Data Science R Basics

The textbook for the Data Science course series is [freely available online](https://rafalab.github.io/dsbook/).

# Learning Objectives

* Learn to read, extract, and create datasets in R
* Learn to perform a variety of operations on datasets using R
* Learn to write your own functions/sub-routines in R

## Course Overview

### Section 1: R Basics, Functions, Data types

You will get started with R, learn about its functions and data types.

### Section 2: Vectors, Sorting

You will learn to operate on vectors and advanced functions such as sorting.

### Section 3: Indexing, Data Manipulation, Plots

You will learn to wrangle and visualize data.

### Section 4: Programming Basics

You will learn to use general programming features like ‘if-else’, and ‘for loop’ commands, and write your own functions to perform various operations on datasets.

# Section 1 Overview

Section 1 introduces you to R Basics, Functions and Datatypes.

In Section 1, you will learn to:

* Appreciate the rationale for data analysis using R
* Define objects and perform basic arithmetic and logical operations
* Use pre-defined functions to perform operations on objects
* Distinguish between various data types

The textbook for this section is available [here](https://rafalab.github.io/dsbook/r-basics.html)

## Motivation

Here is a link to the [textbook section on the motivation for this course](https://rafalab.github.io/dsbook/r-basics.html#case-study-us-gun-murders).

## Getting started

Here is a link to the textbook section on [Getting Started](https://rafalab.github.io/dsbook/getting-started.html) with R.

**Key Points**

* R was developed by statisticians and data analysts as an interactive environment for data analysis.
* Some of the advantages of R are that (1) it is free and open source, (2) it has the capability to save scripts, (3) there are numerous resources for learning, and (4) it is easy for developers to share software implementation.
* Expressions are evaluated in the R console when you type the expression into the console and hit Return.
* A great advantage of R over point and click analysis software is that you can save your work as scripts.
* “Base R” is what you get after you first install R. Additional components are available via packages.

# installing the dslabs package  
if(!require(dslabs)) install.packages("dslabs")

## Loading required package: dslabs

# loading the dslabs package into the R session  
library(dslabs)

## Installing R and R Studio

### Installing R

To install R to work on your own computer, you can download it freely from the [Comprehensive R Archive Network](https://cran.r-project.org) (CRAN). Note that CRAN makes several versions of R available: versions for multiple operating systems and releases older than the current one. You want to read the CRAN instructions to assure you download the correct version. If you need further help, you read the walkthrough in this [Chapter of the textbook](https://rafalab.github.io/dsbook/installing-r-rstudio.html#installing-r).

### Installing RStudio

RStudio is an integrated development environment (IDE). We highly recommend installing and using RStudio to edit and test your code. You can install RStudio through the [RStudio website](https://rstudio.com/products/rstudio/download/). Their [cheatsheet](https://rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf) is a great resource. You must install R before installing RStudio.

### Textbook Link

Here is a link to the textbook section on [Installing R and RStudio](https://rafalab.github.io/dsbook/installing-r-rstudio.html).

## R Basics - Objects

Here is a link to the textbook section on [objects in R](https://rafalab.github.io/dsbook/r-basics.html#objects).

**Key Points**

* To define a variable, we may use the assignment symbol “<-“.
* There are two ways to see the value stored in a variable: (1) type the variable into the console and hit Return, or (2) type print(“variable name”) and hit Return.
* Objects are stuff that is stored in R. They can be variables, functions, etc.
* The ls() function shows the names of the objects saved in your workspace.

*Solving the equation x2+x−1=0*

# assigning values to variables  
a <- 1  
b <- 1  
c <- -1  
  
# solving the quadratic equation  
(-b + sqrt(b^2 - 4\*a\*c) ) / ( 2\*a )

## [1] 0.618034

(-b - sqrt(b^2 - 4\*a\*c) ) / ( 2\*a )

## [1] -1.618034

## R Basics - Functions

Here is a link to the textbook section on [functions](https://rafalab.github.io/dsbook/r-basics.html#functions).

**Key points**

* In general, to evaluate a function we need to use parentheses. If we type a function without parenthesis, R shows us the code for the function. Most functions also require an argument, that is, something to be written inside the parenthesis.
* To access help files, we may use the help function help(“function name”), or write the question mark followed by the function name.
* The help file shows you the arguments the function is expecting, some of which are required and some are optional. If an argument is optional, a default value is assigned with the equal sign. The args() function also shows the arguments a function needs.
* To specify arguments, we use the equals sign. If no argument name is used, R assumes you’re entering arguments in the order shown in the help file.
* Creating and saving a script makes code much easier to execute.
* To make your code more readable, use intuitive variable names and include comments (using the “#” symbol) to remind yourself why you wrote a particular line of code.

## Assessment - R Basics

1. What is the sum of the first n positive integers? We can use the formula to quickly compute this quantity.

# Here is how you compute the sum for the first 20 integers  
20\*(20+1)/2

## [1] 210

# However, we can define a variable to use the formula for other values of n  
n <- 20  
n\*(n+1)/2

## [1] 210

n <- 25  
n\*(n+1)/2

## [1] 325

# Below, write code to calculate the sum of the first 100 integers  
n<-100  
n\*(n+1)/2

## [1] 5050

1. What is the sum of the first 1000 positive integers? We can use the formula to quickly compute this quantity.

# Below, write code to calculate the sum of the first 1000 integers   
n<-1000  
n\*(n+1)/2

## [1] 500500

1. Run the following code in the R console.

n <- 1000  
x <- seq(1, n)  
sum(x)

## [1] 500500

Based on the result, what do you think the functions seq and sum do?

* ☐ A. sum creates a list of numbers and seq adds them up.
* ☒ B. seq creates a list of numbers and sum adds them up.
* ☐ C. seq computes the difference between two arguments and sum computes the sum of 1 through 1000.
* ☐ D. sum always returns the same number.

1. In math and programming we say we evaluate a function when we replace arguments with specific values. So if we type log2(16) we evaluate the log2 function to get the log base 2 of 16 which is 4.

In R it is often useful to evaluate a function inside another function. For example, sqrt(log2(16)) will calculate the log to the base 2 of 16 and then compute the square root of that value. So the first evaluation gives a 4 and this gets evaluated by sqrt to give the final answer of 2.

# log to the base 2   
log2(16)

## [1] 4

# sqrt of the log to the base 2 of 16:  
sqrt(log2(16))

## [1] 2

# Compute log to the base 10 (log10) of the sqrt of 100. Do not use variables.  
log10(sqrt(100))

## [1] 1

1. Which of the following will always return the numeric value stored in x? You can try out examples and use the help system in the R console.

* ☐ A. log(10^x)
* ☐ B. log10(x^10)
* ☒ C. log(exp(x))
* ☐ D. exp(log(x, base = 2))

## Data Types

You can find the [section of the textbook on data types here](https://rafalab.github.io/dsbook/r-basics.html#data-types).

**Key Points**

* The function “class” helps us determine the type of an object.
* Data frames can be thought of as tables with rows representing observations and columns representing different variables.
* To access data from columns of a data frame, we use the dollar sign symbol, which is called the accessor.
* A vector is an object consisting of several entries and can be a numeric vector, a character vector, or a logical vector.
* We use quotes to distinguish between variable names and character strings.
* Factors are useful for storing categorical data, and are more memory efficient than storing characters.

*Code*

# loading the the murders dataset  
data(murders)  
  
# determining that the murders dataset is of the "data frame" class  
class(murders)

## [1] "data.frame"

# finding out more about the structure of the object  
str(murders)

## 'data.frame': 51 obs. of 5 variables:  
## $ state : chr "Alabama" "Alaska" "Arizona" "Arkansas" ...  
## $ abb : chr "AL" "AK" "AZ" "AR" ...  
## $ region : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...  
## $ population: num 4779736 710231 6392017 2915918 37253956 ...  
## $ total : num 135 19 232 93 1257 ...

# showing the first 6 lines of the dataset  
head(murders)

## state abb region population total  
## 1 Alabama AL South 4779736 135  
## 2 Alaska AK West 710231 19  
## 3 Arizona AZ West 6392017 232  
## 4 Arkansas AR South 2915918 93  
## 5 California CA West 37253956 1257  
## 6 Colorado CO West 5029196 65

# using the accessor operator to obtain the population column  
murders$population

## [1] 4779736 710231 6392017 2915918 37253956 5029196 3574097 897934  
## [9] 601723 19687653 9920000 1360301 1567582 12830632 6483802 3046355  
## [17] 2853118 4339367 4533372 1328361 5773552 6547629 9883640 5303925  
## [25] 2967297 5988927 989415 1826341 2700551 1316470 8791894 2059179  
## [33] 19378102 9535483 672591 11536504 3751351 3831074 12702379 1052567  
## [41] 4625364 814180 6346105 25145561 2763885 625741 8001024 6724540  
## [49] 1852994 5686986 563626

# displaying the variable names in the murders dataset  
names(murders)

## [1] "state" "abb" "region" "population" "total"

# determining how many entries are in a vector  
pop <- murders$population  
length(pop)

## [1] 51

# vectors can be of class numeric and character  
class(pop)

## [1] "numeric"

class(murders$state)

## [1] "character"

# logical vectors are either TRUE or FALSE  
z <- 3 == 2  
z

## [1] FALSE

class(z)

## [1] "logical"

# factors are another type of class  
class(murders$region)

## [1] "factor"

# obtaining the levels of a factor  
levels(murders$region)

## [1] "Northeast" "South" "North Central" "West"

## Assessment - Data Types

1. We’re going to be using the following dataset for this module. Run this code in the console.

library(dslabs)   
data(murders)

Next, use the function str to examine the structure of the murders object. We can see that this object is a data frame with 51 rows and five columns.

str(murders)

## 'data.frame': 51 obs. of 5 variables:  
## $ state : chr "Alabama" "Alaska" "Arizona" "Arkansas" ...  
## $ abb : chr "AL" "AK" "AZ" "AR" ...  
## $ region : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...  
## $ population: num 4779736 710231 6392017 2915918 37253956 ...  
## $ total : num 135 19 232 93 1257 ...

