Near the end of the school year, students are eager to know their grades. In many cases, teachers don't post scores until the school year ends. As a result, many students try to figure out their final grades based on their exam, project & homework performance.

Let's say you're a high school senior looking to calculate your grade point average. Grade point average represents the average value of the accumulated final scores earned in all classes. You are currently taking seven classes. You are graded on exams, homework and projects, each equally weighted.

In this mission, we'll learn the basics of R by writing a program that calculates your grade point average. In this mission, grade point average will be measured on a 0 to 100 scale.

Let's say we wanted to see how well you performed in math class. You scored a 92 on exams, 87 on homework and 85 on projects. To calculate the average math score, we could do the following:

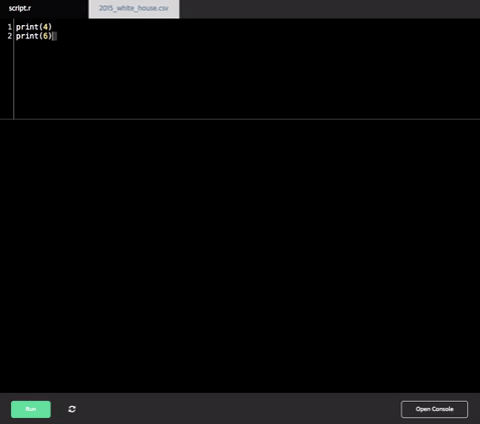


(92 + 87 + 85)/3

We *could* perform tasks like calculating the average, by hand. However, if we had to calculate the averages for a thousand students, hand calculations won't be an effective use of our time.

To ask a computer to carry out calculations will require us to learn how to program. **Programming** involves expressing what we want the computer to do, in the form of a computer program.

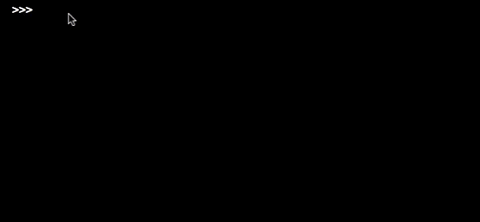
With each step in this course, we'll teach a concept and then you'll have a coding exercise to complete. The **R Interpreter** on our Dataquest servers, will interpret your code and provide feedback on how you did. You can write your code on the screen to your right. This area is called a *script*. When finished, press **Run** to check the code:



After running the code, our servers will check the code against our solutions. If the result of your code doesn't match our solution, the server will display an error message.

In addition to running your code, you can also write code in the **console**:

While you can write your code in a script to check your answer, think of the console as a sandbox where you can test or play with your code. This area does not have any answer checking. You can only write one line at a time. Press **enter** on your keypad to run these lines:



In later missions, we'll teach you how to install an R interpreter (R Studio) so you can practice programming on your own machine.

**Performing Calculations**

To start off, R can perform the actions of a basic calculator. Let's calculate the average math score:



(92 + 87 + 85)/3

These calculations are also called *expressions*. In R, once we finish writing out our expressions, we can show the results of our calculation using a print() statement. The print() statement shows the results of the calculation in between (). Like this:



print((92 + 87 + 85)/3)

If we click **Run**, your result will display:



[1] 88

We'll dive into what [1] means later in this course. In our displayed result, both the calculation and the print() statement have pairs of matching parentheses. To make this clearer, here's the same calculation:



print(

(92 + 87 + 85)/3

)

Here's the same expression, without the closing parenthesis:



print((92 + 87 + 85)/3

Instead of printing the result, the interpreter will return:

Error in parse(text = x, srcfile = src): :7:0: unexpected end of input

The interpreter is telling us we made an error. An error stops our code from running. The interpreter will also display what the error is.

The text unexpected end of input indicates to us that the input to the R interpreter (our code) was missing a closing parentheses ). Every starting parenthesis needs a closing parenthesis.

In this example, we've calculated your average math score. Let's calculate your average chemistry score & display the results using the print() statement.

instructions

* The following values are your exam, homework and project scores, respectively in your chemistry class:

90,81,92

* Calculate the average score in chemistry class.

In the previous exercise, we used the print() statement to display the results.

