Data Structures

STAT 133

Gaston Sanchez

`github.com/ucb-stat133/stat133-fall-2016`

Data Types and Structures

To make the best of the R language, you'll need a strong understanding of the basic **data types** and **data structures** and how to operate on them.

Data Structures

There are various data structures in R (we'll describe them in detail later):

- vectors
- matrices (2d arrays)
- arrays (in general)
- factors
- lists
- data frames

Vectors

Vectors

- A vector is the most basic data structure in R
- Vectors are contiguous cells containing data
- Can be of any length (including zero)
- R has five basic type of vectors:
  integer, double, complex, logical, character

Vectors

The most simple type of vectors are scalars or single values:

```
# integer
x <- 1L
# double (real)
y <- 5
# complex
z <- 3 + 5i
# logical
a <- TRUE
# character
b <- "yosemite"
```

Data modes

- A **double** vector stores regular (i.e. real) numbers
- An **integer** vector stores integers (no decimal component)
- A **character** vector stores text
- A **logical** vector stores `TRUE`'s and `FALSE`'s values
- A **complex** vector stores complex numbers

Data Types (or modes)

value	example	mode	storage
integer	`1L, 2L`	`numeric`	`integer`
real	`1, -0.5`	`numeric`	`double`
complex	`3 + 5i`	`complex`	`complex`
logical	`TRUE, FALSE`	`logical`	`logical`
character	`"hello"`	`character`	`character`

Special Values

There are some special values

- `NULL` is the null object (it has length zero)
- Missing values are referred to by the symbol `NA`
- `Inf` indicates positive infinite
- `-Inf` indicates negative infinite
- `NaN` indicates Not a Number

Vectors

The function to create a vector from individual values is `c()`, short for **concatenate**:

```
# some vectors
x <- c(1, 2, 3, 4, 5)

y <- c("one", "two", "three")

z <- c(TRUE, FALSE, FALSE)
```

Separate each element by a comma

Atomic Vectors

- vectors are **atomic** structures
- the values in a vector must be ALL of the same type
- either all integers, or reals, or complex, or characters, or logicals
- you cannot have a vector of different data types

Atomic Vectors

If you mix different data values, R will **implicitly** coerce them so they are all of the same type

```
# mixing numbers and characters
x <- c(1, 2, 3, "four", "five")
x

## [1] "1"    "2"    "3"    "four" "five"

# mixing numbers and logical values
y <- c(TRUE, FALSE, 3, 4)
y

## [1] 1 0 3 4
```

Atomic Vectors

```r
# mixing numbers and logical values
z <- c(TRUE, FALSE, "TRUE", "FALSE")
z
```

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

```r
# mixing integer, real, and complex numbers
w <- c(1L, -0.5, 3 + 5i)
w
```

```
## [1]  1.0+0i -0.5+0i  3.0+5i
```

How does R coerce data types?

R follows two basic rules of implicit coercion

If a character is present, R will coerce everything else to characters

If a vector contains logicals and numbers, R will convert the logicals to numbers (`TRUE` to 1, `FALSE` to 0)

Coercion functions

Coercion Functions

R provides a set of **explicit** coercion functions that allow us to "convert" one type of data into another

- `as.character()`
- `as.numeric()`
- `as.logical()`

Conversion between types

from	to	function	conversions
logical	numeric	`as.numeric`	`FALSE` $\rightarrow$ `0` `TRUE` $\rightarrow$ `1`
logical	character	`as.character`	`FALSE` $\rightarrow$ `"FALSE"` `TRUE` $\rightarrow$ `"TRUE"`
character	numeric	`as.numeric`	`"1", "2"` $\rightarrow$ `1, 2` `"A"` $\rightarrow$ `NA`

character	logical	`as.logical`	`"FALSE"` $\rightarrow$ `FALSE`
			`"TRUE"` $\rightarrow$ `TRUE`
			other $\rightarrow$ `NA`

numeric	logical	`as.logical`	`0` $\rightarrow$ `FALSE` *other* $\rightarrow$ `1`
numeric	character	`as.character`	`1, 2` $\rightarrow$ `"1", "2"`

Properties of Vectors

- all vectors have a length
- vector elements can have associated names
- vectors are objects of class `"vector"`
- vectors have a mode (storage mode)

Properties of Vectors

```r
# vector with named elements
x <- c(a = 1, b = 2.5, c = 3.7, d = 10)
x
```

```
##    a    b    c    d
##  1.0  2.5  3.7 10.0
```

```r
length(x)
```

