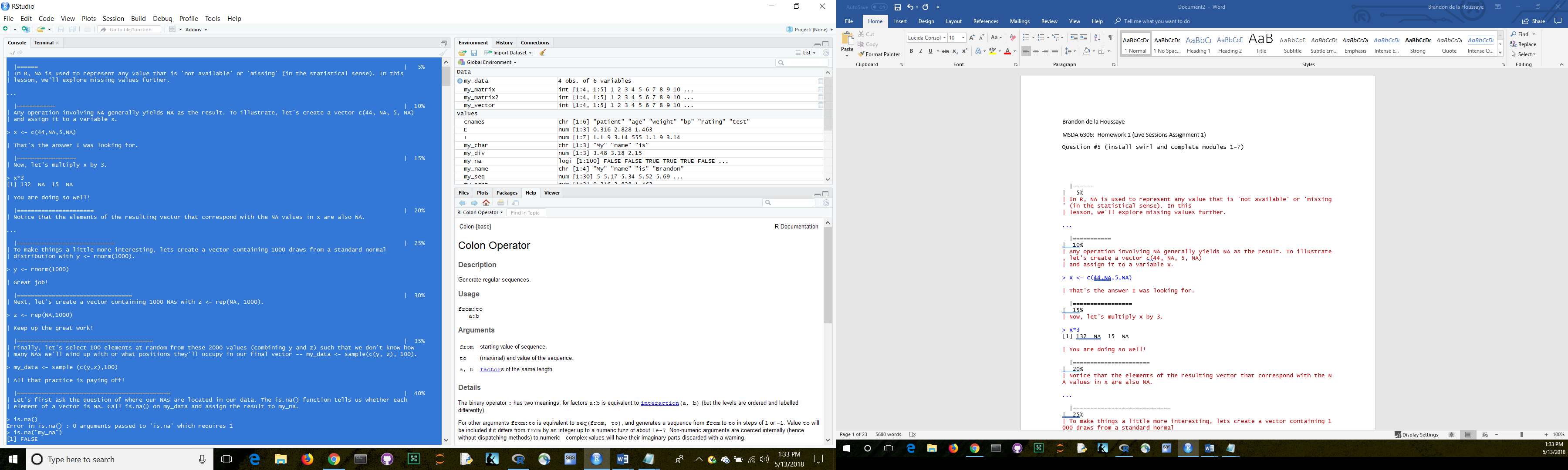
Brandon de la Houssaye

MSDA 6306 (4033): Homework 1 (Live Sessions Assignment 1)

Question #5 (install swirl and complete modules 1-7)



|====== | 5%

| In R, NA is used to represent any value that is 'not available' or 'missing' (in the statistical sense). In this

| lesson, we'll explore missing values further.

...

|=========== | 10%

| Any operation involving NA generally yields NA as the result. To illustrate, let's create a vector c(44, NA, 5, NA)

| and assign it to a variable x.

> x <- c(44,NA,5,NA)

| That's the answer I was looking for.

|================= | 15%

| Now, let's multiply x by 3.

> x\*3

[1] 132 NA 15 NA

| You are doing so well!

|====================== | 20%

| Notice that the elements of the resulting vector that correspond with the NA values in x are also NA.

...

|============================ | 25%

| To make things a little more interesting, lets create a vector containing 1000 draws from a standard normal

| distribution with y <- rnorm(1000).

> y <- rnorm(1000)

| Great job!

|================================= | 30%

| Next, let's create a vector containing 1000 NAs with z <- rep(NA, 1000).

> z <- rep(NA,1000)

| Keep up the great work!

|======================================= | 35%

| Finally, let's select 100 elements at random from these 2000 values (combining y and z) such that we don't know how

| many NAs we'll wind up with or what positions they'll occupy in our final vector -- my\_data <- sample(c(y, z), 100).

> my\_data <- sample (c(y,z),100)

| All that practice is paying off!

|============================================ | 40%

| Let's first ask the question of where our NAs are located in our data. The is.na() function tells us whether each

| element of a vector is NA. Call is.na() on my\_data and assign the result to my\_na.