Which of the following best describes the variables represented in this data frame?

* ☐ A. The 51 states.
* ☐ B. The murder rates for all 50 states and DC.
* ☒ C. The state name, the abbreviation of the state name, the state’s region, and the state’s population and total number of murders for 2010.
* ☐ D. str shows no relevant information.

1. In the previous question, we saw the different variables that are a part of this dataset from the output of the str() function. The function names() is specifically designed to extract the column names from a data frame.

# Load package and data  
library(dslabs)  
data(murders)

# Use the function names to extract the variable names   
names(murders)

## [1] "state" "abb" "region" "population" "total"

1. In this module we have learned that every variable has a class. For example, the class can be a *character*, *numeric* or *logical*. The function class() can be used to determine the class of an object.

Here we are going to determine the class of one of the variables in the murders data frame. To extract variables from a data frame we use $, referred to as the accessor.

# To access the population variable from the murders dataset use this code:  
p <- murders$population   
  
# To determine the class of object `p` we use this code:  
class(p)

## [1] "numeric"

# Use the accessor to extract state abbreviations and assign it to a  
a <- murders$abb  
  
# Determine the class of a  
class(a)

## [1] "character"

1. An important lesson you should learn early on is that there are multiple ways to do things in R. For example, to generate the first five integers we note that 1:5 and seq(1,5) return the same result.

There are also multiple ways to access variables in a data frame. For example we can use the square brackets [[ instead of the accessor $.

If you instead try to access a column with just one bracket,

murders["population"]

R returns a subset of the original data frame containing just this column. This new object will be of class data.frame rather than a vector. To access the column itself you need to use either the $ accessor or the double square brackets [[.

Parentheses, in contrast, are mainly used alongside functions to indicate what argument the function should be doing something to. For example, when we did class(p) in the last question, we wanted the function class to do something related to the argument p.

This is an example of how R can be a bit idiosyncratic sometimes. It is very common to find it confusing at first.

# We extract the population like this:  
p <- murders$population  
  
# This is how we do the same with the square brackets:  
o <- murders[["population"]]  
  
# We can confirm these two are the same  
identical(o, p)

## [1] TRUE

# Use square brackets to extract `abb` from `murders` and assign it to b  
b <- murders[["abb"]]  
  
# Check if `a` and `b` are identical   
identical(a, b)

## [1] TRUE

1. Using the str() command, we saw that the *region* column stores a factor. You can corroborate this by using the class command on the *region* column.

The function levels shows us the categories for the factor.

# We can see the class of the region variable using class  
class(murders$region)

## [1] "factor"

# Determine the number of regions included in this variable   
length(levels(murders$region))

## [1] 4

1. The function table takes a vector as input and returns the frequency of each unique element in the vector.

# Here is an example of what the table function does  
x <- c("a", "a", "b", "b", "b", "c")  
table(x)

## x  
## a b c   
## 2 3 1

# Write one line of code to show the number of states per region  
table(murders$region)

##   
## Northeast South North Central West   
## 9 17 12 13

## Section 1 Assessment

1. To find the solutions to an equation of the format , use the quadratic equation: .

What are the two solutions to ? Use the quadratic equation. (Report the greater of the two solutions first, using 3 significant digits for both solutions)

options(digits = 3)  
a <- 2  
b <- -1  
c <- -4  
(-b+sqrt(b^2-4\*a\*c))/(2\*a)

## [1] 1.69

(-b-sqrt(b^2-4\*a\*c))/(2\*a)

## [1] -1.19

1. Use R to compute log base 4 of 1024. You can use the help function to learn how to use arguments to change the base of the log function.

log(1024, base = 4)

## [1] 5

1. Load the movielens dataset

data(movielens)  
str(movielens)

## 'data.frame': 100004 obs. of 7 variables:  
## $ movieId : int 31 1029 1061 1129 1172 1263 1287 1293 1339 1343 ...  
## $ title : chr "Dangerous Minds" "Dumbo" "Sleepers" "Escape from New York" ...  
## $ year : int 1995 1941 1996 1981 1989 1978 1959 1982 1992 1991 ...  
## $ genres : Factor w/ 901 levels "(no genres listed)",..: 762 510 899 120 762 836 81 762 844 899 ...  
## $ userId : int 1 1 1 1 1 1 1 1 1 1 ...  
## $ rating : num 2.5 3 3 2 4 2 2 2 3.5 2 ...  
## $ timestamp: int 1260759144 1260759179 1260759182 1260759185 1260759205 1260759151 1260759187 1260759148 1260759125 1260759131 ...

How many rows are in the dataset? 100004

How many different variables are in the dataset? 7

What is the variable type of title?

* ☐ A. It is a text (txt) variable
* ☐ B. It is a chronological (chr) variable
* ☐ C. It is a string (str) variable
* ☐ D. It is a numeric (num) variable
* ☐ E. It is an integer (int) variable
* ☐ F. It is a factor (Factor) variable
* ☒ G. It is a character (chr) variable

What is the variable type of genres?

* ☐ A. It is a text (txt) variable
* ☐ B. It is a chronological (chr) variable
* ☐ C. It is a string (str) variable
* ☐ D. It is a numeric (num) variable
* ☐ E. It is an integer (int) variable
* ☒ F. It is a factor (Factor) variable
* ☐ G. It is a character (chr) variable

1. We already know we can use the levels() function to determine the levels of a factor. A different function, nlevels(), may be used to determine the number of levels of a factor.

Use this function to determine how many levels are in the factor genres in the movielens data frame.

nlevels(movielens$genres)

## [1] 901

# Section 2 Overview

In Section 2.1, you will:

* Create numeric and character vectors.
* Name the columns of a vector.
* Generate numeric sequences.
* Access specific elements or parts of a vector.
* Coerce data into different data types as needed.

In Section 2.2, you will:

* Sort vectors in ascending and descending order.
* Extract the indices of the sorted elements from the original vector.
* Find the maximum and minimum elements, as well as their indices, in a vector.
* Rank the elements of a vector in increasing order.

In Section 2.3, you will:

* Perform arithmetic between a vector and a single number.
* Perform arithmetic between two vectors of same length.

## Vectors

The textbook for this section is available [here](https://rafalab.github.io/dsbook/r-basics.html#vectors)

**Key Points**

* The function c(), which stands for concatenate, is useful for creating vectors.
* Another useful function for creating vectors is the seq() function, which generates sequences.
* Subsetting lets us access specific parts of a vector by using square brackets to access elements of a vector.

*Code*

# We may create vectors of class numeric or character with the concatenate function  
codes <- c(380, 124, 818)  
country <- c("italy", "canada", "egypt")  
  
# We can also name the elements of a numeric vector  
# Note that the two lines of code below have the same result  
codes <- c(italy = 380, canada = 124, egypt = 818)  
codes <- c("italy" = 380, "canada" = 124, "egypt" = 818)  
  
# We can also name the elements of a numeric vector using the names() function  
codes <- c(380, 124, 818)  
country <- c("italy","canada","egypt")  
names(codes) <- country  
  
# Using square brackets is useful for subsetting to access specific elements of a vector  
codes[2]

## canada   
## 124

codes[c(1,3)]

## italy egypt   
## 380 818

codes[1:2]

## italy canada   
## 380 124

# If the entries of a vector are named, they may be accessed by referring to their name  
codes["canada"]

## canada   
## 124

codes[c("egypt","italy")]

## egypt italy   
## 818 380

## Vectors - Vector Coercion

The textbook for this section is available [here](https://rafalab.github.io/dsbook/r-basics.html#coercion)

**Key Points**

* In general, *coercion* is an attempt by R to be flexible with data types by guessing what was meant when an entry does not match the expected. For example, when defining x as

x <- c(1, “canada”, 3)

R *coerced* the data into characters. It guessed that because you put a character string in the vector, you meant the 1 and 3 to actually be character strings “1” and “3”.

* The function as.character() turns numbers into characters.
* The function as.numeric() turns characters into numbers.
* In R, missing data is assigned the value NA.

## Assessment - Vectors

1. A vector is a series of values, all of the same type. They are the most basic data type in R and can hold numeric data, character data, or logical data. In R, you can create a vector with the concatenate (or combine) function c()

You place the vector elements separated by a comma between the parentheses. For example a numeric vector would look something like this:

cost <- c(50, 75, 90, 100, 150)

# Here is an example creating a numeric vector named cost  
cost <- c(50, 75, 90, 100, 150)  
  
# Create a numeric vector to store the temperatures listed in the instructions into a vector named temp  
# Make sure to follow the same order in the instructions  
temp <- c("Beijing"=35, "Lagos"=88, "Paris"=42, "Rio de Janeiro"=84, "San Juan"=81, "Toronto"=30)  
cost

## [1] 50 75 90 100 150

temp

## Beijing Lagos Paris Rio de Janeiro San Juan   
## 35 88 42 84 81   
## Toronto   
## 30

class(temp)

## [1] "numeric"

1. As in the previous question, we are going to create a vector. Only this time, we learn to create *character* vectors. The main difference is that these have to be written as strings and so the names are enclosed within double quotes.

A *character* vector would look something like this:

food <- c("pizza", "burgers", "salads", "cheese", "pasta")

# here is an example of how to create a character vector  
food <- c("pizza", "burgers", "salads", "cheese", "pasta")  
  
# Create a character vector called city to store the city names  
# Make sure to follow the same order as in the instructions  
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro", "San Juan", "Toronto")

1. We have successfully assigned the temperatures as *numeric* values to temp and the city names as character values to city. But can we associate the temperature to its related city? Yes! We can do so using a code we already know - names. We assign names to the *numeric* values.

