Advanced R - Chapter 1 - Data Structures

Matthew Strimas-Mackey

# Data structures

## Quiz

1. What are the three properties of a vector, other than its contents?

* length
* type

1. What are the four common types of atomic vectors? What are the two rare types?

* character
* boolean
* numeric
* dimension

1. What are attributes? How do you get them and set them?
2. How is a list different from an atomic vector? How is a matrix different from a data frame?

* Only one datatype allowed in vector and matrix

1. Can you have a list that is a matrix? Can a data frame have a column that is a matrix?

* No

## Vectors

Vectores can be atomic vectors, i.e. all elements of the same type, or lists, with mixed types. They have 3 basic properties:

* Type, typeof(), what it is.
* Length, length(), how many elements it contains.
* Attributes, attributes(), additional arbitrary metadata.

v <- c(1,2,3)  
names(v) <- c('one', 'two', 'three')  
l <- list(v)  
  
typeof(v)

## [1] "double"

length(v)

## [1] 3

attributes(v)

## $names  
## [1] "one" "two" "three"

# Testing identity  
is.vector(v) # Vector with no attributes other than name

## [1] TRUE

is.atomic(v) # Atomic vector

## [1] TRUE

is.list(l) # List

## [1] TRUE

is.atomic(v) || is.list(v) # Either type of vector

## [1] TRUE

### Atomic vectors

Four common types: logical, integer, double (often called numeric), and character. Two rare types: complex and raw.

Atomic vectors are usually created with c(), short for combine:

# By default all numbers are stored as doubles  
dbl\_var <- c(1, 2.5, 4.5)  
# With the L suffix, you get an integer rather than a double  
int\_var <- c(1L, 6L, 10L)  
  
# Missing values specified with NA  
miss <- c(1, 2, 3, NA)  
# NA a logical by default, but always coerced to correct type, but can also specify explicitely  
c(1, 2, 3, NA\_real\_)

## [1] 1 2 3 NA

c(1L, 2L, 3L, NA\_integer\_)

## [1] 1 2 3 NA

c('1', '2', '3', NA\_character\_)

## [1] "1" "2" "3" NA

#### Types and tests

Given a vector, you can determine its type with typeof(), or check if it’s a specific type with an “is” function: is.character(), is.double(), is.integer(), is.logical(), or, more generally, is.atomic().

int\_var <- c(1L, 6L, 10L)  
typeof(int\_var)

## [1] "integer"

is.integer(int\_var)

## [1] TRUE

is.atomic(int\_var)

## [1] TRUE

dbl\_var <- c(1, 2.5, 4.5)  
typeof(dbl\_var)

## [1] "double"

is.double(dbl\_var)

## [1] TRUE

is.atomic(dbl\_var)

## [1] TRUE

# is.numeric() returns T for double or integer  
is.numeric(int\_var)

## [1] TRUE

is.numeric(dbl\_var)

## [1] TRUE

#### Coercion

When you attempt to combine different types they will be **coerced** to the most flexible type. Types from least to most flexible are: logical, integer, double, and character. If confusion is likely, explicitly coerce with as.character(), as.double(), as.integer(), or as.logical().

### Lists

Lists are vectors whose elements can be of any type, including lists. list() creates a list out of it’s elements, c() will combine lists together, unlist() turns a list into an atomic vector, making the required coercions.

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

## List of 4  
## $ : int [1:3] 1 2 3  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE  
## $ : num [1:2] 2.3 5.9

# Difference between list and c  
x <- list(list(1, 2), c(3, 4))  
y <- c(list(1, 2), c(3, 4))  
str(x)

## List of 2  
## $ :List of 2  
## ..$ : num 1  
## ..$ : num 2  
## $ : num [1:2] 3 4

str(y)

## List of 4  
## $ : num 1  
## $ : num 2  
## $ : num 3  
## $ : num 4

# unlist  
str(unlist(x))

## num [1:4] 1 2 3 4

str(unlist(z)) # Coercion to string

## chr [1:9] "1" "2" "3" "a" "TRUE" "FALSE" "TRUE" "2.3" "5.9"

### Exercises

1. What are the six types of atomic vector? How does a list differ from an atomic vector?

* double, integer, logical, character; complex, raw
* elements of a list can be of different types; atomic vectors are all of the same type

1. What makes is.vector() and is.numeric() fundamentally different to is.list() and is.character()?

* the first 2 will match 2 types of data, while the second will only match one each

1. Test your knowledge of vector coercion rules by predicting the output of the following uses of c():

c(1, FALSE) # c(1,0)