> is.na()

Error in is.na() : 0 arguments passed to 'is.na' which requires 1

> is.na("my\_na")

[1] FALSE

| Not quite right, but keep trying. Or, type info() for more options.

| Assign the result of is.na(my\_data) to the variable my\_na.

> skip()

| Entering the following correct answer for you...

> my\_na <- is.na(my\_data)

| Great job!

|================================================== | 45%

| Now, print my\_na to see what you came up with.

> my\_na

[1] FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE

[20] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE

[39] TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE TRUE FALSE TRUE FALSE TRUE TRUE FALSE

[58] FALSE FALSE TRUE FALSE TRUE TRUE FALSE TRUE FALSE FALSE TRUE TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE

[77] TRUE FALSE TRUE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE TRUE FALSE FALSE TRUE TRUE TRUE

[96] TRUE FALSE TRUE TRUE FALSE

| You are doing so well!

|======================================================== | 50%

| Everywhere you see a TRUE, you know the corresponding element of my\_data is NA. Likewise, everywhere you see a FALSE,

| you know the corresponding element of my\_data is one of our random draws from the standard normal distribution.

...

|============================================================= | 55%

| In our previous discussion of logical operators, we introduced the `==` operator as a method of testing for equality

| between two objects. So, you might think the expression my\_data == NA yields the same results as is.na(). Give it a

| try.

> my\_data == NA

[1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA

[39] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA

[77] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA

| Great job!

|=================================================================== | 60%

| The reason you got a vector of all NAs is that NA is not really a value, but just a placeholder for a quantity that is

| not available. Therefore the logical expression is incomplete and R has no choice but to return a vector of the same

| length as my\_data that contains all NAs.

...

|======================================================================== | 65%

| Don't worry if that's a little confusing. The key takeaway is to be cautious when using logical expressions anytime

| NAs might creep in, since a single NA value can derail the entire thing.

...

|============================================================================== | 70%

| So, back to the task at hand. Now that we have a vector, my\_na, that has a TRUE for every NA and FALSE for every

| numeric value, we can compute the total number of NAs in our data.

...

|=================================================================================== | 75%

| The trick is to recognize that underneath the surface, R represents TRUE as the number 1 and FALSE as the number 0.

| Therefore, if we take the sum of a bunch of TRUEs and FALSEs, we get the total number of TRUEs.

...

|========================================================================================= | 80%

| Let's give that a try here. Call the sum() function on my\_na to count the total number of TRUEs in my\_na, and thus the

| total number of NAs in my\_data. Don't assign the result to a new variable.

> sum(my\_na)

[1] 44

| You are doing so well!

|============================================================================================== | 85%

| Pretty cool, huh? Finally, let's take a look at the data to convince ourselves that everything 'adds up'. Print

| my\_data to the console.

> my\_data

[1] 1.48605688 -1.24640043 NA NA NA 0.02652878 0.20278642 0.55719288 NA

[10] NA NA -0.17807043 NA NA NA NA -2.50860750 -1.14645456

[19] 1.28717107 -0.07177880 -0.49584205 -0.59152017 0.57436678 -0.44147144 -0.25148609 0.98185316 -0.60598651

[28] NA NA -0.74928901 1.40624862 -0.61643348 -0.29390194 0.86425690 NA NA

[37] NA NA NA -3.03690597 -0.38526851 NA 0.21172315 -0.41533578 -0.36413541

[46] -1.07091890 -0.15629848 NA NA -0.42714086 NA -0.08280260 NA -0.63165741

[55] NA NA 0.53235045 -0.68790147 -0.68132948 NA -0.18712731 NA NA

[64] -0.86815077 NA -0.10607909 -0.50308760 NA NA -0.52913220 0.37086655 -0.23975945

[73] NA NA 1.02454689 -1.52072262 NA 1.26650751 NA NA 3.09893649

[82] -1.24961739 0.69505883 0.64126813 NA 1.07863878 -0.71757993 NA 1.01185497 NA

[91] 0.09358678 -1.01451759 NA NA NA NA 0.43958281 NA NA

[100] -0.41031287

| You are really on a roll!

|==================================================================================================== | 90%

| Now that we've got NAs down pat, let's look at a second type of missing value -- NaN, which stands for 'not a number'.

| To generate NaN, try dividing (using a forward slash) 0 by 0 now.