It would look like this:

cost <- c(50, 75, 90, 100, 150)  
food <- c("pizza", "burgers", "salads", "cheese", "pasta")  
names(cost) <- food

# Associate the cost values with its corresponding food item  
cost <- c(50, 75, 90, 100, 150)  
food <- c("pizza", "burgers", "salads", "cheese", "pasta")  
names(cost) <- food  
  
# You already wrote this code  
temp <- c(35, 88, 42, 84, 81, 30)  
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro", "San Juan", "Toronto")  
  
# Associate the temperature values with its corresponding city  
names(temp) <- city  
temp

## Beijing Lagos Paris Rio de Janeiro San Juan   
## 35 88 42 84 81   
## Toronto   
## 30

1. If we want to display only selected values from the object, R can help us do that easily.

For example, if we want to see the cost of the last 3 items in our food list, we would type:

cost[3:5]

Note here, that we could also type cost[c(3,4,5)] and get the same result. The : operator helps us condense the code and get consecutive values.

# cost of the last 3 items in our food list:  
cost[3:5]

## salads cheese pasta   
## 90 100 150

# temperatures of the first three cities in the list:  
temp[1:3]

## Beijing Lagos Paris   
## 35 88 42

1. In the previous question, we accessed the temperature for consecutive cities (1st three). But what if we want to access the temperatures for any 2 specific cities?

An example: To access the cost of pizza (1st) and pasta (5th food item) in our list, the code would be:

cost[c(1,5)]

# Access the cost of pizza and pasta from our food list   
cost[c(1,5)]

## pizza pasta   
## 50 150

# Define temp  
temp <- c(35, 88, 42, 84, 81, 30)  
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro", "San Juan", "Toronto")  
names(temp) <- city  
  
# Access the temperatures of Paris and San Juan  
temp[c(3,5)]

## Paris San Juan   
## 42 81

1. The : operator helps us create sequences of numbers. For example, 32:99 would create a list of numbers from 32 to 99.

Then, if we want to know the length of this sequence, all we need to do is use the length command.

# Create a vector m of integers that starts at 32 and ends at 99.  
m <- 32:99  
  
# Determine the length of object m.  
length(m)

## [1] 68

# Create a vector x of integers that starts at 12 and ends at 73.  
x <- 12:73  
  
# Determine the length of object x.  
length(x)

## [1] 62

1. We can also create different types of sequences in R. For example, in seq(7, 49, 7), the first argument defines the start, and the second the end. The default is to go up in increments of 1, but a third argument lets us tell it by what interval.

# Create a vector with the multiples of 7, smaller than 50.  
seq(7, 49, 7)

## [1] 7 14 21 28 35 42 49

# Create a vector containing all the positive odd numbers smaller than 100.  
# The numbers should be in ascending order  
seq(1,99,2)

## [1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49  
## [26] 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99

1. The second argument of the function seq is actually a maximum, not necessarily the end.

So if we type

seq(7, 50, 7)

we actually get the same vector of integers as if we type

seq(7, 49, 7)

This can be useful because sometimes all we want are sequential numbers that are smaller than some value. Let’s look at an example.

# We can create a vector with the multiples of 7, smaller than 50 like this   
seq(7, 49, 7)

## [1] 7 14 21 28 35 42 49

# But note that the second argument does not need to be the last number  
# It simply determines the maximum value permitted  
# so the following line of code produces the same vector as seq(7, 49, 7)  
seq(7, 50, 7)

## [1] 7 14 21 28 35 42 49

# Create a sequence of numbers from 6 to 55, with 4/7 increments and determine its length  
length(seq(6,55,4/7))

## [1] 86

1. The seq() function has another useful argument. The argument *length.out*. This argument lets us generate sequences that are increasing by the same amount but are of the prespecified length.

For example, this line of code

x <- seq(0, 100, length.out = 5)  
produces the numbers 0, 25, 50, 75, 100.

Let’s create a vector and see what is the class of the object produced.

# Store the sequence in the object a  
a <- seq(1, 10, length.out = 100)  
  
# Determine the class of a  
class(a)

## [1] "numeric"

1. We have discussed the numeric class. We just saw that the seq function can generate objects of this class.

For another example, type

class(seq(1, 10, 0.5))

into the console and note that the class is *numeric*. R has another type of vector we have not described, the *integer* class. You can create an *integer* by adding the letter L after a whole number. If you type

class(3L)

in the console, you see this is an *integer* and not a *numeric*. For most practical purposes, integers and numerics are indistinguishable. For example 3, the integer, minus 3 the numeric is 0. To see this type this in the console

3L - 3

The main difference is that integers occupy less space in the computer memory, so for big computations using integers can have a substantial impact.

# Store the sequence in the object a  
a <- seq(1,10)  
  
# Determine the class of a  
class(a)

## [1] "integer"

1. Let’s confirm that 1L is an *integer* not a *numeric*.

# Check the class of 1, assigned to the object a  
class(1)

## [1] "numeric"

# Confirm the class of 1L is integer  
class(1L)

## [1] "integer"

1. The concept of coercion is a very important one. Watching the video, we learned that when an entry does not match what an R function is expecting, R tries to guess what we meant before throwing an error. This might get confusing at times.

As we’ve discussed in earlier questions, there are numeric and character vectors. The character vectors are placed in quotes and the numerics are not.

We can avoid issues with coercion in R by changing characters to numerics and vice-versa. This is known as typecasting. The code, as.numeric(x) helps us convert character strings to numbers. There is an equivalent function that converts its argument to a string, as.character(x).

Let’s practice doing this!

# Define the vector x  
x <- c(1, 3, 5,"a")  
  
# Note that the x is character vector  
x

## [1] "1" "3" "5" "a"

# Typecast the vector to get an integer vector  
# You will get a warning but that is ok  
x <- as.numeric(x)

## Warning: NAs introduced by coercion

x

## [1] 1 3 5 NA

## Sorting

The textbook for this section is available [here](https://rafalab.github.io/dsbook/r-basics.html#sorting)

**Key Points**

* The function sort() sorts a vector in increasing order.
* The function order() produces the indices needed to obtain the sorted vector, e.g. a result of 2 3 1 5 4 means the sorted vector will be produced by listing the 2nd, 3rd, 1st, 5th, and then 4th item of the original vector.
* The function rank() gives us the ranks of the items in the original vector.
* The function max() returns the largest value while which.max() returns the index of the largest value. The functions min() and which.min() work similarly for minimum values.

## Assessment - Sorting

1. When looking at a dataset, we may want to sort the data in an order that makes more sense for analysis. Let’s learn to do this using the murders dataset as an example

# Access the `state` variable and store it in an object   
states <- murders$state   
  
# Sort the object alphabetically and redefine the object   
states <- sort(states)   
  
# Report the first alphabetical value   
states[1]

## [1] "Alabama"

# Access population values from the dataset and store it in pop  
pop <- murders$population  
  
# Sort the object and save it in the same object  
pop <- sort(pop)  
  
# Report the smallest population size   
pop[1]

## [1] 563626

1. The function order() returns the index vector needed to sort the vector. This implies that sort(x) and x[order(x)] give the same result.

This can be useful for finding row numbers with certain properties such as “the row for the state with the smallest population”. Remember that when we extract a variable from a data frame the order of the resulting vector is the same as the order of the rows of the data frame. So for example, the entries of the vector murders$state are ordered in the same way as the states if you go down the rows of murders.

# Access population from the dataset and store it in pop  
pop <- murders$population  
  
# Use the command order to find the vector of indexes that order pop and store in object ord  
ord <- order(pop)  
  
# Find the index number of the entry with the smallest population size  
ord[1]

## [1] 51

1. We can actually perform the same operation as in the previous exercise using the function which.min. It basically tells us which is the minimum value.

# Find the index of the smallest value for variable total   
which.min(murders$total)

## [1] 46

# Find the index of the smallest value for population  
which.min(murders$population)

## [1] 51

1. Now we know how small the smallest state is and we know which row represents it. However, which state is it?

# Define the variable i to be the index of the smallest state  
i <- which.min(murders$population)  
  
# Define variable states to hold the states  
states <- murders$state  
  
# Use the index you just defined to find the state with the smallest population  
states[i]

## [1] "Wyoming"

1. You can create a data frame using the data.frame function.

Here is a quick example:

temp <- c(35, 88, 42, 84, 81, 30)  
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro", "San Juan", "Toronto")  
city\_temps <- data.frame(name = city, temperature = temp)

# Store temperatures in an object   
temp <- c(35, 88, 42, 84, 81, 30)  
  
# Store city names in an object   
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro", "San Juan", "Toronto")  
  
# Create data frame with city names and temperature   
city\_temps <- data.frame(name = city, temperature = temp)  
  
# Define a variable states to be the state names   
states <- murders$state  
  
# Define a variable ranks to determine the population size ranks   
ranks <- rank(murders$population)  
  
# Create a data frame my\_df with the state name and its rank  
my\_df <- data.frame(name = states, rank = ranks)

1. This exercise is somewhat more challenging. We are going to repeat the previous exercise but this time order `my\_df so that the states are ordered from least populous to most.

# Define a variable states to be the state names from the murders data frame  
states <- murders$state  
  
# Define a variable ranks to determine the population size ranks   
ranks <- rank(murders$population)  
  
# Define a variable ind to store the indexes needed to order the population values  
ind <- order(murders$population)  
  
# Create a data frame my\_df with the state name and its rank and ordered from least populous to most   
my\_df <- data.frame(states = states[ind], ranks = ranks[ind])

1. The na\_example dataset represents a series of counts. It is included in the *dslabs* package.

You can quickly examine the object using

library(dslabs)  
data(na\_example)  
str(na\_example)

However, when we compute the average we obtain an NA. You can see this by typing

mean(na\_example)

# Using new dataset   
library(dslabs)  
data(na\_example)  
  
# Checking the structure   
str(na\_example)

## int [1:1000] 2 1 3 2 1 3 1 4 3 2 ...

# Find out the mean of the entire dataset   
mean(na\_example)

## [1] NA

# Use is.na to create a logical index ind that tells which entries are NA  
ind <- is.na(na\_example)  
  
# Determine how many NA ind has using the sum function  
sum(ind)

## [1] 145

1. We previously computed the average of na\_example using mean(na\_example) and obtain NA. This is because the function mean returns NA if it encounters at least one NA. A common operation is therefore removing the entries that are NA and after that perform operations on the rest.

