Introduction to Data Science

Yohn Jairo

7/9/2019

Data science: It's one of the most exciting and fastest-growing fields out there. Data scientists bring value to all kinds of businesses and organizations, and we can thank data science for many of the technologies that make our lives easier, like web search engines and smartphone personal assistants.

Data scientist extract information from data and use it to create valuable predictions, visualizations, and technologies. To learn from data, we often need to perform billions of computations over large data sets. We do this with the aid of computers, which need to be given a set of instructions. We refer to writing these instructions as programming.

This worked well enough for a program with seven lines of code. As our programming tasks become more complex, however, assigning values to variables will improve our workflow. Using **variables** allows us to store values in computer memory with an associated name that we can use to access the values. For example, let's say we have a variable named human_population that stores the value 7,000,000,000. When we type human_population into the code editor, the computer accesses the value stored in the variable and returns it:

```
math <- 88
chemistry <- 87.66667
writing <- 86
art <- 91.33333
history <- 84
music <- 91
physical_education <- 89.33333
```

Storing values as variables can make them easier to track. We'll illustrate this by writing some code using the variables that we assigned class grade values to.

```
math <- 88
chemistry <- 87.66667
writing <-  86
art <- 91.33333
history <- 84
music <- 91
physical_education <- 89.33333
gpa <- (math + chemistry + writing + art + history + music + physical_education) / 7
```

This solution worked well enough for the small data set we're analyzing, but will not scale well as we begin to work with more data. To prepare to work with larger data sets, we'll work with storage objects that can hold multiple values: **vectors**.

In R, vectors contain a sequence of values that can be assigned to a single variable. For example, we could create a vector, **math_chem**, that contains the math and chemistry final grades:

```
math_chem <- c(math, chemistry)
```

On the previous, we discussed how programming with vectors will allow us to work with large data sets, since we can perform the same operation on all elements of a vector at once.

Let's take a look at how working with vectors can improve data analysis efficiency.

Earlier, we used arithmetic operators to calculate the grade point average:

```
gpa <- (math + chemistry + writing + art + history + music + physical_education) / 7
```

Now that we have stored the grades in a vector, we can calculate the gpa more efficiently. We will do this using one of **R's built-in functions: mean()**. Like the c() function, mean() takes inputs, performs an action, and returns an output. The input to mean() is a vector, and the output is the average of the values contained in the vector.

```
final_scores <- c(math, chemistry, writing, art, history, music, physical_education)
gpa <- mean(final_scores)
```

we calculated the grade point average using the mean() function. Now, let's learn about some additional built-in R functions that we can use to ask more questions about our grades data set:

What was the highest score? What was the lowest score? How many classes did we take? To answer these questions and more, let's introduce some useful R functions.

min(): Takes a vector as input, output is the smallest value in the vector. **max()**: Takes a vector as input, output is the largest value in the vector. **length()**: Takes a vector as input, output is the total number of values in the vector. **sum()**:: Takes a vector as input, output is the sum of all values in the vector.

```
max(final_scores)
```

```
## [1] 91.33333
```

```
min(final_scores)
```

```
## [1] 84
```

```
length(final_scores)
```

```
## [1] 7
```

We've seen how useful working with vectors and functions is for quickly performing calculations in R. In this mission, we'll get a more in-depth look at manipulating vectors for data analysis:

Working with a subset of values in a vector. Assigning names to elements of a vector. Using comparison operators to answer questions about data stored in vectors. Let's continue working with the data on class grades we introduced in the previous mission. We'll begin by investigating how our grades in STEM (science, technology, engineering, and math) classes compare with those in non-STEM classes.

First, we'll need to extract **subsets** of the data stored in our final_scores vector: **STEM classes** and **non-STEM classes**.

We can index vectors to select a subset of the elements they contain. Within a vector, every element has a position. **R is a 1-indexed programming language**, which means that the first element in a vector is assigned a position of one.

```
final_scores <- c(88, 87.66667, 86, 91.33333, 84, 91, 89.33333)
stem_grades <- final_scores[c(1,2)]
non_stem_grades <- final_scores[c(3:7)]

mean(stem_grades)
```

```
## [1] 87.83334
```

```
mean(non_stem_grades)
```

```
## [1] 88.33333
```

we've been working with class grade data; the data has consisted entirely of **numbers**. In R, this data type is referred to as **numeric**. Numeric data may include integer data, or whole numbers (88), and **double** data, or decimals (87.666667).

To display the data type of a vector, we'll use the typeof() function. Let's display the data type of final_scores.

```
typeof(final_scores)
```

```
## [1] "double"
```

In R, "characters" refer to all symbols that are used to make up a language, including letters, special characters like "%", "&", or "$", and numbers. Some functions you'll use to perform calculations, such as min() and max(), will not work on character data.

To create a vector containing character data, you can use the c() function as you did when you added grades to the final_scores vector. However, you need to specify the elements you are including consist of characters by surrounding them with quotation marks (either ” or “”).

To create a vector containing the names of two of your classes, we would write:

```
math_chemistry <- c("math", "chemistry")
typeof(math_chemistry)
```

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

```
class_names <- c("math","chemistry","writing","art","history","music","physical_education")
```

We have now created two vectors:

final_scores, containing your class grades (numeric data). class_names, containing the class names (character data). In R, vectors may have attributes assigned to them. Attributes provide information, such as names, about the values stored in the vector. To assign names to vector elements, we can use the **names()** function.

To illustrate how to use the names() function, let's create two vectors.

```
class_names <- c("math", "chemistry", "writing", "art", "history", "music", "physical_education")
final_scores <- c(88, 87.66667, 86, 91.33333, 84, 91, 89.33333)
names(final_scores) <- class_names

final_scores
```

```
##               math          chemistry            writing
##           88.00000           87.66667           86.00000
##                art            history              music
##           91.33333           84.00000           91.00000
## physical_education
##           89.33333
```

```
liberal_arts <- final_scores[c("writing","history")]
fine_arts <- final_scores[c("art","music")]
mean(liberal_arts)
```

```
## [1] 85
```

```
mean(fine_arts)
```

```
## [1] 91.16666
```

We have now indexed the final_scores vector to visually compare averages of the grades in a few subsets of classes:

STEM vs. non-STEM Liberal arts vs. fine arts This method worked well with the small data set we've been working with, but there are other methods that will scale more effectively as our data sets grow in size and complexity.

Instead visually comparing pairs of grades, we can write code using comparison operators to compare values based on specific conditions, such as "greater than," "less than," or "equal to."

When we compare two values using a comparison operator, if the values satisfy the condition, the R interpreter will return TRUE. If the values do not satisfy the condition, the R interpreter will return FALSE.

```
liberal_arts <- final_scores[c("writing", "history")]
fine_arts <- final_scores[c("art", "music")]
mean(liberal_arts) > mean(fine_arts)
```

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