Week 1 R Quiz

|  |
| --- |
| > my\_data  [1] 0.6742841 NA 1.4310072 -0.3987273 0.2087032 0.2388870 NA NA 0.1188924 -0.1665823 NA NA NA  [14] 1.9337871 NA NA -0.1721988 NA NA NA 0.1826903 NA 0.3449658 NA NA -0.6289643  [27] -0.2356352 NA -1.6488669 NA NA NA -0.6143092 NA 1.7196410 0.1670003 NA NA 0.1534375  [40] -0.5129245 NA NA NA -1.8436074 0.4785066 NA NA NA 0.4369445 0.6150833 NA 0.9135801  [53] NA NA NA 0.5377129 NA NA NA NA 0.1783982 -0.6984531 1.0019841 1.8232178 1.0790879  [66] NA NA -1.9496951 NA -0.6541084 NA NA NA NA -0.7470551 -0.6470831 NA NA  [79] NA NA NA NA NA -0.9162522 1.2828151 -0.9347724 -0.9437486 NA 0.1397368 NA 1.2322816  [92] -1.6595730 0.3265106 -0.2440135 -1.6723572 -1.2070996 NA 1.7194646 -1.4657697 NA  | Excellent job!  |================================================================================================================================================== | 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 are really on a roll!  |========================================================================================================================================================== | 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  [1] NaN  | Keep up the great work!  |==================================================================================================================================================================| 100%  | Would you like to receive credit for completing this course on Coursera.org?  1: Yes  2: Generate Code  3: No  Selection: 2  Submit the following code as the answer  to a quiz question on Coursera.  #########################  NoawKtpqDOc4Szi9qyqX  #########################  | Perseverance, that's the answer.  | You've reached the end of this lesson! Returning to the main menu...  | Please choose a course, or type 0 to exit swirl.  1: The R Programming Environment  2: Take me to the swirl course repository!  Selection: 1  | Please choose a lesson, or type 0 to return to course menu.  1: Setting Up Swirl 2: Basic Building Blocks 3: Sequences of Numbers 4: Vectors  5: Missing Values 6: Subsetting Vectors 7: Matrices and Data Frames 8: Logic  9: Workspace and Files 10: Reading Tabular Data 11: Looking at Data 12: Data Manipulation  13: Text Manipulation Functions 14: Regular Expressions 15: The stringr Package  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] 0.82356980 NA NA -0.13872447 -0.26461747 NA NA -0.44626884 -1.20361830 0.02094342 NA -1.05550816  [13] NA NA 0.75191437 -0.19775039 0.27848848 NA NA -1.98745988 NA 0.85019207 -0.05317058 NA  [25] NA NA 0.66974784 -0.09775127 NA 0.21543083 NA 0.73644386 NA NA 0.53958429 -0.96475584  [37] NA 0.08317012 NA NA  | All that practice is paying off!  |=========== | 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] 0.82356980 NA NA -0.13872447 -0.26461747 NA NA -0.44626884 -1.20361830 0.02094342  | All that practice is paying off!  |================== | 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 all NAs  2: A vector of TRUEs and FALSEs  3: A vector of length 0  4: A vector with no NAs  Selection: 4  | Almost! 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 with no NAs  2: A vector of TRUEs and FALSEs  3: A vector of length 0  4: A vector of all NAs  Selection: 0  | Give it another try.  | 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 TRUEs and FALSEs  2: A vector with no NAs  3: A vector of all NAs  4: A vector of length 0  Selection: 3  | Great job!  |============================== | 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  | You're the best!  |================================= | 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 <-[is.na(x)]  Error: unexpected '[' in "y <-["  > y <- [!is.na(x)]  Error: unexpected '[' in "y <- ["  > y <- x[!is.na(x)]  | That's the answer I was looking for.  |===================================== | 26%  | Print y to the console.  > y  [1] 0.82356980 -0.13872447 -0.26461747 -0.44626884 -1.20361830 0.02094342 -1.05550816 0.75191437 -0.19775039 0.27848848 -1.98745988 0.85019207  [13] -0.05317058 0.66974784 -0.09775127 0.21543083 0.73644386 0.53958429 -0.96475584 0.08317012  | You are really on a roll!  |========================================= | 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 all NAs  2: A vector of all the positive elements of y  3: A vector of length 0  4: A vector of all the negative elements of y  5: A vector of TRUEs and FALSEs  Selection: 2  | Excellent job!  |================================================ | 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] 0.82356980 0.02094342 0.75191437 0.27848848 0.85019207 0.66974784 0.21543083 0.73644386 0.53958429 0.08317012  | Keep up the great work!  |==================================================== | 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 > o]  Error: object 'o' not found  > x[x > 0]  [1] 0.82356980 NA NA NA NA 0.02094342 NA NA NA 0.75191437 0.27848848 NA NA  [14] NA 0.85019207 NA NA NA 0.66974784 NA 0.21543083 NA 0.73644386 NA NA 0.53958429  [27] NA 0.08317012 NA NA  | You are really on a roll!  |======================================================= | 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) > 0]  [1] 0.82356980 -0.13872447 -0.26461747 -0.44626884 -1.20361830 0.02094342 -1.05550816 0.75191437 -0.19775039 0.27848848 -1.98745988 0.85019207  [13] -0.05317058 0.66974784 -0.09775127 0.21543083 0.73644386 0.53958429 -0.96475584 0.08317012  | Not quite, but you're learning! Try again. Or, type info() for more options.  | Try x[!is.na(x) & x > 0] to see what you get.  > x[!is.na(x) & x > 0]  [1] 0.82356980 0.02094342 0.75191437 0.27848848 0.85019207 0.66974784 0.21543083 0.73644386 0.53958429 0.08317012  | All that practice 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.  > x <- c(x: [3, 5, 7])  Error: unexpected '[' in "x <- c(x: ["  > x <- c(x = 3, x = 5, x = 7)  | That's not the answer I was looking for, but try again. Or, type info() for more options.  | Create a vector of indexes with c(3, 5, 7), then put that inside of the square brackets.  > c(3, 5, 7)  [1] 3 5 7  | Give it another try. Or, type info() for more options.  | Create a vector of indexes with c(3, 5, 7), then put that inside of the square brackets.  > info()  | When you are at the R prompt (>):  | -- Typing skip() allows you to skip the current question.  | -- Typing play() lets you experiment with R on your own; swirl will ignore what you do...  | -- UNTIL you type nxt() which will regain swirl's attention.  | -- Typing bye() causes swirl to exit. Your progress will be saved.  | -- Typing main() returns you to swirl's main menu.  | -- Typing info() displays these options again.  > [c(3, 5, 7)]  Error: unexpected '[' in "["  > c[3, 5, 7]  Error in c[3, 5, 7] : object of type 'builtin' is not subsettable  > is.vector(x, double(OL))  Error in double(OL) : object 'OL' not found  > cls  Error: object 'cls' not found  > ?sub  > ?v  No documentation for ‘v’ in specified packages and libraries:  you could try ‘??v’  > ?`vector-class`  > as.numeric(x)  [1] 3 5 7  | Try again. Getting it right on the first try is boring anyway! Or, type info() for more options.  | Create a vector of indexes with c(3, 5, 7), then put that inside of the square brackets.  > [c(3, 5, 7)]  Error: unexpected '[' in "["  > x is.vector(x, mode = 'any')  Error: unexpected symbol in "x is.vector"  > x [c(3, 5, 7)]  x <NA> <NA>  7 NA NA  | Keep working like that and you'll get there!  |============================================================================== | 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]  named numeric(0)  | You got it!  |================================================================================= | 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]  <NA>  NA  | Keep up the great work!  |===================================================================================== | 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)]  x x  3 7  | You are doing so well!  |================================================================================================ | 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)]  x x  3 7  | Nice work!  |==================================================================================================== | 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)  | That's correct!  |=========================================================================================================== | 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  | Excellent work!  |=============================================================================================================== | 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"  | You got it right!  |================================================================================================================== | 79%  | Alternatively, we can create an unnamed vector vect2 with c(11, 2, NA). Do that now.  > vect2 <- c(11, 2, NA)  | Nice work!  |====================================================================================================================== | 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")  | Excellent job!  |========================================================================================================================== | 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: could not find function "indentical"  > names(vect)  [1] "foo" "bar" "norf"  | Almost! Try again. Or, type info() for more options.  | The identical() function tells us if its first two arguments are, well, identical.  > identical(vect, vect2)  [1] TRUE  | Perseverance, that's the answer.  |============================================================================================================================== | 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[bar]  2: vect["bar"]  3: vect["2"]  Selection: 2  | All that practice is paying off!  |===================================================================================================================================== | 92%  | Now, try it out.  > vect["bar"]  bar  2  | You're the best!  |========================================================================================================================================= | 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  | You're the best!  |============================================================================================================================================ | 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: Generate Code  3: No  Selection: 2  Submit the following code as the answer  to a quiz question on Coursera.  #########################  WxGMrFld23A8mrmNXMjw  #########################  | 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: The R Programming Environment  2: Take me to the swirl course repository!  Selection: 1  | Please choose a lesson, or type 0 to return to course menu.  1: Setting Up Swirl 2: Basic Building Blocks 3: Sequences of Numbers 4: Vectors  5: Missing Values 6: Subsetting Vectors 7: Matrices and Data Frames 8: Logic  9: Workspace and Files 10: Reading Tabular Data 11: Looking at Data 12: Data Manipulation  13: Text Manipulation Functions 14: Regular Expressions 15: The stringr Package  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)  | You almost had it, but 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 `:`.  > my\_vector <- [1;20]  Error: unexpected '[' in "my\_vector <- ["  > my\_vector <- 1:20  | You're the best!  |============ | 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 are really on a roll!  |================ | 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  | Your dedication is inspiring!  |==================== | 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\_vecotr)  Error: object 'my\_vecotr' not found  > 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\_vecor) <- c(4, 5)  Error in dim(my\_vecor) <- c(4, 5) : object 'my\_vecor' not found  > 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  | Nice work!  |==================================== | 25%  | Another way to see this is by calling the attributes() function on my\_vector. Try it now.  > attributes(my\_vector)  $dim  [1] 4 5  | Your dedication is inspiring!  |======================================== | 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 are really on a roll!  |================================================ | 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"  | All that hard work is paying off!  |==================================================== | 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\_matirc <- my\_vector  | You're close...I can feel it! Try it 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  | You are amazing!  |======================================================== | 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.  > ?function  +  + '?'function  Error: unexpected string constant in:  "  '?'"  > `?`function  Error: unexpected 'function' in "`?`function"  > `?`  function (e1, e2)  {  if (missing(e2)) {  type <- NULL  topicExpr <- substitute(e1)  }  else {  type <- substitute(e1)  topicExpr <- substitute(e2)  }  search <- (is.call(topicExpr) && topicExpr[[1L]] == "?")  if (search) {  topicExpr <- topicExpr[[2L]]  if (is.call(te <- topicExpr) && te[[1L]] == "?" && is.call(te <- topicExpr[[2L]]) &&  te[[1L]] == "?") {  cat("Contacting Delphi...")  flush.console()  Sys.sleep(2 + rpois(1, 2))  cat("the oracle is unavailable.\nWe apologize for any inconvenience.\n")  return(invisible())  }  }  if (is.call(topicExpr) && (topicExpr[[1L]] == "::" || topicExpr[[1L]] ==  ":::")) {  package <- as.character(topicExpr[[2L]])  topicExpr <- topicExpr[[3L]]  }  else package <- NULL  if (search) {  if (is.null(type))  return(eval(substitute(help.search(TOPIC, package = PACKAGE),  list(TOPIC = as.character(topicExpr), PACKAGE = package))))  else return(eval(substitute(help.search(TOPIC, fields = FIELD,  package = PACKAGE), list(TOPIC = as.character(topicExpr),  FIELD = as.character(type), PACKAGE = package))))  }  else {  if (is.null(type)) {  if (is.call(topicExpr))  return(utils:::.helpForCall(topicExpr, parent.frame()))  topic <- if (is.name(topicExpr))  as.character(topicExpr)  else e1  return(eval(substitute(help(TOPIC, package = PACKAGE),  list(TOPIC = topic, PACKAGE = package))))  }  else {  type <- if (is.name(type))  as.character(type)  else e1  topic <- if (is.name(topicExpr))  as.character(topicExpr)  else {  if (is.call(topicExpr) && identical(type, "method"))  return(utils:::.helpForCall(topicExpr, parent.frame(),  FALSE))  e2  }  if (type == "package")  package <- topic  doHelp <- RevoMods:::.tryHelp(topicName(type, topic),  package = package)  if (inherits(doHelp, "try-error")) {  if (is.language(topicExpr))  topicExpr <- deparse(topicExpr)  stop(gettextf("no documentation of type %s and topic %s (or error in processing help)",  sQuote(type), sQuote(topicExpr)), domain = NA)  }  doHelp  }  }  }  <environment: namespace:RevoMods>  | Keep trying! Or, type info() for more options.  | The command ?matrix will do the trick.  > ?matrix  | Nice work!  |================================================================ | 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.  > is.matric(as.matric(1:20))  Error: could not find function "is.matric"  > is.matrix(as.matrix(1:20))  [1] TRUE  | That's not exactly what I'm looking for. 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).  > mattrix(1:20, nrow = 4, ncol = 5, byrow = FALSE)  Error: could not find function "mattrix"  > matrix(1:20, nrow = 4, ncol = 5, byrow = FALSE)  [,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  | Keep trying! 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).  > my\_matrix2 <- matrix(1:20, 4, 6)  Warning message:  In matrix(1:20, 4, 6) :  data length [20] is not a sub-multiple or multiple of the number of columns [6]  | 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).  > my\_matrix2 <- matrix(1:20, 4, 5)  | All that hard work is paying off!  |==================================================================== | 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.  > identical(my\_matrix, my\_matrix2)  [1] TRUE  | Keep up the great work!  |======================================================================== | 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.  > patients("Bill"", "Gina"", "Kelly"", "Sean")  Error: unexpected string constant in "patients("Bill"", ""  > patients("Bill", "Gina"", "Kelly"", "Sean")  Error: unexpected string constant in "patients("Bill", "Gina"", ""  > patients <- matirx("Bill", "Gina"", "Kelly"", "Sean")  Error: unexpected string constant in "patients <- matirx("Bill", "Gina"", ""  > patients <- ("Bill", "Gina"", "Kelly"", "Sean")  Error: unexpected ',' in "patients <- ("Bill","  > patients <- c("Bill", "Gina"", "Kelly"", "Sean")  Error: unexpected string constant in "patients <- c("Bill", "Gina"", ""  > matric("Bill", "Gina"", "Kelly"", "Sean")  Error: unexpected string constant in "matric("Bill", "Gina"", ""  > matrix("Bill", "Gina"", "Kelly"", "Sean")  Error: unexpected string constant in "matrix("Bill", "Gina"", ""  > patients <- matrix(c("Bill", "Gina"", "Kelly"", "Sean"))  Error: unexpected string constant in "patients <- matrix(c("Bill", "Gina"", ""  > patients <- (c("Bill", "Gina"", "Kelly"", "Sean"))  Error: unexpected string constant in "patients <- (c("Bill", "Gina"", ""  > patients <- c("Bill", "Gina"", "Kelly"", "Sean"))  Error: unexpected string constant in "patients <- c("Bill", "Gina"", ""  > patients <- c("Bill", "Gina"", "Kelly"", "Sean")  Error: unexpected string constant in "patients <- c("Bill", "Gina"", ""  > ?matrix  > patients <- c("Bill", "Gina", "Kelly", "Sean")  | You nailed it! Good job!  |==================================================================================== | 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.  > 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're the best!  |======================================================================================== | 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)  > my\_data <- data.frame(patients, my\_matrix)  | You are really on a roll!  |======================================================================================================== | 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  | Your dedication is inspiring!  |============================================================================================================ | 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.  > clas(my\_data)  Error: could not find function "clas"  > class(data.frame)  [1] "function"  | Not quite right, but keep trying. Or, type info() for more options.  | The class function takes one argument -- the object of which you want to determine the class.  > Class(my\_data)  Error: could not find function "Class"  > class(my\_data)  [1] "data.frame"  | You're the best!  |======================================================================================================================== | 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")  | Excellent work!  |==================================================================================================================================== | 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.  > colnames(my\_data)  [1] "patients" "X1" "X2" "X3" "X4" "X5"  | Almost! Try again. Or, type info() for more options.  | Try colnames(my\_data) <- cnames.  > colnames(my\_data) <- matrix(c("patient", "age", "weight", "bp", "rating", "test"), nrow = 2, ncol = 3  +  + colnames(my\_data) <- cnames  Error: unexpected symbol in:  "  colnames"  > colnames(my\_data) <- cnames  | You are really on a roll!  |======================================================================================================================================== | 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  | All that hard work is paying off!  |============================================================================================================================================ | 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: Generate Code  3: Yes  Selection: 2  Submit the following code as the answer  to a quiz question on Coursera.  #########################  vudJn2TG19ARluGDFhYb  #########################  | That's correct!  | You've reached the end of this lesson! Returning to the main menu...  | Please choose a course, or type 0 to exit swirl.  1: The R Programming Environment  2: Take me to the swirl course repository! |
|  |
| |  | | --- | |  | |