Lecture 3: Data Structures in R

UNIVERSITY OF SAN FRANCISCO

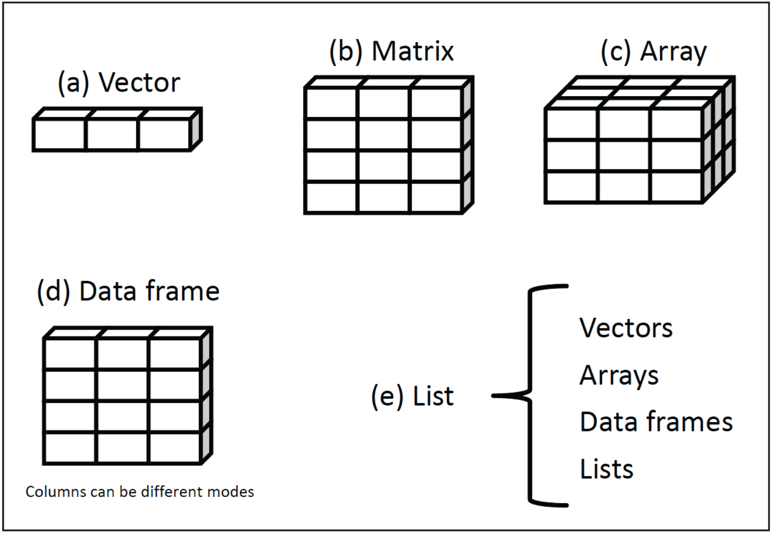
James D. Wilson

ICPSR: Network Analysis I

Data Structures

- A **data structure** is a format or organization of data in software that enables efficient use.
- Every programming language has its own types of data structures
- In R, you can create your own type of data structure; however, there are some that are automatically recognized by the software.
- **Examples**: list, array, data.frame, vector, matrix, string

Data Structures



Data Structures & Dimensionality

Dimension	Homogeneous	Heterogeneous
1	Atomic Vector	List
2	Matrix	Data Frame
n	Array	

Homogeneous: All contents must be of the same type

Heterogeneous: Contents can be of different types

Note: There are no 0-dimensional (scalar) types in R, only vectors of length one

Part I: Vectors

Vectors

- The basic data structure in `R` is the vector
- There two types of vectors: atomic vectors and lists

Properties of Vectors

- Type (`typeof()`)
- Length (`length()`)
- Attributes (`attributes()`)

Use `is.atomic()` or `is.list()` to determine if an object is a vector, **not** `is.vector()`

Atomic Vectors

Four Common Types of Vectors

- Logical
- Integer
- Double (numeric)
- Character

```
> doubleAtomicVector <- c(1, 3.14, 99.999)

# use L prefix to get integers instead of doubles
> integerAtomicVector <- c(1L, 3L, 19L)

> logicalAtomicVector <- c(TRUE, FALSE, T, F)

> characterAtomicVector <- c("this", "is a", "string")
```

Example: Try This

1. Create the vector `myFavNum` of you favorite fractional number
2. Create the vector `myNums` of your seven favorite numbers
3. Create the vector `firstNames` of the first names of two people next to you
4. Create the vector `myVec` of the last name and age of someone you know

Example: Answer these

1. Guess and then check what types your vectors are.
2. Check the length of each vector.
3. Did you write the code in the console window or the editor?
4. How do you execute a line of code in the editor?
5. How do you execute multiple lines of code simultaneously in the editor?
6. Did you leverage the `TAB` button for auto-completion?

