R Fundamentals Part 1: Introduction

Evan Muzzall, Rochelle Terman, Dillon Niederhut

November 10, 2016

Table of Contents

Learning objectives

0. Front matter
1. Introduction: What is R Studio?, objects, global environment, packages, variable assignment, help
2. Atomic data types: numeric, character, and logical
3. Data type coercion
4. Data structures: vector, list, matrix, data.frame
5. Saving your work

0. Front matter/ Pre-introduction

Start by having the class go to our github page at github.com/dlab-berkeley/R-Fundamentals to get the course materials by:

1. clicking the green `download zip` button on the right hand side of the screen, or
2. `git clone https://github.com/dlab-berkeley/R-Fundamentals.git`

While everything is downloading, you can go on to:

0. Front matter/ Introduction to the class

These materials are meant to be guides for you, ***the instructor***. Your students will retain more of this content if they type these commands themselves than if they simply enter them off this markdown file.

D-Lab works with Berkeley faculty, research staff, and students to advance data-intensive social science and humanities research. Our goal at D-Lab is to provide students with practical training, staff support, resources, and space to enable them to use R for their own research applications. Our services cater to all skill levels and no programming, statistical, or computer science background is necessary. We offer these services in the form of workshops such as this one, one-to-one consulting, and working groups that cover a variety of topics and programming languages:

D-Lab Workshop Calendar
D-Lab Consulting
D-lab Working Groups

It is a good idea to start off the class by explpaining to folks why you (the instructor) use R. Also describe in general why social scientists use R as well as some of its pitfalls. Then, break the ice by going around the room and asking each student to state their name, department, and why they want to learn R. Common responses include:

1. Other programs are too expensive
2. I saw a pretty graph someone made in R
3. My field uses analytical packages written for R
4. I have a deep and burning desire for open and reproducible research

As the instructor, you should draw on your own experiences to include further examples and advice, especially for students who do not fall into one of the four categories above and/or are just beginning to learn R.

If you are a student and you are looking at this markdown file, please use this as a guide if you have fallen behind, can't see the screen very well, or want to review these materials.

1. Introduction/ what is R Studio?

R is old programming language based on the "S" and "S-PLUS" languages and is both *functional* and *object oriented*. These themes make R very flexible.

However, we do not need to think of R exclusively in these terms. At D-Lab, we want to provide you with basic programming competencies so you can get started on your own research!

RStudio is an interactive R environment that permits the user to enter and save code and data for exploration, manipulation, plotting, and testing.

You can download R Studio here

If this installation says that you are lacking the binary files, download R binary files here

After opening R Stuio, by default you should see four windows after opening a new script:

Open a new script. *PC:* `Ctrl + Shift + N` *MAC:* `command + shift + N`

Top left window: enter code in this savable `script` file

Bottom left window: code output is displayed here in the `console`

Top right window: contains the defined objects currently in the `global environment`

Bottom right window: contains file import/export, plots, package installations, and help

control + 1, 2, 3, 4 will keyboard shortcut between these windows

To run a line of code: *MAC*: `command + return` *PC*: `Ctrl + r`

To clear your console *MAC*: `control + l` *PC*: `Ctrl + l`

Commenting your script file: Using a hashtag `#` will tell R that you do not want that particular line or block of code to be run. This is handy for making notes to yourself! You can even add hashtags after lines of runable code - on the same line!

In the console, the prompt `>` looks like a greater than symbol. If your prompt begins to look like a + symbol by mistake, simply click in your console and press the `esc` key on your keyboard to return to `>`.

R uses + when code is broken up across multiple lines and R is still expecting more code. A line of code does not usually stop until it finds an appropriate stop parameter, often a closed parenthesis `)` or closed bracket`]`.

NOTE: the name of your script file is in the tab at the top of your script window - the name defaults to `Untitled1`. If the name is red and followed by an asterisk * it means ***your script is not saved***. Save your script by clicking "File" then "Save", or `command + s` (Mac) or `Ctrl + s` (PC).

1. Introduction/ package installation

Many people write clever software that makes R smarter/better/faster/stronger. You can install packages by clicking the "Packages" tab in the bottom-right window, clicking install, and then searching for the package you wish to install.