To run our code, the R interpreter:

1. Scanned and looked for syntax errors.
2. Interpreted and ran each line of code, from top to bottom.
3. Exited when the last line of code is run.

To understand the sequential way R code is interpreted, let's look the math and chemistry score calculations. What happens if we run both calculations on separate lines?



print((92 + 87 + 85)/3)

print((90 + 81 + 92)/3)

Running this code, the R interpreter will display:



[1] 88

[1] 87.66667

Does R always display two lines if we write two lines of code? What if we break up our code into multiple lines?



print(

(92 + 87 + 85)/3

)

print(

(90 + 81 + 92)/4

)

The R interpreter will still display the same values:



[1] 88

[1] 87.66667

Notice how R interprets our code. Each print statement corresponds to it's own line in the result:

Now that we've calculated the math and chemistry scores, let's calculate scores for writing and art class!

instructions

* Here are your exam, homework and project scores for writing class, respectively:

84,95,79

* Here are your exam, homework and project scores for art class, respectively::

95,86,93

* Calculate the average score for each class.
* Print the results for each calculation.

In the previous exercise, you calculated your average class scores using the + and /. The + or / are called **arithmetic operators**. Arithmetic operators are used to carry out mathematical operations. In the following diagram, here is a list of the most common operators and a simple expression using each operator:

For those who are unfamiliar with **exponentiation**, exponentiation will raise a number to its power, using the \*\* or ^ operator.

Exponentiation is a way of multiplying a number by itself a specific number of times. If we wanted to multiply the value 4 by itself 3 times, this would look like the following using the \* operator:



4 \* 4 \* 4

Unfortunately, This quickly gets cumbersome for larger exponents. For example, if we wanted to multiply the value 4 by itself 20 times, we'd need to type out many values. Instead, we can express the calculation as an exponent:



4\*\*20

Running 4\*\*3 will return:



[1] 64

Let's use arithmetic operators to calculate the scores for the rest of your classes.

instructions

* So far, you've calculated your average scores for four classes: math, writing, art, chemistry. Let's use arithmetic operators to calculate your average scores for the last three classes: history, music and physical education.
* Here are the corresponding exam, homework and project scores for each subject:

history: 77, 85, 90

music: 92, 90, 91

physical education: 85, 88, 95

* For each class, calculate your average score. Write each expression on a separate line and print the results.

In the previous sections, we learned how to use arithmetic operators to calculate the average scores for each class. Let's return to our average calculation for math:



print(

(92 + 87 + 85)/3

)

What if we deleted the parenthesis surrounding 92 + 87 + 85?



print(

92 + 87 + 85/3

)

This will display:



207.333

By deleting the parentheses surrounding 92 + 87 + 85, the R interpreter makes a different calculation. When using multiple operators, there are rules that determine the order in which calculations are performed.

A simple way to determine the order of your calculations, is to throw a *parenthesis* around the calculation you want performed first. This is useful for a more complex calculation like this:



print(

(92 + 87 + 85 + 67 + 92 + 84)/6 - (77 + 90 + 98)/3

)

In this scenario, we've thrown a parentheses around the 92 + 87 + 85 + 67 + 92 + 84 and 77 + 90 + 98. We're telling the interpreter to execute the addition operator before executing the division.

The R interpreter follows the order of operations rules in mathematics. An easy way to remember this is [PEMDAS](https://en.wikipedia.org/wiki/Order_of_operations):

* **P**arentheses
* **E**xponent
* **M**ultiplication or **D**ivision
* **A**ddition or **S**ubtraction

Let's take a look at an example without the parentheses. For 92 + 87 + 85/3, the R interpreter will calculate the expression in this sequence:

When you don't include a parentheses surrounding 92 + 87 + 85, based on PEMDAS, the R interpreter will calculate the *division* operator first.

Now, let's re-add the parentheses onto our expression. For (1150 + 1550 + 1420)/3. The R interpreter will calculate the expression in a difference sequence:

Now that we have our final scores for each class, let's calculate your overall average while keeping PEMDAS in mind.

instructions

* Write a single expression that calculates the average of the following values:

88,87.66667,86,91.33333,84,91,89.33333

* On the same expression, subtract this average from your math score 88.
* Print the entire expression.

