In the last mission, you wrote a program to calculate your overall grade point average. A program is a set of coded computer instructions you write telling the computer to perform a specific task. In this mission, we'll dive deeper into vectors. We'll write a program that figures out which class you're struggling in. Then, we'll help your friend Johnny calculate his scores.

From the previous mission, you learned the min() function. Let's try applying the min() function to our final\_scores vector:



print(min(final\_scores))

This displays:



[1] 84

Calling a min() on our final\_scores vector doesn't provide us with the *name* of the class like math or chemistry. It still prints our score. We'd like the R interpreter to print both name *and* score. Like this:



math

88

*Only* using min() won't work as our final solution. We'll return to min() later on in this mission. In the meantime, let's find another method that can print both name and the score.

One method, is to use **indexing**. Indexing means to select a subset of values for use. Within a vector, every value has a position in the vector. R is a **1-indexed** programming language. This means, that the first value has a position of 1.

We'll go over what names() is in a later screen. Let's use our math\_chemistry vector and return the value in the *second* position. In order to return a specific value from our vector, follow these steps:

When you index into a vector, the interpreter will display both the value *and* the name.

Let's index into our final\_scores vector!

instructions

* Index into the final\_scores vector and return the score for the 3rd class. Print results.
* Store this result in third.

In the previous section, you learned how to index into a vector to return a specific score. When we're indexing into final\_scores, we're returning a *number*:



final\_scores <- c(88, 87.66667, 86, 91.33333, 84, 91, 89.33333)

third <- final\_scores[3]

print(third)

This displays:



[1] 86

Whenever you store a number or return a number, this value is a **numeric** data type. Numeric types are both whole numbers & decimal numbers(88,87.666667). However, not all values stored in R are of the numeric data type. We'll introduce a different data type later on in this mission, called the *character* type. First, let's figure out how check our data type.

To display the data type of a vector, you'll use the [class()](https://stat.ethz.ch/R-manual/R-devel/library/base/html/class.html) function. The class() function operates similar to the mean() function. Just place the vector in between the parentheses.

Let's display the data type of math\_chemistry.



math\_chemistry <- c(88,87.66667)

Let's pass math\_chemistry through the class() function:



class(math\_chemistry)

This will display:

[1] "numeric"

The displayed result is "numeric". We'll go over what the quotations(" ") are in the next screen. Within the numeric data type, there is more complexity between different types of numbers, which you can read about [here](http://uc-r.github.io/integer_double/) if you'd like. Let's put the class() function to use!

instructions

* Use the class() function to look up the data type for final\_scores:

final\_scores <- c(88, 87.66667, 86, 91.33333, 84, 91, 89.33333)

In the previous screen, we introduced numeric data types. However, we're not looking to return a number 86, we're looking to return a class name (math, chemistry etc). In addition, asking a question like "what was my score in position 3?", doesn't have much real-world utility.

As a result, there are a few more steps we need to take to return *both* a final score and class name. To return a name, let's learn about the character data type. The character data type is a common non-numerical data type. To represent the character data type, we'll surround our text with double quotes ("). "math" would be an example of a character data type.



>>> print("math")

math

Unlike variable names, you can include special characters within character types. Similar to assigning a number to a variable, you can also assign a character to a variable:



class\_name <- "math"

When you print() a variable with a character type, the interpreter will return the value *with* the quotations:



print(class\_name)

This would return:



"math"

Let's return to our math\_chemistry vector. Within this vector, we have our math score 88 and our chemistry score 87.66667:



math\_chemistry <- c(88,87.66667)

Now, let's create a new vector with the character type:



class\_names <- c("math","chemistry")

Let's double check, just to make sure:



class(class\_names)

This returns:

[1] "character"

A common mistake is to forget to include the " ". Let's see what happens when we forget to use " ":



math\_chemistry <- c(88,87.66667)

Let's try creating a vector of text with math and chemistry, without the quotation wrapper(" "):



class\_names <- c(math, chemistry)

This will return an error:

Error: object 'math' not found

Without quotes surrounding math and chemistry, the R interpreter will treat these two values as variables. This will return an error because we didn't create these variables in this program.