> 0/0

[1] NaN

| You got it right!

|========================================================================================================= | 95%

| Let's do one more, just for fun. In R, Inf stands for infinity. What happens if you subtract Inf from Inf?

> inf-inf

Error: object 'inf' not found

> Inf - Inf

[1] NaN

| That's correct!

|===============================================================================================================| 100%

| Would you like to receive credit for completing this course on Coursera.org?

1: Yes

2: No

Selection: 2

| That's a job well done!

| You've reached the end of this lesson! Returning to the main menu...

| Please choose a course, or type 0 to exit swirl.

1: R Programming

2: Take me to the swirl course repository!

Selection: 1

| Please choose a lesson, or type 0 to return to course menu.

1: Basic Building Blocks 2: Workspace and Files 3: Sequences of Numbers 4: Vectors

5: Missing Values 6: Subsetting Vectors 7: Matrices and Data Frames 8: Logic

9: Functions 10: lapply and sapply 11: vapply and tapply 12: Looking at Data

13: Simulation 14: Dates and Times 15: Base Graphics

Selection: 6

| | 0%

| In this lesson, we'll see how to extract elements from a vector based on some conditions that we specify.

...

|=== | 3%

| For example, we may only be interested in the first 20 elements of a vector, or only the elements that are not NA, or

| only those that are positive or correspond to a specific variable of interest. By the end of this lesson, you'll know

| how to handle each of these scenarios.

...

|====== | 5%

| I've created for you a vector called x that contains a random ordering of 20 numbers (from a standard normal

| distribution) and 20 NAs. Type x now to see what it looks like.

> x

[1] NA NA -0.86867902 NA NA -0.23893261 NA NA 0.25109672

[10] -0.27582977 0.34398630 NA NA 0.35154977 NA NA NA NA

[19] NA -0.67859122 0.08877186 NA -0.27815706 NA NA -0.99114632 NA

[28] -1.62448744 -0.82246287 1.38536080 -0.31329923 0.23056499 NA NA -0.14378969 -1.23546524

[37] 0.14058996 0.58282334 NA -1.60875462

| Excellent job!

|========= | 8%

| The way you tell R that you want to select some particular elements (i.e. a 'subset') from a vector is by placing an

| 'index vector' in square brackets immediately following the name of the vector.

...

|=========== | 10%

| For a simple example, try x[1:10] to view the first ten elements of x.

> x[1:10]

[1] NA NA -0.8686790 NA NA -0.2389326 NA NA 0.2510967 -0.2758298

| You are doing so well!

|============== | 13%

| Index vectors come in four different flavors -- logical vectors, vectors of positive integers, vectors of negative

| integers, and vectors of character strings -- each of which we'll cover in this lesson.

...

|================= | 15%

| Let's start by indexing with logical vectors. One common scenario when working with real-world data is that we want to

| extract all elements of a vector that are not NA (i.e. missing data). Recall that is.na(x) yields a vector of logical

| values the same length as x, with TRUEs corresponding to NA values in x and FALSEs corresponding to non-NA values in

| x.

...

|==================== | 18%

| What do you think x[is.na(x)] will give you?

1: A vector of TRUEs and FALSEs

2: A vector with no NAs

3: A vector of length 0

4: A vector of all NAs

Selection: 1

| Keep trying!

| Remember that is.na(x) tells us where the NAs are in a vector. So if we subset x based on that, what do you expect to

| happen?

1: A vector of length 0

2: A vector of all NAs

3: A vector of TRUEs and FALSEs

4: A vector with no NAs

Selection: 1

| That's not exactly what I'm looking for. Try again.

| Remember that is.na(x) tells us where the NAs are in a vector. So if we subset x based on that, what do you expect to

| happen?

1: A vector of length 0

2: A vector of TRUEs and FALSEs

3: A vector of all NAs

4: A vector with no NAs

Selection: 3

| You're the best!

|======================= | 21%

| Prove it to yourself by typing x[is.na(x)].

> x[is.na(x)]

[1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA

| Keep up the great work!

|========================== | 23%

| Recall that `!` gives us the negation of a logical expression, so !is.na(x) can be read as 'is not NA'. Therefore, if

| we want to create a vector called y that contains all of the non-NA values from x, we can use y <- x[!is.na(x)]. Give

| it a try.