Packages can also be installed through the command line using `install.packages()`. The package name must be wrapped in quotation marks so

that R knows it is searching for that particular package named "psych", rather than previously defined data named psych:

```
install.packages("psych", dependencies=TRUE)
```

Anytime you want to utilize a previously installed package, you must retreive it with `library()` when you begin a new R session. You do not need to reinstall packages each time you quit and restart your R session

```
library(psych)

## Warning: package 'psych' was built under R version 3.2.5
```

1. Introduction/ objects

Objects are special data structures that allow you to enter data into R. Objects are stored in R's memory and can be retrieved ("called") when you need them.

You define objects through `variable assignment` and they are stored in your `global environment`.

1. Introduction/ the R global environment

Objects in R are stored in your global environment. `ls()` will list previously defined objects in your environment.

```
ls() #There is currently nothing in our global environment

## character(0)
```

Click the "Environment" tab in the upper-right window - it says "Environment is empty".

1. Introduction/ variable assignment

You define objects through variable assignment using the ***assignment operator*** <-

This is a "less than" < symbol immediately followed by a hyphen -

Object definition requires three pieces of information:
1) object_name
2) <-
3) definition

Let's define an object named `numeric_oject` and define it as the number 4:

```
numeric_object <- 4
ls()

## [1] "numeric_object"
```

We now have an object defined in our global environment!

"Call" (retrieve) the data contained wihtin the object by typing its name into your script and running the line of code

```
numeric_object # 4 is returned

## [1] 4
```

Let's try another example, this time using character data. Note that character data is ***always*** contained within quotation marks " "

```
welcome_object <- "Welcome to the D-Lab"
ls()

## [1] "numeric_object" "welcome_object"
```

We now have two objects in our global environment. Call the data contained within the object by typing its name and running the line of code:

```
welcome_object

## [1] "Welcome to the D-Lab"
```

1. Introduction/ variable assignment/ naming R objects

Object names can be anything, but are always case sensitive. However, they cannot begin with a number and generally do not begin with symbols.

However you choose to name your objects, be consistent and use brief descriptions of their contents.

Names must be ***unique***. If you resuse an old name, the old object definition will be overwritten.

Let's overwrite the object `welcome_object` from above.

```
welcome_object <- "Welcome to Barrows Hall"
```

See how the definition of `welcome_object` changed in your global environment window? However, there are still only two objects in your global environment.

It also prints the new information when called:

```
welcome_object

## [1] "Welcome to Barrows Hall"
```

If we want to preserve the original object `welcome_object` we have to define a new object

```
welcome_object2 <- "Welcome to the D-Lab in Barrows Hall"
ls()

## [1] "numeric_object"  "welcome_object"  "welcome_object2"
```

We now have three objects defined in our global environment! This third object contains new information and did not overwrite the old oblect:

```
welcome_object2

## [1] "Welcome to the D-Lab in Barrows Hall"
```

1. Introduction/ variable assignment/ `class()`

Each object in R has a `class()` that summarizes the type of the data stored within the object.

```
class(numeric_object)

## [1] "numeric"

class(welcome_object)

## [1] "character"

class(welcome_object2)

## [1] "character"
```

1. Introduction/ variable assignment/ `rm()`

We remove individual variables from our environment with `rm()`. For example, to remove `numeric_object`, we type:

```
rm(numeric_object)
ls()

## [1] "welcome_object"  "welcome_object2"
```

See how `numeric_object` disappeared from our global environment?

We can also wipe the entire environment with `rm(list = ls())`

```
rm(list=ls())
ls()

## character(0)
```

Now, all objects are gone from our global environment.

1. Introduction/ help

A single question mark `?` will call the help pages. Notice that we must wrap symbols in quotation marks to access their help page:

```
?"<-"
?ls
?dir
```

```
?rm
?iris
```

NOTE: We will use the `iris` dataset in Parts 2, 3, and 4

Double question marks ?? will lead you to coding walkthroughs called "vignettes":

```
??"<-"
??rm
??iris
```

If you have questions that don't seem to be answered by the help pages and vignettes, we recommend to type your question into Google along with the name of one of the many community based resources:
Quick-R
UCLA idre
R-bloggers
Stack Overflow - R

1. Introduction/ living in R

Like in Unix, in R you are always in a directory. Your actions are all relative to that directory. You will find this useful for loading data from files and when you save your output it will be located here.

