3 Working with vectors

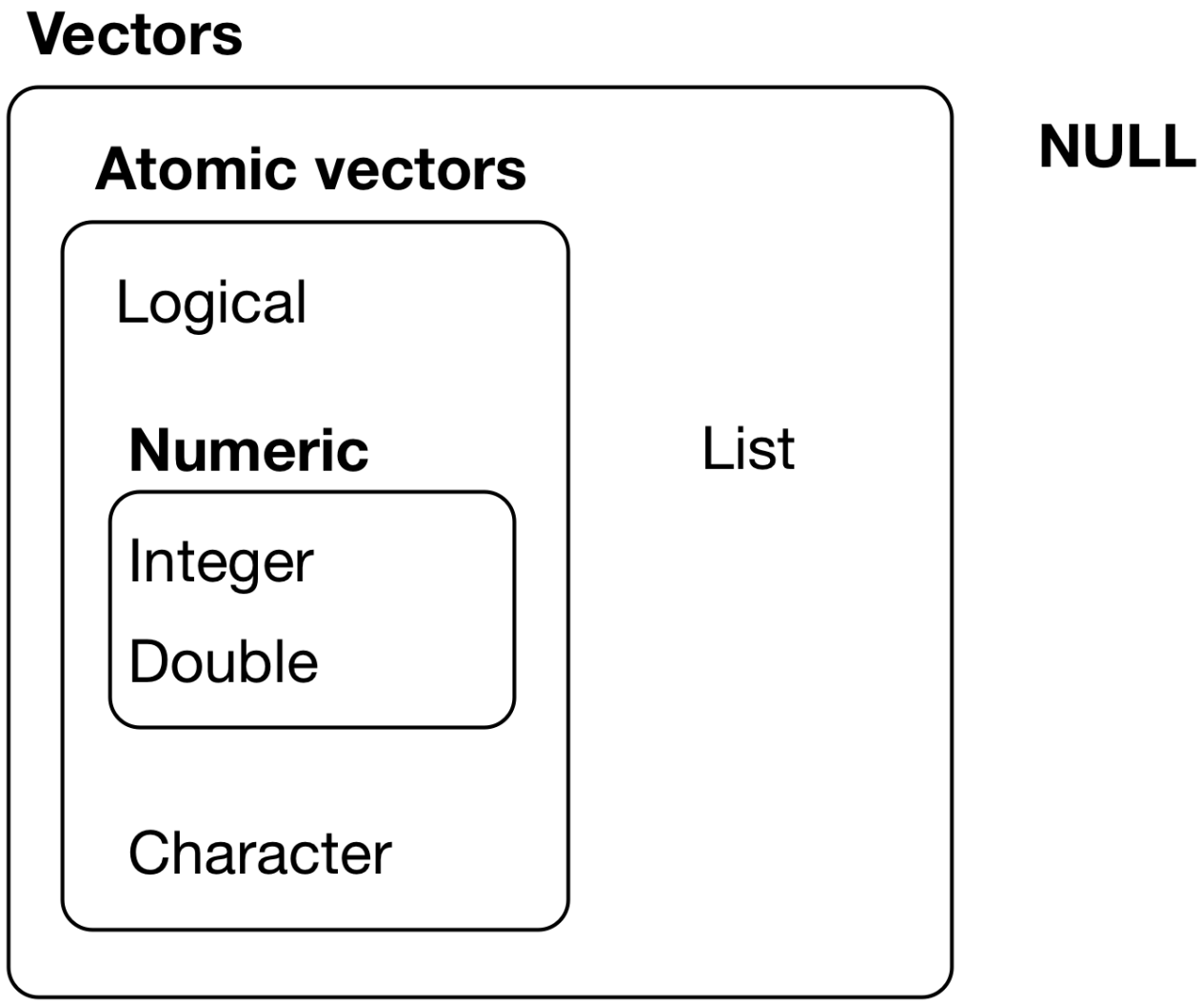
(AST230) R for Data Science

Md Rasel Biswas

Vectors

- A variable or an R object with more than one value is known as a **vector**, and in R, there are two types of vectors: **atomic vectors** and **lists**
  - Atomic vector consists of the same type of elements, e.g. all doubles or all characters.
  - List can have elements of different data types, i.e. one element of a list could be a numeric value, and the other could be a character value. [More on lists later]
- Most of the time, atomic vectors are just called vectors (we've already done this in the last section, and we'll keep doing it throughout the course!).
- While lists are also technically vectors, we like to keep things clear by simply calling them "lists." It makes things easier to understand

Vectors


Vectors
Atomic vectors
Logical
Numeric
Integer
Double
Character
List
NULL

Vector operations:

Combining vectors

- The function `c()` merges an arbitrary number of vectors to one vector

```
x <- c(10, 15)
c(1:5, x, 100, x)
```

```
 [1]   1   2   3   4   5  10  15 100  10  15
```

- R will quite happily do arithmetic operations with vectors as well

```
x+3
```

```
[1] 13 18
```

```
x/3
```

```
[1] 3.333333 5.000000
```

