Notes on Data Science: R Basics

Igor Luciano de Paula

January 30, 2018

Instructor

- **Rafael Irizarry**: Professor of Biostatistics at the Harvard T.H. Chan School of Public Health and a Professor of Biostatistics and Computational Biology at the Dana Farber Cancer Institute.

This is a book of notes on Data Science: R Basics (code PH125.1x), a course offered by Harvard University on learning platform edX. Some of the content of this summary was made available by HarvardX at edX.

Introduction

The demand for skilled data science practitioners in industry, academia, and government is rapidly growing. The HarvardX Data Science Series prepares the necessary knowledge base and skills to tackle real-world data analysis challenges. The series covers concepts such as probability, inference, regression and machine learning and helps you develop a skill set that includes R programming, data wrangling with dplyr, data visualization with ggplot2, file organization with UNIX/Linux, version control with git and GitHub, and reproducible document preparation with RStudio. In the R Basics course, we learn the basic building blocks of R. As done in all our courses, we use motivating case studies, we ask specific questions, and learn by answering these through data analysis. Our assessments use code checking technology that will permit get hands-on practice during the courses.

Throughout the series, we will be using the R software environment. Learn R, statistical concepts, and data analysis techniques simultaneously. In this course, we will introduce the necessary basic R syntax to get you going. However, rather than cover every R skill you need, we introduce just enough so you can continue learning in the next courses, which will provide more in depth coverage. We believe that you can better retain R knowledge when you learn it to solve a specific problem. The motivating question in this course relates to crime in the United States and we provide a relevant dataset. You will learn some basic R skills to permit us to answer specific questions about differences across the different states.

1 R Basics, Functions, and Data Types

Overview:

- 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
- Common errors and how to fix them

1.1 Motivation

In this series, we'll be using the R software environment for all our analyses. We find that we better retain our knowledge when we learn it to solve a specific problem.As in throughout the series, in this course, we'll use a motivating case study. In this case, it's related to crime in the United States. A relevant data set and some basic skills will permit us to answer specific questions related to the topic. You have options of where to live in US, and want to find out how safe each state is. We will do this using R. We will examine data related to gun homicides in the US. We will delve into the homicide data, and answer practical, real world questions.

1.1.1 Getting Started

R is not a programming language like C or Java. It was not created by software engineers for software development. Instead, it was developed by statisticians and data analysts as an interactive environment for data analysis. The interactivity is an indispensable feature in data science, because as you will soon learn, the ability to quickly explore data is a necessity for success in this field. However, like in any other programming language, you can save your work in scripts, which you can easily execute at any moment. These scripts serve as a record of the analysis you performed, a key feature that facilitates reproducible work. Other attractive features of R are the following. R as free and open source, meaning that you can look at the code. It runs across all major platforms– Windows, Mac OS, Unix, Linux. And data objects and scripts can be shared seamlessly across these platforms. There's also a large and growing active community of our users. And as a result, there are numerous resources for learning and asking questions. It's easy for others to contribute add-ons, which enable developers to share software implementations of new data science techniques.

Interactive data analysis usually occurs on the R console. In the R console, you can execute commands as you type them. There are several ways to gain access to an R console. The most common way is to download and install R, and then start it up on your computer. When you type an expression into the console and hit Return, the expression is evaluated. You can edit and save these scripts using a text editor.

We highly recommend working on an interactive, integrated development environment in IDE. This includes editor, R-specific features, and then it has a console where you execute your code.

R has many, many, many functions, many, many tools for you to use. However, the functionality provided by a fresh install of R is only a small fraction of that. We refer to what you get after you first install R as base R. The extra functionality comes from add-ons available from developers. There are currently hundreds of these available from CRAN, and many others shared via other repositories such as GitHub. However, because not everybody needs all available functionality, we instead make different components available via packages. R makes it very easy to install packages from within R itself.

1.2 R Basics

1.2.1 Objects

Before we get started with a motivating data set, we need to cover the very basics of R, like objects.

One advantage of programming language is that we can define variables and keep such expressions general. In R, we use the assignment symbol, whichis less than followed by a minus. Note that we can also use the equals sign for assignment, but we recommend against it, because using it can cause confusion. To see the value stored in a, variable we simply type it like this. We type a in the console.A more explicit way to ask R to show us the value that's saved in a variable is to use a print function. We use the term object to describe stuff that is stored in R. Variables are examples, but objects can also be more complicated entities such as functions, which are described later. As we define objects in the console, we are actually changing what is called the workspace. You can see all the variables saved in your workspace by typing ls(). This is a function that shows you the names of the objects saved in your workspace. Now, if you try to recover the value of a variable that is not in your workspace, you'll receive an error.

1.2.2 Functions

Once you define variables, a data analysis process can usually be described as a series of functions applied to the data. R includes several predefined functions. Now there are many more functions, and even more can be added through packages. Here we review some important information related to functions, and we show you how to learn about functions and tell you a little bit about what's available. Note that these functions do not appear in the workspace, but they are available for use.

In general, to evaluate a function, we need to use parentheses. Note, for example, that if you don't, let's say we type the function ls without using parentheses, what it does is it shows us the code for ls. It doesn't evaluate the function. Unlike ls, which doesn't require any arguments, most functions require at least one, between the parentheses. In R, functions can be what we call nested. What we mean by nested is that you can call a function to get the argument that's going to be used by another function. So the important thing to remember is that functions are evaluated from the inside out when you nest them. Another very important thing to know about functions. You can learn about them by using the help system.

A very nice feature of R is that it documents its functions and that we can call help files and see, get much information about them. Help files are like user manuals for the functions. You can find out what the function expects and what it does by reviewing the very useful manuals included in R. You get help by using the help function like this. Help, open parentheses, and then the function name in quotes. For most functions, you can use as a shorthand, which is the question mark followed by the function name. And that will show you the same help file. The help file will show you what the function is expecting. We call these arguments. For example, log expects an x, a value. And it also expects space. You can see that from looking at the help file for log. However, some arguments are required, and others are optional. You can determine which arguments are optional by noting in the help document that a default value is a sign with the equal sign. If you already know how the function works but need a quick reminder of the arguments, you can use the args function. If I type args of log, it shows us the two arguments that it needs. We can save ourselves some time,because if no argument name is used, R assumes you're entering arguments in the order shown in the help file or by args. So by not using the names, R assumes that the arguments are x, and then the next one is base. We said that functions need parentheses to be evaluated, but there are some exceptions. Among these, the most commonly used are the arithmetic and relational operators. For example, 2 to the 3, that function that takes 2 to the power of 3 doesn't need a parentheses. We just write it out as we would do in a mathematical formula. You can see the arithmetic operators by looking at the help file. For, for example, the plus operator. So we would type help, and then in quotes, plus. That will show us all the mathematical operators. We can also get help with a question mark, but we need to include quotes for these particular operators.