> y <- x[!is.na(x)]

| You are really on a roll!

|============================ | 26%

| Print y to the console.

> y

[1] -0.86867902 -0.23893261 0.25109672 -0.27582977 0.34398630 0.35154977 -0.67859122 0.08877186 -0.27815706

[10] -0.99114632 -1.62448744 -0.82246287 1.38536080 -0.31329923 0.23056499 -0.14378969 -1.23546524 0.14058996

[19] 0.58282334 -1.60875462

| That's a job well done!

|=============================== | 28%

| Now that we've isolated the non-missing values of x and put them in y, we can subset y as we please.

...

|================================== | 31%

| Recall that the expression y > 0 will give us a vector of logical values the same length as y, with TRUEs

| corresponding to values of y that are greater than zero and FALSEs corresponding to values of y that are less than or

| equal to zero. What do you think y[y > 0] will give you?

1: A vector of TRUEs and FALSEs

2: A vector of all the positive elements of y

3: A vector of all NAs

4: A vector of length 0

5: A vector of all the negative elements of y

Selection: 2

| You are really on a roll!

|===================================== | 33%

| Type y[y > 0] to see that we get all of the positive elements of y, which are also the positive elements of our

| original vector x.

> y[y>0]

[1] 0.25109672 0.34398630 0.35154977 0.08877186 1.38536080 0.23056499 0.14058996 0.58282334

| All that practice is paying off!

|======================================== | 36%

| You might wonder why we didn't just start with x[x > 0] to isolate the positive elements of x. Try that now to see

| why.

> x[x>0]

[1] NA NA NA NA NA NA 0.25109672 0.34398630 NA NA

[11] 0.35154977 NA NA NA NA NA 0.08877186 NA NA NA

[21] NA 1.38536080 0.23056499 NA NA 0.14058996 0.58282334 NA

| Perseverance, that's the answer.

|=========================================== | 38%

| Since NA is not a value, but rather a placeholder for an unknown quantity, the expression NA > 0 evaluates to NA.

| Hence we get a bunch of NAs mixed in with our positive numbers when we do this.

...

|============================================== | 41%

| Combining our knowledge of logical operators with our new knowledge of subsetting, we could do this -- x[!is.na(x) & x

| > 0]. Try it out.

> x[!is.na(x) & x > 0]

[1] 0.25109672 0.34398630 0.35154977 0.08877186 1.38536080 0.23056499 0.14058996 0.58282334

| All that hard work is paying off!

|================================================ | 44%

| In this case, we request only values of x that are both non-missing AND greater than zero.

...

|=================================================== | 46%

| I've already shown you how to subset just the first ten values of x using x[1:10]. In this case, we're providing a

| vector of positive integers inside of the square brackets, which tells R to return only the elements of x numbered 1

| through 10.

...

|====================================================== | 49%

| Many programming languages use what's called 'zero-based indexing', which means that the first element of a vector is

| considered element 0. R uses 'one-based indexing', which (you guessed it!) means the first element of a vector is

| considered element 1.

...

|========================================================= | 51%

| Can you figure out how we'd subset the 3rd, 5th, and 7th elements of x? Hint -- Use the c() function to specify the

| element numbers as a numeric vector.

> skip()

| Entering the following correct answer for you...

> x[c(3, 5, 7)]

[1] -0.868679 NA NA

| That's a job well done!

|============================================================ | 54%

| It's important that when using integer vectors to subset our vector x, we stick with the set of indexes {1, 2, ...,

| 40} since x only has 40 elements. What happens if we ask for the zeroth element of x (i.e. x[0])? Give it a try.

> x[0]

numeric(0)

| You nailed it! Good job!

|=============================================================== | 56%

| As you might expect, we get nothing useful. Unfortunately, R doesn't prevent us from doing this. What if we ask for

| the 3000th element of x? Try it out.

> x[3000]

[1] NA

| Excellent job!

|================================================================= | 59%

| Again, nothing useful, but R doesn't prevent us from asking for it. This should be a cautionary tale. You should

| always make sure that what you are asking for is within the bounds of the vector you're working with.

