R Fundamentals Part 1: Introduction

Evan Muzzall, Rochelle Terman, Dillon Niederhut

December 6, 2016

Table of Contents

Workshop goals 2

Part 1 Learning objectives 3

0. Front matter: Pre-introduction 3

0. Front matter: Introduction to the class 3

About the UC Berkeley Social Sciences Data Laboratory (D-Lab) 3

1. Introduction - what is RStudio? 4

1. Introduction - package installation 5

1. Introduction - objects 5

1. Introduction - the R global environment 6

1. Introduction - variable assignment 6

1. Introduction - variable assignment - naming R objects 7

1. Introduction - variable assignment - class() 8

1. Introduction - variable assignment - rm() 8

1. Introduction - help 8

1. Introduction -living in R 9

Challenge 1 10

2. Atomic data types in R 10

2. Atomic data types in R - numeric 10

2. Atomic data types in R - character 11

2. Atomic data types in R - character - paste() 12

2. Atomic data types in R - character - "arguments" 12

2. Atomic data types in R - character - substr() 12

2. Atomic data types in R - character - strsplit() 13

2. Atomic data types in R - character - gsub() 13

Atomic data types in R - logical 14

2. Atomic data types in R - logical - logical tests 14

Challenge 2 15

3. Data testing and type coercion - is.type() 15

Data testing and type coercion - as.type() 16

Data testing and type coercion - as.integer() and L 17

Challenge 3 18

4. Data structures 18

4. Data structures - vector 18

4. Data structures - vector - seq() 19

4. Data structures - vector - : 20

4. Data structure - vector - random sampling 20

4. Data structures - list 21

4. Data structures - matrix 22

4. Data structures - data frame 23

4. Data structures - data frame - learn about your data frame 24

4. Data structures - data frame - change the order of the columns 24

4. Data structures - data frame - factor data type 25

4. Data structures - data frame - factor data type - changing factor "levels" 26

Challenge 4 26

5. Saving your work - write.csv() 28

5. Saving your work - sink() 28

6. Homework (optional) 28

Acknowledgements 29

This is an R markdown (.Rmd) file. It allows you create documents of various kinds. Check out the [R Markdown Quick Tour](http://rmarkdown.rstudio.com/authoring_quick_tour.html)

# Workshop goals

The goals of this workshop are divided into four parts:  
Part 1) Introduction to R, data types, data structures  
Part 2) Importing, subsetting, and merging data; identifying missing data  
Part 3) Data exploration and statistical testing  
Part 4) For-loops and functions

# Part 1 Learning objectives

1. Front matter
2. Introduction: What is RStudio?, objects, global environment, packages, variable assignment, help
3. Atomic data types: numeric, character, and logical
4. Data type coercion
5. Data structures: vector, list, matrix, data frame
6. 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.

## About the UC Berkeley Social Sciences Data Laboratory (D-Lab)

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 practical training, staff support, resources, and space to enable you to use R for your own research applications. Our services cater to all skill levels and no programming, statistical, or computer science backgrounds are 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](http://dlab.berkeley.edu/calendar-node-field-date)  
[D-Lab Consulting](http://dlab.berkeley.edu/consulting)  
[D-lab Working Groups](http://dlab.berkeley.edu/working-groups)

It is a good idea to start off the class by explpaining to folks why you (the instructor) use R. As the instructor, you should 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
5. It is open-sourced with strong community support and help and is used by virtually all academic disciplines as well as the private sector

Be sure to draw on your own experiences to include further examples and advice, especially for students who do not fall into one of the five categories above and/or are just beginning to learn R. Be sure to discuss a motivating example personal to you.

**If you are a student and you are looking at this R 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.

Remember, our goal is to teach you research applications in R. Check out the UC Berkeley [Department of Statistics](http://statistics.berkeley.edu/), [Interdepartmental Group in Biostatistics](https://www.stat.berkeley.edu/biostat/), and [EECS](https://cs.berkeley.edu/).

# 1. Introduction - what is RStudio?

R is old programming language based on the "S" and "S-PLUS" language and graphics user interface (GUI) and is both *functional* and *object oriented*. These themes make R highly extensible.