Functions are not the only pre-built objects in R. There are also data sets that are included for users to practice and test out functions. You can see all the available data sets by typing data, parentheses, parentheses. This shows you the object name for these data sets. And these data sets are objects that you can use by simply typing the name. For example, if you type CO2, one of the pre-built data objects, you will see CO2 measurements from a particular measurement station. There are also other mathematical objects that are also pre-built, such as the constant for pi and the infinity number. You can see them by typing pi or typing Inf.

Now when we are writing code, you want to think about the names you use for your variables. You want your code to be readable. There are some rules in R. Some basic rules are that they have to start with a letter, and they can't contain spaces. You also want to avoid using variable names of objects that are already defined in R, because that will cause a lot of confusion. For example, don't define a variable and call it installed or packages, because that's already a function in R. A nice convention to follow is to use meaningful words that describe what is stored, stick to lowercase, and instead of spaces, use underscores.

By creating and saving a script, like the code above, we would not need to retype everything, and again, simply change the variables.

Finally, another way you can make your code more readable is by including comments. If a line of R code starts with a hashtag, with the number symbol, it is not evaluated. We can use this to write reminders of why we wrote the code we wrote. For example, in the script that we just wrote for the quadratic equation, we can include simple comments like these. When we read it in the future, we'll be reminded of why we wrote these particular lines of code.

1.2.3 DataCamp

In this course we will be using the DataCamp platform for all assessments. DataCamp provides an R console and and a script editor right here on your browser. DataCamp, a learn-to-code company that is serving up all of the assignments for Data Science: R Basics. This first assignment teaches you the basics of R.

1.2.4 Assessment 1

1.3 Data Types

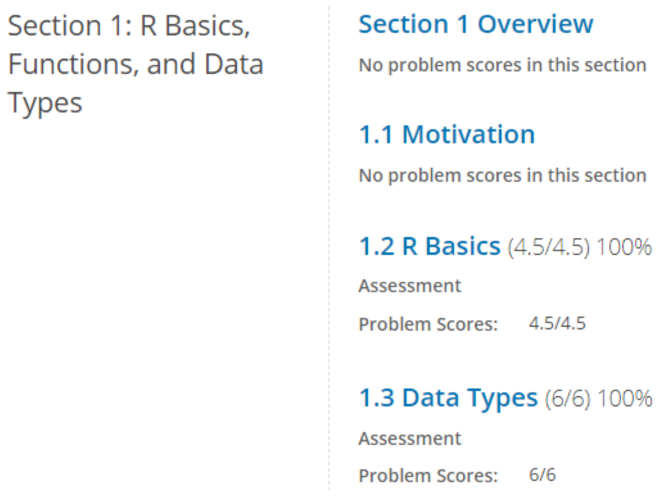
Variables in R can be of different types. For example, we need to distinguish numbers from character strings and tables from simple lists of numbers. The function class helps us determine the type of an object. Up to now, the variables we have defined have been just one number. This is not very useful for storing data. The most common way of storing data sets in R is with **data frames**. Conceptually, we can think of **data frames** as tables.Rows represent observations, and different variables are represented by different columns. We're going to see an example soon. Data frames are particularly useful for data sets because we can combine different types into one single object. We store the data for a motivating example in a data frame. You can access this data set by loading the dslabs library, and then after that, loading the murders data set. We do that using the function data. To see that this is, in fact, the data frame, we type class murders, and we see that R tells us that in fact, it is a data frame. This is where the relevant data for answering our questions is stored. So we know our data is stored in the object murders. But what is this object? How can we find out more about this object? The function str is useful for this. str stands for structure, and it shows us the structure of an object. If we type str murders, it shows us that, for example, it's a data frame. It has 51 observations, 51 rows, and five variables. On the left, we can see the variable names, state, abb, which is abbreviation, region, population, and total. So this all makes sense. We can see that this is going to all be useful for answering our questions. We can also show the first six lines of this data frame using the function head. We type head murders, we see Alabama, Alaska, Arizona, Arkansas, California, and Colorado, along with the variables associated with each state. This object follows the convention that we describe. It's quite common in data science. Rows are the different observation, in this case, states. And columns represent different variables. In this case, state, abbreviation, region, population, and total. OK, now we want to start accessing data from this object. So before we go any further in answering our original question about different states, let's get to know the components of this object a little better.

For our analysis, we will need to access the different variables represented by columns. To access these variables, we use the dollar sign symbol. It's called the accessor. When we used str to reveal the structure of the object, we saw the names of the columns. We can also get the names of the columns using the names function. It is important to know, for the rest of the analysis we're going to do, that the order of the entries in the murders dollar sign population list that we get, it preserves the order of the rows in our data table. As you will see, this will later permit us to manipulate one variable based on the results of another. For example, we might want to manipulate the state names by the number of murders. Note that the object, murders dollar sign population, is not one number. It's 51. We call these types of objects vectors. A single number is technically a vector. But in general, they have several entries. The function length tells you how many. So if I define a new object called pop for population, as the column in the murders data set that has the population sizes, and then I type length pop, I will see that it's a vector with 51 entries, one for each state. This particular vector is a numeric vector since population size, there are numbers. We can store characters in R as well. Because variables also use character strings, we're going to use quotes to distinguish between variable names and character strings. An example of a character vector is the column with the state names. So if I grab that column by using the dollar sign and I look at the class of that column, you will see that it's a character. As with numeric vectors, all entries in a character vector need to be a character. Logical vectors or another type of vector too, these must be either true or false. So we can create a logical vector by, for example, assigning 3 equal equal to 2 to the object z. When I type z, we see that it's false. That's because 3 is not equal to 2. And if we type class of z, it returns illogical. This is because equal equal is a relational operato, that is different "=".