# Note what we can do with the ! operator  
x <- c(1, 2, 3)  
ind <- c(FALSE, TRUE, FALSE)  
x[!ind]

## [1] 1 3

# Create the ind vector  
library(dslabs)  
data(na\_example)  
ind <- is.na(na\_example)  
  
# We saw that this gives an NA  
mean(na\_example)

## [1] NA

# Compute the average, for entries of na\_example that are not NA  
mean(na\_example[!ind])

## [1] 2.3

## Vector arithmetic

The textbook for this section is available [here](https://rafalab.github.io/dsbook/r-basics.html#vector-arithmetics)

**Key Points**

* In R, arithmetic operations on vectors occur element-wise.

*Code*

# The name of the state with the maximum population is found by doing the following  
murders$state[which.max(murders$population)]

## [1] "California"

# how to obtain the murder rate  
murder\_rate <- murders$total / murders$population \* 100000  
  
# ordering the states by murder rate, in decreasing order  
murders$state[order(murder\_rate, decreasing=TRUE)]

## [1] "District of Columbia" "Louisiana" "Missouri"   
## [4] "Maryland" "South Carolina" "Delaware"   
## [7] "Michigan" "Mississippi" "Georgia"   
## [10] "Arizona" "Pennsylvania" "Tennessee"   
## [13] "Florida" "California" "New Mexico"   
## [16] "Texas" "Arkansas" "Virginia"   
## [19] "Nevada" "North Carolina" "Oklahoma"   
## [22] "Illinois" "Alabama" "New Jersey"   
## [25] "Connecticut" "Ohio" "Alaska"   
## [28] "Kentucky" "New York" "Kansas"   
## [31] "Indiana" "Massachusetts" "Nebraska"   
## [34] "Wisconsin" "Rhode Island" "West Virginia"   
## [37] "Washington" "Colorado" "Montana"   
## [40] "Minnesota" "South Dakota" "Oregon"   
## [43] "Wyoming" "Maine" "Utah"   
## [46] "Idaho" "Iowa" "North Dakota"   
## [49] "Hawaii" "New Hampshire" "Vermont"

## Assessment - Vector Arithmetic

1. Previously we created this data frame.

{r, eval=FALSE, echo=TRUE temp <- c(35, 88, 42, 84, 81, 30) city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro", "San Juan", "Toronto") city\_temps <- data.frame(name = city, temperature = temp)

# Assign city names to `city`   
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro", "San Juan", "Toronto")  
  
# Store temperature values in `temp`  
temp <- c(35, 88, 42, 84, 81, 30)  
  
# Convert temperature into Celsius and overwrite the original values of 'temp' with these Celsius values  
temp <- 5/9 \* (temp -32)  
  
# Create a data frame `city\_temps`   
city\_temps <- data.frame(name = city, temperature = temp)

1. We can use some of what we have learned to perform calculations that would otherwise be quite complicated. Let’s see an example.

# Define an object `x` with the numbers 1 through 100  
x <- seq(1,100)  
  
# Compute the sum   
sum((1/x)^2)

## [1] 1.63

1. Compute the per 100,000 murder rate for each state and store it in the object murder\_rate. Then compute the average murder rate for the US using the function mean. What is the average?

# Store the per 100,000 murder rate for each state in murder\_rate  
murder\_rate <- murders$total / murders$population \* 100000  
  
# Calculate the average murder rate in the US   
mean(murder\_rate)

## [1] 2.78

## Section 2 Assessment

1. Consider the vector x <- c(2, 43, 27, 96, 18).

Match the following outputs to the function which produces that output. Options include sort(x), order(x), rank(x) and none of these.

x <- c(2, 43, 27, 96, 18)  
sort(x)

## [1] 2 18 27 43 96

order(x)

## [1] 1 5 3 2 4

rank(x)

## [1] 1 4 3 5 2

1, 2, 3, 4, 5 none of these

1, 5, 3, 2, 4 order(x)

1, 4, 3, 5, 2 rank(x)

2, 18, 27, 43, 96 sort(x)

1. Continue working with the vector x <- c(2, 43, 27, 96, 18).

x <- c(2, 43, 27, 96, 18)  
min(x)

## [1] 2

which.min(x)

## [1] 1

max(x)

## [1] 96

which.max(x)

## [1] 4

min(x) 2

which.min(x) 1

max(x) none of these

which.max(x) 4

1. Mandi, Amy, Nicole, and Olivia all ran different distances in different time intervals. Their distances (in miles) and times (in minutes) are as follows:

name <- c("Mandi", "Amy", "Nicole", "Olivia")  
distance <- c(0.8, 3.1, 2.8, 4.0)  
time <- c(10, 30, 40, 50)

Write a line of code to convert time to hours. Remember there are 60 minutes in an hour. Then write a line of code to calculate the speed of each runner in miles per hour. Speed is distance divided by time.

How many hours did Olivia run?

hours <- time/60  
hours[4]

## [1] 0.833

What was Mandi’s speed in miles per hour?

speed <- distance/hours  
speed[1]

## [1] 4.8

Which runner had the fastest speed?

name[which.max(speed)]

## [1] "Amy"

# Section 3 Overview

Section 3 introduces to the R commands and techniques that help you wrangle, analyze, and visualize data.

In Section 3.1, you will:

* Subset a vector based on properties of another vector.
* Use multiple logical operators to index vectors.
* Extract the indices of vector elements satisfying one or more logical conditions.
* Extract the indices of vector elements matching with another vector.
* Determine which elements in one vector are present in another vector.

In Section 3.2, you will:

* Wrangle data tables using the functions in ‘dplyr’ package.
* Modify a data table by adding or changing columns.
* Subset rows in a data table.
* Subset columns in a data table.
* Perform a series of operations using the pipe operator.
* Create data frames.

In Section 3.3, you will:

* Plot data in scatter plots, box plots and histograms.

## Indexing

The textbook for this section is available [here](https://rafalab.github.io/dsbook/r-basics.html#indexing)

**Key Points**

* We can use logicals to index vectors.
* Using the function >sum()on a logical vector returns the number of entries that are true.
* The logical operator “&” makes two logicals true only when they are both true.

*Code*

# defining murder rate as before  
murder\_rate <- murders$total / murders$population \* 100000  
# creating a logical vector that specifies if the murder rate in that state is less than or equal to 0.71  
index <- murder\_rate <= 0.71  
# determining which states have murder rates less than or equal to 0.71  
murders$state[index]

## [1] "Hawaii" "Iowa" "New Hampshire" "North Dakota"   
## [5] "Vermont"

# calculating how many states have a murder rate less than or equal to 0.71  
sum(index)

## [1] 5

# creating the two logical vectors representing our conditions  
west <- murders$region == "West"  
safe <- murder\_rate <= 1  
# defining an index and identifying states with both conditions true  
index <- safe & west  
murders$state[index]

## [1] "Hawaii" "Idaho" "Oregon" "Utah" "Wyoming"

## Indexing - Indexing Functions

The textbook for this section is available [here](https://rafalab.github.io/dsbook/r-basics.html#which)

**Key Points**

* The function which() gives us the entries of a logical vector that are true.
* The function match() looks for entries in a vector and returns the index needed to access them.
* We use the function %in% if we want to know whether or not each element of a first vector is in a second vector.

*Code*

# to determine the murder rate in Massachusetts we may do the following  
ind <- which(murders$state == "Massachusetts")  
murder\_rate[ind]

## [1] 1.8

# to obtain the indices and subsequent murder rates of New York, Florida, Texas, we do:  
ind <- match(c("New York", "Florida", "Texas"), murders$state)  
ind

## [1] 33 10 44

murder\_rate[ind]

## [1] 2.67 3.40 3.20

# to see if Boston, Dakota, and Washington are states  
c("Boston", "Dakota", "Washington") %in% murders$state

## [1] FALSE FALSE TRUE

## Assessment - Indexing

1. Here we will be using logical operators to create a logical vector. Compute the per 100,000 murder rate for each state and store it in an object called murder\_rate. Then use logical operators to create a logical vector named low that tells us which entries of murder\_rate are lower than 1.

# Store the murder rate per 100,000 for each state, in `murder\_rate`  
murder\_rate <- murders$total / murders$population \* 100000  
  
# Store the `murder\_rate < 1` in `low`   
low <- murder\_rate < 1

1. The function `which() helps us know directly, which values are low or high, etc. Let’s use it in this question.

# Store the murder rate per 100,000 for each state, in murder\_rate  
murder\_rate <- murders$total/murders$population\*100000  
  
# Store the murder\_rate < 1 in low   
low <- murder\_rate < 1  
  
# Get the indices of entries that are below 1  
ind <- which(low)  
ind

## [1] 12 13 16 20 24 30 35 38 42 45 46 51

1. Note that if we want to know which entries of a vector are lower than a particular value we can use code like this.

small <- murders$population < 1000000  
murders$state[small]

The code above shows us the states with populations smaller than one million.

# Store the murder rate per 100,000 for each state, in murder\_rate  
murder\_rate <- murders$total/murders$population\*100000  
  
# Store the murder\_rate < 1 in low   
low <- murder\_rate < 1  
  
# Names of states with murder rates lower than 1  
murders$state[low]

## [1] "Hawaii" "Idaho" "Iowa" "Maine"   
## [5] "Minnesota" "New Hampshire" "North Dakota" "Oregon"   
## [9] "South Dakota" "Utah" "Vermont" "Wyoming"

1. Now we will extend the code from the previous exercises to report the states in the Northeast with a murder rate lower than 1.