Accessing Elements of a Vector

- To access the individual elements of a vector

```
> (myAtomicVector <- c(1, 2, 3, 4, -99, 5, NA, 4, 22.223))
[1]   1.000   2.000   3.000   4.000 -99.000   5.000      NA
[8]   4.000  22.223

#look at fifth element of the vector
> myAtomicVector[5]
[1] -99

> myAtomicVector[c(1, 2, 5, 9)]
[1]   1.000   2.000 -99.000  22.223

> myAtomicVector[10]
[1] NA

#look at the third through eigth elements of the vector
> myAtomicVector[3:8]
[1]   3   4 -99   5  NA   4
```

Accessing Elements of a Vector

- To look at the first and last 6 elements of a vector

```
> (myAtomicVector <- c(1, 2, 3, 4, -99, 5, NA, 4, 22.223))
[1]   1.000   2.000   3.000   4.000 -99.000   5.000      NA
[8]   4.000  22.223

#look at the first and last six elements of the vector
> head(myAtomicVector)
[1] 1.000   2.000   3.000   4.000 -99.000   5.000

> tail(myAtomicVector)
[1] 4.000 -99.000   5.000     NA   4.000  22.223
```

Example, continued

1. Add `myFavNum` to the seventh entry of `myNums` and store the result in a variable named `myFirstAddition`
2. Add `myFavNum` to each of the seven entries of `myNums` and store the result in a variable named `mySecondAddition`
3. Add `myFavNum` to **all** of the values in `myNums` and store the result in a variable named `myFirstSum`
4. Add `myFavNum` to the smallest number in `myNums` and store the result in a variable named `thisIsGettingMoreComplex`
5. Add the second entry of `myNums` to the age of the person you select for `myVec` and store the result in a variable named `whatTypeOfVectorIsThis`
    - Does what we did make sense? Did it work? Why?

Solution

```
# preamble
myFavNum <- 3.1415
myNums <- c(1, 3, 55, 33, 86, -sqrt(2), -110)
# also works myNums <- 1:7
firstNames <- c("Jeff", "Terence", "David")
myVec <- c("Parr", 99)
```

1. `myFirstAddition <- myFavNum + myNums[7]`
2. `mySecondAddition <- myFavNum + myNums`
3. `myFirstSum <- myFavNum + sum(myNums)`
4. `thisIsGettingMoreComplex <- myFavNum + min(myNums)`
5. `whatTypeOfVectorIsThis <- sum(c(myNums[2], myVec[2]))`

```
Error in sum(c(myNums[2], myVec[2])) :
  invalid 'type' (character) of argument
```

Missing Values

Missing values are specified with `NA`, a logical vector of length one.

- `NA` will always be coerced to the correct type if used inside `c()`

```
> c(1, 2, 3, NA)
[1]  1  2  3 NA

> x <- c(1, 2, 3, NA)

> typeof(x)
[1] "double"
```

```
> x[1]
[1] 1

> x[4]
[1] NA

> typeof(x[4])
[1] "double"
```

Argument `na.rm = TRUE`

- Certain functions will fail when applied to vectors with an `NA`

```
> myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4, NA)
[1] 99.1 98.2 97.3 96.4   NA

> sum(myAtomicVector_01)
[1] NA

> mean(myAtomicVector_01)
[1] NA
```

Argument `na.rm = TRUE`

- You can avoid this by providing the argument `na.rm = TRUE`

```
> sum(myAtomicVector_01, na.rm = TRUE)
[1] 391

> mean(myAtomicVector_01, na.rm = TRUE)
[1] 97.75
```

Types & Tests

To check the type of a vector, use `typeof()`, or more specifically

- `is.character()`
- `is.double()`
- `is.integer()`
- `is.logical()`
- `is.na()`

Coercion

Coercion is a great feature in R which can make coding easy, but may also have unintended consequences.