Figure out where you are with `getwd()`

```
getwd()

## [1] "/Users/E/Desktop/R-Fundamentals-master"
```

Tell R you would like to set your working directory to the R-Fundamentals folder you just downloaded with `setwd()`. Follow the format as shown by `getwd()`

My computer is named "E" and my "R-Fundamentals-master" folder is located on the Desktop, so the file path would look like this:

```
setwd("/Users/E/Desktop/R-Fundamentals")
getwd()
```

Note the subtle difference between Mac and PC users:

```
setwd("/Users/MacName/Desktop/R-Fundamentals-master")
setwd("C:/Users/PCName/Desktop/R-Fundamentals-master")  #C:/...
```

You can view the contents of your working directory with `dir()`.

```
dir()

## [1] "animals.csv"
"animals.txt"
## [3] "basket.csv"                                          "R
```

```
Fundamentals Part 1 Introduction.Rmd"
## [5] "R Fundamentals Part 2 Subsetting and reshaping.Rmd"      "R
Fundamentals Part 3 Data exploration and analysis.Rmd"
## [7] "R Fundamentals Part 4 For-loops and functions.Rmd"
"R_Fundamentals_Part_1_Introduction.html"
## [9] "R_Fundamentals_Part_1_Introduction.Rmd"
```

1. *Challenge 1*

1. Define two numeric objects
2. Define a character object
3. Remove an individual object using rm()
4. Wipe your environment using rm(list=ls())
5. Set your working directory to your "Documents" folder. Now set it back to the "R-Fundamentals" folder.

2. Atomic data types in R/

Numeric, character, and logical (aka "boolean") data types all exist at the atomic level. Normally this means that they cannot be broken down any further and are the raw inputs for commands in R. Other R objects are frequently built upon these atomic types.

2. Atomic data types in R/ numeric

Numeric data are numbers and integers. You may also hear numeric data referred to as `float` or `double` data types. By default, R stores everything as `doubles` (64 bit floating point numbers) which makes R very memory hungry.

Define an object called `num` and then check its class

```
num <- 2 * pi
num

## [1] 6.283185

class(num)

## [1] "numeric"
```

Standard mathematical operators apply to the creation of numeric data: `+ - * ^ ** /` `%*% (matrix multiplication) %/% (integer division) %% (modular division)`

```
5 + 2

## [1] 7

5 - 2
```

```
## [1] 3
```

```
5 * 2
```

```
## [1] 10
```

```
5 ^ 2
```

```
## [1] 25
```

```
5 ** 2 # same as ^
```

```
## [1] 25
```

```
5 / 2
```

```
## [1] 2.5
```

```
5 %*% 2
```

```
##      [,1]
## [1,]   10
```

```
5 %/% 2
```

```
## [1] 2
```

```
5 %% 2
```

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

2. Atomic data types in R/ character

Character (aka string or text) data are always contained within quotation marks " ". Character handling in R is fairly close to character handling in a Unix terminal.

Let's create an object called `char` and define it with some character data:

```
char <- "This is character data"
char
```

```
## [1] "This is character data"
```

```
class(char)
```

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

Character data have some useful commands such as `paste()`, `substr()`, `strsplit()`, and `gsub()`

2. Atomic data types in R/ character/ `paste()`

Use `paste()` to combine/concatenate character data. This will paste together separate words to form a sentence.

```
char2 <- paste("Hey", "momma", "I'm", "a", "string")
char2

## [1] "Hey momma I'm a string"
```

NOTE: The following section on string operations can be skipped if time is an issue.

Blankspace is the default separater in the `paste()` function. If you don't want this and want the words to be smushed together, use the argument `sep=""`

```
char3 <- paste("Hey", "momma", "I'm", "a", "string", sep=" ")
char3

## [1] "Hey momma I'm a string"
```

Note here that R is not a zero-indexed language - lists begin at the number 1. The one and only element in this object is the sentence "Hey momma I'm a string".

2. Atomic data types in R/ character/ "arguments"

`sep=""` is what is called an argument. This is a command within the parentheses of another command `paste(sep="")`

Most commands require one or two arguments to be defined in order for the command to properly run. You will find that commands are full of default and optional arguments that you can manipulate. Use `?` to get their definitions.

Return to line 300 above and put a space between the quotation marks so that `sep=" "`. This will ensure the words in your sentence will be separated by a blankspace.

2. Atomic data types in R/ character/ `substr()`

`substr()` lets you extract text from certain character positions in character data. If we want to extract just the first four characters of the `char2` object we type:

```
substr(char2, 1, 4)

## [1] "Hey "
```

"Hey " (Hey + blankspace) is returned.

You can use `substr()` and the assignment operator `<-` to redefine the first four characters in `char2` with the word "Yes " followed by a blankspace

```
substr(char2, 1, 4) <- "Yes "
char2

## [1] "Yes momma I'm a string"
```

What changed?