# Store the murder rate per 100,000 for each state, in `murder\_rate`  
murder\_rate <- murders$total/murders$population\*100000  
  
# Store the `murder\_rate < 1` in `low`   
low <- murder\_rate < 1  
  
# Create a vector ind for states in the Northeast and with murder rates lower than 1.   
northeast <- murders$region == "Northeast"  
ind <- low & northeast  
  
# Names of states in `ind`   
murders$state[ind]

## [1] "Maine" "New Hampshire" "Vermont"

1. In a previous exercise we computed the murder rate for each state and the average of these numbers. How many states are below the average?

# Store the murder rate per 100,000 for each state, in murder\_rate  
murder\_rate <- murders$total/murders$population\*100000  
  
  
# Compute the average murder rate using `mean` and store it in object named `avg`  
avg <- mean(murder\_rate)  
  
# How many states have murder rates below avg ? Check using sum   
ind <- murder\_rate < avg  
sum(ind)

## [1] 27

1. In this exercise we use the match function to identify the states with abbreviations AK, MI, and IA.

# Store the 3 abbreviations in a vector called `abbs` (remember that they are character vectors and need quotes)  
abbs <- c("AK", "MI", "IA")  
  
# Match the abbs to the murders$abb and store in ind  
ind <- match(abbs, murders$abb)  
  
# Print state names from ind  
murders$state[ind]

## [1] "Alaska" "Michigan" "Iowa"

1. If rather than an index we want a logical that tells us whether or not each element of a first vector is in a second, we can use the function %in%.

For example:

x <- c(2, 3, 5)  
y <- c(1, 2, 3, 4)  
x%in%y

Gives us two TRUE followed by a FALSE because 2 and 3 are in y but 5 is not.

# Store the 5 abbreviations in `abbs`. (remember that they are character vectors)  
abbs <- c("MA", "ME", "MI", "MO", "MU")  
  
# Use the %in% command to check if the entries of abbs are abbreviations in the the murders data frame  
abbs%in%murders$abb

## [1] TRUE TRUE TRUE TRUE FALSE

1. In a previous exercise we computed the index abbs%in%murders$abb. Based on that, and using the which function and the ! operator, get the index of the entries of abbs that are **not** abbreviations.

# Store the 5 abbreviations in abbs. (remember that they are character vectors)  
abbs <- c("MA", "ME", "MI", "MO", "MU")   
  
# Use the `which` command and `!` operator to find out which index abbreviations are not actually part of the dataset and store in `ind`  
ind <- which(!abbs%in%murders$abb)  
  
# Names of abbreviations in `ind`  
abbs[ind]

## [1] "MU"

## Basic Data Wrangling

The textbook for this section is available [here](https://rafalab.github.io/dsbook/tidyverse.html#manipulating-data-frames) and [here](https://rafalab.github.io/dsbook/tidyverse.html#the-pipe)

In the textbook, the dplyr package is introduced in the context of the tidyverse, a collection of R packages

**Key Points**

* To change a data table by adding a new column, or changing an existing one, we use the mutate function.
* To filter the data by subsetting rows, we use the function filter.
* To subset the data by selecting specific columns, we use the select function.
* We can perform a series of operations by sending the results of one function to another function using what is called the pipe operator, %>%.

*Code*

# installing and loading the dplyr package  
if(!require(dplyr)) install.packages("dplyr")

## Loading required package: dplyr

##   
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':  
##   
## filter, lag

## The following objects are masked from 'package:base':  
##   
## intersect, setdiff, setequal, union

library(dplyr)  
  
# adding a column with mutate  
library(dslabs)  
data("murders")  
murders <- mutate(murders, rate = total / population \* 100000)  
  
# subsetting with filter  
filter(murders, rate <= 0.71)

## state abb region population total rate  
## 1 Hawaii HI West 1360301 7 0.515  
## 2 Iowa IA North Central 3046355 21 0.689  
## 3 New Hampshire NH Northeast 1316470 5 0.380  
## 4 North Dakota ND North Central 672591 4 0.595  
## 5 Vermont VT Northeast 625741 2 0.320

# selecting columns with select  
new\_table <- select(murders, state, region, rate)  
  
# using the pipe  
murders %>% select(state, region, rate) %>% filter(rate <= 0.71)

## state region rate  
## 1 Hawaii West 0.515  
## 2 Iowa North Central 0.689  
## 3 New Hampshire Northeast 0.380  
## 4 North Dakota North Central 0.595  
## 5 Vermont Northeast 0.320

## Basic Data Wrangling - Creating Data Frames

**Key Points**

* We can use the data.frame() function to create data frames.
* By default, the data.frame() function turns characters into factors. To avoid this, we utilize the stringsAsFactors argument and set it equal to false.

*Code*

# creating a data frame with stringAsFactors = FALSE  
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),   
 exam\_1 = c(95, 80, 90, 85),   
 exam\_2 = c(90, 85, 85, 90),  
 stringsAsFactors = FALSE)

## Assessment - Basic Data Wrangling

1. You can add columns using the dplyr function mutate.

This function is aware of the column names and inside the function you can call them unquoted. Like this:

murders <- mutate(murders, population\_in\_millions = population / 10^6)

Note that we can write population rather than murders$population. The function mutate knows we are grabing columns from murders.

# Redefine murders so that it includes a column named rate with the per 100,000 murder rates  
murders <- mutate(murders, rate = total / population \* 100000)

1. Note that if rank(x) gives you the ranks of x from lowest to highest, rank(-x) gives you the ranks from highest to lowest.

# Note that if you want ranks from highest to lowest you can take the negative and then compute the ranks   
x <- c(88, 100, 83, 92, 94)  
rank(-x)

## [1] 4 1 5 3 2

# Defining rate  
rate <- murders$total/ murders$population \* 100000  
  
# Redefine murders to include a column named rank  
# with the ranks of rate from highest to lowest  
murders <- mutate(murders, rank = rank(-rate))

1. With dplyr we can use select to show only certain columns. For example with this code we would only show the states and population sizes:

select(murders, state, population)

# Use select to only show state names and abbreviations from murders  
select(murders, state, abb)

## state abb  
## 1 Alabama AL  
## 2 Alaska AK  
## 3 Arizona AZ  
## 4 Arkansas AR  
## 5 California CA  
## 6 Colorado CO  
## 7 Connecticut CT  
## 8 Delaware DE  
## 9 District of Columbia DC  
## 10 Florida FL  
## 11 Georgia GA  
## 12 Hawaii HI  
## 13 Idaho ID  
## 14 Illinois IL  
## 15 Indiana IN  
## 16 Iowa IA  
## 17 Kansas KS  
## 18 Kentucky KY  
## 19 Louisiana LA  
## 20 Maine ME  
## 21 Maryland MD  
## 22 Massachusetts MA  
## 23 Michigan MI  
## 24 Minnesota MN  
## 25 Mississippi MS  
## 26 Missouri MO  
## 27 Montana MT  
## 28 Nebraska NE  
## 29 Nevada NV  
## 30 New Hampshire NH  
## 31 New Jersey NJ  
## 32 New Mexico NM  
## 33 New York NY  
## 34 North Carolina NC  
## 35 North Dakota ND  
## 36 Ohio OH  
## 37 Oklahoma OK  
## 38 Oregon OR  
## 39 Pennsylvania PA  
## 40 Rhode Island RI  
## 41 South Carolina SC  
## 42 South Dakota SD  
## 43 Tennessee TN  
## 44 Texas TX  
## 45 Utah UT  
## 46 Vermont VT  
## 47 Virginia VA  
## 48 Washington WA  
## 49 West Virginia WV  
## 50 Wisconsin WI  
## 51 Wyoming WY

1. The dplyr function filter is used to choose specific rows of the data frame to keep. Unlike select which is for columns, filter is for rows.

For example you can show just the New York row like this:

filter(murders, state == "New York")

You can use other logical vectors to filter rows.

# Add the necessary columns  
murders <- mutate(murders, rate = total/population \* 100000, rank = rank(-rate))  
  
# Filter to show the top 5 states with the highest murder rates  
filter(murders, rank <= 5)

## state abb region population total rate rank  
## 1 District of Columbia DC South 601723 99 16.45 1  
## 2 Louisiana LA South 4533372 351 7.74 2  
## 3 Maryland MD South 5773552 293 5.07 4  
## 4 Missouri MO North Central 5988927 321 5.36 3  
## 5 South Carolina SC South 4625364 207 4.48 5

1. We can remove rows using the != operator.

For example to remove Florida we would do this:

no\_florida <- filter(murders, state != “Florida”)

# Use filter to create a new data frame no\_south  
no\_south <- filter(murders, region != "South")  
  
# Use nrow() to calculate the number of rows  
nrow(no\_south)

## [1] 34

1. We can also use the %in% to filter with dplyr.

For example you can see the data from New York and Texas like this:

filter(murders, state %in% c(“New York”, “Texas”))

# Create a new data frame called murders\_nw with only the states from the northeast and the west  
murders\_nw <- filter(murders, region %in% c("Northeast", "West"))  
  
# Number of states (rows) in this category   
nrow(murders\_nw)

## [1] 22

1. Suppose you want to live in the Northeast or West **and** want the murder rate to be less than 1.

We want to see the data for the states satisfying these options. Note that you can use logical operators with filter:

filter(murders, population < 5000000 & region == “Northeast”)

# add the rate column  
murders <- mutate(murders, rate = total / population \* 100000, rank = rank(-rate))  
  
# Create a table, call it my\_states, that satisfies both the conditions  
my\_states <- filter(murders, rate < 1 & region %in% c("Northeast", "West"))  
  
# Use select to show only the state name, the murder rate and the rank  
select(my\_states, state, rate, rank)

## state rate rank  
## 1 Hawaii 0.515 49  
## 2 Idaho 0.766 46  
## 3 Maine 0.828 44  
## 4 New Hampshire 0.380 50  
## 5 Oregon 0.940 42  
## 6 Utah 0.796 45  
## 7 Vermont 0.320 51  
## 8 Wyoming 0.887 43

