Untitled

SenthilRajah

2/6/2020

## Till now

You are amazing!

|=================================================================== | 76% | Take a look at the contents of my\_name.

my\_name [1] “My” “name” “is” “Senthil”

That’s correct!

|===================================================================== | 78% | Now, use the paste() function once more to join the words in my\_name together into a single | character string. Don’t forget to say collapse = " "!

paste(my\_char, collapse = " “) [1]”My name is"

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

Use paste(my\_name, collapse = " ") to join all four words together, separated by single spaces.

paste(my\_name, collapse = " “) [1]”My name is Senthil"

Keep up the great work!

|======================================================================= | 81% | In this example, we used the paste() function to collapse the elements of a single character | vector. paste() can also be used to join the elements of multiple character vectors.

…

|========================================================================== | 84% | In the simplest case, we can join two character vectors that are each of length 1 (i.e. join | two words). Try paste(“Hello”, “world!”, sep = " "), where the sep argument tells R that we | want to separate the joined elements with a single space.

paste(“Hello”, “world!”, sep = " “) [1]”Hello world!"

You are doing so well!

|============================================================================ | 86% | For a slightly more complicated example, we can join two vectors, each of length 3. Use paste() | to join the integer vector 1:3 with the character vector c(“X”, “Y”, “Z”). This time, use sep = | "" to leave no space between the joined elements.

paste(1:3, c(“X”,“Y”,“Z”), sep = "“) [1]”1X" “2Y” “3Z”

Keep up the great work!

|============================================================================== | 89% | What do you think will happen if our vectors are of different length? (Hint: we talked about | this in a previous lesson.)

…

|================================================================================= | 92% | Vector recycling! Try paste(LETTERS, 1:4, sep = “-”), where LETTERS is a predefined variable in | R containing a character vector of all 26 letters in the English alphabet.

paste(LETTES, 1:4, sep = “-”) Error in paste(LETTES, 1:4, sep = “-”) : object ‘LETTES’ not found paste(LETTERS, 1:4, sep = “-”) [1] “A-1” “B-2” “C-3” “D-4” “E-1” “F-2” “G-3” “H-4” “I-1” “J-2” “K-3” “L-4” “M-1” “N-2” “O-3” [16] “P-4” “Q-1” “R-2” “S-3” “T-4” “U-1” “V-2” “W-3” “X-4” “Y-1” “Z-2”

Excellent job!

|=================================================================================== | 95% | Since the character vector LETTERS is longer than the numeric vector 1:4, R simply recycles, or | repeats, 1:4 until it matches the length of LETTERS.

…

|====================================================================================== | 97% | Also worth noting is that the numeric vector 1:4 gets ‘coerced’ into a character vector by the | paste() function.

…

|========================================================================================| 100% | We’ll discuss coercion in another lesson, but all it really means is that the numbers 1, 2, 3, | and 4 in the output above are no longer numbers to R, but rather characters “1”, “2”, “3”, and | “4”.

…

You’ve reached the end of this lesson! Returning to the main menu…

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

1: 14 310x Intro to R 2: Take me to the swirl course repository!

Selection: 1

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

1: Welcome 2: Basic Building Blocks  
3: Workspace and Files 4: Sequences of Numbers  
5: Vectors 6: Missing Values  
7: Subsetting Vectors 8: Matrices and Data Frames  
9: Looking at Data 10: Base Graphics  
11: Manipulating Data with dplyr 12: Getting and Cleaning Data  
13: Tidying Data with tidyr

Selection: 6

| | 0%

Missing values play an important role in statistics and data analysis. Often, missing values  
must not be ignored, but rather they should be carefully studied to see if there’s an  
underlying pattern or cause for their missingness.

…

|===== | 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.

…

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

You are doing so well!

|============== | 16% | Now, let’s multiply x by 3.

x \* 3 [1] 132 NA 15 NA

Excellent work!

|=================== | 21% | Notice that the elements of the resulting vector that correspond with the NA values in x are | also NA.

…

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

That’s a job well done!

|============================ | 32% | Next, let’s create a vector containing 1000 NAs with z <- rep(NA, 1000).

z <- rep(NA, 1000)

Great job!

|================================ | 37% | 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)

Perseverance, that’s the answer.

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

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

You got it!

|========================================== | 47% | Now, print my\_na to see what you came up with.

my\_na [1] TRUE TRUE FALSE TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE TRUE FALSE TRUE [16] TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE TRUE TRUE TRUE [31] TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE FALSE FALSE TRUE [46] FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE TRUE FALSE TRUE TRUE TRUE [61] FALSE FALSE TRUE TRUE FALSE TRUE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE [76] TRUE FALSE TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE [91] TRUE TRUE FALSE FALSE FALSE TRUE TRUE FALSE TRUE FALSE

Great job!

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

…

|=================================================== | 58% | 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 [32] 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 [63] 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 [94] NA NA NA NA NA NA NA

You are doing so well!

|======================================================== | 63% | 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.