- All elements in an atomic vector must be the same type
- If you attempt to combine different types in an atomic vector they will be coerced to the most flexible type
- **Most to least flexible types** ↓
  - character
  - double
  - integer
  - logical

- When a logical vector is coerced to numeric (double or integer), TRUE = 1 and FALSE = 0

```
> x <- c("abc", 123)

> typeof(x)
[1] "character"
```

You can explicitly coerce using `as.character()`, `as.double()`, `as.integer()`, and `as.logical()`

A Brief Digression: `str()`

- A quick way to figure out what data structure an object is composed of is to use `str()`, which is short for structure
- `str()` provides a concise description for any R data structure

Conditionally Subsetting Atomic Vectors

- The syntax is awkward and takes some time to get used to
- Once you understand the sequence of events in conditional subsetting, it will feel more natural
- Try to figure out what is happening in the following example:

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
[1] 99.1 98.2 97.3 96.4

> myAtomicVector_01[myAtomicVector_01 > 98]
[1] 99.1 98.2
```

Conditionally Subsetting Atomic Vectors

What is actually happening in the last slide:

1. The `myAtomicVector_01 > 98` part of the statement tests each element of the vector to see whether it is $>$ 98 and returns a `LOGICAL` value for each test which, in this case, returns the logical vector (`T T F F`)

2. The vector (`T T F F`) is passed to `myAtomicVector_01`, which returns the first two elements and omits the final two

    - An equivalent statement would be `myAtomicVector_01[c(T, T, F, F)]`

Handy vector functions

Function	Action
`seq(from, to, by)`	Creates a vector of numbers from from to to in increments of by
`rep(x, times)`	Creates a vector that repeats the values in x exactly times number of times
`x +(-, /, *) y`	For x and y of the *same length*, calculates a vector of the same length where each entry is the **entry-wise** summation (subtraction, division, or product) of x and y

Handy tricks

- If you would like to create a vector that is a sequence of numbers from x to y that increase by exactly one, then you can simply write

```
x:y
```

- rep() can be applied to a seq(), providing a flexible means to create sequences with repeating patterns.

  Example:

```
> rep(seq(1, 1.3, .1), 2)
[1] 1.0 1.1 1.2 1.3 1.0 1.1 1.2 1.3
```

Example

```
> x <- rep(c(1,2), 3)

> y <- seq(from = .5, to = 3, by = .5)

> x
[1] 1 2 1 2 1 2

> y
[1] 0.5 1.0 1.5 2.0 2.5 3.0

> x+y
[1] 1.5 3.0 2.5 4.0 3.5 5.0

> x/y
[1] 2.0000000 2.0000000 0.6666667 1.0000000 0.4000000 0.6666667
```

A List of Logical Operators

Operator	Description
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Exactly equal to
`!=`	Not equal to
`!`x	Not x
x `|` y	x or y
x `&` y	x and y
`isTRUE(`x`)`	Test if x is `TRUE`

Vector example: names

- A name is a vector attribute
- Can be identified using the `names()` function

```
> x <- c(1, 2, 3)
> names(x)
NULL

> x <- c(1, 2, 3); names(x) <- c("a", "b", "c")
> names(x)
[1] "a" "b" "c"

> x <- c(a = 1, b = 2, c = 3)
> names(x)
[1] "a" "b" "c"

> x <- c(a = 1, b = 2, 3)
> names(x)
[1] "a" "b" ""
```

Part II: Matrices and Arrays

Matrices and Arrays

- By giving an atomic vector a dimension attribute, it behaves like a multi-dimensional array
- A special case of the array is a matrix, a two-dimensional array
- A matrix has 2 dimensions, and an array has n $\geq$ 2 - dimensions.
- Matrices and arrays are created with `matrix()` and `array()`

Matrix Example

```
> x <- matrix(1:10, nrow = 2, ncol = 5)
 # can drop nrow and ncol to shorten but keep in this order

 > x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Array Example

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

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Selected Functional Generalizations

1-D Function	n-D Functions
`length()`	`nrow()`, `ncol()`, `dim()`
`names()`	`rownames()`, `colnames()`, `dimnames()`
`c()`	`cbind()`, `rbind()`

Note: a matrix or array can also be one-dimensional, e.g., an object that is defined as a matrix is permitted to only have one column or one row; although they may look and behave alike, a vector and a one-dimensional matrix behave differently and may generate strange output when using certain functions, e.g., `tapply()`