1. The pipe %>% can be used to perform operations sequentially without having to define intermediate objects.

After redefining murder to include rate and rank.

library(dplyr)  
murders <- mutate(murders, rate = total / population \* 100000, rank = (-rate))

in the solution to the previous exercise we did the following:

# Created a table  
my\_states <- filter(murders, region %in% c(“Northeast”, “West”) & rate < 1)  
  
# Used select to show only the state name, the murder rate and the rank  
select(my\_states, state, rate, rank)

The pipe %>% permits us to perform both operation sequentially and without having to define an intermediate variable my\_states

For example we could have mutated and selected in the same line like this:

mutate(murders, rate = total / population \* 100000, rank = (-rate)) %>% select(state, rate, rank)

Note that select no longer has a data frame as the first argument. The first argument is assumed to be the result of the operation conducted right before the %>%

## Define the rate column  
murders <- mutate(murders, rate = total / population \* 100000, rank = rank(-rate))  
  
# show the result and only include the state, rate, and rank columns, all in one line  
filter(murders, region %in% c("Northeast", "West") & rate < 1) %>% select(state, rate, rank)

## state rate rank  
## 1 Hawaii 0.515 49  
## 2 Idaho 0.766 46  
## 3 Maine 0.828 44  
## 4 New Hampshire 0.380 50  
## 5 Oregon 0.940 42  
## 6 Utah 0.796 45  
## 7 Vermont 0.320 51  
## 8 Wyoming 0.887 43

1. Now we will reset murders to the original table by using data(murders).

# Loading the table  
data(murders)  
  
# Create new data frame called my\_states (with specifications in the instructions)  
my\_states <- murders %>% mutate(rate = total / population \* 100000, rank = rank(-rate)) %>% filter(region %in% c("Northeast", "West") & rate < 1) %>% select(state, rate, rank)

## Basic Plots

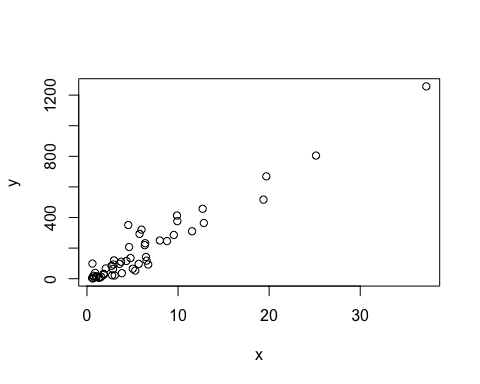
Here is a link to the textbook section on [basic plots](https://rafalab.github.io/dsbook/r-basics.html#basic-plots)

**Key Points**

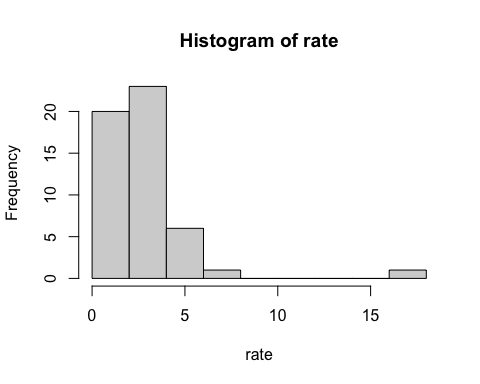
* We can create a simple scatterplot using the function plot().
* Histograms are graphical summaries that give you a general overview of the types of values you have. In R, they can be produced using the hist() function.
* Boxplots provide a more compact summary of a distribution than a histogram and are more useful for comparing distributions. They can be produced using the ```boxplot() function.

*Code*

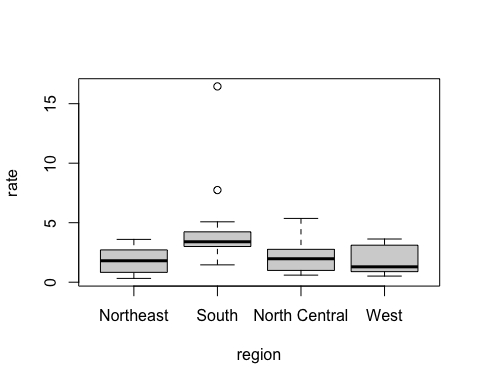
# a simple scatterplot of total murders versus population  
x <- murders$population / 10^6  
y <- murders$total  
plot(x, y)



# a histogram of murder rates  
hist(rate)



# boxplots of murder rates by region  
boxplot(rate~region, data = murders)



## Assessment - Basic Plots

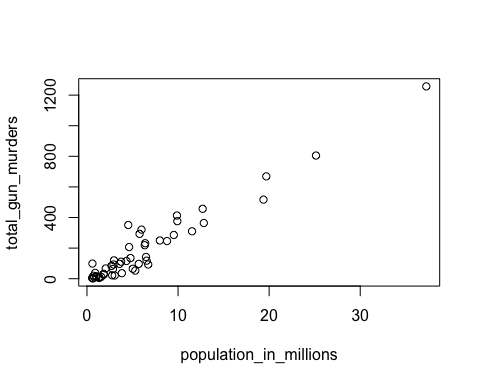
1. We made a plot of total murders versus population and noted a strong relationship: not surprisingly, states with larger populations had more murders.

You can run the code in the console to get the plot.

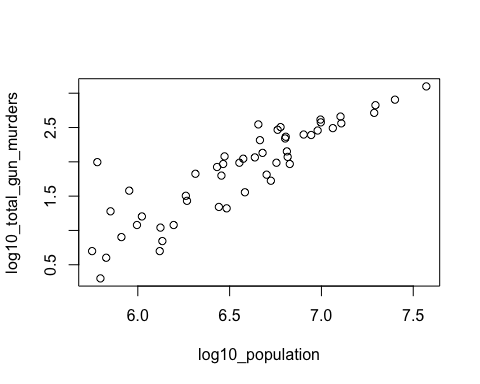
library(dslabs)  
data(murders)  
  
population\_in\_millions <- murders$population/10^6  
total\_gun\_murders <− murders$total  
  
plot(population\_in\_millions, total\_gun\_murders)

Note that many states have populations below 5 million and are bunched up in the plot. We may gain further insights from making this plot in the log scale.

population\_in\_millions <- murders$population/10^6  
total\_gun\_murders <- murders$total  
  
plot(population\_in\_millions, total\_gun\_murders)

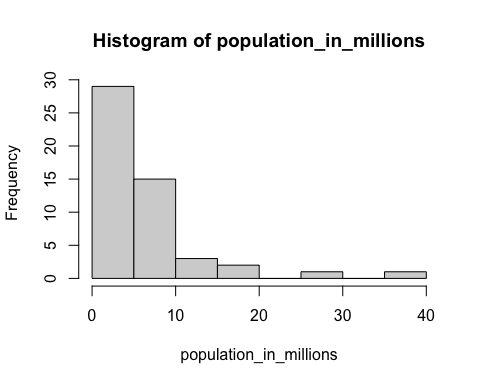


# Transform population using the log10 transformation and save to object log10\_population  
log10\_population <- log10(murders$population)  
  
# Transform total gun murders using log10 transformation and save to object log10\_total\_gun\_murders  
log10\_total\_gun\_murders <- log10(total\_gun\_murders)  
  
# Create a scatterplot with the log scale transformed population and murders   
plot(log10\_population, log10\_total\_gun\_murders)



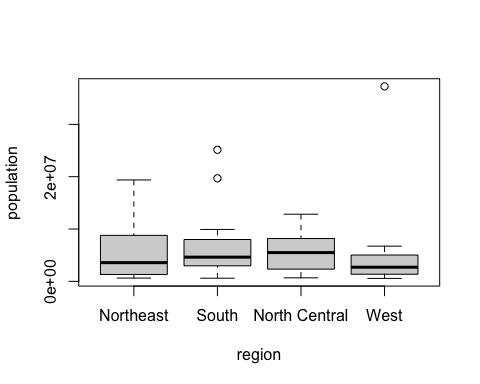
1. Now we are going to make a histogram.

# Store the population in millions and save to population\_in\_millions   
population\_in\_millions <- murders$population/10^6  
  
# Create a histogram of this variable  
hist(population\_in\_millions)



1. Now we are going to make boxplots. Boxplots are useful when we want a summary of several variables or several strata of the same variables. Making too many histograms can become too cumbersome.

# Create a boxplot of state populations by region for the murders dataset  
boxplot(population~region, data = murders)



## Section 3 Assessment

data(heights)  
options(digits = 3) # report 3 significant digits for all answers

1. First, determine the average height in this dataset. Then create a logical vector ind with the indices for those individuals who are above average height.

How many individuals in the dataset are above average height?

ind <- heights$height > mean(heights$height)  
sum(ind)

## [1] 532

1. How many individuals in the dataset are above average height and are female?

sum(ind & heights$sex=="Female")

## [1] 31

1. If you use mean on a logical (TRUE/FALSE) vector, it returns the proportion of observations that are TRUE.

What proportion of individuals in the dataset are female?

mean(heights$sex == "Female")

## [1] 0.227

1. This question takes you through three steps to determine the sex of the individual with the minimum height.

Determine the minimum height in the heights dataset.

min(heights$height)

## [1] 50

Use the match() function to determine the index of the individual with the minimum height.

match(50,heights$height)

## [1] 1032

Subset the sex column of the dataset by the index above to determine the individual’s sex. Male

heights$sex[1032]

## [1] Male  
## Levels: Female Male

1. This question takes you through three steps to determine how many of the integer height values between the minimum and maximum heights are not actual heights of individuals in the heights dataset.

Determine the maximum height.

max(heights$height)

## [1] 82.7

Which integer values are between the maximum and minimum heights? For example, if the minimum height is 10.2 and the maximum height is 20.8, your answer should be x <- 11:20 to capture the integers in between those values. (If either the maximum or minimum height are integers, include those values too.)

