List and data frame

STA 032: Gateway to data science Lecture 4

Jingwei Xiong

April 10, 2023

Reminders

- Homework 1 has been assigned: (Due April 17 midnight, cover lecture 1-4)
    - start as soon as possible
    - PDF files only
    - Submission through Gradescope (accessible through Canvas)
    - If you get stuck, it's encouraged to communicate with your mate about solution.
    - But you should type your solution by your own.
    - If you collaborate with others, write their names in your submission
- Office hours:
    - TBD
- Lecture 1 - 3 we covered vectors and basics of R, R markdown.
- From now on we will start working with data.

Today

- Introduction to lists
- Data frames, or more generally "data sets"
- Auto complete and How to use help in R

Lists

- Lists are a generic container
- Sequence of values, *not* necessarily all of the same type
  - (Vector has to be the same type!)

```
my.distribution <- list("exponential", 7, FALSE)
my.distribution
```

```
[[1]]
[1] "exponential"

[[2]]
[1] 7

[[3]]
[1] FALSE
```

- Most of what you can do with vectors you can also do with lists
- This is an unnamed list

Lists

- Elements can be vectors of **any type**, or other data structures like data frame (We will cover that later this lecture)
- This is a named list

```
l <- list(
  x = 1:4,
  y = c("hi", "hello", "jello"),
  z = data.frame(a = c(1,2), b = c(3,4))
)
l
```

```
$x
[1] 1 2 3 4

$y
[1] "hi"    "hello" "jello"

$z
  a b
1 1 3
2 2 4
```

Lists

Make an empty list to fill in later

```
myList <- vector(mode = "list", length = 4)
myList
```

```
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

Accessing pieces of lists

Can use `[ ]` as with vectors

or use `[[ ]]`, but only with a single index
`[[ ]]` drops names and structures, `[ ]` does not

```
l[1]
```

```
$x
[1] 1 2 3 4
```

```
l[[1]]
```

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

Does `l[[1:2]]` work?

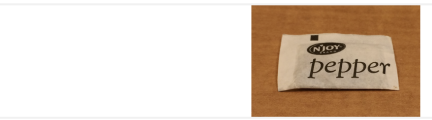
Accessing pieces of lists

Helpful illustration from R for Data Science (Chapter 20.5.3):

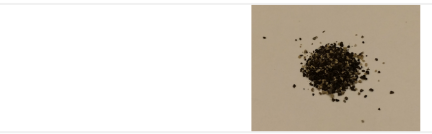
If this pepper shaker is your list `x`, then, `x[1]` is a pepper shaker containing a single pepper packet:

`x[2]` would look the same, but would contain the second packet. `x[1:2]` would be a pepper shaker containing two pepper packets.

`x[[1]]` is:



If you wanted to get the content of the pepper package, you'd need `x[[1]][[1]]`:



> Summary: [] subset will still be a list, not an element. If you want to access the element, use [[]]

Working with lists

```
my.distribution
```

```
[[1]]
[1] "exponential"

[[2]]
[1] 7

[[3]]
[1] FALSE
```

```
is.character(my.distribution)
```

```
[1] FALSE
```

```
is.character(my.distribution[[1]
```

```
[1] TRUE
```

```
my.distribution[[2]]^2
```

```
[1] 49
```

What happens if you try `my.distribution[2]^2`? What happens if you try `[[ ]]` on a vector?

Filling in lists

```
myList[[1]] <- 1:10
# Fill the first element with a vector 1:10
myList
```

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

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

What happens if you try `myList[1] <- 1:10`?

> Summary: Rememeber when you want to **access** or **assign** with an element in a list, use [[]]!!!

Expanding and contracting lists

Add to lists with `c()` (also works with vectors):

```
my.distribution <- c(my.distribution, 7)
my.distribution
```

```
[[1]]
[1] "exponential"

[[2]]
[1] 7

[[3]]
[1] FALSE

[[4]]
[1] 7
```

```
# vector:
a = c(1,2,3)
c(a, 4)
```

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

Chop off the end of a list by setting the length to something smaller (also works with vectors):

```
length(my.distribution)
```

```
[1] 4
```

```
length(my.distribution) <- 3
my.distribution
```

```
[[1]]
[1] "exponential"

[[2]]
[1] 7

[[3]]
[1] FALSE
```

```
length(a) = 3
a
```

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

Naming list elements

- We saw how to name elements of a list while constructing them
- We can also add names later on:

```
my.distribution <- list("exponential", 7, FALSE)
names(my.distribution) <- c("family", "mean", "is.symmetric")
my.distribution
```

```
$family
[1] "exponential"

$mean
[1] 7

$is.symmetric
[1] FALSE
```

Accessor sign $

Lists have a special short-cut way of using names, $ (which removes names and structures):

```
my.distribution[["family"]]
```

```
[1] "exponential"
```

```
my.distribution$family
```

```
[1] "exponential"
```

```
my.distribution[1]
```

```
$family
[1] "exponential"
```

> Using the $ operator can make our code more readable and easier to understand, especially when working with large or complex data structures

Names in lists

Creating a list with names:

```
another.distribution <- list(family="gaussian", mean = 7,
                             sd = 1, is.symmetric = TRUE)
```

Adding named elements:

```
my.distribution$was.estimated <- FALSE
my.distribution[["last.updated"]] <- "2011-08-30"
```

Removing a named list element, by assigning it the value `NULL`:

```
my.distribution$was.estimated <- NULL
```

Structure of lists

- `str()` is particularly useful for lists, since it allows us to easily get an idea of what is in the list.
- We can use str() to see the structure of a list, including the data types and names of each element in the list.

```
str(my.distribution)
```

```
List of 4
 $ family      : chr "exponential"
 $ mean        : num 7
 $ is.symmetric: logi FALSE
 $ last.updated: chr "2011-08-30"
```

Data frames

- A data frame is a special **list** containing vectors of **equal length**
- Data frame = the classic data table, n rows for observations, p columns for variables
- Lots of the statistical parts of R presume data frames
- *columns can have different types*: String, numeric, Date, Boolen, etc

Creating data frames

- Use the data.frame() function to create a dataframe
- We can use `dim()` function to know how many row and columns.

```
# create a data frame with three variables
my.df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  z = c(TRUE, FALSE, TRUE)
)
# print the data frame
my.df
```

```
  x y     z
1 1 a  TRUE
2 2 b FALSE
3 3 c  TRUE
```

```
dim(my.df)
```

```
[1] 3 3
```

Accessing dataframe

- Use $ accessor to access a specific column of a data frame by name.
- For example, `my.df$column_name` returns the values of the `column_name` column.
    - (Which will be a vector)

```
my.df$x
```

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

- Row and column index: we can use the [ ] operator with row and column index to access individual elements of a data frame.
    - For example, `my_df[row_index, col_index]` returns the value in the `row_index`-th row and `col_index`-th column.

```
my.df[2, 3]
```

```
[1] FALSE
```

- Rows or columns by index: we can use the [ ] operator to access entire rows or columns of a data frame by index
  - For example, `my_df[row_index, ]` returns the `row_index`-th row of the data frame, and `my_df[, col_index]` returns the `col_index`-th column.

```
# Use row index to get a row
my.df[3, ]
```

```
  x y    z
3 3 c TRUE
```

```
# Use column index to get a column
my.df[, 2]
```

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

- You can also use index vectors on the row or column arguments.

```
my.df[c(1,2),c(1,2)]
```

```
  x y
1 1 a
2 2 b
```

Adding rows and columns

We can add columns using $ accessor

```
# Adding columns
my.df$new.col <- 4:6
my.df
```

```
  x y     z new.col
1 1 a  TRUE       4
2 2 b FALSE       5
3 3 c  TRUE       6
```

We can also add columns similar to a list

```
my.df[["newer.col"]] <- c(7, 8, 9)
my.df
```

```
  x y     z new.col newer.col
1 1 a  TRUE       4         7
2 2 b FALSE       5         8
3 3 c  TRUE       6         9
```

remove column

Now remove `newCol`

```
# Removing column 3
my.df <- my.df[, -3]
# We can also remove by this way
my.df$new.col <- NULL
```

> Some are very similar to a list operation.

`rbind()` and `cbind()`

We can also add rows or columns to an array or data-frame with `rbind()` and `cbind()`.

rbind | rbind_df | cbind | cbind_df

```
# Create a data frame
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  z = c(TRUE, FALSE, TRUE)
)

# Add a new row
new_row <- data.frame(x = 4, y = "d", z = FALSE)
df <- rbind(df, new_row)
df
```

```
  x y     z
1 1 a  TRUE
2 2 b FALSE
3 3 c  TRUE
4 4 d FALSE
```

`rbind()` and `cbind()`

We can also add rows or columns to an array or data-frame with `rbind()` and `cbind()`.

rbind | rbind_df | cbind | cbind_df