Common R Functions for Working with Data

Function	Purpose
`length(object)`	Number of elements/components.
`dim(object)`	Dimensions of an object.
`str(object)`	Structure of an object.
`class(object)`	Class or type of an object.
`mode(object)`	How an object is stored.
`names(object)`	Names of components in an object.
`c(object, object,...)`	Combines objects into a vector.
`cbind(object, object, ...)`	Combines objects as columns.
`rbind(object, object, ...)`	Combines objects as rows.
`object`	Prints the object.
`head(object)`	Lists the first part of the object.
`tail(object)`	Lists the last part of the object.
`ls()`	Lists current objects.
`rm(object, object, ...)`	Deletes one or more objects. The statement `rm(list = ls())` will remove most objects from the working environment.
`newobject <- edit(object)`	Edits object and saves as newobject.
`fix(object)`	Edits in place.

Part III: Lists

Lists

- Lists are different from atomic vectors as elements of a list can be of any type, including lists
- A list is constructed using `list()` instead of `c()`

```
> myList <- list(10:12, "abc", c(3.1415, 9), c(T, F, F, F))

> str(myList)
List of 4
 $ : int [1:3] 10 11 12
 $ : chr "abc"
 $ : num [1:2] 3.14 9
 $ : logi [1:4] TRUE FALSE FALSE FALSE
```

Lists

- Lists are recursive, i.e., a list can contain lists, making them fundamentally different from atomic vectors
- Handy functions

Function	Action
`is.list()`	test if list
`as.list()`	coerce to list
`unlist()`	convert to atomic vector + coercion

Subsetting Lists

- Entries in a list can contain any type of data structure
- To call a single entry (say the second one) in the list `myList`, use double brackets: `myList[[2]]`
- To call multiple entries in a list (say the first and second), use single brackets: `myList[1:2]`
- If the entries in a list are named, you can call them directly using `myList$Name`

Subsetting Example

```
> myList <- list(10:12, Letters = "abc", c(3.1415, 9), Logicals =
c(T, F, F, F))

> myList[[2]]
[1] "abc"

> myList$Logicals
[1]  TRUE FALSE FALSE FALSE

> myList[1:2]
[[1]]
[1] 10 11 12

$Letters
[1] "abc"
```

Part IV: Data Frames

Data Frames

- Most common way of storing data in R
- A data frame is a list with equal-length vectors
- Each vector must be of the same data type

This is why we use

Data Frame Summary Example

Summary of Data `ToothGrowth`: a data frame with 60 observations on 3 variables.

- [, 1] len numeric: Tooth length
- [, 2] supp factor: Supplement type (VC or OJ)
- [, 3] dose numeric: Dose in milligrams/day

```
> str(ToothGrowth)
'data.frame': 60 obs. of  3 variables:
 $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...

 > ?ToothGrowth
```

Creating Data Frames

Create a data frame using `data.frame()`

```
# this is sloppy coding etiquette and is only for exposition

> (xyz <- data.frame(1:3, c("a", "b", "c")))
  X1.3 c..a....b....c..
1    1                a
2    2                b
3    3                c

> str(xyz)
'data.frame': 3 obs. of  2 variables:
 $ X1.3            : int  1 2 3
 $ c..a....b....c..: Factor w/ 3 levels "a","b","c": 1 2 3
```

Creating Data Frames

Create a data frame using `data.frame()`

- Surround code with `()` to automatically print the result to the console
- After creating the data frame, the first column of untitled numbers are row numbers
- Observe that even though the entries in `letterColumn` are characters that an `str(letterColumn)` shows the column to be a `Factor`