Write code to create a vector x that includes the *integers* between the minimum and maximum heights.

x <- 50:82

How many of the integers in x are NOT heights in the dataset?

sum(!(x %in% heights$height))

## [1] 3

1. Using the heights dataset, create a new column of heights in centimeters named ht\_cm. Recall that 1 inch = 2.54 centimeters. Save the resulting dataset as heights2.

What is the height in centimeters of the 18th individual (index 18)?

heights2 <- mutate(heights, ht\_cm = height\*2.54)  
  
# Then we subset the new heights2 dataset:  
  
heights2$ht\_cm[18]

## [1] 163

What is the mean height in centimeters?

mean(heights2$ht\_cm)

## [1] 174

Create a data frame females by filtering the heights2 data to contain only female individuals.

How many females are in the heights2 dataset?

females <- filter(heights2, sex == "Female")  
nrow(females)

## [1] 238

What is the mean height of the females in centimeters?

mean(females$ht\_cm)

## [1] 165

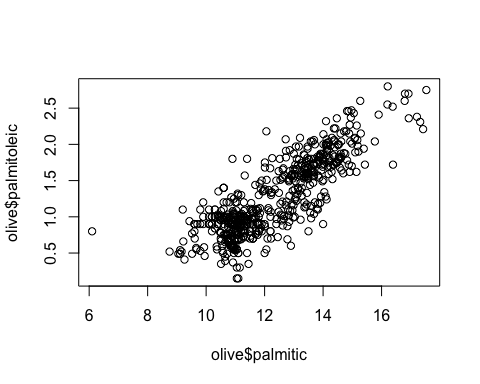
1. The olive dataset in **dslabs** contains composition in percentage of eight fatty acids found in the lipid fraction of 572 Italian olive oils:

data(olive)  
head(olive)

## region area palmitic palmitoleic stearic oleic linoleic  
## 1 Southern Italy North-Apulia 10.75 0.75 2.26 78.2 6.72  
## 2 Southern Italy North-Apulia 10.88 0.73 2.24 77.1 7.81  
## 3 Southern Italy North-Apulia 9.11 0.54 2.46 81.1 5.49  
## 4 Southern Italy North-Apulia 9.66 0.57 2.40 79.5 6.19  
## 5 Southern Italy North-Apulia 10.51 0.67 2.59 77.7 6.72  
## 6 Southern Italy North-Apulia 9.11 0.49 2.68 79.2 6.78  
## linolenic arachidic eicosenoic  
## 1 0.36 0.60 0.29  
## 2 0.31 0.61 0.29  
## 3 0.31 0.63 0.29  
## 4 0.50 0.78 0.35  
## 5 0.50 0.80 0.46  
## 6 0.51 0.70 0.44

Plot the percent palmitic acid versus palmitoleic acid in a scatterplot. What relationship do you see?

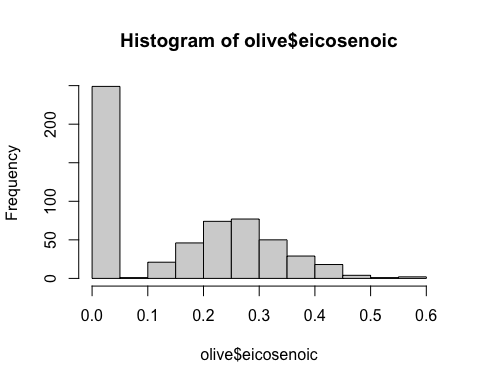
plot(olive$palmitic, olive$palmitoleic)



* ☐ A. There is no relationship between palmitic and palmitoleic.
* ☒ B. There is a positive linear relationship between palmitic and palmitoleic.
* ☐ C. There is a negative linear relationship between palmitic and palmitoleic.
* ☐ D. There is a positive exponential relationship between palmitic and palmitoleic.
* ☐ E. There is a negative exponential relationship between palmitic and palmitoleic.

1. Create a histogram of the percentage of eicosenoic acid in olive. Which of the following is true?

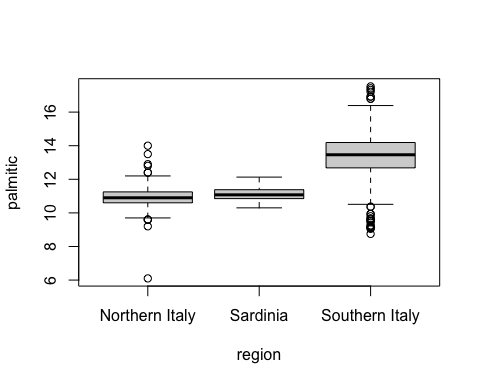
hist(olive$eicosenoic)



* ☒ A. The most common value of eicosenoic acid is below 0.05%.
* ☐ B. The most common value of eicosenoic acid is greater than 0.5%.
* ☐ C. The most common value of eicosenoic acid is around 0.3%.
* ☐ D. There are equal numbers of olive oils with eicosenoic acid below 0.05% and greater than 0.5%.

1. Make a boxplot of palmitic acid percentage in olive with separate distributions for each region.

boxplot(palmitic ~ region, data = olive)



Which region has the highest median palmitic acid percentage? Southern Italy

Which region has the most variable palmitic acid percentage? Southern Italy

# Section 4 Overview

Section 4 introduces you to general programming features like ‘if-else’, and ‘for loop’ commands so that you can write your own functions to perform various operations on datasets.

In Section 4.1, you will:

* Understand some of the programming capabilities of R.

In Section 4.2, you will:

* Use basic conditional expressions to perform different operations.
* Check if any or all elements of a logical vector are TRUE.

In Section 4.3, you will:

* Define and call functions to perform various operations.
* Pass arguments to functions, and return variables/objects from functions.

In Section 4.4, you will:

* Use ‘for’ loop to perform repeated operations.
* Articulate in-built functions of R that you could try for yourself.

## Programming Basics - Introduction to Programming in R

The textbook for this section is available [here](https://rafalab.github.io/dsbook/programming-basics.html)

## Basic Conditionals

The textbook for this section is available [here](https://rafalab.github.io/dsbook/programming-basics.html#conditionals)

**Key Points**

* The most common conditional expression in programming is an if-else statement, which has the form “if [condition], perform [expression], else perform [alternative expression]”.
* The ifelse() function works similarly to an if-else statement, but it is particularly useful since it works on vectors by examining each element of the vector and returning a corresponding answer accordingly.
* The any() function takes a vector of logicals and returns true if any of the entries are true.
* The all() function takes a vector of logicals and returns true if all of the entries are true.

*Code*

# an example showing the general structure of an if-else statement  
a <- 0  
if(a!=0){  
 print(1/a)  
} else{  
 print("No reciprocal for 0.")  
}

## [1] "No reciprocal for 0."

# an example that tells us which states, if any, have a murder rate less than 0.5  
library(dslabs)  
data(murders)  
murder\_rate <- murders$total / murders$population\*100000  
ind <- which.min(murder\_rate)  
if(murder\_rate[ind] < 0.5){  
 print(murders$state[ind])   
} else{  
 print("No state has murder rate that low")  
}

## [1] "Vermont"

# changing the condition to < 0.25 changes the result  
if(murder\_rate[ind] < 0.25){  
 print(murders$state[ind])   
} else{  
 print("No state has a murder rate that low.")  
}

## [1] "No state has a murder rate that low."

# the ifelse() function works similarly to an if-else conditional  
a <- 0  
ifelse(a > 0, 1/a, NA)

## [1] NA

# the ifelse() function is particularly useful on vectors  
a <- c(0,1,2,-4,5)  
result <- ifelse(a > 0, 1/a, NA)  
  
# the ifelse() function is also helpful for replacing missing values  
data(na\_example)  
no\_nas <- ifelse(is.na(na\_example), 0, na\_example)   
sum(is.na(no\_nas))

## [1] 0

# the any() and all() functions evaluate logical vectors  
z <- c(TRUE, TRUE, FALSE)  
any(z)

## [1] TRUE

all(z)

## [1] FALSE

## Functions

The textbook for this section is available [here](https://rafalab.github.io/dsbook/programming-basics.html#defining-functions)

**Key points**

* The R function, called function() tells R you are about to define a new function.
* Functions are objects, so must be assigned a variable name with the arrow operator.
* The general way to define functions is: (1) decide the function name, which will be an object, (2) type function() with your function’s arguments in parentheses, (3) write all the operations inside brackets.
* Variables defined inside a function are not saved in the workspace.

*Code*

# example of defining a function to compute the average of a vector x  
avg <- function(x){  
 s <- sum(x)  
 n <- length(x)  
 s/n  
}  
  
# we see that the above function and the pre-built R mean() function are identical  
x <- 1:100  
identical(mean(x), avg(x))

## [1] TRUE

# variables inside a function are not defined in the workspace  
s <- 3  
avg(1:10)

## [1] 5.5

s

## [1] 3

# the general form of a function  
my\_function <- function(VARIABLE\_NAME){  
 perform operations on VARIABLE\_NAME and calculate VALUE  
 VALUE  
}

# functions can have multiple arguments as well as default values  
avg <- function(x, arithmetic = TRUE){  
 n <- length(x)  
 ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))  
}

## For Loops

The textbook for this section is available [here](https://rafalab.github.io/dsbook/programming-basics.html#for-loops)

**Key points**

* For-loops perform the same task over and over while changing the variable. They let us define the range that our variable takes, and then changes the value with each loop and evaluates the expression every time inside the loop.
* The general form of a for-loop is: “For i in [some range], do operations”. This i changes across the range of values and the operations assume i is a value you’re interested in computing on.
* At the end of the loop, the value of i is the last value of the range.