In the previous exercises, we made multiple calculations using operators. Later on, when we're writing hundreds of lines of code, it's good programming practice to *organize* our code. We can organize our code by inserting **comments**. Comments are notes that help people - including yourself- understand the code. The R interpreter recognizes comments and treats them as plain text and won't attempt to execute them. There are two main types of comments we can add to our code:

* inline comment
* single-line comment

**inline comment**

An inline comment is useful whenever we want to annotate, or add more detail to, a specific statement. To add an inline comment at the end of a statement, start with the hash character (#) and then add the comment:



print(

(92 + 87 + 85)/3 # Finding the math score

)

While we don't need to add a space after the hash character (#), this is considered good style and makes our comments cleaner and easier to read.

**single-line comment**

A single-line comment spans the full line and is useful when we want to separate our code into sections. To specify that we want a line of text to be treated as a comment, start the line with the hash character (#):



# Here, we're finding the average of our scores. Then, subtracting this average from the math score.

print(

88 - ((88 + 87.66667 + 86 + 91.33333 + 84 + 91 + 89.33333)/7)

)

Let's add comments to our code!

instructions

* Add a comment to the code from the previous exercise.

Using R as a "calculator" is useful. However, a more robust approach would be to store these values for later use. This process of storing values is called **variable assignment**. A **variable** in R, is like a named storage unit that can hold values.

The process of assigning a variable requires two steps:

1. Naming the variable.
2. Assigning the value to the name using <-.

When naming a variable, there are a few rules you must follow:

* A variable name consists of letters, numbers, a dot, or an underline.
* We can begin a variable with a letter or a dot. If it's a dot, then we cannot follow it with a number.
* We *cannot* begin a variable with a number.
* No special characters allowed.

For more detail, here is a table detailing what variable names are allowed and which are not:

Let's return to our math score calculation: (92 + 87 + 85)/3, the result of this calculation is 88. To store 88 in a variable called math:



math <- 88

And then if we tried to print() math, like this:



print(math)

This would display:

[1] 88

Variables, not only can hold the result of our calculation, we can also assign the value of an expression:



math <- (92 + 87 + 85)/3

And then if we tried to print() math, like this:



print(math)

This would display the same result as example 1:

[1] 88

We've stored our math grade in a variable. Let's store our other scores in variables.

instructions

* Store the following values in their corresponding variable names. Use the names listed as the variable names:
  + chemistry: 87.66667
  + writing: 86
  + art: 91.33333
  + history: 84
  + music: 91
  + physical\_education: 89.33333

Now that we've stored your grades for each class in a variable, we can now use these variables to find the grade point average.

Let's look at our math and chemistry scores:



math <- 88

chemistry <- 87.66667

When performing a calculation, the variables and values are treated *the same*. Using our math and chemistry variables, 88 + 90 is the same as math + 90. When performing calculations using variables, the PEMDAS rule still applies. t If we wanted to see how much better you did in math, than chemistry, we can use the - arithmetic operator to find the difference:



math <- 88

chemistry <- 87.66667

​

print(math - chemistry)

This would display:



[1] 0.33333

If we wanted to find the average score between math and chemistry, we can use the +,/,() operators on the two variables:



(math + chemistry)/2

This would display:



[1] 87.83334

After we make these calculations, we can also store the result of these expressions in a variable. If we wanted to store the average of math and chemistry in a variable called average, it would look like this:



average <- (math + chemistry)/2

Displaying the average would return the same value 87.83334.

Let's calculate your grade point average using variables!

instructions

* Calculate your grade point average using the following variables:
  + math <- 88
  + chemistry <- 87.66667
  + writing <- 86
  + art <- 91.33333
  + history <- 84
  + music <- 91
  + physical\_education <- 89.33333
* Store this in gpa.
* Subtract your gpa from history to see if history is below the average. Store this difference in history\_difference.