2. Atomic data types in R/ character/ `strsplit()`

You can separate characters with `strsplit()`. We can break the sentence in `char2` apart at its whitespaces with `strsplit()`. Note the use of `" "` to tell R that we want to separate the sentence at the blank spaces between the words.

```
char4 <- strsplit(char2, " ")
char4

## [[1]]
## [1] "Yes"    "momma"  "I'm"    "a"      "string"
```

We are now back to our original words!

2. Atomic data types in R/ character/ `gsub()`

You can also substitute with `gsub()`. Let's say we want to substitute all periods `.` in `char2` with `X`. We would type:

```
gsub(".", "X", char2)

## [1] "XXXXXXXXXXXXXXXXXXXXXX"
```

What happened here? R here calls Perl's regular expressions package, where `.` is a special shorthand for "everything else".

To be safe, put the period in brackets so that R views it as a subsetting operation and not "everything else".

```
gsub("[.]", "X", char2)

## [1] "Yes momma I'm a string"
```

Still nothing happened... Why not? (hint: we don't have any periods in our sentence!). Let's choose another letter:

```
gsub("[m]", "X", char2)

## [1] "Yes XoXXa I'X a string"

# All "m" letters have now been replaced with "X". Remember that the
`char2` object has not changed because we did not reassign it!
char2 <- gsub("[m]", "X", char2)
char2

## [1] "Yes XoXXa I'X a string"
```

Atomic data types in R/ logical

Logical (boolean) data are dichotomous TRUE/FALSE values. R internally stores `FALSE` as 0 and `TRUE` as 1. Define a logical object:

```
bool_object <- TRUE
bool_object
```

```
## [1] TRUE
```

```
class(bool_object)
```

```
## [1] "logical"
```

Note that logical data also take on numeric properties

```
bool_object + 1
```

```
## [1] 2
```

```
TRUE - TRUE
```

```
## [1] 0
```

```
TRUE + FALSE
```

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

```
FALSE - TRUE
```

```
## [1] -1
```

2. Atomic data types in R/ logical/ logical tests

Logical tests are helpful in R if you want to check for equivalence. This is useful for removing missing data and subsetting. Note the use of the double equals == sign.

```
?"=="
?"&"
?"|"
?"!"
```

```
TRUE == TRUE
```

```
## [1] TRUE
```

```
FALSE == FALSE
```

```
## [1] TRUE
```

```
TRUE == FALSE
```

```
## [1] FALSE
```

```
TRUE & TRUE   # and
```

```
## [1] TRUE
```

```
TRUE | FALSE  # or
```

```
## [1] TRUE
```

```
TRUE != FALSE  # not
```

```
## [1] TRUE
```

```
TRUE > FALSE
```

```
## [1] TRUE
```

```
FALSE >= TRUE
```

```
## [1] FALSE
```

2. *Challenge 2*

1. Define a numeric object and check its class
2. Define a boolean object and check its class
3. Define a character object that uses `paste()` to combine more than one word into a sentence. Then use `substr()` to redefine a part of this sentence. Finally, use `gsub()` to substitute part of this new sentence with a new word.

3. Data testing and type coercion/ `is.type()`

"Type coercion" refers to changing the data from one type to another.

Often it is handy to see what types of data you are working with. Similar to `class()` we can see what data type an object is with `is.type`. A logical response is returned:

```
is.numeric(num)
```

```
## [1] TRUE
```

```
is.logical(bool_object)
```

```
## [1] TRUE
```

```
is.logical(char2)
```

```
## [1] FALSE
```

```
class(char2)
```

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

Data testing and type coercion/ `as.type()`

You can change data types with `as.type()`

```r
# Create some character data
char_data <- "9"
class(char_data)

## [1] "character"

# Coerce this character data to numeric data type
as.numeric(char_data)

## [1] 9

class(char_data)

## [1] "character"

# What happened here? Why did `char_data` not change classes? (hint: we
did not overwrite the object!)
char.data <- as.numeric(char_data)
class(char.data)

## [1] "numeric"

char.data

## [1] 9
```

Data testing and type coercion/ `as.integer()` and `L`

You can change numeric type to integer type using `as.integer()` and `L`

```r
num <- 4
class(num)

## [1] "numeric"

int <- as.integer(num)
class(int)

## [1] "integer"

# or
int <- 2L
class(int)

## [1] "integer"
```

Now, create some character data and try to convert it to integer type:

```r
# Create a new object
char.num <- "three"
char.num   #Note that the word three is contained within " "

## [1] "three"
```

```r
class(char.num)
```

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

```r
# What happens if we try to coerce character to numeric data type?
as.integer(char.num)
```

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

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

Why did this fail? Can R change character data to numbers? Why not? (hint: R does not know how to automatically coerce words to numbers). As you can see, trying to coerce data types can lead to weird behavior sometimes.