Arithmetic operations

- Functions work on vectors as they do on individual objects.

```
log(x)
```

```
[1] 2.302585 2.708050
```

- Arithmetic operations can also be done with two vectors

```
x <- c(10, 15, 20)
y <- c(1, 2, 3)
x*y
```

```
[1] 10 30 60
```

The recycling rule of vectors

- It is not necessary to have vectors of the same length in an expression
- If two vectors in an expression are not of the same length then the shorter one will be repeated until it has the same length as the longer one.

```
m <- c(1, 2, 3); n <- c(3, 4, 5, 7, 50)
m*n
```

```
Warning in m * n: longer object length is not a multiple of shorter object
length
```

```
[1]   3   8  15   7 100
```

Some vector functions:

Some vector functions

```
x <- c(10, 8, 9, 16, 9, 8, 16)
sort(x)
```

```
[1]  8  8  9  9 10 16 16
```

```
order(x)
```

```
[1] 2 6 3 5 1 4 7
```

```
table(x)
```

```
x
 8  9 10 16
 2  2  1  2
```

```
unique(x)
```

```
[1] 10  8  9 16
```

Summary statistics on vectors

Function	Example	Output
`sum()`, `prod()`	`sum(1:10)`	55
`min()`, `max()`	`min(1:10)`	1
`mean()`, `median()`	`median(1:10)`	5.5
`sd()`, `var()`	`sd(1:10)`	3.0276504
`quantile()`	`quantile(1:10)`	1, 3.25, 5.5, 7.75, 10

- Check the help pages of the R functions related to these summary statistics

Vector indexing:

Extracting elements of vectors:

- To extract (also known as indexing or subscripting) one or more values (more generally known as elements) from a vector, we use the square bracket `[ ]` notation

Indexing vectors with `[]` (Positional indexing)

A vector of the age of five children

```
age <- c(11, 9, 8, 10, 5)
age
```

```
[1] 11  9  8 10  5
```

Age of a specific child, say the third child

```
age[3]
```

```
[1] 8
```

Age of several children

```
age[c(2, 3, 5)]
```

```
[1] 9 8 5
```

```
age[c(3, 5, 2, 2)]
```

```
[1] 8 5 9 9
```

```
age[-c(1, 3)]
```

```
[1]  9 10  5
```

Note

The positional index starts at 1 rather than 0 like some other programming languages (e.g. C, Python)

Indexing vectors with `[]` (Logical indexing)

```
age
```

```
[1] 11  9  8 10  5
```

```
age >= 8
```

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

Number of children with age 8 years or more

```
sum(age >= 8)
```

```
[1] 4
```

Select the observations greater than 8 years

```
age[age >= 8]
```

```
[1] 11  9  8 10
```

Indexing vectors with [] (Logical indexing)

```
age
```

```
[1] 11  9  8 10  5
```

Children with age 11 or 8 years

```
age[age %in% c(8, 11)]
```

```
[1] 11  8
```

Children with ages not equal to 11 or 8 years

```
age[!age %in% c(8, 11)]
```

```
[1]  9 10  5
```

Indexing vectors with [] (Logical indexing)

```
age
```

```
[1] 11  9  8 10  5
```

Observations with age greater than 9 or less than 8

```
age[age > 9 | age < 8]
```

```
[1] 11 10  5
```

Observations with ages between 8 to 10 inclusive

```
age[age >= 8 & age <= 10]
```

```
[1]  9  8 10
```

The mean age of observations between 8 to 10 inclusive

```
mean(age[age >= 8 & age <= 10])
```

```
[1] 9
```

Replacing elements

Changing values of a vector

```
age <- c(11, 9, 8, 10, 5)
age1 <- age
age1
```

```
[1] 11  9  8 10  5
```

```
age2 <- age
age2
```

```
[1] 11  9  8 10  5
```

Change the first child's age to 15

```
age1[1] <- 15
age1
```

```
[1] 15  9  8 10  5
```

Change it to 20 if it is greater than 9

```
age2[age2 > 9] <- 20
age2
```

```
[1] 20  9  8 20  5
```

Ordering elements

```
age <- c(11, 9, 8, 10, 5)
sort(age)
```

```
[1]  5  8  9 10 11
```

```
sort(age, decreasing = TRUE)
```

```
[1] 11 10  9  8  5
```

```
order(age)
```

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

```
age[order(age)] #equivalent to sort()
```

```
[1]  5  8  9 10 11
```

Exercise 3

The following code generates a vector `nage` of size 1000.

```
set.seed(100)
nage <- sample(x = 30:80, size = 1000, replace = T)
```

- Show that the number of observations
  i. greater than 70 is 176
  ii. less than 40 is 185
  iii. equal to 39 is 19
  iv. greater than 77 or less than 35 is 140
  v. between 50 and 55 (inclusive) is 110
- What percentage of observations lies between 70 to 75 (inclusive)?