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

```r
mode(x)
```

```
## [1] "numeric"
```

Matrices and Arrays

From Vectors to Arrays

We can transform a vector in an **n-dimensional** array by giving it a dimensions attribute `dim`

```
# positive: from 1 to 8
x <- 1:8

# adding 'dim' attribute
dim(x) <- c(2, 4)
x

##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

From Vectors to Arrays

- a vector can be given a `dim` attribute
- a `dim` attribute is a numeric vector of length `n`
- R will reorganize the elements of the vector into `n` dimensions
- each dimension will have as many rows (or columns, etc.) as the `n`-th value of the `dim` vector

From Vectors to Arrays

```
# dim attribute with 3 dimensions
dim(x) <- c(2, 2, 2)
x
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

From Vector to Matrix

A `dim` attribute of length 2 will convert a vector into a matrix

```
# vector to matrix
A <- 1:8
class(A)

## [1] "integer"

dim(A) <- c(2, 4)
class(A)

## [1] "matrix"
```

When using `dim()`, R always fills up each matrix by rows.

From Vector to Matrix

To have more control about how a matrix is filled, (by rows or columns), we use the `matrix()` function:

```r
# vector to matrix (by rows)
A <- 1:8

matrix(A, nrow = 2, ncol = 4, byrow = TRUE)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

Matrices

- Matrices store values in a two-dimensional array
- To create a matrix, give a vector to `matrix()` and specify number of rows and columns
- you can also assign row and column names to a matrix

Creating a Matrix

Exercise: create the following matrix

```
##      [,1]        [,2]
## [1,] "Harry"     "Potter"
## [2,] "Ron"       "Weasley"
## [3,] "Hermione"  "Granger"
```

Creating a Matrix

Solution:

```
# vector of names
hp <- c("Harry", "Potter", "Ron", "Weasley",
        "Hermione", "Granger")

# matrix filled up by rows
matrix(hp, nrow = 3, byrow = TRUE)

##      [,1]       [,2]
## [1,] "Harry"    "Potter"
## [2,] "Ron"      "Weasley"
## [3,] "Hermione" "Granger"
```

Arrays

- Arrays store values in an n-dimensional array
- To create an array, give a vector to `array()` and specify number of dimensions
- you can also assign dim-names to an array

Creating an Array

```
ar <- array(c(1:4, 5:8, 9:12), dim = c(2, 2, 3))
ar
```

```
## , , 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
```

Factors

Factors

- A similar structure to vectors are **factors**
- factors are used for handling categorical (i.e. qualitative) data
- they are represented as objects of class `"factor"`
- internally, factors are stored as integers
- factors behave much like vectors (but they are not vectors)

Factors

To create a factor we use the function `factor()`

```
# factor
cols <- c("blue", "red", "blue", "gray", "red")
cols <- factor(cols)
cols

## [1] blue red  blue gray red
## Levels: blue gray red
```

The different values in a factor are called **levels**

So far ...

- Vectors, matrices, and arrays are atomic structures (they can only store one type of data)
- Many operations in R need atomic structures to make sure all values are of the same mode
- In real life, however, many datasets contain multiple types of information
- R provides other data structures to store different types of data

Lists

Lists

- Lists are the most general class of data container
- Like vectors, lists group data into a one-dimensional set
- Unlike vectors, lists can store all kinds of objects
- Lists can be of any length
- Elements of a list can be named, or not

Lists

To create a list we use the function `list()`. The `list()` function creates a list the same way `c()` creates a vector:

```
lfriends <- list("Harry", "Ron", "Hermione")
lfriends

## [[1]]
## [1] "Harry"
##
## [[2]]
## [1] "Ron"
##
## [[3]]
## [1] "Hermione"
```

Lists

Lists can contain any type of data object (even other lists):

```
lst <- list(
  c("Harry", "Ron", "Hermione"),
  matrix(1:6, nrow = 2, ncol = 3),
  factor(c("yes", "no", "no", "no", "yes")),
  lfriends
)
```

Lists

Elements in a list can be named:

```
list(
  first = c("Harry", "Ron", "Hermione"),
  second = matrix(1:6, nrow = 2, ncol = 3),
  third = factor(c("yes", "no", "no", "no", "yes")),
  fourth = lfriends
)
```

Creating a List

Exercise: create a list with your first name, middle name, and last name