...

|==================================================================== | 62%

| What if we're interested in all elements of x EXCEPT the 2nd and 10th? It would be pretty tedious to construct a

| vector containing all numbers 1 through 40 EXCEPT 2 and 10.

...

|======================================================================= | 64%

| Luckily, R accepts negative integer indexes. Whereas x[c(2, 10)] gives us ONLY the 2nd and 10th elements of x, x[c(-2,

| -10)] gives us all elements of x EXCEPT for the 2nd and 10 elements. Try x[c(-2, -10)] now to see this.

> x[c(-2,-10)]

[1] NA -0.86867902 NA NA -0.23893261 NA NA 0.25109672 0.34398630

[10] NA NA 0.35154977 NA NA NA NA NA -0.67859122

[19] 0.08877186 NA -0.27815706 NA NA -0.99114632 NA -1.62448744 -0.82246287

[28] 1.38536080 -0.31329923 0.23056499 NA NA -0.14378969 -1.23546524 0.14058996 0.58282334

[37] NA -1.60875462

| You are really on a roll!

|========================================================================== | 67%

| A shorthand way of specifying multiple negative numbers is to put the negative sign out in front of the vector of

| positive numbers. Type x[-c(2, 10)] to get the exact same result.

> x[-c(2,10)]

[1] NA -0.86867902 NA NA -0.23893261 NA NA 0.25109672 0.34398630

[10] NA NA 0.35154977 NA NA NA NA NA -0.67859122

[19] 0.08877186 NA -0.27815706 NA NA -0.99114632 NA -1.62448744 -0.82246287

[28] 1.38536080 -0.31329923 0.23056499 NA NA -0.14378969 -1.23546524 0.14058996 0.58282334

[37] NA -1.60875462

| That's correct!

|============================================================================= | 69%

| So far, we've covered three types of index vectors -- logical, positive integer, and negative integer. The only

| remaining type requires us to introduce the concept of 'named' elements.

...

|================================================================================ | 72%

| Create a numeric vector with three named elements using vect <- c(foo = 11, bar = 2, norf = NA).

> vect <- c(foo = 11, bar = 2, norf = NA)

| Keep working like that and you'll get there!

|=================================================================================== | 74%

| When we print vect to the console, you'll see that each element has a name. Try it out.

> vect

foo bar norf

11 2 NA

| All that hard work is paying off!

|===================================================================================== | 77%

| We can also get the names of vect by passing vect as an argument to the names() function. Give that a try.

> names(vect)

[1] "foo" "bar" "norf"

| Great job!

|======================================================================================== | 79%

| Alternatively, we can create an unnamed vector vect2 with c(11, 2, NA). Do that now.

> vect <- c(11,2, NA)

| Not quite, but you're learning! Try again. Or, type info() for more options.

| Create an ordinary (unnamed) vector called vect2 that contains c(11, 2, NA).

> vect <- c(11, 2, NA)

| Not exactly. Give it another go. Or, type info() for more options.

| Create an ordinary (unnamed) vector called vect2 that contains c(11, 2, NA).

> skip()

| Entering the following correct answer for you...

> vect2 <- c(11, 2, NA)

| Perseverance, that's the answer.

|=========================================================================================== | 82%

| Then, we can add the `names` attribute to vect2 after the fact with names(vect2) <- c("foo", "bar", "norf"). Go ahead.

> names(vect2)<- c("foo","bar","norf")

| You are really on a roll!

|============================================================================================== | 85%

| Now, let's check that vect and vect2 are the same by passing them as arguments to the identical() function.

> indentical(vect,vect2)

Error in indentical(vect, vect2) : could not find function "indentical"

> skip()

| Entering the following correct answer for you...

> identical(vect, vect2)

[1] TRUE

| You're the best!

|================================================================================================= | 87%

| Indeed, vect and vect2 are identical named vectors.

...

|==================================================================================================== | 90%

| Now, back to the matter of subsetting a vector by named elements. Which of the following commands do you think would

| give us the second element of vect?

1: vect["2"]

2: vect[bar]

3: vect["bar"]

Selection: 2

| Give it another try.

| If we want the element named "bar" (i.e. the second element of vect), which command would get us that?

1: vect["2"]

2: vect["bar"]

3: vect[bar]

Selection: 2

| Your dedication is inspiring!

|====================================================================================================== | 92%

| Now, try it out.