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 RStudio here](https://www.rstudio.com/products/rstudio/download/)

After opening RStudio, 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 R finds an appropriate stop parameter, often a closed parenthesis ), closed bracket], or other statement.

**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).

It is generally good to keep RStudio and your packages up to date!

## 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.

See ?update.packages for help updating packages.

library(psych)  
# the psych package is now loaded in our current R instance

## 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/variable assignment requires three pieces of information**:  
1) object\_name  
2) <-  
3) definition/assignment

[Brunsdon and Comber 2015, page 15](https://us.sagepub.com/en-us/nam/an-introduction-to-r-for-spatial-analysis-and-mapping/book241031) offer a useful summary table of logical operators in R. We will talk more about these symbols in Parts 2, 3, and 4.

|  |  |
| --- | --- |
| Logical operator | Description |
| == | Equal |
| != | Not equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| ! | Not (goes in front of other expressions) |
| & | And (combines expressions) |
| | | Or (combines expressions) |

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. We will talk more about data types below.

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 of this workshop series.

Double question marks ?? will lead you to coding walkthroughs called "vignettes". These usually come with preloaded data and examples:

??"<-"  
??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, browse their contents, or use their built-in search tools:  
[Quick-R](http://statmethods.net/)  
[UCLA idre](http://www.ats.ucla.edu/stat/r/)  
[R-bloggers](https://www.r-bloggers.com/)  
[Stack Overflow - R](http://stackoverflow.com/questions/tagged/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/evanmuzzall/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 "evanmuzzall" and my "R-Fundamentals-master" folder is located on the Desktop, so the file path would look like this:

setwd("/Users/evanmuzzall/Desktop/R-Fundamentals-master")  
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") # note that PC users must use "C:/..."

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

dir()

## [1] "LICENSE"   
## [2] "R Fundamentals Part 1 Introduction.Rmd"   
## [3] "R Fundamentals Part 2 Subsetting and reshaping.Rmd"   
## [4] "R Fundamentals Part 3 Data exploration and analysis.Rmd"  
## [5] "R Fundamentals Part 4 For-loops and functions.Rmd"   
## [6] "R\_Fundamentals\_Part\_1\_Introduction.Rmd"   
## [7] "README.md"

# 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".

Also see ?grep for more tools.

## 2. Atomic data types in R - character - "arguments"

Arguments go inside of the parentheses of R commands. Unnamed arguments are things like the number 4. sep="" is what is called a named argument. This is a command within the parentheses of another command paste()

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.

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?

You will also see that = is frequently used in place of the assignment operator. This works the same, but we recommend using the assignment operator for consistency. Arguments and functions frequently use = and it is best to consistently assign your variables with <-

## 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"

We recommend to always spell out TRUE and FALSE instead of abbreviating them T and F which might be mistaken for previously defined objects. R has many reserved words that you should avoid using. See ?reserved for more information.

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. Logical tests compare two objects and return a logical output. This is useful for removing missing data and subsetting (you will learn more about this in Part 2). 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

# Challenge 2

1. Define a numeric object as 2 and check its class
2. Define a boolean object as "2" and check its class
3. Use == to compare your numeric and boolean object. Are they equivalent? Why or why not?
4. 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.

[Brunsdon and Comber 2015, page 102](https://us.sagepub.com/en-us/nam/an-introduction-to-r-for-spatial-analysis-and-mapping/book241031) also offer an excellent summary table of logical functions in R.

|  |  |
| --- | --- |
| Logical function | Description |
| any(x) | TRUE if any in a vector of conditions x is true |
| all(x) | TRUE if all of a vector of conditions x is true |
| is.numeric(x) | TRUE if x contains a numeric value |
| is.character(x) | TRUE if x contains a character value |
| is.logical(x) | TRUE if x contains a true or false value |

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().

[Brunsdon and Comber 2015, page 102](https://us.sagepub.com/en-us/nam/an-introduction-to-r-for-spatial-analysis-and-mapping/book241031) also offer an excellent summary table of data types in R.

|  |  |  |
| --- | --- | --- |
| Type | Test | Conversion |
| character | is.character | as.character |
| complex | is.complex | as.complex |
| double | is.double | as.double |
| expression | is.expression | as.expression |
| integer | is.integer | as.integer |
| list | is.list | as.list |
| logical | is.logical | as.logical |
| numeric | is.numeric | as.numeric |
| single | is.single | as.single |
| raw | is.raw | as.raw |

# 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

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:

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

## [1] "three"

class(char.num)

## [1] "character"

# 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 has no protocol for automatically coerce words to numbers). As you can see, trying to coerce data types can lead to weird behavior sometimes.