```
# Creating two data frames to combine
df1 <- data.frame(x = 1:3, y = c("a", "b", "c"))
df2 <- data.frame(x = 4:6, y = c("d", "e", "f"))

# Using rbind to combine rows
df3 <- rbind(df1, df2)
df3
```

```
  x y
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
6 6 f
```

`rbind()` and `cbind()`

We can also add rows or columns to an array or data-frame with `rbind()` and `cbind()`.

rbind | rbind_df | **cbind** | cbind_df

```
# Create a data frame
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  z = c(TRUE, FALSE, TRUE)
)

# Add a new column
new_col <- c(4, 5, 6)
df <- cbind(df, new_col)
df
```

```
  x y     z new_col
1 1 a  TRUE       4
2 2 b FALSE       5
3 3 c  TRUE       6
```

`rbind()` and `cbind()`

We can also add rows or columns to an array or data-frame with `rbind()` and `cbind()`.

rbind | rbind_df | cbind | **cbind_df**

```
# Creating two data frames to combine
df1 <- data.frame(x = 1:3, y = c("a", "b", "c"))
df2 <- data.frame(z = c(TRUE, FALSE, TRUE), w = c(0.5, 1.2, 2.1))

# Using cbind to combine columns
df3 <- cbind(df1, df2)
df3
```

```
  x y     z   w
1 1 a  TRUE 0.5
2 2 b FALSE 1.2
3 3 c  TRUE 2.1
```

More complicated data structures: structures of structures

Words | Code example | Output | Explanation

- Internally, a data frame is basically a list of vectors
- List elements can even be other lists,
  - which may contain other data structures, including other lists,
  - which may contain other data structures…
- This **recursion** lets us build arbitrarily complicated data structures from the basic ones

More complicated data structures: structures of structures

Words | **Code example** | Output | Explanation

```
df1 <- data.frame(x = 1:3, y = c("a", "b", "c"))
df2 <- data.frame(x = 4:6, y = c("d", "e", "f"))
nested_list <- list(df1 = df1, df2 = df2)
str(nested_list)
```

```
List of 2
 $ df1:'data.frame':    3 obs. of  2 variables:
  ..$ x: int [1:3] 1 2 3
  ..$ y: chr [1:3] "a" "b" "c"
 $ df2:'data.frame':    3 obs. of  2 variables:
  ..$ x: int [1:3] 4 5 6
  ..$ y: chr [1:3] "d" "e" "f"
```

More complicated data structures: structures of structures

Words | Code example | Output | Explanation

```
nested_list
```

```
$df1
  x y
1 1 a
2 2 b
3 3 c

$df2
  x y
1 4 d
2 5 e
3 6 f
```

More complicated data structures: structures of structures

Words | Code example | Output | Explanation

In this example, we create two data frames, `df1` and `df2`, and then create a list called `nested_list` containing these two data frames as elements. We can see that nested_list is a list with two elements, each of which is a data frame, by using the `str()` function to print the structure of the list.

Autocomplete and help

Autocomplete | seq() example | Get help of a function | help page

- RStudio has a powerful autocomplete feature that makes it easy to write code and reduce errors
- To use the autocomplete feature, simply start typing a function or variable name and press the TAB key
- RStudio will display a list of possible completions, including function names, variable names, and other objects in your workspace
- You can select the desired completion from the list using the up and down arrow keys or by clicking on the item with your mouse
- Autocomplete also works with other types of text, such as file paths and package names

Autocomplete and help

Autocomplete | seq() example | Get help of a function | help page

- Let's try using `seq()` which makes regular **seq**uences of numbers
- Type `se` and hit TAB. A popup shows you possible completions.
- Specify `seq()` by typing more (a "q") to disambiguate
- Press TAB once more when you've selected the function you want.
  RStudio will add matching opening (() and closing ()) parentheses for you.
- Type the arguments 1, 10 and hit return.

```
seq(1,10)
```

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

Autocomplete and help

Autocomplete | seq() example | **Get help of a function** | help page

- Type `?function_name` in console to get help of a function. The help will be in "help" panel to the right.