> vect["bar"]

bar

2

| That's the answer I was looking for.

|========================================================================================================= | 95%

| Likewise, we can specify a vector of names with vect[c("foo", "bar")]. Try it out.

> vect[c("foo", "bar")]

foo bar

11 2

| Keep up the great work!

|============================================================================================================ | 97%

| Now you know all four methods of subsetting data from vectors. Different approaches are best in different scenarios

| and when in doubt, try it out!

...

|===============================================================================================================| 100%

| Would you like to receive credit for completing this course on Coursera.org?

1: Yes

2: No

Selection: 2

| All that practice is paying off!

| You've reached the end of this lesson! Returning to the main menu...

| Please choose a course, or type 0 to exit swirl.

1: R Programming

2: Take me to the swirl course repository!

Selection: 1

| Please choose a lesson, or type 0 to return to course menu.

1: Basic Building Blocks 2: Workspace and Files 3: Sequences of Numbers 4: Vectors

5: Missing Values 6: Subsetting Vectors 7: Matrices and Data Frames 8: Logic

9: Functions 10: lapply and sapply 11: vapply and tapply 12: Looking at Data

13: Simulation 14: Dates and Times 15: Base Graphics

Selection: 7

| | 0%

| In this lesson, we'll cover matrices and data frames. Both represent 'rectangular' data types, meaning that they are

| used to store tabular data, with rows and columns.

...

|=== | 3%

| The main difference, as you'll see, is that matrices can only contain a single class of data, while data frames can

| consist of many different classes of data.

...

|====== | 6%

| Let's create a vector containing the numbers 1 through 20 using the `:` operator. Store the result in a variable

| called my\_vector.

> my\_vector <- c(1:20)

| Not quite! Try again. Or, type info() for more options.

| You learned about the `:` operator in the lesson on sequences. If you wanted to create a vector containing the numbers

| 1, 2, and 3 (in that order), you could use either c(1, 2, 3) or 1:3. In this case, we want the numbers 1 through 20

| stored in a variable called my\_vector. Also, remember that you don't need the c() function when using `:`.

Error in file(out, "wt") : cannot open the connection

> skip()

| Entering the following correct answer for you...

> my\_vector <- 1:20

| You got it right!

|========= | 8%

| View the contents of the vector you just created.

> my\_vector

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

| You got it!

|============ | 11%

| The dim() function tells us the 'dimensions' of an object. What happens if we do dim(my\_vector)? Give it a try.

> dim(my\_vector)

NULL

| You are quite good my friend!

|=============== | 14%

| Clearly, that's not very helpful! Since my\_vector is a vector, it doesn't have a `dim` attribute (so it's just NULL),

| but we can find its length using the length() function. Try that now.

> length(my\_vector)

[1] 20

| That's a job well done!

|================== | 17%

| Ah! That's what we wanted. But, what happens if we give my\_vector a `dim` attribute? Let's give it a try. Type

| dim(my\_vector) <- c(4, 5).

> dim(my\_vector) <- c(4,5)

| Keep up the great work!

|====================== | 19%

| It's okay if that last command seemed a little strange to you. It should! The dim() function allows you to get OR set

| the `dim` attribute for an R object. In this case, we assigned the value c(4, 5) to the `dim` attribute of my\_vector.

...

|========================= | 22%

| Use dim(my\_vector) to confirm that we've set the `dim` attribute correctly.

> dim(my\_vector)

[1] 4 5

| That's the answer I was looking for.

|============================ | 25%

| Another way to see this is by calling the attributes() function on my\_vector. Try it now.

Error in file(out, "wt") : cannot open the connection

> attributes(my\_vector)

$`dim`

[1] 4 5

| Keep up the great work!

|=============================== | 28%

| Just like in math class, when dealing with a 2-dimensional object (think rectangular table), the first number is the

| number of rows and the second is the number of columns. Therefore, we just gave my\_vector 4 rows and 5 columns.

...

|================================== | 31%

| But, wait! That doesn't sound like a vector any more. Well, it's not. Now it's a matrix. View the contents of

| my\_vector now to see what it looks like.