3. *Challenge 3*

1. Create a character object and check its type using `is.type`

2. Try to change ("coerce") this object to another data type using `as.type`

3. Create a numeric vector of class "integer"

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

4. Data structures/

There are several kinds of data structures in R. These four are the most common:

1. vector
2. list
3. matrix
4. dataframe

4. Data structures/ vector

A ***VECTOR*** is an ordered group of the same kind of data. Vectors are one-dimensional and homogenous, and are thus referred to by their type (e.g., character vector, numeric vector, logical vector).

Create a numeric vector by combining/concatenating elements with `c()`

```r
?c()
```

```r
numeric_vector <- c(3, 5, 6, 5, 3)
numeric_vector
```

```
## [1] 3 5 6 5 3
```

You can also add items to a vector using `c()` and a comma `,` (as long as it is the same data type)

```
numeric_vector2 <- c(numeric_vector, 78)
numeric_vector2

## [1]  3  5  6  5  3 78
```

It doesn't matter what the datatype is for a vector, as long as it is all the same

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

## [1]  TRUE  TRUE FALSE FALSE  TRUE

logical_vector2 <- c(logical_vector, c(FALSE, FALSE, FALSE))
logical_vector2

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

You can also add and multiply vectors, but they need to be the same length

```
logical_vector * logical_vector

## [1] 1 1 0 0 1
```

What happens when we multiply `c(1,2,3,4) * c(TRUE, FALSE)` ?

```
c(1,2,3,4) * c(TRUE, FALSE)

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

Since the number of elements in the first vector (four) is a multiple of the length of the second vector (two), the second vector gets concatenated against itself two times. This is called "recycling".

4. Data structures/ vector/ `seq()`

You might need to create vectors that are sequences of numbers. You can do this via `seq()`. Here we specify a vector from zero to the length of our object `logical_vector2` (eight). The argument `by` tells R that we want only the even numbers!

```
?length

length(logical_vector2)

## [1] 8

seq(from=0,to=length(logical_vector2),by=2)

## [1] 0 2 4 6 8
```

4. Data structures/ vector/ :

R also gives you a shorthand operator for creating sequences in whole number increments of 1. This is the colon symbol :

```
0:8

## [1] 0 1 2 3 4 5 6 7 8

c(28:36)

## [1] 28 29 30 31 32 33 34 35 36

0:length(logical_vector2)

## [1] 0 1 2 3 4 5 6 7 8
```

4. Data structures/ vector/ random sampling

You can also sample random groups of numbers. You can use `set.seed()` to ensure that we all always get the same random draws from the parent universe, even on different machines

```
?set.seed
?runif
?rnorm
?sample
```

Set the seed, and then choose our values:

```
set.seed(1)

uniform <- runif(20, 3, 7) # 20 random samples from uniform
distribution between the numbers 3 and 7
uniform

##  [1] 4.062035 4.488496 5.291413 6.632831 3.806728 6.593559 6.778701
5.643191 5.516456 3.247145 3.823898 3.706227
## [13] 5.748091 4.536415 6.079366 4.990797 5.870474 6.967624 4.520141
6.109781

normal <- rnorm(20, 0, 1) # 20 random samples from the normal
distribution with a mean of 0 and standard deviation of 1
normal

##  [1]  1.51178117  0.38984324 -0.62124058 -2.21469989  1.12493092 -
0.04493361 -0.01619026  0.94383621  0.82122120
## [10]  0.59390132  0.91897737  0.78213630  0.07456498 -1.98935170
0.61982575 -0.05612874 -0.15579551 -1.47075238
## [19] -0.47815006  0.41794156
```

```
integer <- sample(5:10, 20, replace=TRUE) # 20 random samples from
between the numbers 5 and 10. Note that `replace=TRUE` signifies that
it is OK to reuse numbers already selected.
integer
```

```
##  [1] 10  6  7  6  8  6  7  9  5 10  7 10  7  7  7 10 10  7  9 10
```

```
character <- sample(c("Cat", "Dog", "Pig"), 20, replace=TRUE) # 20
random samples of character data
character
```