Missing values

- In R, missing values are coded as `NA` meaning 'Not available'
- Most of the R functions return missing value (i.e. `NA`) if any input vector contains a missing value

```
5 + NA
```

```
[1] NA
```

```
mval <- c(12:15, NA)
mval
```

```
[1] 12 13 14 15 NA
```

```
mean(mval)
```

```
[1] NA
```

Indexing vectors with [] (Logical indexing)

- Most of the R functions have an argument `na.rm`, which takes a logical value to include (or exclude) the missing value in (from) the calculation

```
mval
```

```
[1] 12 13 14 15 NA
```

```
mean(mval)
```

```
[1] NA
```

```
mean(mval, na.rm = T)
```

```
[1] 13.5
```

```
is.na(mval)
```

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

```
sum(!is.na(mval))
```

```
[1] 4
```

Coercion:

Coercion

- In R, atomic vectors are homogeneous, i.e., all elements of an atomic vector will be of the same data type
- If you attempt to create an atomic vector with more than one data type, e.g. `nvec <- c(1, 2, "all")`, then R will create an atomic vector, i.e. all elements of `nvec` will be of the same data type, which is known as **coercion**

```
nvec <- c(1, 2, "all")
nvec
```

```
[1] "1"   "2"   "all"
```

```
typeof(nvec)
```

```
[1] "character"
```

Coercion

- In R, coercion occurs in the following (decreasing) order of precedence

1. Character 2. Numeric 3. Integer 4. Logical

```
c(1L, 5L, FALSE, TRUE)
```

```
[1] 1 5 0 1
```

```
typeof(c(1L, 2L, 4.0))
```

```
[1] "double"
```

```
typeof(c(2.0, "new"))
```

```
[1] "character"
```

```
"A" > 10
```

```
[1] TRUE
```

Explicit coercion

- Objects can be explicitly coerced from one type to another using `as.**` functions, if available

```
x <- 0:5
typeof(x)
```

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

```
as.numeric(x)
```

```
[1] 0 1 2 3 4 5
```

```
as.logical(x)
```

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

```
as.character(x)
```

```
[1] "0" "1" "2" "3" "4" "5"
```

Explicit coercion

- Nonsensical coercion results in NAs

```
x <- c("a", "b", "0", "522")
as.numeric(x)
```

```
Warning: NAs introduced by coercion
```

```
[1]  NA  NA   0 522
```

```
as.logical(x)
```

```
[1] NA NA NA NA
```

Attributes:

Attributes

- An attribute is a piece of information that you can attach to an atomic vector (or any R object) and it won't affect any of the values in the object, and it will usually not appear when displaying the object.
- Attributes are metadata and R will normally ignore it, but some R functions will check for specific attributes
- Atomic vectors can be transformed into some other important R data structures, e.g., matrices, arrays, factors, or date-times by adding attributes
- Attributes can be retrieved and modified by `attr()` or `attributes()`

Attributes

```
# an atomic vector that initially has no attributes
die <- 1:6
attributes(die)
```

```
NULL
```

```
# Setting an attribute named "x"
attr(die, "x") <- "abcd"
attributes(die)
```

```
$x
[1] "abcd"
```

```
# Now it has an attribute
die
```

```
[1] 1 2 3 4 5 6
attr(,"x")
[1] "abcd"
```

Attributes

Two mostly used attributes are:

1. **names**, a character vector giving each element a name.
2. **dim**, short for dimensions, an integer vector, used to turn vectors into matrices or arrays.

1. Names

`names` is one of the common attributes of an R object. We can set names to an atomic vector in various ways. Two of them are:

```
# Naming vector when creating it:
x <- c(a = 1, b = 2, c = 3)
x
```

```
a b c
1 2 3
```

```
# Naming vector by assigning a character vector to names()
x <- 1:3
names(x) <- c("a", "b", "c")
x
```

```
a b c
1 2 3
```

1. Names

```
# Subsetting vector by names
x[["a"]]
```

```
[1] 1
```

```
# Attributes are preserved by most operations
y <- x^2 + 1
names(y)
```

```
[1] "a" "b" "c"
```

```
# Removing the attributes
names(x) <- NULL
x
```

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

2. Dimensions

- An atomic vector can be transformed into n-dimensional array by adding a dimension attribute with `dim`

```
# Adding the dim attribute
die1 <- 1:6
dim(die1) <-  c(2, 3)

# it's a matrix now
die1
```

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

```
# Modifying the dim attribute
dim(die1) <- c(3, 2)
die1
```

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

```
# See the attributes
attributes(die1)
```

```
$dim
[1] 3 2
```

2. Dimensions

```
# Creating 3 dimensional array
# by adding dim attributes
die <- 1:6
dim(die) <- c(1, 2, 3)
die
```

```
, , 1

     [,1] [,2]
[1,]    1    2

, , 2

     [,1] [,2]
[1,]    3    4

, , 3

     [,1] [,2]
[1,]    5    6
```

```
# Creating 3 dimensional array
# by adding dim attributes
die12 <- 1:12
dim(die12) <- c(2, 2, 3)
die12
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

2. Dimensions

- R will always use the first value in `dim` for the number of rows and the second value for the number of columns
- R always fills up each matrix by columns, instead of by rows
- R functions `matrix()` and `array()` can be used to control how the columns and rows of a matrix will be arranged (More on next section)