# 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. Data structures are collections of data objects (e.g., numeric, character, and logical vectors, lists, and matrices) that work together. 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. "Ordered" means that their position matters. 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()

?c

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=2 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

sequence\_object <- c(28:36)  
sequence\_object

## [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 structure - vector - random sampling

You can also sample random groups of numbers. 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  
## [8] 5.643191 5.516456 3.247145 3.823898 3.706227 5.748091 4.536415  
## [15] 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  
## [6] -0.04493361 -0.01619026 0.94383621 0.82122120 0.59390132  
## [11] 0.91897737 0.78213630 0.07456498 -1.98935170 0.61982575  
## [16] -0.05612874 -0.15579551 -1.47075238 -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"  
## [12] "Cat" "Dog" "Pig" "Pig" "Pig" "Dog" "Dog" "Pig" "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  
## [12] FALSE TRUE TRUE TRUE TRUE FALSE TRUE TRUE 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

*It is worth emphasizing the importance of data frames in R!* Inside R, a **DATA FRAME** 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", "Type", "Healthy")  
  
# Pass this vector into `colnames()`  
colnames(animals) <- new\_df\_names  
head(animals)

## Weight Progress Height Type 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 ...  
## $ Type : 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" "Type" "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] "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("Type", "Healthy", "Weight", "Height", "Progress")]  
head(animals)

## Type 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.

For example, in our animals dataframe, we can coerce "Type" from character to factor data type. Cat, Dog, and Pig are the factor levels. If we might want to compare heights and weights between Cat, Dog, and Pigs, we set the character "Type" 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:  
## $ Type : 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$Type <- factor(animals$Type)  
str(animals) # "Name" is now factor data type!

## 'data.frame': 20 obs. of 5 variables:  
## $ Type : 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 "Type" vector.

animals$Type

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

levels(animals$Type)

## [1] "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$Type 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$Type # 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  
## [18] Dog Pig Dog  
## Levels: Cat Dog Pig

# What if we want to change the factor level sort to Levels: Dog Pig Cat?  
animals$Type <- factor(animals$Type, levels = c("Dog", "Pig", "Cat"))  
animals$Type # Now when we call animals$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  
## [18] 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.

# Challenge 4

Vectors 1.1 Create a vector of a sequence of numbers between 1 to 10. 1.2 Coerce that vector into a character vector 1.3 Add the element "11" to the end of the vector 1.4 Evaluate the str of the vector. 1.5 Create a sentence from separate words using paste(). Can you guess how to add another word to this vector without creating a new vector?

Lists 2.1 How does a list differ from an atomic vector? 2.2 Create three objects of different types and lengths and then combine them into a list names x. 2.3 If x is a list, what is the class of x[1]? How about x[[1]]? (this is a preview of Part 2).

Data frames 3.1 Create a 3x2 data frame called basket. List the name of each fruit in the first column and its price in teh second column. 3.2 Now give your dataframe appropriate column and row names. 3.3 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 solution:

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  
sink()

If you ever find that your code stops working after saving via sink(), you are probably stuck in the 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!

# 6. Homework (optional)

swirl is a package that helps you learn R by using R. Visit the [swirl](http://swirlstats.com/) homepage to learn more

install.packages("swirl")  
library(swirl)  
swirl()  
# follow the instructions until you can select number 1 "R Programming: The basics of programming in R"

# Acknowledgements

[Wickham H. 2014. Advanced R](http://adv-r.had.co.nz/)  
[Lander J. 2013. R for everyone: Advanced analytics and graphics](http://www.jaredlander.com/r-for-everyone/)  
[Matloff N. 2011. The art of R programming: A tour of statistical software design](https://www.nostarch.com/artofr.htm)  
[Brunsdon C, Comber L. 2015. An Introduction to R for Spatial Analysis and Mapping](https://us.sagepub.com/en-us/nam/an-introduction-to-r-for-spatial-analysis-and-mapping/book241031)