```
##  [1] "Dog" "Pig" "Dog" "Cat" "Pig" "Cat" "Pig" "Cat" "Cat" "Cat"
"Cat" "Cat" "Dog" "Pig" "Pig" "Pig" "Dog" "Dog" "Pig"
## [20] "Dog"
```

```
logical <- sample(c(TRUE, FALSE), 20, replace=TRUE) # 20 random samples
of logical data
logical
```

```
##  [1] FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
## [20] FALSE
```

4. Data structures/ list

A ***LIST*** is an ordered group of data that are not of the same type. Lists are heterogenous. Instead of using `c()` like in vector creation, use `list()` to create a list:

```
?list
```

```
list_object <- list(TRUE, "one", 1) # include three kinds of data:
logical, character, and integer
list_object
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] "one"
##
## [[3]]
## [1] 1
```

```
class(list_object)
```

```
## [1] "list"
```

Lists are simple containers and are not additive or multiplicative like vectors and matrices are:

```
list_object * list(FALSE, "zero", 0) # Error
```

4. Data structures/ matrix

Matrices are homogenous like vectors. They are tables comprised of data all of the same type. Matrices are organized into rows and columns.

Use `matrix()` to create a matrix

```
?matrix
```

We can also specify how we want the matrix to be organized using the `nrow` and `ncol` arguments:

```
matrix1 <- matrix(1:12, nrow = 4, ncol = 3) # this makes a 4 x 3 matrix
matrix1
```

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

```
class(matrix1)
```

```
## [1] "matrix"
```

We can also coerce a vector to a matrix, because a vector is comprised of homogenous data of the same kind, just like a matrix is:

```
# Create a numeric vector from 1 to 20
vec1 <- c(1:20)
vec1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
class(vec1)
```

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

```
# Coerce this vector to a matrix with 10 rows and 2 columns:
matrix2 <- matrix(vec1, ncol=2)
matrix2
```

```
##       [,1] [,2]
##  [1,]    1   11
##  [2,]    2   12
##  [3,]    3   13
##  [4,]    4   14
##  [5,]    5   15
##  [6,]    6   16
##  [7,]    7   17
##  [8,]    8   18
```

```
##  [9,]    9   19
## [10,]   10   20

class(matrix2)

## [1] "matrix"
```

4. Data structures/ data frame

Inside R, a dataframe is a list of equal-length vectors. These vectors can be of different types. Data frames are heterogenous like lists.

Let's create a dataframe called `animals` using the vectors we already created:

```
uniform
normal
integer
character
logical
```

We do this using `data.frame()`

```
?data.frame

animals <- data.frame(uniform, normal, integer, character, logical,
stringsAsFactors=FALSE)
# NOTE: `stringsAsFactors=FALSE` means that R will NOT try to interpret
character data as factor type. More on this below.
```

Take a peek at the `animals` data frame to see what it looks like:

```
head(animals)

##    uniform      normal integer character logical
## 1 4.062035  1.51178117      10       Dog   FALSE
## 2 4.488496  0.38984324       6       Pig    TRUE
## 3 5.291413 -0.62124058       7       Dog    TRUE
## 4 6.632831 -2.21469989       6       Cat   FALSE
## 5 3.806728  1.12493092       8       Pig   FALSE
## 6 6.593559 -0.04493361       6       Cat    TRUE
```

We can change the names of the columns by passing into it a vector with our desired names

```
# Create a vector called `new_df_names` with the new column names:
new_df_names <- c("Weight", "Progress", "Height", "Name", "Healthy")

# Pass this vector into `colnames()`
colnames(animals) <- new_df_names
head(animals)
```

```
##     Weight    Progress Height Name Healthy
## 1 4.062035  1.51178117     10  Dog   FALSE
## 2 4.488496  0.38984324      6  Pig    TRUE
## 3 5.291413 -0.62124058      7  Dog    TRUE
## 4 6.632831 -2.21469989      6  Cat   FALSE
## 5 3.806728  1.12493092      8  Pig   FALSE
## 6 6.593559 -0.04493361      6  Cat    TRUE
```

We can check the structure of our data frame via `str()`

```
?str

str(animals)

## 'data.frame':    20 obs. of  5 variables:
##  $ Weight  : num  4.06 4.49 5.29 6.63 3.81 ...
##  $ Progress: num  1.512 0.39 -0.621 -2.215 1.125 ...
##  $ Height  : int  10 6 7 6 8 6 7 9 5 10 ...
##  $ Name    : chr  "Dog" "Pig" "Dog" "Cat" ...
##  $ Healthy : logi  FALSE TRUE TRUE FALSE FALSE TRUE ...
```

Here, we can see that a data frame is just a list of equal-length vectors! In `str()`, our columns are displayed top to bottom, while the data are displayed left to right.

4. Data structures/ data frame/ learn about your data frame

