x[1]</code>

Practice


```

df <- data.frame (

  col_a = rnorm(100),

  col_b = rnorm(100),

  col_c = rnorm(100),

  col_d = rnorm(100)

)

library(tibble)

as_tibble(df)

# A tibble: 100 x 4

   col_a    col_b    col_c    col_d
  <dbl>    <dbl>    <dbl>    <dbl>
1 -0.09754680 -1.0265275 -1.1441489  0.1355666
2 -0.01121655  0.3643875  0.8441854  1.0353159
3  0.26600646 -0.8164279 -0.6364876  1.1521661
4  0.46999005  0.1305813  1.8666679 -0.1122410
5  0.77085928  0.1764415  0.7265543  1.2569743
6  0.85352310  1.2663297 -0.2828890 -2.2197279
7 -0.13918735 -0.5980350 -1.1016172 -0.2847439
8  1.56235829  0.4744131  1.1063214  0.5207370
9 -0.84576242  0.1435361 -1.4697943 -1.1066756
10 -1.24986940 -1.7620736  0.1561537  3.6725506

# ... with 90 more rows

```

Find the median of each column

(hint: look up the *median* function in R)

Find the median of each row

```
median(df[[1]])
```

```
median(df[[2]])
```

```
median(df[[3]])
```

```
median(df[[4]])
```

Why `df[[1]]` instead of `df[1]`?

```
str(df[1])
```

```
'data.frame': 100 obs. of 1 variable:
```

```
$ col_a: num -0.0975 -0.0112 0.266 0.47 0.7709 ...
```

```
str(df[[1]])
```

```
num [1:100] -0.0975 -0.0112 0.266 0.47 0.7709 ...
```

Because data frames are lists of vectors, you can use `[[` to extract a column from data frames

Conditional Statements and Loops

Conditional Statements and Loops

- `if...else`
- `for`
- `while`

if...else

```
if (condition) {  
    statement  
  
} else {  
    statement  
  
}
```

if...else

```
x <- 6
```

```
if (x>0) {
```

```
  print("X is a positive number")
```

```
} else {
```

```
  print("X is a negative number")
```

```
}
```

```
[1] "X is a positive number"
```


for

```
for (name in vector) {  
  statement  
}
```

for

```
y <- c(1,2,3,4,5)
for (number in y) {
  print(number)
}
```

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

for

```
y <- c(1,2,3,4,5)
```

```
z <- vector()
```

```
for (number in y) {
```

```
  z[number] <- number + 1
```

```
  z
```

```
}
```

```
z
```

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

Three ways to loop over a vector

- loop over elements
- loop over numeric indices
- loop over the names

for (loop over elements)

```
y <- c(1,2,3,4,5)
```

```
z <- vector()
```

```
for (number in y) {
```

```
  z[number] <- number + 1
```

```
  z
```

```
}
```

```
z
```

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

for (loop over numeric indices)

```
y <- c(1,2,3,4,5)
```

```
z <- vector()
```

```
for (number in seq_along(y)) {
```

```
  z[number] <- number + 1
```

```
}
```

```
z
```

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

for (loop over the names)

```
y <- c(1,2,3,4,5)
```

```
for (number in names(y)) {
```

```
  z[number] <- number + 1
```

```
}
```

```
z
```

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

while

```
while condition {  
  statement  
}
```


while

```
x <- 1
```

```
while (x < 4) {
```

```
  print(x)
```

```
  x = x+1
```

```
}
```

while

```
x <- 1  
while (x < 4) {  
  print(x)  
  x = x+1  
}
```

[1] 1

[1] 2

[1] 3

```
bx <- replicate(5, runif(10), simplify = FALSE)
```

```
ry <- replicate(5, rpois(10,5)+1, simplify = FALSE)
```

Calculate the median of each element in bx

Calculate the mean of each element in ry

The Usual Way

```
median(bx[[1]])
```

```
median(bx[[2]])
```

```
mean(ry[[1]])
```

```
mean(ry[[2]])
```

```
out <- vector("double", length(bx))
```

```
for (i in seq_along(bx)) {
```

```
  out[i] <- median(bx[[i]])
```

```
}
```

```
out
```

```
out <- vector("double", length(ry))
```

```
for (i in seq_along(ry)) {
```

```
  out[i] <- mean(ry[[i]])
```

```
}
```

```
out
```

Functions

Functions

```
my_fun <- function(arg1, arg2) {  
  body  
}
```


Functions

```
f <- function(x) {
```

```
  x^2
```

```
}
```

Function Components

formals(), the list of arguments

body(), the code inside the function

environment(), the “map” of the location of the function’s variables

Function Components

`formals(f)`

`#$x`

`body(f)`

`#{`

`# x^2`

`#}`

`environment(f)`

`#<environment: R_GlobalEnv>`

Not true of primitive functions like `sum()`, written in C code

Function Scoping

```
f <- function() {  
  
  x <- 1  
  
  y <- 2  
  
  c(x, y)  
  
}
```