In the murders data set, we have a column called regions. These are the regions of the US, so which state is in which region. We would think that this would be a character, because the regions are things like Northeast and South, et cetera. But when we look at the class of the regions column, we see something new, it says **factor**. So let's learn about factors, as they appear often in R and in data science. Factors are useful for storing what is called categorical data. The regions are categorical. There is only four categories. Each state is in one of these four. We can see the four levels in this particular factor using the function levels. So why do we do this? Turns out that saving categorical data this way is more memory efficient. So in R, in the background, we store integers. We store the levels as integers. Technically, integers are smaller memory-wise than characters. So this is what makes it more efficient. However, factors are also a source of confusion, as they can easily be confused with characters. If we just look at this variable, they would appear to be characters. It's only when we ask using class what type it is that we see there are factors. In general, we recommend avoiding factors as much as possible. Although, as we will learn later, they are sometimes necessary to fit statistical models that depend on categorical data.

1.3.1 Assessment 2

1.3.2 Scores Section 1: R Basics, Functions, and Data Types



Section 1: R Basics, Functions, and Data Types

Section 1 Overview

No problem scores in this section

1.1 Motivation

No problem scores in this section

1.2 R Basics (4.5/4.5) 100%

Assessment

Problem Scores: 4.5/4.5

1.3 Data Types (6/6) 100%

Assessment

Problem Scores: 6/6

2 Vectors, Sorting

Overview:

- Section 2.1:

  - 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.

- Section 2.2:

  - 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.

- Section 2.3:

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

‘

2.1 Vectors

The most basic unit available in R to store data are vectors. Complex datasets can usually be broken down into components that are vectors, like, each column can be a vector.

Let's start by learning how to create vectors. One way we can do that is by using the function C, which stands for concatenate.

We can also create character vectors. We use the quotes to denote that the entries are characters rather than variables. If you don't use the quotes, R looks for variables with those names, and in this case, it will return an error, because you haven't defined objects named Italy, Canada, or Egypt.

Sometimes it's useful to name the entries of a vector. For example, when defining a vector of country codes as we did, we can use the names to connect the two. So here we would type codes, then use concatenate.But this time, instead of just writing out the three numbers, we would assign a name to each one.

Now, if we use strings without quotes, it looks a little confusing to some. But you should know that you can use strings here. You can do the same exact expression. But this time, you use quotes to define the names.

We can use that does exactly the same thing as the previous two chunks of code, but using the names function.

Another useful function for creating vectors, and we use this often, is a function that generates sequences. The function is seq, stands for sequence. So if I type seq, then from 1, 10, it writes out the numbers from 1 to 10, consecutive integers from 1 to 10. In this function, the first argument defines the start, and the second defines the end. The default is to go in increments of 1. But a third argument, which defaults to 1, lets us tell seq how much to jump by. So for example, if I type seq(1, 10, 2), it would write out the odd numbers from 1 to 9. Note that if we want consecutive integers, we can use the following shorthand. We can type 1:10 and we get the integers from 1 to 10.

Now let's go over subsetting, an important topic. It lets us access specific parts of a vector. We use square brackets to access elements of a vector. So for example, we can access the second element of codes by simply typing codes, square bracket, 2, close square bracket. So we can type codes, and then create the vector 1, 3. And now we get the first and third element. The sequences defined above are particularly useful if you want to access, say, the first two elements. So we can type codes and then 1 through 2, we get the first two elements of our vector. If the elements have names, we can also access the entries using these names. For example, if we type codes, open brackets, and then canada in quotes, it will access the entry that has the name Canada. We can also have vector of names that are longer than 1.

In general, coercion is an attempt by R to be flexible with data types. When an entry does not match the expected, R tries to guess what we meant before throwing it in there. But this can also lead to confusion. Failing to understand coercion can drive programmers crazy when attempting to code in R, since it behaves quite differently from most other languages. We earlier said that vectors must be all of the same type. So if we try to combine say numbers and characters, you might expect an error. But if we type x and assign 1, canada, 3, we don't get an error. We don't even get a warning. Even though one and 3 were originally numbers when we wrote it out, it has converted them to character. We say that R coerced the data into a character string. R also offers functions to force a specific coercion. For example, you can turn numbers into characters with the as.character function. We can turn them back using the as.numeric function, which converts characters or other data types into numeric variables. This function is actually quite useful in practice, because many datasets, many public datasets that include numbers, include them in a form that makes them appear to be character strings.

Missing data is very common in practice. In R, we have a special value for missing data. It's the NA("Not Available"). We can get to NA's from coercion. For example, when R fails to coerce something, it tries to coerce but it can't, we will get NA. Note that, as a data scientist, you will encounter the NA often, as they are used as missing data. And as I said, this is a very common problem in real life data sets. So be sure to know what NA means. And be ready to see a lot of them.

2.1.1 Assessment 3

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.

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 integers. There is an equivalent function that converts integers to strings, as.character(x).

2.2 Sorting

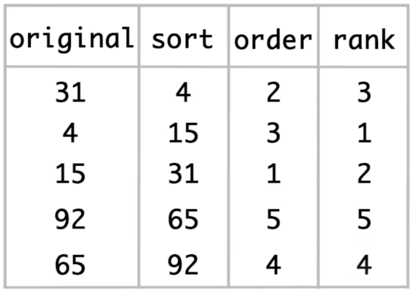
The function sort, sorts the vector in increasing order.

The order takes a vector and returns the indices that sorts that vector. Is the index that puts x in order.

Now, there's actually a much faster way of doing this, of getting the biggest or the smallest. If we are only interested in the entry with the largest value, we can use the function max. For the minimum, we can do the same with min and which.min.

So is California the most dangerous state?

In the next section, we argue that we should be considering rates, not total.
But before doing that, we introduce one last order related function– rank. For any given list, it gives you a vector with the rank of the first entry, second entry, et cetera.
All right, now that we have learned about sorting in R, we have some useful tools to answer questions about the data at hand.



original	sort	order	rank
31	4	2	3
4	15	3	1
15	31	1	2
92	65	5	5
65	92	4	4

2.2.1 Assessment 4

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.
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")
citytemps <- data.frame(name = city, temperature = temp)
```

Find out the mean of the entire dataset: mean(example). This function 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.
The is.na() returns a logical vector that tells us which entries are NA.

2.3 Vector Arithmetic

We saw that California had the most murders of any state. But does this mean it is the most dangerous state? What if it just has many, many more people than any other one? Using the tools we have learned, we can very quickly confirm that indeed, California has the largest population.