## [1] 1 0

c("a", 1) # c('a', '1')

## [1] "a" "1"

c(list(1), "a") # list(1, 'a')

## [[1]]  
## [1] 1  
##   
## [[2]]  
## [1] "a"

c(TRUE, 1L) # c(1L, 1L)

## [1] 1 1

1. Why do you need to use unlist() to convert a list to an atomic vector? Why doesn’t as.vector() work?

* a list is already a vector

1. Why is 1 == "1" true? Why is -1 < FALSE true? Why is "one" < 2 false?

* Coercion

1. Why is the default missing value, NA, a logical vector? What’s special about logical vectors? (Hint: think about c(FALSE, NA\_character\_).)

* logical is the least flexible type

## Attributes

All objects can have arbitrary additional attributes, used to store metadata about the object. Attributes can be thought of as a named list (with unique names). Attributes can be accessed individually with attr() or all at once (as a list) with attributes().

y <- 1:10  
names(y) <- 1:10  
attr(y, "my\_attribute") <- "This is a vector"  
attr(y, "my\_attribute")

## [1] "This is a vector"

str(attributes(y))

## List of 2  
## $ names : chr [1:10] "1" "2" "3" "4" ...  
## $ my\_attribute: chr "This is a vector"

str(y)

## Named int [1:10] 1 2 3 4 5 6 7 8 9 10  
## - attr(\*, "names")= chr [1:10] "1" "2" "3" "4" ...  
## - attr(\*, "my\_attribute")= chr "This is a vector"

#### Names

You can name a vector in three ways:

# When creating it  
c(a = 1, b = 2, c = 3)

## a b c   
## 1 2 3

# Modifying in place  
x <- 1:3; names(x) <- c("a", "b", "c"); x

## a b c   
## 1 2 3

setNames(1:3, c("a", "b", "c"))

## a b c   
## 1 2 3

### Factors

A factor is a vector that can contain only predefined values, and is used to store categorical data. Factors are built on top of integer vectors using two attributes: the class(), “factor”, which makes them behave differently from regular integer vectors, and the levels(), which defines the set of allowed values.

x <- factor(c("a", "b", "b", "a"))  
x

## [1] a b b a  
## Levels: a b

class(x)

## [1] "factor"

levels(x)

## [1] "a" "b"

# You can't use values that are not in the levels  
x[2] <- "c"

## Warning in `[<-.factor`(`\*tmp\*`, 2, value = "c"): invalid factor level, NA  
## generated

x

## [1] a <NA> b a   
## Levels: a b

Factors are useful when you know the possible values a variable may take, even if you don’t see all values in a given dataset. Using a factor instead of a character vector makes it obvious when some groups contain no observations:

sex\_char <- c("m", "m", "m")  
sex\_factor <- factor(sex\_char, levels = c("m", "f"))  
  
table(sex\_char)

## sex\_char  
## m   
## 3

table(sex\_factor)

## sex\_factor  
## m f   
## 3 0

### Exercises

1. An early draft used this code to illustrate structure():

* structure(1:5, comment = "my attribute")
* ## [1] 1 2 3 4 5
* But when you print that object you don’t see the comment attribute. Why? Is the attribute missing, or is there something else special about it? (Hint: try using help.)
* comment attributes are not printed by default

1. What happens to a factor when you modify its levels?

* f1 <- factor(letters)  
  levels(f1) <- rev(levels(f1))  
  levels(f1) <- 1:26
* the mapping between integers and levels is changed, so the vector is now labelled wrong

1. What does this code do? How do f2 and f3 differ from f1?

* f2 <- rev(factor(letters)); f2
* ## [1] z y x w v u t s r q p o n m l k j i h g f e d c b a  
  ## Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z
* f3 <- factor(letters, levels = rev(letters)); f3
* ## [1] a b c d e f g h i j k l m n o p q r s t u v w x y z  
  ## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
* First changes order of vector, second changes the order of the integers that are used to represent the levels

## Matrices and arrays

Adding a dim() attribute to an atomic vector allows it to behave like a multi-dimensional **array**. A special case of the array is the **matrix**, which has two dimensions. Matrices and arrays are created with matrix() and array(), or by using the assignment form of dim():

# Two scalar arguments to specify rows and columns  
a <- matrix(1:6, ncol = 3, nrow = 2); a; class(a)

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

## [1] "matrix"

# One vector argument to describe all dimensions  
b <- array(1:12, c(2, 3, 2)); b; class(b)

## , , 1  
##   
## [,1] [,2] [,3]  
## [1,] 1 3 5  
## [2,] 2 4 6  
##   
## , , 2  
##   
## [,1] [,2] [,3]  
## [1,] 7 9 11  
## [2,] 8 10 12