> my\_vector

[,1] [,2] [,3] [,4] [,5]

[1,] 1 5 9 13 17

[2,] 2 6 10 14 18

[3,] 3 7 11 15 19

[4,] 4 8 12 16 20

| You nailed it! Good job!

|===================================== | 33%

| Now, let's confirm it's actually a matrix by using the class() function. Type class(my\_vector) to see what I mean.

> class(my\_vector)

[1] "matrix"

| You nailed it! Good job!

|======================================== | 36%

| Sure enough, my\_vector is now a matrix. We should store it in a new variable that helps us remember what it is. Store

| the value of my\_vector in a new variable called my\_matrix.

> my\_matrix <- c(my\_vector)

| That's not exactly what I'm looking for. Try again. Or, type info() for more options.

| Since we can't just change the name of our my\_vector variable, we'll assign its value to a new variable with my\_matrix

| <- my\_vector.

> my\_matrix <- my\_vector

| That's the answer I was looking for.

|=========================================== | 39%

| The example that we've used so far was meant to illustrate the point that a matrix is simply an atomic vector with a

| dimension attribute. A more direct method of creating the same matrix uses the matrix() function.

...

|============================================== | 42%

| Bring up the help file for the matrix() function now using the `?` function.

> ?matrix()

| You almost had it, but not quite. Try again. Or, type info() for more options.

| The command ?matrix will do the trick.

> ?matrix

| Keep working like that and you'll get there!

|================================================= | 44%

| Now, look at the documentation for the matrix function and see if you can figure out how to create a matrix containing

| the same numbers (1-20) and dimensions (4 rows, 5 columns) by calling the matrix() function. Store the result in a

| variable called my\_matrix2.

Error in file(out, "wt") : cannot open the connection

> my\_matrix2 <- c(1:20,4,5)

| You almost had it, but not quite. Try again. Or, type info() for more options.

| Call the matrix() function with three arguments -- 1:20, the number of rows, and the number of columns. Be sure to

| specify arguments by their proper names and store the result in my\_matrix2 (not in my\_matrix).

> skip()

| Entering the following correct answer for you...

> my\_matrix2 <- matrix(1:20, nrow=4, ncol=5)

| You are really on a roll!

|==================================================== | 47%

| Finally, let's confirm that my\_matrix and my\_matrix2 are actually identical. The identical() function will tell us if

| its first two arguments are the same. Try it out.

Error in file(out, "wt") : cannot open the connection

> identical(my\_matrix, my\_matrix2)

[1] TRUE

| Excellent job!

|======================================================== | 50%

| Now, imagine that the numbers in our table represent some measurements from a clinical experiment, where each row

| represents one patient and each column represents one variable for which measurements were taken.

...

|=========================================================== | 53%

| We may want to label the rows, so that we know which numbers belong to each patient in the experiment. One way to do

| this is to add a column to the matrix, which contains the names of all four people.

...

|============================================================== | 56%

| Let's start by creating a character vector containing the names of our patients -- Bill, Gina, Kelly, and Sean.

| Remember that double quotes tell R that something is a character string. Store the result in a variable called

| patients.

Error in file(out, "wt") : cannot open the connection

> Patient <- c("Bill","Gina", "kelly", "Sean")

| You almost had it, but not quite. Try again. Or, type info() for more options.

| Make sure to capitalize the first letter of each name and to store the result in a variable called patients. Also,

| don't get the order of the patients mixed up! That would be a disaster!

> Patient <- c("bill", "Gina", "Kelly", "sean")

| You're close...I can feel it! Try it again. Or, type info() for more options.

| Make sure to capitalize the first letter of each name and to store the result in a variable called patients. Also,

| don't get the order of the patients mixed up! That would be a disaster!

> skip()

| Entering the following correct answer for you...

> patients <- c("Bill", "Gina", "Kelly", "Sean")

| All that practice is paying off!

|================================================================= | 58%

| Now we'll use the cbind() function to 'combine columns'. Don't worry about storing the result in a new variable. Just

| call cbind() with two arguments -- the patients vector and my\_matrix.