Let's create a vector of all our class names!

instructions

* Create a vector called class\_names of the character data type, in this order:
  + math
  + chemistry
  + writing
  + art
  + history
  + music
  + physical\_education
* Call class() on this vector to check its data type.
* Then, store the data type in type.

Now that you've created a vector of class names of character type, we now have two vectors: final\_scores and class\_names. To allow our program to return math,chemistry etc, let's find a way to *label* our scores in final\_scores with our class\_names. That way, when you display the classes we're struggling with, it'll return *both* the score and the class name.

Vectors have a feature where you can *name* the values within the vector. You'll use the [name()](https://stat.ethz.ch/R-manual/R-devel/library/base/html/names.html) function to assign these values. However, the name() function works differently than the mean() and other functions we've used.

When you look at a vector, you'll only see its values. Let's take math\_chemistry <- c(88,87.66667). However, vectors have an additional feature that can store the *names* of every value. This feature is called an **attribute**. Attributes are labeled values we can attach to our vector.

To access an attribute, use the names() function on your vector: names(math\_chemistry). This function that accesses the attribute of a vector is called an **accessor** function. By default, attributes are empty, so calling names(math\_chemistry) will return an NA:

While functions like mean() and min() accept input values and return a new, computed value, functions like print() and names() return information about the value that's passed in.

To name our values in a vector, there are three steps: 1. Create a vector of names. 2. Call the names() function on the original vector (not the vector of names). Store the vector of names in names().

Let's give a class name for each final score!

instructions

* Name the final\_scores vector using the class\_names vector.
* Store final\_scores in named\_final\_scores, like this: named\_final\_scores <- final\_scores.

Now that we've named our final\_scores vector, we can use a *named index* to answer the question "what was my score in math class?"

Before we use a named index, there is one key requirement when indexing by name:

**To index by name, you must have already named the values in your vector using the names() function.**

By default, attributes are empty, so calling names(vector) will be NA. You cannot index a vector by name, if the name is NA. This is why you need to name your values first.

Indexing by name is similar to indexing by position, except we're accessing the value by *name*. To answer our question, "what was my score in math class?", indexing "math" into your vector will answer your question!

Let's index our final scores vector by name!

instructions

Index into final\_scores and perform the following:

* Get the score for history and store it in history. Print results.
* Get the score for art and store it in art. Print results.
* Get the score for music and store it in music. Print the results.

In the last screen, we used indexing to answer a basic question: "what was my score in math class?" To figure out what class you're struggling with, let's take a step closer by asking a different question: "Did I score higher in math or in chemistry?"

While you could answer this question by indexing with math and chemistry, final\_scores["math"] and final\_scores["chemistry"], and then visually comparing the result of 88 and 87.6667, this method is not robust if we had thousands of data points.

Rather than eyeball the comparison, you can represent this comparison in code by using a comparison operator. A comparison operator compares two values based on a specific condition. A comparison operator will create a condition that compares 88 against 87.6667. There are multiple conditions you can use, such as greater than or less than. See table below for most common conditions.

After you use a comparison operator, if the 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. TRUE and FALSE are not numeric or character data types, they're called **boolean** or **logical** data types. The logical data type can only take on two values, TRUE or FALSE.

Let's compare 88 against 87.6667 using all the comparison operators:

Let's use comparison operators to compare our class scores!

instructions

* Check if your history score is greater than your math score. Store this in history\_math
* Check if your writing score is less than or equal to your art score. Store this in writing\_art
* Check if your music score is equal to your chemistry score. Store this in music\_chem

In the previous section, we compared 88 against 87.6667. In this scenario, we're comparing a single value against another single value. Returning to our original question, if we wanted to display the class you're struggling in, one method would be to compare every value against each other:

final\_scores["math"] > final\_scores["chemistry"]

final\_scores["math"] > final\_scores["writing"]

final\_scores["math"] > final\_scores["art"]

final\_scores["math"] > final\_scores["history"].............

