

# Data Science R Basics

## 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](#)

## Motivation

Here is a link to the textbook section on the motivation for this course.

## Getting started

Here is a link to the textbook section on Getting Started 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 (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.

### 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. Their cheatsheet 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.

## R Basics - Objects

Here is a link to the textbook section on objects in R.

### 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  $x^2+x-1=0$*

```
# 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.

### 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  $n(n+1)/2$  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
```

2. What is the sum of the first 1000 positive integers? We can use the formula  $n(n+1)/2$  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
```

3. 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.

4. 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
```

5. 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](#).

### 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.

2. 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"
```

3. 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"
```

4. 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
```

5. 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
```

6. 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  $ax^2 + bx + c$ , use the quadratic equation:  $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ .

What are the two solutions to  $2x^2 - x - 4 = 0$ ? 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
```

## 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.

The textbook for this section is available [here](#)

## Assessment 3

1. Use the function `c` to create a vector with the average high temperatures in January for Beijing, Lagos, Paris, Rio de Janeiro, San Juan and Toronto, which are 35, 88, 42, 84, 81, and 30 degrees Fahrenheit. Call the object `temp`.

```
# 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(35, 88, 42, 84, 81, 30)
```

2. Now create a vector with the city names and call the object city.

```
# 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")
```

3. Use the names function and the objects defined in the previous exercises to associate the temperature data with its corresponding city.

```
# 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
```

4. Use the [ and : operators to access the temperature of the first three cities on the list.

```
# 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
```

5. Use the [ operator to access the temperature of Paris and San Juan.

```
# 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
```

6. Use the : operator to create a sequence of numbers 12, 13, 14,...,73.

```
# 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 12 and ends at 73.
x <- 12:73
```

```
# Determine the length of object x.
length(x)
```

```
## [1] 62
```

7. Create a vector containing all the positive odd numbers smaller than 100.

```
# 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(1,100,2)
```

```
## [1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45
## [24] 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91
## [47] 93 95 97 99
```

8. Create a vector of numbers that starts at 6, does not pass 55, and adds numbers in increments of  $4/7$ :  $6, 6+4/7, 6+8/7$ , etc.. How many numbers does the list have? Hint: use seq and length.

```
# We can 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 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
```

9. What is the class of the following object `a <- seq(1, 10, length.out = 100)`?

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

```
# Determine the class of a
class(a)
```

```
## [1] "numeric"
```

10. What is the class of the following object `a <- seq(1, 10)`?

```
# Store the sequence in the object a
a <- seq(1, 10)
```

```
# Determine the class of a
class(a)
```

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

11. The class of `class(a<-1)` is numeric, not integer. R defaults to numeric and to force an integer, you need to add the letter L. Confirm that the class of `1L` is integer.

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

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

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

12. Define the following vector:

```
x <- c("1", "3", "5", "a")
```

and coerce it to get integers.

```

# 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
x <- as.numeric(x)

# You will get a warning but that is ok

```

## Assessment 4

For these exercises we will use the US murders dataset. Make sure you load it prior to starting.

```

library(dslabs)
data("murders")

```

1. Use the \$ operator to access the population size data and store it as the object pop. Then use the sort function to redefine pop so that it is sorted. Finally, use the [ operator to report the smallest population size.

```

# 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

```

2. Now instead of the smallest population size, find the index of the entry with the smallest population size. Hint: use order instead of sort.

```
# Access population from the dataset and store it in pop
pop <- murders$population

# Use the command order, to order pop and store in object o
o <- order(pop)

# Find the index number of the entry with the smallest population size
which.min(murders$population)

## [1] 51
```

3. We can actually perform the same operation as in the previous exercise using the function `which.min`. Write one line of code that does this.

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

## [1] 46

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

## [1] 51
```

4. Now we know how small the smallest state is and we know which row represents it. Which state is it? Define a variable `states` to be the state names from the `murders` data frame. Report the name of the state with the smallest population.

```
# 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"
```

5. 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)
```

Use the `rank` function to determine the population rank of each state from smallest population size to biggest. Save these ranks in an object called `ranks`, then create a data frame with the state name and its rank. Call the data frame `my_df`.

```

# 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, ranks)

```

6. Repeat the previous exercise, but this time order my\_df so that the states are ordered from least populous to most populous. Hint: create an object ind that stores the indexes needed to order the population values. Then use the bracket operator [ to re-order each column in the data frame.

```

# 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])

```

7. The na\_example vector represents a series of counts. You can quickly examine the object using:

```

data("na_example")
str(na_example)
nt [1:1000] 2 1 3 2 1 3 1 4 3 2 ...

```

However, when we compute the average with the function mean, we obtain an NA:

```

mean(na_example)
[1] NA

```

The is.na function returns a logical vector that tells us which entries are NA. Assign this logical vector to an object called ind and determine how many NAs does na\_example have.

```

# 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
```

8. Now compute the average again, but only for the entries that are not NA. Hint: remember the ! operator.

```
# 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] 2.301754
```

## Assessment 5

1. Previously we created this data frame:

```
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)
```

Remake the data frame using the code above, but add a line that converts the temperature from Fahrenheit to Celsius. The conversion is  $C = (F - 32) \times \frac{5}{9}$ .

```
# 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)
```

2. What is the following sum  $1 + 1/2^2 + 1/3^2 + .1/100^2$  ? Hint: thanks to Euler, we know it should be close to  $\pi^2/6$ .

```
# Define an object `x` with the numbers 1 through 100
x <- c(1:100)
```

```
# Compute the sum
sum(1/x^2)
```

```
## [1] 1.634984
```

3. 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?

```
# Load the data
library(dslabs)
data(murders)
```

```
# 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] 2.779125
```

## 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.

The textbook for this section is available [here](#)

## Assessment 6

Start by loading the library and data.

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

1. 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
```

2. Now use the results from the previous exercise and the function `which` to determine the indices of `murder_rate` associated with values 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

# Get the indices of entries that are below 1
which(low)

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

3. Use the results from the previous exercise to report the names of the states with murder rates 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

# 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"
```

4. Now extend the code from exercise 2 and 3 to report the states in the Northeast with murder rates lower than 1. Hint: use the previously defined logical vector `low` and the logical operator `&`.

```

# 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
ind <- low & murders$region=='Northeast'

# Names of states in `ind`
murders$state[ind]

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

```

5. 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 average murder rate and store in avg using `mean`
avg <- mean(murder_rate)

# How many states have murder rates below avg ? Check using sum
sum(murder_rate<avg)

## [1] 27

```

6. Use the match function to identify the states with abbreviations AK, MI, and IA. Hint: start by defining an index of the entries of murders\$abb that match the three abbreviations, then use the [ operator to extract the states.

```

# Store the 3 abbreviations in abbs in a vector (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"

```

7. Use the %in% operator to create a logical vector that answers the question: which of the following are actual abbreviations: MA, ME, MI, MO, MU ?

```

# 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

```