```
> murders$state[which.max(murders$population)]
[1] "California"
> max(murders$population)
[1] 37253956
```

With over 37 million inhabitants, it might be unfair to compare California to other states. What we really should be computing is the murders per capita. Here, the powerful vector arithmetic capabilities of R come in handy. In R, arithmetic operations on vectors occur element-wise, one by one. See the examples:

```
heights <- c(69,62,66,70,70,73,67,73,67,70)
> heights * 2.54
 [1] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 185.42 170.18
     177.80
> heights - 69
 [1]  0 -7 -3 1  1  4 -2  4 -2  1
> #this is the difference
```

Now, let's calculate the murders per capita:

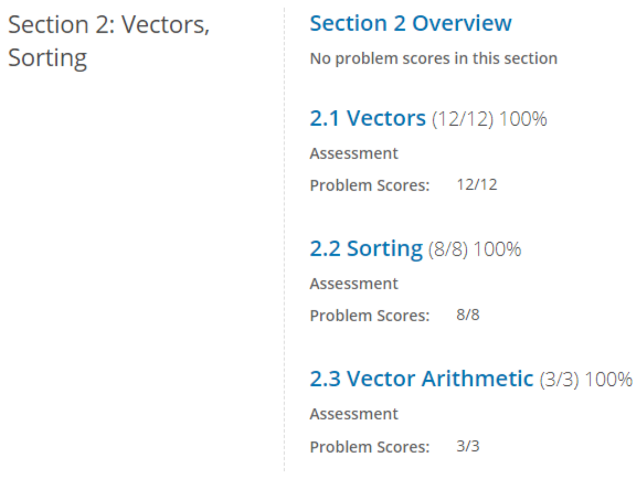
```
> murder_rate <- murders$total/murders$population*100000
> murder_rate
 [1]  2.8244238 2.6751860 3.6295273 3.1893901 3.3741383 1.2924531
 [7]  2.7139722 4.2319369 16.4527532 3.3980688 3.7903226 0.5145920
[13]  0.7655102 2.8369608 2.1900730 0.6893484 2.2081106 2.6732010
[19]  7.7425810 0.8280881 5.0748655 1.8021791 4.1786225 0.9992600
[25]  4.0440846 5.3598917 1.2128379 1.7521372 3.1104763 0.3798036
[31]  2.7980319 3.2537239 2.6679599 2.9993237 0.5947151 2.6871225
[37]  2.9589340 0.9396843 3.5977513 1.5200933 4.4753235 0.9825837
[43]  3.4509357 3.2013603 0.7959810 0.3196211 3.1246001 1.3829942
[49]  1.4571013 1.7056487 0.8871131
> # multiply by 100,000 to get it in the right units.
> 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"
>
```

Once we do this, we notice that California is no longer near the top of the list. We can use what we've learned to order the states by murder rate. Here I'm going to look at the states ordered by murder rate, in this case, in decreasing order. And now we see that California is not even in the top 10. The highest murder rate is in the District of Columbia, the second one is in Louisiana, et cetera, and California is only the 14th.

2.3.1 Assessment 5

2.3.2 Scores Section 2: Vectors, Sorting



Section 2: Vectors, Sorting

Section 2 Overview

No problem scores in this section

2.1 Vectors (12/12) 100%

Assessment

Problem Scores: 12/12

2.2 Sorting (8/8) 100%

Assessment

Problem Scores: 8/8

2.3 Vector Arithmetic (3/3) 100%

Assessment

Problem Scores: 3/3

3 Indexing, Data Wrangling, Plots

Overview:

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

Overview:

- Section 3.1:

- 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.

- Section 3.2:

- 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.

- Section 3.3:

- Plot data in scatter plots, box plots and histograms.
'

3.0.1 Indexing

R provides a powerful and convenient way of indexing vectors, like logicals to index factors. Furthermore, the principles of vector arithmetics that we've already learned apply to logical operations. For example, if we compare a vector to a single number, it actually performs the test for each entry. Can leverage the fact that vectors can meet the index with logicals.

Another nice feature is that to count how many entries are true, the function sum returns the sum of these entries. This is because the logical vector gets coerced in numeric. True turns into a 1. False gets turned into a 0. So when we sum them, we're basically counting the cases that are true.

We can use the logical operators in R. And one of them is the and sign, which makes two logicals true only when they're both true. So true and true is true, but true and false is false. And false and false, of course, is false. And we've any others logical operators, like:

- <
- <=
- >
- >=
- ==
- !=
- !
- |
- &

3.0.2 Indexing Function

which, match, and %in%

- Which gives us the entries of a logical vector that are true.

- Match looks for entries in a vector and returns the index needed to access them.

- If rather than an index, we want to know whether or not each element of a first vector is in a second vector, we use the function %in%.

3.0.3 Assessment 6

3.1 Basic Data Wrangling

3.1.1 Basic Data Wrangling

The dplyr package provides intuitive functionality for working with tables. This package introduces functions that perform the most common operations in data manipulation and uses names for these functions that are relatively easy to remember.

For example, to change the data table by adding a new colum or changing an existing one, we use mutate. To filter the data by subsetting rows, we use the function filter. And to subset the data by selecting specific columns, we use select.We can also perform a series of operations –for example, select this and then filter that– by sending the results of one function to another function using what is called the pipe operator.

3.1.2 Creating a Data Frame

For many of the analyses we perform, we will find it necessary to create data frames in R. You can do this using the data.frame function. Now, be warned, by default, data-frame turns characters into factors.

3.1.3 Assessment 7

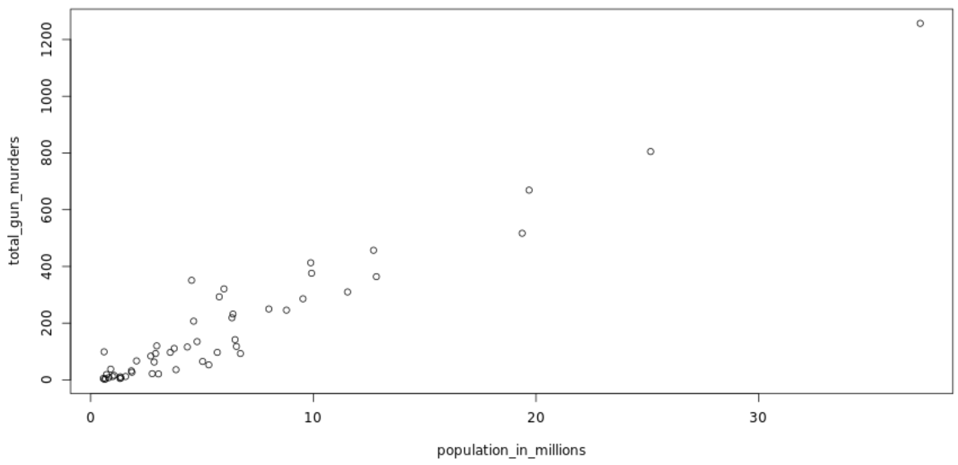
The dplyr function filter is used to choose specific rows of the data frame to keep. Unlke select which is for columns, filter is for rows. We can also use the %in% to filter with dplyr. The pipe %>% can be used to perform operations sequentially without having to define intermediate objects.

3.2 Basic Plots

Exploratory data visualization is perhaps the main strength of R. One can quickly go from idea to data to plot with a unique balance of flexibility and ease. For example, Excel may be easier than R, but it is nowhere as near as flexible. D3, an interactive data visualization programming language, may be more flexible and powerful than R, but it takes much longer to generate a plot.

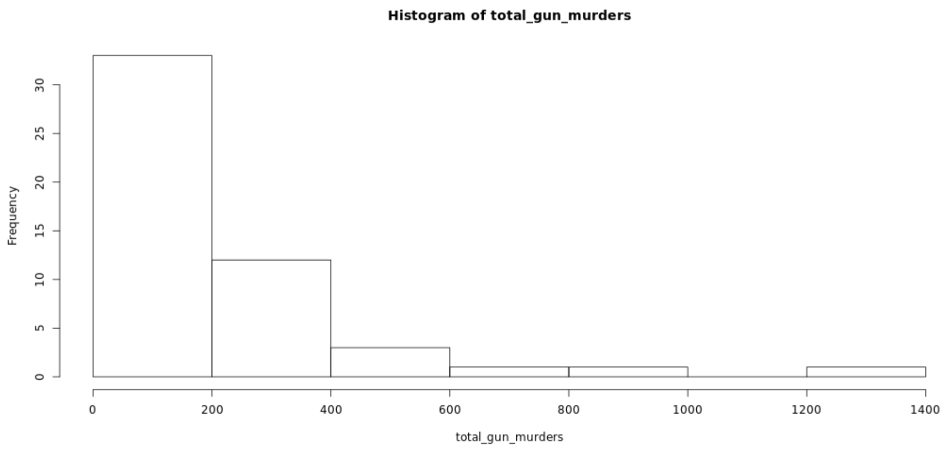
- Scatter Plots: plot()