```
?seq
```

The help documentation is divided into several sections, including:

- Description: a brief summary of what the function does.
- Usage: the syntax of the function, including any required and optional arguments.
- Arguments: a description of each argument, including its name, data type, and default value.
- Details: additional information about the function's behavior or how to use it.
- Value: the type of output produced by the function.
- Examples: example code demonstrating how to use the function.

Autocomplete and help

Autocomplete | seq() example | Get help of a function | **help page**

links here

- It will be the same when you use `?seq` on your own computer

Recap: install packages

- What you get after your first install is base R
- extra functionality comes from add-ons available from developers
- R makes it very easy to install packages from within R. For example, type this in console

```
install.packages("tidyverse")
install.packages("ggplot2")
install.packages("dslabs")
```

After we install the package, we can then load the package into our R sessions using the library function:

```
library(tidyverse)
library(dslabs)
```

If you want to use the add-on functions in the package, you need to library the package first.

Built-in Dataframes

For example, we stored the data for US gun murder in a data frame. You can access this dataset by loading the **dslabs** library and loading the `murders` dataset using the `data` function:

```
library(dslabs)
data(murders)
```

To see that this is in fact a data frame, we type:

```
class(murders)
```

```
[1] "data.frame"
```

Data frames, data sets

- We've seen data frames. This is a commonly used data structure that we get after reading in a data set into R.
- In a data set in general,
    - Each row is an **observation**, n
    - Each column is a **variable**, p
- Often, the first things we want to do when given a data set are to figure out
    1. What is in it (what dimensions, what variables)
    2. What the main characteristics of the variables are.
- We've seen a few tools and functions for working with data frames in "base R," now we will look at some tools from `dplyr`

Tidyverse

Packages Blog Learn Help Contribute

R packages for data science

The tidyverse is an opinionated **collection of R packages** designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

https://www.tidyverse.org/

- What we've seen so far: "base R"
- `ggplot2` for plotting, `dplyr` for data manipulation

First question: What's in a data set?

Example: Star Wars data

- `starwars` data set in the `dplyr` package

(A `tibble` is the `tidyverse` version of the data frame.)

```
dplyr::starwars
```

```
# A tibble: 87 x 14
   name       height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex
   <chr>       <int> <dbl> <chr>   <chr>   <chr>     <dbl> <chr>
 1 Luke Skyw~    172    77 blond   fair    blue       19   male
 2 C-3PO         167    75 <NA>    gold    yellow    112   none
 3 R2-D2          96    32 <NA>    white,~ red        33   none
 4 Darth Vad~    202   136 none    white   yellow     41.9 male
 5 Leia Orga~    150    49 brown   light   brown      19   fema~
 6 Owen Lars     178   120 brown,~ light   blue       52   male
 7 Beru Whit~    165    75 brown   light   blue       47   fema~
 8 R5-D4          97    32 <NA>    white,~ red        NA   none
 9 Biggs Dar~    183    84 black   light   brown      24   male
10 Obi-Wan K~    182    77 auburn~ fair    blue-g~    57   male
# ... with 77 more rows, 6 more variables: gender <chr>,
#   homeworld <chr>, species <chr>, films <list>,
```

We've seen `str()`. `dplyr::glimpse()` produces cleaner output in this case:

```
dplyr::glimpse(starwars)
```

```
Rows: 87
Columns: 14
$ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth ~
$ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, ~
$ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0,~
$ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, g~
$ skin_color <chr> "fair", "gold", "white, blue", "white", "li~
$ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown",~
$ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, ~
$ sex        <chr> "male", "none", "none", "male", "female", "~
$ gender     <chr> "masculine", "masculine", "masculine", "mas~
$ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine"~
$ species    <chr> "Human", "Droid", "Droid", "Human", "Human"~
$ films      <list> <"The Empire Strikes Back", "Revenge of th~
$ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, ~
$ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "T~
```

How many rows and columns does this data set have? What does each row represent? What does each column represent?

```
?starwars
```

starwars {dplyr} R Documentation

Starwars characters

Description

This data comes from SWAPI, the Star Wars API, https://swapi.dev/

Usage

```
starwars
```

Format

A tibble with 87 rows and 14 variables:

name
: Name of the character

height
: Height (cm)

mass
: Weight (kg)

hair_color,skin_color,eye_color
: Hair, skin, and eye colors

birth_year
: Year born (BBY = Before Battle of Yavin)

How many rows and columns does this data set have?

```
nrow(starwars) # number of rows
```

```
[1] 87
```

```
ncol(starwars) # number of columns
```

```
[1] 14
```

```
dim(starwars)  # dimensions (row column)
```

```
[1] 87 14
```

As we've seen, columns (variables) in data frames can be accessed with `$`:

```
dataframe$var_name
```

Next lecture we will start data manipulation!

Readings

- R for Data Science Chapter 4, 20
- Chapter 2:R basics
- Chapter 3