From our previous example, calculating your grade point average using variables is useful. However, in data science, we often work with thousands of data points. If you had the score of each individual homework assignment, exam or project for each class, our dataset would get large. Returning to our math, chemistry example, let's look at the current variables:

Rather than store these two values in two variables, we need a storage unit that can store *multiple* values. In R, we can use a **vector** to store these values. A vector is a storage container that can store a sequence of values. We can then name a vector using a variable. Like this:

To create a vector, you'll be using c(). In R, c() is known as a **function**. Similar to the print() statement, the c() function takes in multiple inputs and stores these values in one place. The c() function *doesn't* perform any arithmetic operation on the values, it just stores those values. You can read more about the c() function [here](https://cran.r-project.org/doc/manuals/r-release/R-intro.html#The-concatenation-function-c_0028_0029-with-arrays).

Here are the steps to creating a vector:

1. Identify the values you want to store in a vector and place these values within the c() function. Separate these values using a comma(,).
2. Assign the vector to a name of your choice using <-.

Let's look at an example of how you create a vector using the c() function and an example of an improper way to create a vector.

Let's create a vector that contains your math and chemistry scores. The math score was 88 and the chemistry score was 87.66667.



math\_chemistry <- c(88,87.66667)

We could also create the vector using your variable names as well:



math\_chemistry <- c(math,chemistry)

If we were to print(math\_chemistry), it would look like this:



[1] 88.00000 87.66667

On the other hand, if we tried to store a sequence of values, like this:



math\_chemistry <- 88, 87.66667

The R interpreter will only try to assign 88 to math\_chemistry but will not be able to interpret the comma after 88:

Error: unexpected ',' in "math\_chemistry <- 88,"

Let's store our final scores in a vector!

instructions

* Create a vector containing the score of each class using the variable names:

math <- 88

chemistry <- 87.66667

writing <- 86

art <- 91.33333

history <- 84

music <- 91

physical\_education <- 89.33333

* Store this value in final\_scores.

Now that we've stored your grades in a vector, we can calculate the grade point average. In a previous exercise, you used an arithmetic operator to calculate your grade point average:



(88 + 87.66667 + 86 + 91.33333 + 84 + 91 + 89.33333)/7

While this solution works, this solution isn't scalable. Now that you created a vector, we have an easier way of calculating the grade point average.

To calculate the grade point average using a vector, use the mean() function. The mean() function will take an input(the vector) and calculate the average of that input. The interpreter will then display the result.

Let's apply the mean() function to our math\_chemistry vector:



math\_chemistry <- c(88,87.66667)

​

mean(math\_chemistry)

This would return:



[1] 87.83334

We can then store the result of mean(math\_chemistry) in a variable for later use:



average\_score <- mean(math\_chemistry)

Let's apply the mean() function on your final grades vector!

instructions

* Use the mean() function to calculate your grade point average. Store this in gpa.

In the last screen, you calculated your final grade using the mean() function and a vector. In data science, there are always multiple questions you can answer with your data.

Let's dig deeper into our final\_grades vector and ask it a few more questions:

* What was the highest score?
* What was the lowest score?
* How many classes did you take?

To answer these questions, let's introduce a few more functions you can use that would be useful for you:

* min(): Finding the smallest value within the vector
* max(): Finding the largest value within the vector
* length(): Finding the total number of values the vector holds
* sum():: Taking the sum of all the values in the vector( *Note: Will not be used in exercise.*)

Applying these functions on a vector, is the same as how you would use the mean() function. To find the max score in our math\_chemistry vector, we'll apply the max() function on this vector:



math\_chemistry <- c(88,87.66667)

​

max(math\_chemistry)

This would display:

[1] 88

Let's answer a few more questions about your grades!

instructions

* Use the max() function to get your highest score and store this in highest\_score.
* Use the min() function to get the lowest score and store in lowest\_score.
* Use the length() function to get the total number of classes and store in num\_classes.

In this mission, we learned the basics of variables, arithmetic operators and got a nice introduction to vectors. In the next mission, we'll dive deeper into vectors and data types. We'll learn how to perform arithmetic operations between *vectors*:

We'll also learn how to add conditions into our vectors, so they can better answer our questions. Click the **Finish** button below to continue to the next mission.