```
> population_in_millions <- murders$population/10^6
+ total_gun_murders <- murders$total
> plot(population_in_millions, total_gun_murders)
```



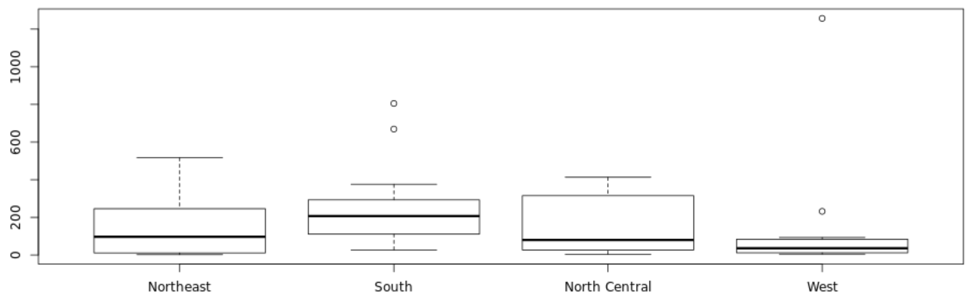
- Histograms: hist()

```
> hist(total_gun_murders)
```



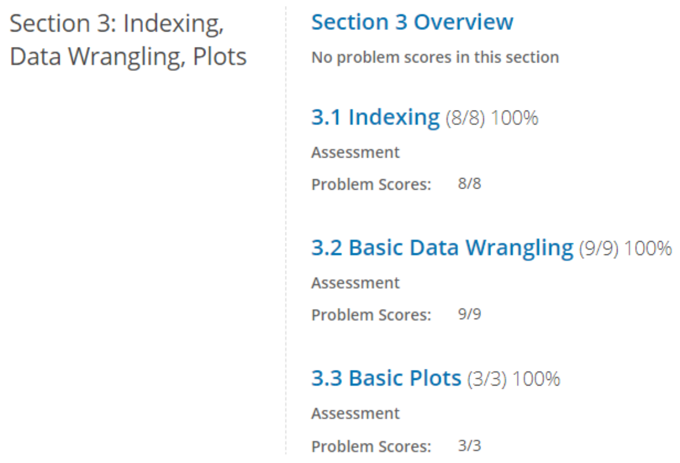
- Boxplot: boxplot()

```
> boxplot(total~region, data = murders)
```



3.2.1 Assessment 8

3.2.2 Scores Section 3: Indexing, Data Wrangling, Plots



Section 3: Indexing, Data Wrangling, Plots

Section 3 Overview

No problem scores in this section

3.1 Indexing (8/8) 100%

Assessment

Problem Scores: 8/8

3.2 Basic Data Wrangling (9/9) 100%

Assessment

Problem Scores: 9/9

3.3 Basic Plots (3/3) 100%

Assessment

Problem Scores: 3/3

4 Programming Basics

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.

Overview:

- Section 4.1:

- Understand some of the programming capabilities of R.

- Section 4.2:

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

- Section 4.3:

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

- Section 4.4:

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

4.1 Introduction to Programming in R

We teach R because it greatly acilitates data analysis, the main topic of this series.Coding in R, we can efficiently perform exploratory data analysis, build data analysis pipelines, and prepare data visualizations to communicate results. However, R is not just a data analysis environment, but a programming language. Advanced R programmers can develop complex packages, and even suggest ways to improve R itself. But we do not cover advanced programming in this course. However, in the next videos, we introduce three key programming concepts– conditional expressions, for-loops, and building functions. These are not just key building blocks for advanced programming, but often come in handy during data analysis.

4.2 Basic Conditionals

Conditional expressions are one of the basic features of programming. The most common conditional expression is the if-else statement. Here's a very simple example showing the general structure