```
## $first
## [1] "Gaston"
##
## $middle
## NULL
##
## $last
## [1] "Sanchez"
```

Data Frames

Data Frames

Data Frame

A `data.frame` is the primary data structure that R provides for handling tabular data sets (e.g. spreadsheet like).

Function `data.frame()`

The `data.frame()` function allows us to create data frames

Lists

- Data frames are the two-dimensional version of a list
- They are the conventional data storage structure for data analysis
- A data frame is displayed like a table (or matrix)
- A data frame is stored as a list

Creating a Data Frame

```
# creating a data frame (manually)
elements <- data.frame(
  name = c('hydrogen', 'nitrogen', 'oxygen'),
  symbol = c('H', 'N', 'O'),
  number = c(1, 7, 8)
)

elements
```

```
##       name symbol number
## 1 hydrogen      H      1
## 2 nitrogen      N      7
## 3   oxygen      O      8
```

(vectors defined inside the data frame function)

Creating a Data Frame

- Give `data.frame()` any number of vectors, each separated with a comma
- Each vector should be set equal to a name that describes the vector
- `data.frame()` will turn each vector into a column of the new data frame
- All vectors should be of the same length
- By default, `data.frame()` converts strings into factors

Creating a Data Frame

Create the following data frame:

```
##        state abbreviation    capital   area
## 1 California           CA Sacramento 163707
## 2   New York           NY     Albany  54475
## 3      Texas           TX     Austin 268601
```

Creating a Data Frame

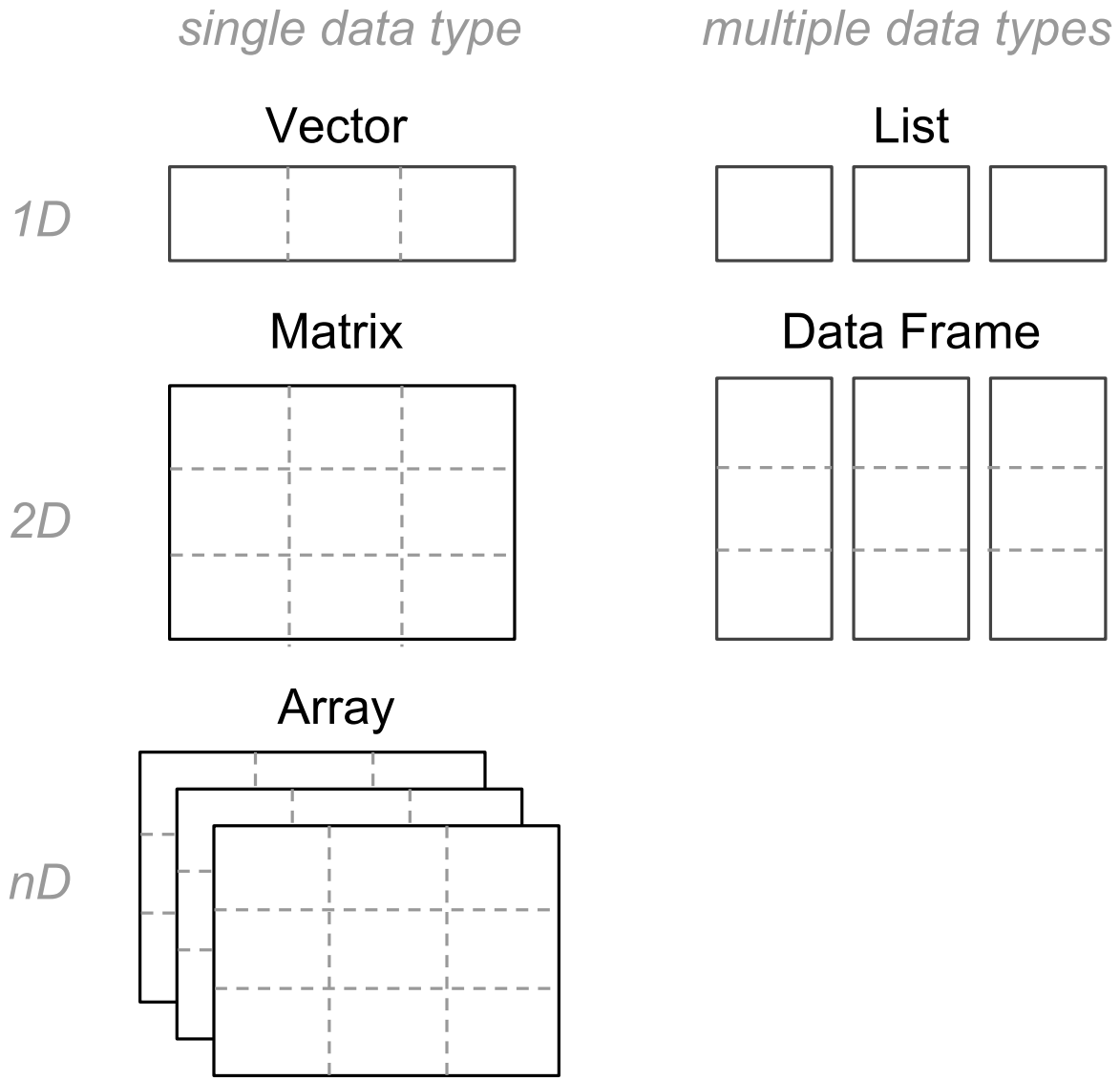
Solution:

```
states <- data.frame(
  state = c('California', 'New York', 'Texas'),
  abbreviation = c('CA', 'NY', 'TX'),
  capital = c('Sacramento', 'Albany', 'Austin'),
  area = c(163707, 54475, 268601),
  stringsAsFactors = FALSE
)

states

##        state abbreviation    capital   area
## 1 California           CA Sacramento 163707
## 2   New York           NY     Albany  54475
## 3      Texas           TX     Austin 268601
```

R common data structures



Selecting Values

Selecting Values

Data manipulation requires you to learn how to select and retrieve values from each of the data structures

Notation System

Notation system to extract values from R objects

- to extract values use brackets: `[ ]`
- inside the brackets specify indices
- use as many indices as dimensions in the object
- each index is separated by comma
- indices can be numbers, logicals, or names

Values of Vectors

```r
# some vector
vec <- 1:5

# adding names
names(vec) <- letters[1:5]

vec
```

```
## a b c d e
## 1 2 3 4 5
```

Extracting values with numbers

```
# first element
vec[1]

## a
## 1

# third element
vec[3]

## c
## 3

# fifth element
vec[5]

## e
## 5
```

Extracting values with positive vectors

```
# range 1 to 3
vec[1:3]

## a b c
## 1 2 3

# vector of positive numbers
vec[c(1, 3, 4)]

## a c d
## 1 3 4
```

Extracting values with negative numbers

```
# all values except the first one
vec[-1]

## b c d e
## 2 3 4 5

# all values except 2nd and 4th
vec[-c(2, 4)]

## a c e
## 1 3 5
```

Extracting values with logicals

```
# first element
vec[c(TRUE, FALSE, FALSE, FALSE, FALSE)]

## a
## 1

# 4th and 5th elements
vec[c(FALSE, FALSE, FALSE, TRUE, TRUE)]

## d e
## 4 5

# logical negation (2nd and 4th)
vec[!c(TRUE, FALSE, TRUE, FALSE, TRUE)]

## b d
## 2 4
```

Extracting values with names

Since `vec` has names, we can use characters to extract named values

```
# element 'a'
vec['a']

## a
## 1

# elements 'b' and 'e'
vec[c('b', 'e')]

## b e
## 2 5
```

Extracting values with names

You can use repeated indices:

```
# element 2 (3-times)
vec[c(2, 2, 2)]

## b b b
## 2 2 2

# element 'a' (four times)
vec[c('a', 'a', 'a', 'a')]

## a a a a
## 1 1 1 1
```

Extracting values of two-dimensional objects

```
states

##        state abbreviation    capital   area
## 1 California           CA Sacramento 163707
## 2   New York           NY     Albany  54475
## 3      Texas           TX     Austin 268601
```

Extracting values of two-dimensional objects

Extracting a single cell

```
# value in row 1, and column 1
states[1,1]

## [1] "California"

# value in row 3, and column 3
states[3,3]

## [1] "Austin"
```

Extracting values of two-dimensional objects

To extract just rows, leave the column index empty

```
# values in row 1
states[1, ]

##        state abbreviation    capital   area
## 1 California           CA Sacramento 163707

# values in rows 1 to 2
states[1:2, ]

##        state abbreviation    capital   area
## 1 California           CA Sacramento 163707
## 2   New York           NY     Albany  54475
```

Extracting values of two-dimensional objects

To extract just columns, leave the row index empty

```
# values in column 1
states[ , 1]

## [1] "California" "New York"   "Texas"

# values in columns 2 to 4
states[ , 2:4]

##   abbreviation    capital   area
## 1           CA Sacramento 163707
## 2           NY     Albany  54475
## 3           TX     Austin 268601
```

Extracting values of two-dimensional objects

Extracting rows with negative indices