*Code*

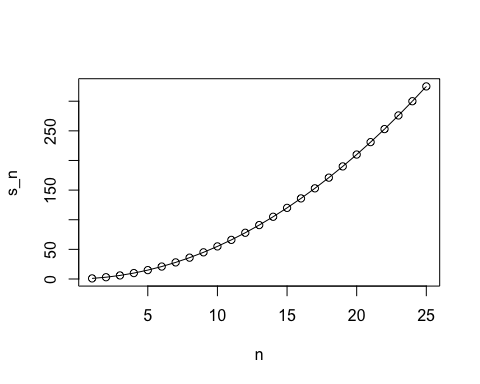
# creating a function that computes the sum of integers 1 through n  
compute\_s\_n <- function(n){  
 x <- 1:n  
 sum(x)  
}  
  
# a very simple for-loop  
for(i in 1:5){  
 print(i)  
}

## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5

# a for-loop for our summation  
m <- 25  
s\_n <- vector(length = m) # create an empty vector  
for(n in 1:m){  
 s\_n[n] <- compute\_s\_n(n)  
}  
  
# creating a plot for our summation function  
n <- 1:m  
plot(n, s\_n)  
  
# a table of values comparing our function to the summation formula  
head(data.frame(s\_n = s\_n, formula = n\*(n+1)/2))

## s\_n formula  
## 1 1 1  
## 2 3 3  
## 3 6 6  
## 4 10 10  
## 5 15 15  
## 6 21 21

# overlaying our function with the summation formula  
plot(n, s\_n)  
lines(n, n\*(n+1)/2)



## Assessment - Programming Basics

1. What will this conditional expression return?

x <- c(1,2,-3,4)  
if(all(x>0)){  
print("All Postives")  
} else{  
print("Not all positives")  
}

## [1] "Not all positives"

* ☐ A. All Positives
* ☒ B. Not All Positives
* ☐ C. N/A
* ☐ D. None of the above

1. Which of the following expressions is always FALSE when at least one entry of a logical vector x is TRUE?

* ☐ A. all(x)
* ☐ B. any(x)
* ☐ C. any(!x)
* ☒ D. all(!x)

1. The function nchar tells you how many characters long a character vector is.

For example:

char\_len <- nchar(murders$state)  
head(char\_len)

The function ifelse is useful because you convert a vector of logicals into something else. For example, some datasets use the number -999 to denote NA. A bad practice! You can convert the -999 in a vector to NA using the following ifelse call:

x <- c(2, 3, -999, 1, 4, 5, -999, 3, 2, 9)  
ifelse(x == -999, NA, x)

If the entry is -999 it returns NA, otherwise it returns the entry.

# Assign the state abbreviation when the state name is longer than 8 characters   
char\_len <- nchar(murders$state)  
new\_names <- ifelse(char\_len > 8, murders$abb, murders$state)

1. You will encounter situations in which the function you need does not already exist. R permits you to write your own.

Let’s practice one such situation, in which you first need to define the function to be used. The functions you define can have multiple arguments as well as default values.

To define functions we use function. For example the following function adds 1 to the number it receives as an argument:

{r, eval=FALSE, echo=TRUE my\_func <- function(x){ y <- x + 1 y }

The last value in the function, in this case that stored in y, gets returned.

If you run the code above R does not show anything. This means you defined the function. You can test it out like this:

my\_func(5)

# Create function called `sum\_n`  
sum\_n <- function(n){  
 x <- 1:n  
 sum(x)  
}  
  
# Use the function to determine the sum of integers from 1 to 5000  
sum\_n(5000)

## [1] 12502500

1. We will make another function for this exercise. We will define a function altman\_plot that takes two arguments x and y and plots the difference y-x in the y-axis against the sum x+y in the x-axis.

You can define functions with as many variables as you want. For example, here we need at least two, x and y. The following function plots log transformed values:

log\_plot <- function(x, y){  
 plot(log10(x), log10(y))  
}

This function does not return anything. It just makes a plot.

# Create `altman\_plot`   
altman\_plot <- function(x, y) {  
 plot(x+y, y-x)  
}

1. Lexical scoping is a convention used by many languages that determine when an object is available by its name.

When you run the code below you will see which x is available at different points in the code.

x <- 8  
my\_func <- function(y){  
 x <- 9  
 print(x)  
 y + x  
}  
my\_func(x)  
print(x)

Note that when we define x as 9, this is inside the function, but it is 8 after you run the function. The x changed inside the function but not outside.

# Run this code   
x <- 3  
 my\_func <- function(y){  
 x <- 5  
 y+5  
}  
  
# Print the value of x   
x <- 3  
my\_func <- function(y){  
 x <- 5  
 y  
 print(x)  
}  
my\_func(x)

## [1] 5

print(x)

## [1] 3

1. In the next exercise we are going to write a for-loop. In that for-loop we are going to call a function. We define that function here.

# Here is an example of a function that adds numbers from 1 to n  
example\_func <- function(n){  
 x <- 1:n  
 sum(x)  
}  
  
# Here is the sum of the first 100 numbers  
example\_func(100)

## [1] 5050

# Write a function compute\_s\_n with argument n that for any given n computes the sum of 1 + 2^2 + ...+ n^2  
compute\_s\_n <- function(n){  
 x <- 1:n  
 sum(x^2)  
}  
  
# Report the value of the sum when n=10  
compute\_s\_n(10)

## [1] 385

1. Now we are going to compute the sum of the squares for several values of *n*. We will use a for-loop for this.

Here is an example of a for-loop:

results <- vector("numeric", 10)  
n <- 10  
for(i in 1:n){  
 x <- 1:i  
 results[i] <- sum(x)  
}

Note that we start with a call to vector which constructs an empty vector that we will fill while the loop runs.

# Define a function and store it in `compute\_s\_n`  
compute\_s\_n <- function(n){  
 x <- 1:n  
 sum(x^2)  
}  
  
# Create a vector for storing results  
s\_n <- vector("numeric", 25)  
  
# write a for-loop to store the results in s\_n  
for(n in 1:length(s\_n)){  
 s\_n[n] <- compute\_s\_n(n)  
}

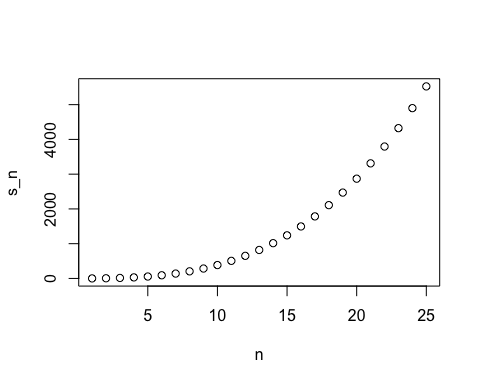
1. If we do the math, we can show that

We have already computed the values of from 1 to 25 using a for loop.

If the formula is correct then a plot of versus *n* should look cubic.

Let’s make this plot.

# Define the function  
compute\_s\_n <- function(n){  
 x <- 1:n  
 sum(x^2)  
}  
  
# Define the vector of n  
n <- 1:25  
  
# Define the vector to store data  
s\_n <- vector("numeric", 25)  
for(i in n){  
 s\_n[i] <- compute\_s\_n(i)  
}  
  
# Create the plot   
plot(n,s\_n)



1. Now let’s actually check if we get the exact same answer.

# Define the function  
compute\_s\_n <- function(n){  
 x <- 1:n  
 sum(x^2)  
}  
  
# Define the vector of n  
n <- 1:25  
  
# Define the vector to store data  
s\_n <- vector("numeric", 25)  
for(i in n){  
 s\_n[i] <- compute\_s\_n(i)  
}  
  
# Check that s\_n is identical to the formula given in the instructions.  
identical(s\_n,(n\*(n+1)\*(2\*n+1))/6)

## [1] TRUE

## Section 4 Assessment

1. Load the **heights** dataset from dslabs:

library(dslabs)  
data(heights)

Write an ifelse statement that returns 1 if the sex is Female and 2 if the sex is Male.

What is the sum of the resulting vector?

sum(ifelse(heights$sex == "Female", 1, 2))

## [1] 1862

1. Write an ifelse statement that takes the height column and returns the height if it is greater than 72 inches and returns 0 otherwise.

What is the mean of the resulting vector?

mean(ifelse(heights$height > 72, heights$height, 0))

## [1] 9.65

1. Write a function inches\_to\_ft that takes a number of inches x and returns the number of feet. One foot equals 12 inches.

What is inches\_to\_ft(144)?

inches\_to\_ft <- function(x){x/12}  
inches\_to\_ft(144)

## [1] 12

How many individuals in the heights dataset have a height less than 5 feet?

sum(inches\_to\_ft(heights$height) < 5)

## [1] 20

1. Which of the following are TRUE?

Select ALL that apply.

* ☒ A. any(TRUE, TRUE, TRUE)
* ☒ B. any(TRUE, TRUE, FALSE)
* ☒ C. any(TRUE, FALSE, FALSE)
* ☐ D. any(FALSE, FALSE, FALSE)
* ☒ E. all(TRUE, TRUE, TRUE)
* ☐ F. all(TRUE, TRUE, FALSE)
* ☐ G. all(TRUE, FALSE, FALSE)
* ☐ H. all(FALSE, FALSE, FALSE)

1. Given an integer x, the factorial of x is called x! and is the product of all integers up to and including x. The factorial() function computes factorials in R. For example, factorial(4) returns .

Complete the code below to generate a vector of length m where the first entry is 1!, the second entry is 2!, and so on up to m!.

# define a vector of length m  
m <- 10  
f\_n <- vector(length = m)  
  
# make a vector of factorials  
for(n in 1:m){  
f\_n[n] <- factorial(n)  
}  
  
# inspect f\_n  
f\_n

## [1] 1 2 6 24 120 720 5040 40320 362880  
## [10] 3628800

* ☐ A. function(n)
* ☐ B. if(n < m)
* ☒ C. for(n in 1:m)
* ☐ D. function(m,n)
* ☐ E. if(m < n)
* ☐ F. for(m in 1:n)