## [1] "array"

# You can also modify an object in place by setting dim()  
c <- 1:6; dim(c) <- c(3, 2); c

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

dim(c) <- c(2, 3); c

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

length() and names() have high-dimensional generalisations:

* length() generalises to nrow() and ncol() for matrices, and dim() for arrays.
* names() generalises to rownames() and colnames() for matrices, and dimnames(), a list of character vectors, for 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

## a b c  
## A 1 3 5  
## B 2 4 6

length(b)

## [1] 12

dim(b)

## [1] 2 3 2

dimnames(b) <- list(c("one", "two"), c("a", "b", "c"), c("A", "B"))  
b

## , , A  
##   
## a b c  
## one 1 3 5  
## two 2 4 6  
##   
## , , B  
##   
## a b c  
## one 7 9 11  
## two 8 10 12

### Exercises

1. What does dim() return when applied to a vector?

* NULL

1. If is.matrix(x) is TRUE, what will is.array(x) return?

* TRUE

1. How would you describe the following three objects? What makes them different to 1:5?

* x1 <- array(1:5, c(1, 1, 5)); x1
* ## , , 1  
  ##   
  ## [,1]  
  ## [1,] 1  
  ##   
  ## , , 2  
  ##   
  ## [,1]  
  ## [1,] 2  
  ##   
  ## , , 3  
  ##   
  ## [,1]  
  ## [1,] 3  
  ##   
  ## , , 4  
  ##   
  ## [,1]  
  ## [1,] 4  
  ##   
  ## , , 5  
  ##   
  ## [,1]  
  ## [1,] 5
* x2 <- array(1:5, c(1, 5, 1)); x2
* ## , , 1  
  ##   
  ## [,1] [,2] [,3] [,4] [,5]  
  ## [1,] 1 2 3 4 5
* x3 <- array(1:5, c(5, 1, 1)); x3
* ## , , 1  
  ##   
  ## [,1]  
  ## [1,] 1  
  ## [2,] 2  
  ## [3,] 3  
  ## [4,] 4  
  ## [5,] 5
* 3d arrays, unlike 1:5 they have a dimension attribute

## Data frames

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

### Creation

You create a data frame using data.frame(), which takes named vectors as input:

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

## 'data.frame': 3 obs. of 2 variables:  
## $ x: int 1 2 3  
## $ y: Factor w/ 3 levels "a","b","c": 1 2 3

# Notice that, by default, strings were converted to factors. To suppress:  
df <- data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = F)  
str(df)

## 'data.frame': 3 obs. of 2 variables:  
## $ x: int 1 2 3  
## $ y: chr "a" "b" "c"

### Testing and coercion

Because a data.frame is an S3 class, its type reflects the underlying vector used to build it: the list. To check if an object is a data frame, use class() or test explicitly with is.data.frame():

typeof(df)

## [1] "list"

class(df)

## [1] "data.frame"

is.data.frame(df)

## [1] TRUE

### Combining data frames

You can combine data frames using cbind() and rbind():

# Combine columns  
cbind(df, data.frame(z = 3:1))

## x y z  
## 1 1 a 3  
## 2 2 b 2  
## 3 3 c 1

# Combine Rows  
rbind(df, data.frame(x = 10, y = "z"))

## x y  
## 1 1 a  
## 2 2 b  
## 3 3 c  
## 4 10 z

It’s a common mistake to try and create a data frame by cbind()ing vectors together. This doesn’t work because cbind() will create a matrix unless one of the arguments is already a data frame. Instead use data.frame() directly:

bad <- data.frame(cbind(a = 1:2, b = c("a", "b")))  
str(bad)

## 'data.frame': 2 obs. of 2 variables:  
## $ a: Factor w/ 2 levels "1","2": 1 2  
## $ b: Factor w/ 2 levels "a","b": 1 2

good <- data.frame(a = 1:2, b = c("a", "b"),  
 stringsAsFactors = FALSE)  
str(good)

## 'data.frame': 2 obs. of 2 variables:  
## $ a: int 1 2  
## $ b: chr "a" "b"

### Exercises

1. What attributes does a data frame possess?

* names, rownames, class

1. What does as.matrix() do when applied to a data frame with columns of different types?

* Coercion

1. Can you have a data frame with 0 rows? What about 0 columns?

* yes