Going through each class you took and comparing every value against each other *can* work. However, you're going to have to write 21 different expressions to make this calculation.

Luckily, comparison operators do not limit you to comparing single values against single values. In fact, you can compare a single value against an *entire* vector. Let's compare your math score against your final\_scores vector and see which classes math scored higher than.

Writing this comparison, would look like this:



final\_scores["math"] > final\_scores

This displays:



math chemistry writing art history music

FALSE TRUE TRUE FALSE TRUE FALSE physical\_education

FALSE

Let's dig into how this comparison works. Here's the comparison in visual form:

The single value will then be compared against *every* value in the vector. The reason behind this is the *recycling rule*, which we'll dive into, later in this mission.

Let's look at a visual diagram to better solidify this concept.

Let's store this expression in a vector:



logical\_vector <- (final\_scores["math"] > final\_scores)

Recall, the operations within the parentheses will be performed first. In this case, this is the comparison. The logical\_vector then, would be a vector of boolean values:



logical\_vector <- c(FALSE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE)

Let's create a boolean vector to figure out which classes we're struggling with!

instructions

* Use the min() function to display the lowest score in final\_scores. Store this in lowest\_score.
* Find which values in final\_scores *equals to* the lowest\_score. Store this in lowest\_logical.

Now that we've created a vector that tells us whether each value is the lowest score, the problem we have is that this is in logical form. A vector of TRUE or FALSE values doesn't explicitly tell us which classes we're struggling with.

In the previous exercises, you learned how to index by *position* and by *name*. Let's introduce a new type of indexing called **logical indexing**. Logical indexing will help us figure out which classes you're struggling with.

Logical indexing will check each value of your target vector, against the corresponding value in the logical vector. If the corresponding value is TRUE, then the resulting slice will contain that value. If the corresponding value is FALSE, the resulting slice *will not* contain that value.

In the previous section, we wanted to compare our math score against our score vector using the *greater than* operator. We made the following comparison final\_scores["math"] > final\_scores to create the logical vector: c(FALSE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE).

If we were to index into final\_scores using our logical vector, this would tell us *explicitly* the classes and the scores that are lower than math:

In review, here are the steps we took over the last two screens to display the classes math scored higher than:

Let's use logical indexing to figure out which class you're struggling with!

instructions

* From the previous section, you created the following logical vector:

lowest\_logical <- lowest\_score == final\_scores

* Use lowest\_logical to index into final\_scores. Store this in lowest\_class.
* Print your variable.

You've figured out the class you're struggling with! As you've been writing this program, your friends have expressed interest in using your program to calculate *their* grades.

Your friend Johnny, has given you all his exam, homework and project scores for the same seven courses. However, Johnny is a bit disorganized, so you'll need to re-organize the scores to make the calculation. Here are Johnny's scores:



tests <- c(76, 89, 78, 88, 79, 93, 89)

homework <- c(85, 90, 88, 79, 88, 95, 74)

projects <- c(77, 93, 87, 90, 77, 82, 80)

Johnny first, wants you to calculate the final scores for each class. Assuming that each value corresponds to the same class, we'll use **vector arithmetic** to make these calculations.

Vector arithmetic is similar to arithmetic in your first mission. However, we're now performing these operations on vectors. Vector arithmetic is performed member-by-member. This means, that the operation will be performed between each values, *by position*.

If we took Johnny's tests & homework scores vectors, added them together and stored them in sum:



tests <- c(76, 89, 78, 88, 79, 93, 89)

homework <- c(85, 90, 88, 79, 88, 95, 74)

sum <- tests + homework

The operation and the resulting vector would look like:

We've done one addition together. Now let's calculate your friend Johnny's final scores for each class!

instructions

* Here are the following vectors we have:

tests <- c(76, 89, 78, 88, 79, 93, 89)

homework <- c(85, 90, 88, 79, 88, 95, 74)

sum <- tests + homework

projects <- c(77, 93, 87, 90, 77, 82, 80)

* Create a johnny\_scores vector by adding the projects vector to sum and dividing the resulting vector by 3.
* Use the mean() function on johnny\_scores to calculate Johnny's overall score. Store this in johnny\_overall.