Creating and Manipulating Data Frames

- If you want to suppress R's default behavior of turning strings into factors, use the options `stringsAsFactors = FALSE`

```
> (xyz <- data.frame(numberColumn = 1:3, letterColumn = c("a", "b", "c"),
    stringsAsFactors = F))

  numberColumn letterColumn
1            1            a
2            2            b
3            3            c

> str(xyz)

'data.frame': 3 obs. of  2 variables:
 $ numberColumn: int  1 2 3
 $ letterColumn: chr  "a" "b" "c"
```

Creating and Manipulating Data Frames

- **Note:** A data frame is a list, which means that `typeof(myDataFrame)` will output a list
- Instead use `class()` or `is.data.frame()`
- An object can be coerced to a data frame using `as.data.frame()`

Combine/Append Data Frames

- **When a data frame already exists**, you can easily combine/append another data frame or a vector to the original data frame

1. Use `cbind()` to column-bind two data frames
   - **Note**: the number of columns in each data frame must be equal, and row names are ignored
2. Use `rbind()` to row-bind two data frames
   - **Note**: the **number** and the **names** of columns must match

Examples: cbind()

```
> (myDataFrame_01 <- data.frame(x = 1:3, y = c("A", "B", "c")))
  x y
1 1 A
2 2 B
3 3 c

> (myDataFrame_02 <- cbind(myDataFrame_01, data.frame(z = -1:-3)))
  x y  z
1 1 A -1
2 2 B -2
3 3 c -3
```

Examples: rbind()

```
> (myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002))
  x   y    z
1 1  98 1000
2 2  99 1001
3 3 100 1002

> (myDataFrame_06 <- rbind(myDataFrame_05, qqq = -1:-3))
     x   y    z
1    1  98 1000
2    2  99 1001
3    3 100 1002
qqq -1  -2   -3
```

Example: Try these

```
> myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_06 <- rbind(myDataFrame_05, ???)
```

- Based on the `myDataFrame_06` code, what happens if we replace ??? with:
  (a) `qqq = -1`
  (b) `qqq = -1:-2`
  (c) `qqq = -1:-99`
  (d) `qqq = c(-1, -2)`
  (e) `qqq = c("-1", -2)`
  (f) `qqq = c("a", -2, -3)))`

Solution

```
> myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_06 <- rbind(myDataFrame_05, ???)
```

(a) Entire additional row of -1's

(b) Entire additional row of repeating -1's and -2's

(c) Additional row: -1, -2, -3

(d) Entire additional row of repeating -1's and -2's

(e) Entire additional row of repeating -1's and -2's as characters (non numeric), thereby changing **all** all data frame column types to characters

(f) Additional row: a, -2, -3 as characters (non numeric), thereby changing **all** all data frame columns types to characters

Combine/Append Data Frames

- Use `cbind()` to column-bind a data frame with a vector
  - **Note**: This will only work if the vector has the same length as the number of rows in the data frame.

```
> (myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002))
  x   y    z
1 1  98 1000
2 2  99 1001
3 3 100 1002

> (myDataFrame_08 <- cbind(myDataFrame_05, qqq = -1:-3))
  x   y    z qqq
1 1  98 1000  -1
2 2  99 1001  -2
3 3 100 1002  -3
```

Example: Try these

```
> myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_08 <- cbind(myDataFrame_07, ???)
```

- Based on the `myDataFrame_08` code, what happens if we replace ??? with:
  (a) `qqq = -1`
  (b) `qqq = -1:-2`
  (c) `qqq = -1:-99`
  (d) `qqq = c("-1", -2)`
  (e) `qqq = c("a", -2, -3)))`

Solution

```
> myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_08 <- cbind(myDataFrame_05, ???)
```

(a) Entire additional column of -1's

(b) `<arguments imply differing number of rows: 3, 2>`

(c) Extends the length of all other columns and repeats those values until -99

(d) `<arguments imply differing number of rows: 3, 2>`

(e) `<arguments imply differing number of rows: 3, 2>`

(f) Additional column: a, -2, -3 as factors (non numeric)