```
if(all(x>0)){
    print("All Positives")
} else{
    print("Not All Positives")
}
```

A related function that is very useful is that ifelse function, one word, ifelse(). This function takes three arguments, a logical, and two possible answers. If the logical is true, the first answer is returned. If it's false, the second answers returned.

But the function is particularly useful, because it works on vectors. It examines each element of the logical vector and returns a corresponding answer accordingly. Example: replacing NA's with some other value, like ”0”.

Two more functions, any() and all(). These are also quite useful. Any function takes a vector of logicals and it returns true if any of the entries is true. The all function takes a vector of logicals and returns TRUE if all the entries are true.

4.3 Functions

As you become more experienced, you'll find yourself needing to perform the same operation over and over. A simple example of this is computing the average. You compute the average all the time when you're doing data science. You can compute the average of a vector x by computing the sum and then dividing by the length. This is longer than it really needs to be, because we can define a function that does this automatically. Because we do this so often, it's more efficient to write a function that performs this operation. And this has already been done. This is what the mean function does in R. However, you will encounter situations in which the function that you need does not already exist. So you have to write your own. A simple version of a function that computes the average can be defined like this.

```
> avg <- function(x){
    s <- sum(x)
    n <- length(x)
    s/n
  }
> #The last value in the function gets returned.
> z <- 1:100
> avg(z)
[1] 50.5
> identical(mean(z), avg(z))
[1] TRUE
```

Lexical Scope: note that variables defined inside a function are not saved in the workspace.

Note that functions can have more than one variable. The way you define them is by having multiple arguments.

4.4 For Loops

We're going to use an example from math. There's a formula that tells you what the sum of 1 plus 2 plus 3 plus dot dot dot, plus n is. The formula is:

$$1 + 2 + ... + n \quad = \tfrac{n \cdot (n+1)}{2} \tag{1}$$

But what if we weren't sure that that was the right formula? How could we check we're going to use R to check. Using what we learned about functions, we can easily create one that computes the sum. I'm going to call it s_n.

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
```

Say we want a compute it for 1, for 2 up to 25. So now we're computing 25 sums. Do we write 25 lines of code, one from each n? No. That's what loops are for. Note that we are performing exactly the same task over and over again, except we're changing n. For loops let us define the range that our variable takes. In our example, it would go from 1 to 25. Then change the value as you loop and evaluate the expression every time inside your loop. The general form looks like this:

```
for (i in range of values){
  operations that use i,
  which is changing across
  the range of values
}
```

here's a example:

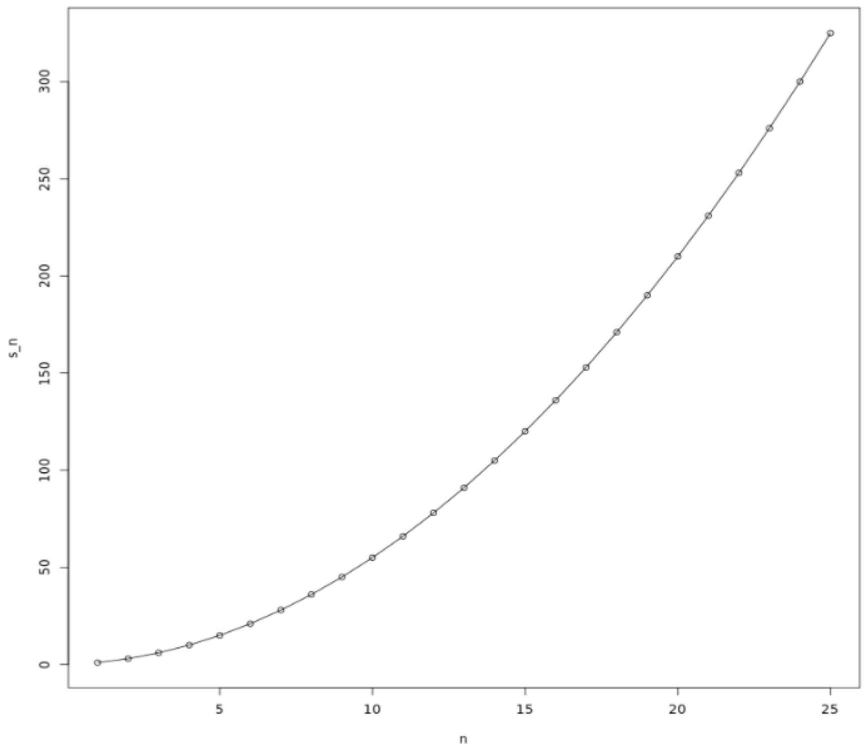
```
> for (i in 1:5){
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Note that we like to use i in for loops. But it can be any variable. Also note that at the end of the loop, the value of i is the last value of the range. So if I type i after that for loop, I get back 5.

```
> m <- 25
> s_n <- vector(length = m)
> for (n in 1:m){
+   s_n[n] <- compute_s_n(n)
+ }
> s_n
 [1]   1   3   6  10  15  21  28  36  45  55  66  78  91 105 120 136 153 171
     190
[20] 210 231 253 276 300 325
```

We can check to see if we did this right by, for example, making a plot.