```
# excluding 3rd row
states[-3, ]

##        state abbreviation    capital   area
## 1 California           CA Sacramento 163707
## 2   New York           NY     Albany  54475

# excluding 1st and 3rd row
states[-c(1, 3), ]

##      state abbreviation capital  area
## 2 New York           NY  Albany 54475
```

Extracting values of two-dimensional objects

Extracting columns with negative indices

```
# excluding 2nd column
states[ , -2]

##        state    capital   area
## 1 California Sacramento 163707
## 2   New York     Albany  54475
## 3      Texas     Austin 268601

# excluding 1st and 2dn columns
states[ , -c(1, 2)]

##      capital   area
## 1 Sacramento 163707
## 2     Albany  54475
## 3     Austin 268601
```

Extracting values of two-dimensional objects

You can use repetition of indices

```
# values in column 1
states[ , c(1, 1)]

##        state    state.1
## 1 California California
## 2   New York   New York
## 3      Texas      Texas

# values in columns 2 to 4
states[c(2, 2, 3), ]

##        state abbreviation capital   area
## 2   New York           NY  Albany  54475
## 2.1 New York           NY  Albany  54475
## 3      Texas           TX  Austin 268601
```

Extracting values of two-dimensional objects

Extracting values with logicals

```
# exclude 3rd row
states[c(TRUE, TRUE, FALSE), ]

##        state abbreviation    capital   area
## 1 California           CA Sacramento 163707
## 2   New York           NY     Albany  54475

# exclude 3rd column
states[  , c(TRUE, TRUE, FALSE, TRUE)]

##        state abbreviation   area
## 1 California           CA 163707
## 2   New York           NY  54475
## 3      Texas           TX 268601
```

Extracting values of two-dimensional objects

Extracting columns by name

```
# column 'state'
states[ , 'state']

## [1] "California" "New York"   "Texas"

# columns 'state' and 'abbreviation'
states[ , c('state', 'abbreviation')]

##        state abbreviation
## 1 California           CA
## 2   New York           NY
## 3      Texas           TX
```

Extracting values of two-dimensional objects

When you select one column from a two-dimensional array, R will return a vector. To get a column output, use the argument `drop = FALSE`

```
# columns 'state' and 'abbreviation'
states[ , 1, drop = FALSE]

##         state
## 1 California
## 2   New York
## 3      Texas
```

Dollar signs and Double Brackets

Dollar signs

R lists and data frames obey an optional second system of notation for extracting values: using the dollar sign $

Dollar signs with data frames

The dollar sign $ notation works for selecting a column of a data frame using its name

```
states$state
```

```
## [1] "California" "New York"   "Texas"
```

```
states$abbreviation
```

```
## [1] "CA" "NY" "TX"
```

```
states$capital
```

```
## [1] "Sacramento" "Albany"     "Austin"
```

Dollar signs with data frames

You don't need to use quote marks, but you can if you want

```
states$'state'
```

```
## [1] "California" "New York"   "Texas"
```

```
states$"abbreviation"
```

```
## [1] "CA" "NY" "TX"
```

```
states$`capital`
```

```
## [1] "Sacramento" "Albany"     "Austin"
```

Dollar signs with lists

Elements in a named list can also be extracted with the dollar sign:

```
lst <- list(numbers = 1:3, letter = c('A', 'B', 'C'))

lst$numbers

## [1] 1 2 3

lst$letters

## NULL
```

Double brackets

In addition, lists (and data frames) accept a third type of notation that uses double brackets: `[[ ]]`

```
lst[[1]]

## [1] 1 2 3

lst[[2]]

## [1] "A" "B" "C"
```

Double brackets

Double brackets are used when we want to get access to the individual elements; use double brackets followed by single brackets

```
lst[[2]]

## [1] "A" "B" "C"

lst[[2]][3]

## [1] "C"
```

Elements-Extraction Notation System

object	notation	example
vector	`[ ]`	`v[1:5]`
factor	`[ ]`	`g[1:5]`
matrix	`[ , ]`	`m[1:5, 1:3]`
array	`[ , , ]`	`arr[1, 2, 3]`
	`[ , , , ]`	`arr[1, 2, 3, 4]`
list	`[ ]`	`lst[3]`
	`[[ ]]`	`lst[[3]]`
	`$`	`lst$name`
data frame	`[ , ]`	`df[1, 2]`
	`$`	`df$name`