In the previous screen, you calculated Johnny's scores by performing vector arithmetic between his scores. In that scenario, all the vectors had the *same* length. However, what happens if Johnny forgets to give us a value. So instead of:

He forgets to give us a homework value:

Earlier in this mission, we learned when comparing a single value with an entire vector, the R interpreter compares the single value against every value in the vector. Whenever there's a mismatch in the length of 2 objects that are compared, the shorter object is recycled (or repeated) until it matches the longer one.

When we perform an arithmetic operation on two vectors that have *different* lengths, the R interpreter will also reuse the values of the shorter vector.

This recycling behavior is called the **recycling rule**. The recycling rule states that when performing an operation between two vectors of unequal length, the R interpreter will automatically *recycle* the shorter one, until it's long enough to match the longer one.

Here's what happening when we employ the recycling rule. Let's shorten our homework vector to only two values:



tests <- c(76, 89, 78, 88, 79, 93, 89)

homework <- c(85, 90)

And then we'll add tests with homework to see the recycling rule in action. In the diagram, we've color coded the values in homework to show how the values recycle:

The R interpreter will see that the homework vector is shorter than the tests vector. As a result, it'll automatically *recycle* the homework values, starting with the first element:

Once the first value is used, the interpreter will look at the next value:

And then, it'll keep recycling the values, in this pattern, until it matches the length of the longer vector:

Once the vector lengths match, the R interpreter will perform the specified arithmetic operation.

In the section where you compared final\_scores["math"] > final\_scores, you compared a single value (math) against the entire vector. In this scenario, the R interpreter used the recycling rule by re-using the single value, until it matched the length of final\_scores.

Let's see the vector recycling rule in action!

instructions

* Here are Johnny's tests, homework and project scores. We've removed the last four values in tests:

tests <- c(76, 89, 78)

homework <- c(85, 90, 88, 79, 88, 95, 74)

projects <- c(77, 93, 87, 90, 77, 82, 80)

* Add tests and homework together and store this in recycling.
* Display the results using print(). Notice the warning message.

In the previous section, you learned about the vector recycling rule by adding tests and homework together. Recycling the three test scores(76, 89, 78) to look like c(76, 89, 78, 76, 89, 78, 76) would allow us to perform an operation. However, for Johnny, recycling his scores for three classes, won't give him an accurate score for each class. Johnny didn't score 76, 89, 78, 76 in his last four classes. To fix this, let's *append* Johnny's *real* scores to our shorter vector.

To append data points to our shorter vector, let's look at the steps to appending by appending one value to tests:

In addition, you could use the same method to append *multiple* values. To append another value, follow the same steps, except, add another value to the expression:

We've added two values to the test vector. Let's finish adding the rest of Johnny's scores!

instructions

* Here's the current tests vector:

tests <- c(76, 89, 78)

* Add the following values to the tests vector:

88, 79, 93, 89

* Store this in tests

In the next mission, we'll dive into a real-world university rankings dataset. We'll expand our R capabilities by introducing the two-dimensional version of a vector, matrices:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **world\_rank** | **quality\_of\_education** | **influence** | **broad\_impact** | **patents** |
| Harvard | 1 | 1 | 1 | 1 | 3 |
| Stanford | 2 | 9 | 3 | 4 | 10 |
| MIT | 3 | 3 | 2 | 2 | 1 |
| Cambridge | 4 | 2 | 6 | 13 | 48 |
| Oxford | 5 | 7 | 12 | 9 | 15 |
| Columbia | 6 | 13 | 13 | 12 | 4 |

Click the **Finish** button below to continue to this mission.