```
n <- 1:m
plot(n, s_n)
lines(n, n*(n+1)/2) #we get the same answers.
```

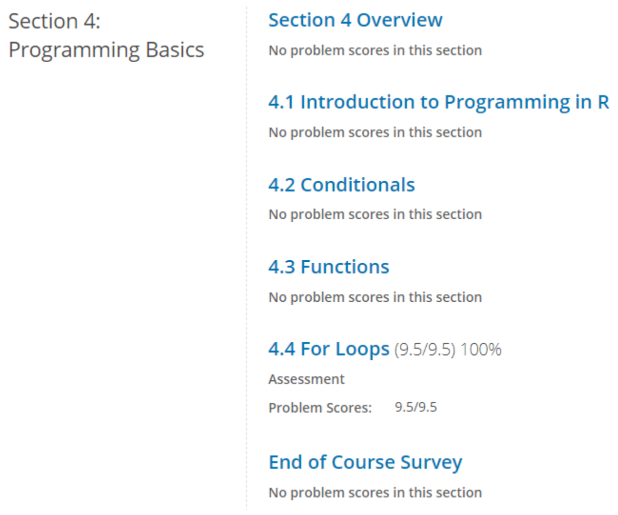


4.4.1 Other Functions

In a previously, we introduced for loops, but it turns out that we rarely use them in R. This is because there are usually more powerful ways to perform the same task. But the concept is still important, so it's good that we covered for loops. Functions that are typically used instead of for loops in R are apply, sapply, tapply, and mapply. These are part of what we call the apply family. We do not cover these functions in this course, but they are worth learning if you intend to go beyond this introduction. Other functions that are widely used are split, cut, quantile, reduce, identical, unique, and many others.

4.4.2 Assessment 9

4.4.3 Scores Section 4: Programming Basics



Section 4:
Programming Basics

Section 4 Overview
No problem scores in this section

4.1 Introduction to Programming in R
No problem scores in this section

4.2 Conditionals
No problem scores in this section

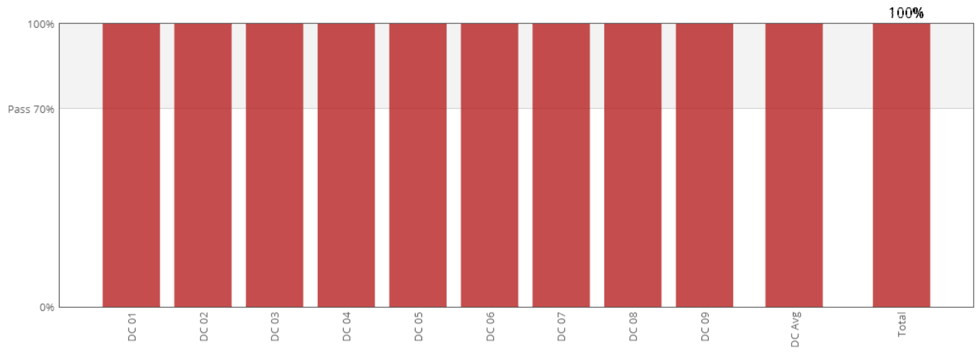
4.3 Functions
No problem scores in this section

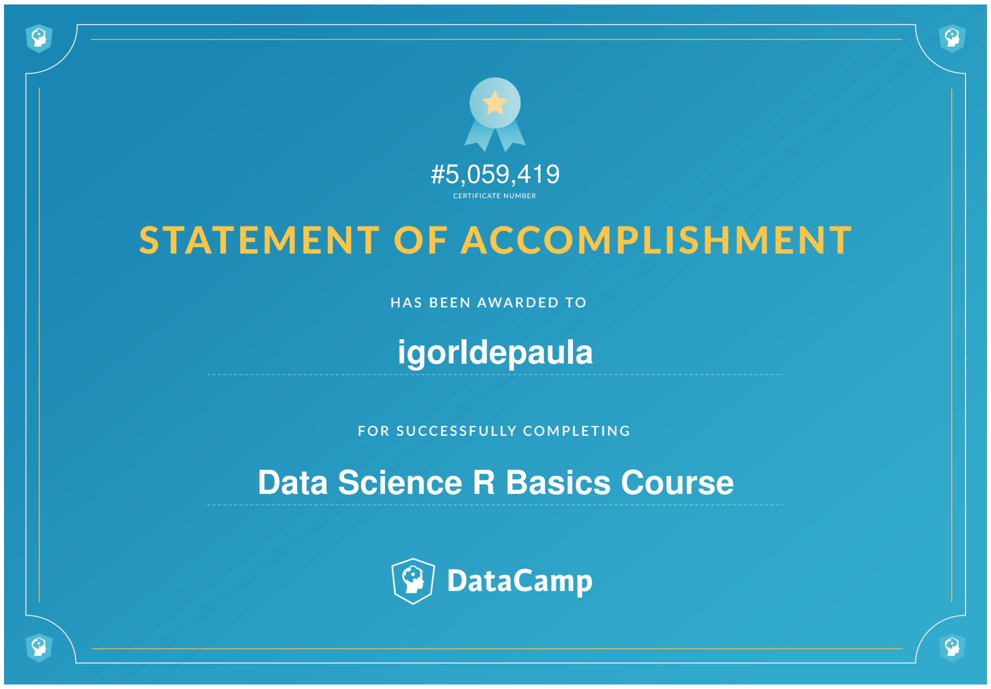
4.4 For Loops (9.5/9.5) 100%
Assessment
Problem Scores: 9.5/9.5

End of Course Survey
No problem scores in this section

4.5 Final Score and Certifications

Final Score: 100%





#5,059,419

CERTIFICATE NUMBER

STATEMENT OF ACCOMPLISHMENT

HAS BEEN AWARDED TO

igorldepaula

FOR SUCCESSFULLY COMPLETING

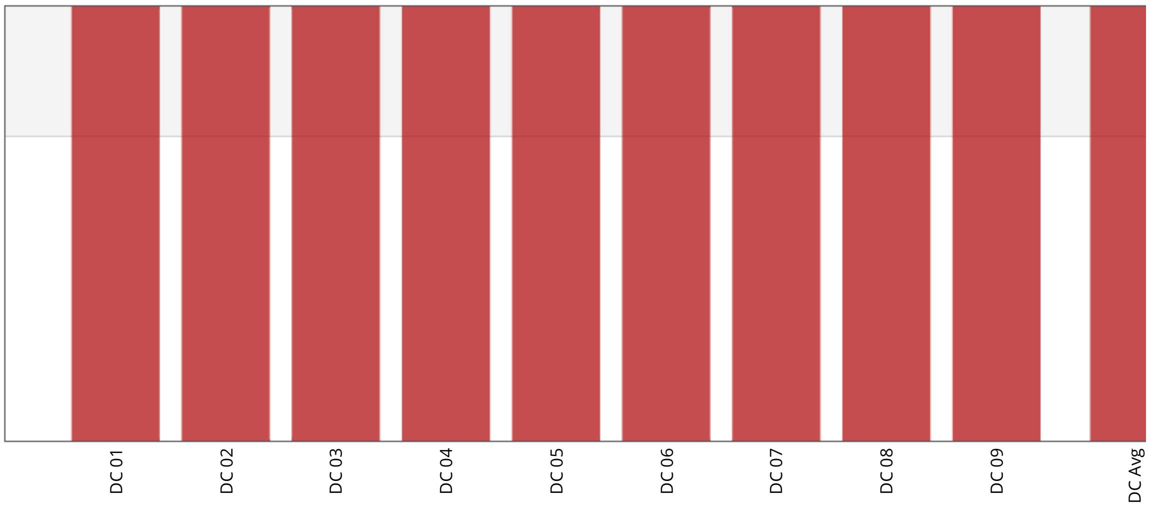
Data Science R Basics Course

DataCamp

Course Progress for Student 'igorldep' (igorldepaula@gmail.com)

Your enrollment: Audit track