Error in file(out, "wt") : cannot open the connection

> cbind(patients, my\_matrix)

patients

[1,] "Bill" "1" "5" "9" "13" "17"

[2,] "Gina" "2" "6" "10" "14" "18"

[3,] "Kelly" "3" "7" "11" "15" "19"

[4,] "Sean" "4" "8" "12" "16" "20"

| You got it!

|==================================================================== | 61%

| Something is fishy about our result! It appears that combining the character vector with our matrix of numbers caused

| everything to be enclosed in double quotes. This means we're left with a matrix of character strings, which is no

| good.

...

|======================================================================= | 64%

| If you remember back to the beginning of this lesson, I told you that matrices can only contain ONE class of data.

| Therefore, when we tried to combine a character vector with a numeric matrix, R was forced to 'coerce' the numbers to

| characters, hence the double quotes.

...

|========================================================================== | 67%

| This is called 'implicit coercion', because we didn't ask for it. It just happened. But why didn't R just convert the

| names of our patients to numbers? I'll let you ponder that question on your own.

...

|============================================================================= | 69%

| So, we're still left with the question of how to include the names of our patients in the table without destroying the

| integrity of our numeric data. Try the following -- my\_data <- data.frame(patients, my\_matrix)

Error in file(out, "wt") : cannot open the connection

> my\_data <- data.frame(patients, my\_matrix)

| Perseverance, that's the answer.

|================================================================================ | 72%

| Now view the contents of my\_data to see what we've come up with.

> my\_data

patients X1 X2 X3 X4 X5

1 Bill 1 5 9 13 17

2 Gina 2 6 10 14 18

3 Kelly 3 7 11 15 19

4 Sean 4 8 12 16 20

| That's a job well done!

|=================================================================================== | 75%

| It looks like the data.frame() function allowed us to store our character vector of names right alongside our matrix

| of numbers. That's exactly what we were hoping for!

...

|====================================================================================== | 78%

| Behind the scenes, the data.frame() function takes any number of arguments and returns a single object of class

| `data.frame` that is composed of the original objects.

...

|========================================================================================= | 81%

| Let's confirm this by calling the class() function on our newly created data frame.

> class(my\_data)

[1] "data.frame"

| That's correct!

|============================================================================================ | 83%

| It's also possible to assign names to the individual rows and columns of a data frame, which presents another possible

| way of determining which row of values in our table belongs to each patient.

...

|================================================================================================ | 86%

| However, since we've already solved that problem, let's solve a different problem by assigning names to the columns of

| our data frame so that we know what type of measurement each column represents.

...

|=================================================================================================== | 89%

| Since we have six columns (including patient names), we'll need to first create a vector containing one element for

| each column. Create a character vector called cnames that contains the following values (in order) -- "patient",

| "age", "weight", "bp", "rating", "test".

> cnames <- c("patient", "age", "weight", "bp", "rating", "test")

| You're the best!

|====================================================================================================== | 92%

| Now, use the colnames() function to set the `colnames` attribute for our data frame. This is similar to the way we

| used the dim() function earlier in this lesson.

Error in file(out, "wt") : cannot open the connection

> colnames(cnames)

NULL

| Almost! Try again. Or, type info() for more options.

| Try colnames(my\_data) <- cnames.

> colnames(cnames)

NULL

| Try again. Getting it right on the first try is boring anyway! Or, type info() for more options.

| Try colnames(my\_data) <- cnames.

> skip()

| Entering the following correct answer for you...

> colnames(my\_data) <- cnames

| You are quite good my friend!

|========================================================================================================= | 94%

| Let's see if that got the job done. Print the contents of my\_data.

> my\_data

patient age weight bp rating test

1 Bill 1 5 9 13 17

2 Gina 2 6 10 14 18

3 Kelly 3 7 11 15 19

4 Sean 4 8 12 16 20

| Your dedication is inspiring!

|============================================================================================================ | 97%

| In this lesson, you learned the basics of working with two very important and common data structures -- matrices and

| data frames. There's much more to learn and we'll be covering more advanced topics, particularly with respect to data

| frames, in future lessons.

...

|===============================================================================================================| 100%

| Would you like to receive credit for completing this course on Coursera.org?

1: No

2: Yes

Selection: 1

| Excellent work!

| You've reached the end of this lesson! Returning to the main menu...

| Please choose a course, or type 0 to exit swirl.

1: R Programming

2: Take me to the swirl course repository!