Function Scoping

```
f <- function() {
```

```
  x <- 1
```

```
  y <- 2
```

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

```
}
```

```
f()
```

```
[1] 1 2
```

Function Scoping

```
x <- 1  
  
h <- function() {  
  
  y <- 2  
  
  i <- function() {  
  
    z <- 3  
  
    c(x, y, z)  
  
  }  
  
  i()  
  
}
```

Function Scoping

```
x <- 1
```

```
h <- function() {
```

```
  y <- 2
```

```
  i <- function() {
```

```
    z <- 3
```

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

```
  }
```

```
  i()
```

```
}
```

```
h()
```

```
[1] 1 2 3
```

Practice


```
func_median <- function(x) {  
  out <- vector(mode = "double", length = length(x))  
  for (i in seq_along(x)) {  
    out[i] <- median(x[[i]])  
  }  
  out  
}
```

```
median_values <- func_median(ry)
```

```
func_mean <- function(x) {  
  out <- vector(mode = "double", length = length(x))  
  for (i in seq_along(x)) {  
    out[i] <- mean(x[[i]])  
  }  
  out  
}  
  
mean_values <- func_mean(bx)
```

```
func_median <- function(x) {  
  out <- vector(mode = "double", length = length(x))  
  for (i in seq_along(x)) {  
    out[i] <- median(x[[i]])  
  }  
  out  
}
```

```
func_mean <- function(x) {  
  out <- vector(mode = "double", length = length(x))  
  for (i in seq_along(x)) {  
    out[i] <- mean(x[[i]])  
  }  
  out  
}
```

```
element_func <- function(x, f) {  
  out <- vector(mode = "double", length = length(x))  
  for (i in seq_along(x)) {  
    out[i] <- f(x[[i]])  
  }  
  out  
}  
  
mean_values <- element_func(bx, mean)  
median_values <- element_func(ry, median)  
multiplication_values <- element_func(bx, function(x) {3*mean(x)})
```

```
data(mtcars)
```

```
names(mtcars)
```

Motor Trend Car Road Tests

Description

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Usage

```
mtcars
```

Format

A data frame with 32 observations on 11 variables.

- [, 1] mpg Miles/(US) gallon
- [, 2] cyl Number of cylinders
- [, 3] disp Displacement (cu.in.)
- [, 4] hp Gross horsepower
- [, 5] drat Rear axle ratio
- [, 6] wt Weight (1000 lbs)
- [, 7] qsec 1/4 mile time
- [, 8] vs V/S
- [, 9] am Transmission (0 = automatic, 1 = manual)
- [,10] gear Number of forward gears
- [,11] carb Number of carburetors

Get the mean of each column in the mtcars dataset

- using a function
- using lapply/sapply
- using map functions from the purrr package

Functionals

Functionals

Is a function that takes functions as an input and returns vector as output

Commonly used as alternatives for loops

lapply

lapply takes a function, applies to each element in a list, and returns a list

```
lapply(bx, mean)
```

```
lapply(ry, median)
```

```
lapply(by, function(x) { mean(x * 3)})
```

sapply

like lapply, but the output is simplified to produce an atomic vector

```
sapply(bx , mean)
```

```
sapply(ry, median)
```

```
sapply(ry, function(x) { mean(x * 3)})
```

Get the mean of each column in the mtcars dataset

- using a function
- using lapply/sapply
- using map functions from the purrr package

Case Study 1

purrr package in R

R now has a purrr package with functionals

`map_dbl(x, f)`

- 1) Loop over x like a for loop
- 2) Apply function f to each element
- 3) Return output

purrr package in R

```
install.packages("purrr")
```

```
library(purrr)
```

```
map_dbl(bx, mean)
```

```
[1] 0.5217805 0.6357506 0.6055213 0.5305683 0.6793977
```

```
map_dbl(ry, median)
```

```
[1] 6.0 7.0 4.5 8.0 5.5
```

Get the mean of each column in the mtcars dataset

- using a function
- using lapply/sapply
- using map functions from the purrr package

Case Study 2

Resources

R for Data Science: <http://r4ds.had.co.nz/>

Advanced R: <http://adv-r.had.co.nz/>

ggplot2: <http://ggplot2.org/>

The Art of R Programming

Tidy Data: <https://www.jstatsoft.org/article/view/v059i10>

Sources

Wickham, Hadley. Advanced R. Boca Raton, FL: CRC, 2015. Print.