```
# View the dimensions (nrow x ncol) of the data frame:
dim(animals)

## [1] 20  5

# View column names:
colnames(animals)

## [1] "Weight"   "Progress" "Height"   "Name"     "Healthy"

# View row names (we did not change these, so they default to character
type)
rownames(animals)

##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12"
"13" "14" "15" "16" "17" "18" "19" "20"

class(rownames(animals))

## [1] "character"
```

4. Data structures/ data frame/ change the order of the columns

You can change the order of the columns by specifying their new order using `c()` within what is called "bracket notation" `[]`.

This will be covered with the rest of subsetting in Part 2.

```
animals <- animals[,c("Name", "Healthy", "Weight", "Height",
"Progress")]
head(animals)

##   Name Healthy   Weight Height    Progress
## 1  Dog   FALSE 4.062035     10  1.51178117
## 2  Pig    TRUE 4.488496      6  0.38984324
## 3  Dog    TRUE 5.291413      7 -0.62124058
## 4  Cat   FALSE 6.632831      6 -2.21469989
## 5  Pig   FALSE 3.806728      8  1.12493092
## 6  Cat    TRUE 6.593559      6 -0.04493361
```

4. Data structures/ data frame/ factor data type

Factor data are categorical types used to make comparisons between data. Factors group the data by their "levels" (the different categories within a factor) for comparative purposes. One overtly simple example of a factor variable is "type of fruit" - its factor "levels" would be fruits such as apple, banana, watermelon, orange, etc.

For example, in our `animals` dataframe, we might want to compare heights and weights between Cat, Dog, and Pigs. In this case, we want to set the character type "Name" vector to factor data type. We can do so with `factor()`

The dollar sign operator `$` is used to call a single vector from a data frame. This will be discussed more in Part 2 along with the rest of subsetting.

```
str(animals)   # "Name" is character data type. See how each column
name is preceded by `$`?

## 'data.frame':    20 obs. of  5 variables:
##  $ Name    : chr  "Dog" "Pig" "Dog" "Cat" ...
##  $ Healthy : logi  FALSE TRUE TRUE FALSE FALSE TRUE ...
##  $ Weight  : num  4.06 4.49 5.29 6.63 3.81 ...
##  $ Height  : int  10 6 7 6 8 6 7 9 5 10 ...
##  $ Progress: num  1.512 0.39 -0.621 -2.215 1.125 ...

animals$Name <- factor(animals$Name)
str(animals)   # "Name" is now factor data type!

## 'data.frame':    20 obs. of  5 variables:
##  $ Name    : Factor w/ 3 levels "Cat","Dog","Pig": 2 3 2 1 3 1 3 1 1
1 ...
##  $ Healthy : logi  FALSE TRUE TRUE FALSE FALSE TRUE ...
##  $ Weight  : num  4.06 4.49 5.29 6.63 3.81 ...
##  $ Height  : int  10 6 7 6 8 6 7 9 5 10 ...
##  $ Progress: num  1.512 0.39 -0.621 -2.215 1.125 ...
```

Notice that R stores factors internally as integers and uses the character strings as labels. Also notice that by default R orders factors alphabetically and returns them when we call the "Name"" vector.

```
animals$Name

##  [1] Dog Pig Dog Cat Pig Cat Pig Cat Cat Cat Cat Cat Dog Pig Pig Pig
Dog Dog Pig Dog
## Levels: Cat Dog Pig
```

4. Data structures/ data frame/ factor data type/ changing factor "levels"

Each animal type (Cat, Dog, and Pig) within the factor `animals$Name` vector are the factor levels.

If we want to change how R stores the factor levels, we can specify their levels using the `levels()` argument. For example:

```
animals$Name  # Levels: Cat Dog Pig (default alphabetical sort)

##  [1] Dog Pig Dog Cat Pig Cat Pig Cat Cat Cat Cat Cat Dog Pig Pig Pig
Dog Dog Pig Dog
## Levels: Cat Dog Pig

# What if we want to change the factor level sort to Levels: Dog Pig
Cat?
animals$Name <- factor(animals$Name, levels = c("Dog", "Pig", "Cat"))
animals$Name # Now when we call animals$Name, we can see that the
levels have changed