You are enrolled in the audit track for this course. The audit track does not include a certificate.



Introduction and Welcome

Introduction and Welcome

No problem scores in this section

0.1 Important Pre-Course Survey

No problem scores in this section

Section 1: R Basics, Functions, and Data Types

Section 1 Overview

No problem scores in this section

1.1 Motivation

No problem scores in this section

1.2 R Basics (4.5/4.5) 100%

Assessment

Problem Scores: 4.5/4.5

1.3 Data Types (6/6) 100%

Assessment

Problem Scores: 6/6

Section 2: Vectors, Sorting

Section 2 Overview

No problem scores in this section

2.1 Vectors (12/12) 100%

Assessment

Problem Scores: 12/12

2.2 Sorting (8/8) 100%

Assessment

Problem Scores: 8/8

2.3 Vector Arithmetic (3/3) 100%

Assessment

Problem Scores: 3/3

Section 3: Indexing, Data Wrangling, Plots

Section 3 Overview

No problem scores in this section

3.1 Indexing (8/8) 100%

Assessment

Problem Scores: 8/8

3.2 Basic Data Wrangling (9/9) 100%

Assessment

Problem Scores: 9/9

3.3 Basic Plots (3/3) 100%

Assessment

Problem Scores: 3/3

Section 4: Programming Basics

Section 4 Overview

No problem scores in this section

4.1 Introduction to Programming in R

No problem scores in this section

4.2 Conditionals

No problem scores in this section

4.3 Functions

No problem scores in this section

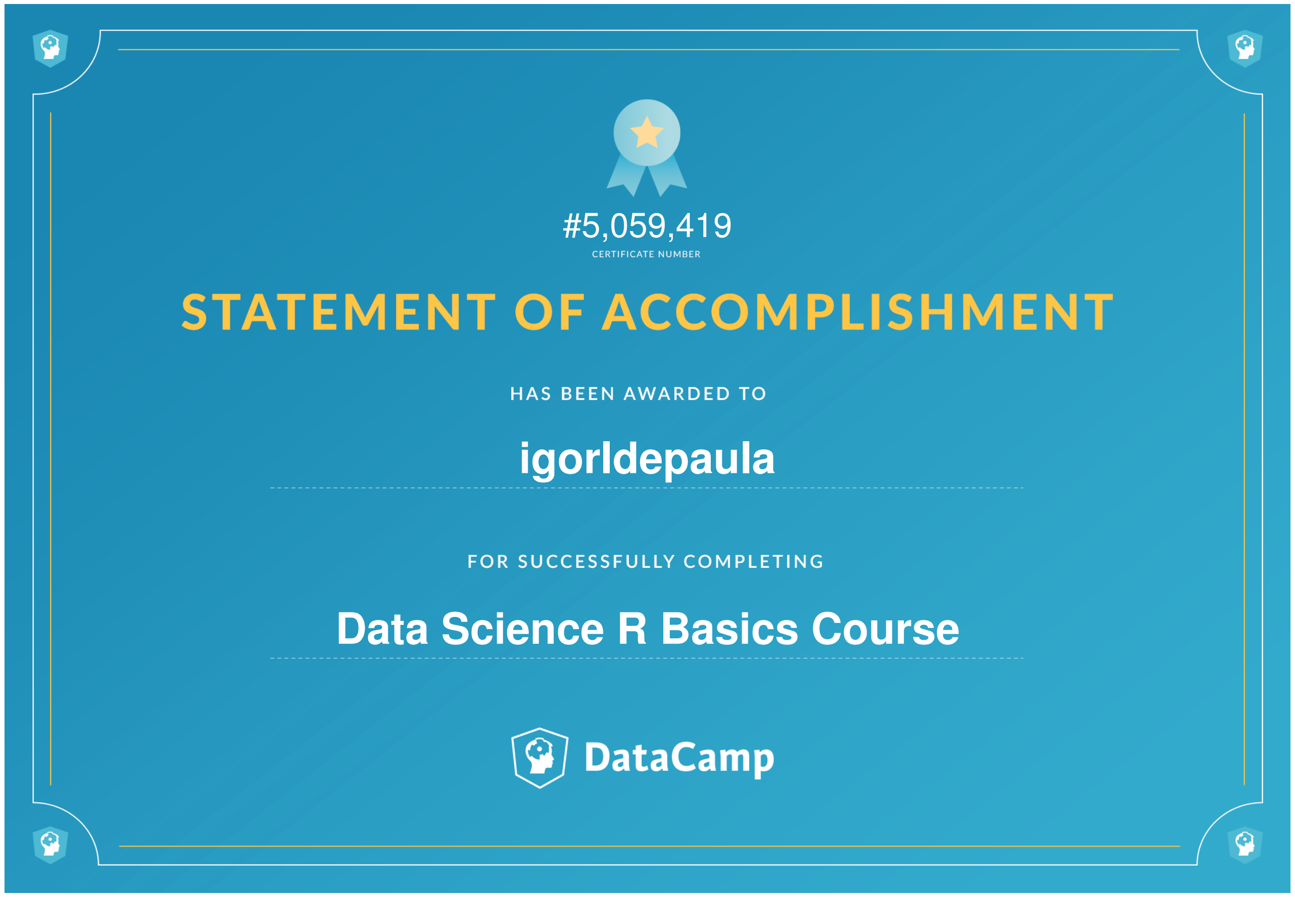
4.4 For Loops (9.5/9.5) 100%

Assessment

Problem Scores: 9.5/9.5

End of Course Survey

No problem scores in this section



#5,059,419

CERTIFICATE NUMBER

STATEMENT OF ACCOMPLISHMENT

HAS BEEN AWARDED TO

igorldepaula

FOR SUCCESSFULLY COMPLETING

Data Science R Basics Course

DataCamp