##  [1] Dog Pig Dog Cat Pig Cat Pig Cat Cat Cat Cat Cat Dog Pig Pig Pig
Dog Dog Pig Dog
## Levels: Dog Pig Cat
```

NOTE: Make sure to keep the `animals` data frame saved in your working folder because we will use it for Parts 2, 3 and 4.

4. *Challenge 4*

1. Vectors
2. Create a vector of a sequence of numbers between 1 to 10.
3. Coerce that vector into a character vector
4. Add the element "11" to the end of the vector
5. Evaluate the `str` of the vector.
6. Create a sentence from separate words using `paste()`. Can you guess how to add another word to this vector without creating a new vector?

7. Lists
8. How does a list differ from an atomic vector?
9. Create three objects of different types and lengths and then combine them into a list names `x`.

10. If x is a list, what is the class of `x[1]`? How about `x[[1]]`? (this is a preview of Part 2).

11. Data frames
12. Create a 3x2 data frame called `basket`. List the name of each fruit in the first column and its price in teh second column.
13. Now give your dataframe appropriate column and row names.
14. We can add a new column using `$` (this will be covered with data subsetting in Part 2). Can you guess how to add a third column called "Color", that shows the color of each fruit?

Basket example:

```
f <- factor(c("Apple", "Orange", "Pear"))
p <- c(10, 28, 36)
basket <- data.frame(f, p)
basket
```

```
##        f  p
## 1  Apple 10
## 2 Orange 28
## 3   Pear 36
```

```
colnames(basket) <- c("Fruit", "Price")
basket
```

```
##    Fruit Price
## 1  Apple    10
## 2 Orange    28
## 3   Pear    36
```

```
basket$Color <- factor(c("Red", "Orange", "Green"))
str(basket)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ Fruit: Factor w/ 3 levels "Apple","Orange",..: 1 2 3
##  $ Price: num  10 28 36
##  $ Color: Factor w/ 3 levels "Green","Orange",..: 3 2 1
```

```
basket
```

```
##    Fruit Price  Color
## 1  Apple    10    Red
## 2 Orange    28 Orange
## 3   Pear    36  Green
```

5. Saving your work/ `write.csv()`

It is always handy to save your work. Saving a dataframe as a .CSV file is a convenient way to store it for future use. ***Anything saved will be placed into your working directory!***

The syntax looks like this: `write.csv(object_name, "nameOfFile.csv", row.names=TRUE)`

```
?write.csv

write.csv(animals, "animals.csv", row.names=FALSE)
write.csv(basket, "basket.csv", row.names=TRUE)
```

Check your working directory! :)

5. Saving your work/ `sink()`

You can also save a dataframe as a .txt file using sink(). To use a sink, put `sink("FileName.txt")` as the line before the code they want to save, and `sink()` as the line after.

```
?sink

sink("animals.txt")
animals

##    Name Healthy   Weight Height    Progress
## 1   Dog   FALSE 4.062035     10  1.51178117
## 2   Pig    TRUE 4.488496      6  0.38984324
## 3   Dog    TRUE 5.291413      7 -0.62124058
## 4   Cat   FALSE 6.632831      6 -2.21469989
## 5   Pig   FALSE 3.806728      8  1.12493092
## 6   Cat    TRUE 6.593559      6 -0.04493361
## 7   Pig    TRUE 6.778701      7 -0.01619026
## 8   Cat    TRUE 5.643191      9  0.94383621
## 9   Cat   FALSE 5.516456      5  0.82122120
## 10  Cat   FALSE 3.247145     10  0.59390132
## 11  Cat   FALSE 3.823898      7  0.91897737
## 12  Cat   FALSE 3.706227     10  0.78213630
## 13  Dog    TRUE 5.748091      7  0.07456498
## 14  Pig    TRUE 4.536415      7 -1.98935170
## 15  Pig    TRUE 6.079366      7  0.61982575
## 16  Pig    TRUE 4.990797     10 -0.05612874
## 17  Dog   FALSE 5.870474     10 -0.15579551
## 18  Dog    TRUE 6.967624      7 -1.47075238
## 19  Pig    TRUE 4.520141      9 -0.47815006
## 20  Dog   FALSE 6.109781     10  0.41794156

sink()
```

If you ever get stuck in a sink(), run `sink()` as many times as necessary until you get a warning message that says "no sink to remove".

We will talk about saving plots as .PNG and .PDF file types in Part 3!

Acknowledgements

Hadley Wickham