…

|============================================================ | 68% | 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.

…

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

…

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

…

|========================================================================== | 84% | 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] 42

Keep working like that and you’ll get there!

|=============================================================================== | 89% | 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] NA NA 0.766507358 NA 0.576171908 NA 0.273997105 [8] -2.447378692 0.362546986 -0.359519867 NA -1.221925291 NA 0.060183896 [15] NA NA 0.732260025 NA -0.295260022 -0.002423927 0.312456255 [22] 0.648350268 NA NA 1.149631429 1.066018579 -0.418162607 NA [29] NA NA NA -1.386591250 -0.059167133 NA 0.536160824 [36] 1.124209486 -0.560361349 2.282656305 NA 0.295510459 NA NA [43] -0.475792369 0.338839111 NA 1.289500304 -0.556358616 NA NA [50] NA -0.461328937 -1.233981793 0.461613381 NA 0.374663703 NA [57] 0.689714397 NA NA NA -0.803696021 -0.511354566 NA [64] NA 0.786351130 NA 1.026750612 NA NA 1.191509630 [71] 0.122104732 -0.864595759 1.484332640 -0.733896140 -1.503108224 NA 0.054387702 [78] NA 0.127982148 NA 0.464243202 -0.390298982 0.271760866 -1.836376245 [85] NA 0.122959622 0.614978040 0.949844737 -0.130257800 0.876887575 NA [92] NA -0.019466265 1.613487876 -1.294899453 NA NA -0.472664932 [99] NA 1.739427177

Nice work!

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

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

Your dedication is inspiring!

You’ve reached the end of this lesson! Returning to the main menu…

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

1: 14 310x Intro to R 2: Take me to the swirl course repository!

Selection: 1

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

1: Welcome 2: Basic Building Blocks  
3: Workspace and Files 4: Sequences of Numbers  
5: Vectors 6: Missing Values  
7: Subsetting Vectors 8: Matrices and Data Frames  
9: Looking at Data 10: Base Graphics  
11: Manipulating Data with dplyr 12: Getting and Cleaning Data  
13: Tidying Data with tidyr

Selection: 7

| | 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] 0.34810102 NA NA -1.33973993 NA 0.60510744 -0.61203694 [8] 0.19072382 NA 1.32142810 -1.00855756 NA NA -0.09567495 [15] 1.01203604 -0.80640531 -1.27939940 NA NA NA NA [22] -0.55819941 NA NA NA 0.43595781 1.20779399 NA [29] -0.46312613 NA 0.82276140 NA NA NA -1.03353306 [36] -0.16572335 NA -0.21960928 -0.22751462 NA

Keep up the great work!

|======= | 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.

…

|========= | 11% | For a simple example, try x[1:10] to view the first ten elements of x.

x[1:10] [1] 0.3481010 NA NA -1.3397399 NA 0.6051074 -0.6120369 0.1907238 [9] NA 1.3214281

That’s correct!

|============ | 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.

…

|============== | 16% | 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 with no NAs 4: A vector of length 0

Selection: 1

Excellent work!

|=================== | 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

All that practice is paying off!

|===================== | 24% | 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 quite good my friend!

|======================= | 26% | Print y to the console.

y [1] 0.34810102 -1.33973993 0.60510744 -0.61203694 0.19072382 1.32142810 -1.00855756 [8] -0.09567495 1.01203604 -0.80640531 -1.27939940 -0.55819941 0.43595781 1.20779399 [15] -0.46312613 0.82276140 -1.03353306 -0.16572335 -0.21960928 -0.22751462

You’re the best!

|========================= | 29% | Now that we’ve isolated the non-missing values of x and put them in y, we can subset y as we | please.

…

|============================ | 32% | 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 negative elements of y 3: A vector of TRUEs and FALSEs 4: A vector of all the positive elements of y 5: A vector of length 0

Selection: 4

Great job!

|============================== | 34% | 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] 0.3481010 0.6051074 0.1907238 1.3214281 1.0120360 0.4359578 1.2077940 0.8227614

All that practice is paying off!

|================================ | 37% | 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] 0.3481010 NA NA NA 0.6051074 0.1907238 NA 1.3214281 NA [10] NA 1.0120360 NA NA NA NA NA NA NA [19] 0.4359578 1.2077940 NA NA 0.8227614 NA NA NA NA [28] NA

You got it right!

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

…

|===================================== | 42% | 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] 0.3481010 0.6051074 0.1907238 1.3214281 1.0120360 0.4359578 1.2077940 0.8227614

That’s a job well done!

|======================================= | 45% | In this case, we request only values of x that are both non-missing AND greater than zero.

…

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

…

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

…

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

c(x[3],x[5],x[7]) [1] NA NA -0.6120369

That’s not exactly what I’m looking for. 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.

x[c(3,5,7)] [1] NA NA -0.6120369

You nailed it! Good job!

|================================================= | 55% | 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 got it!

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

You are quite good my friend!

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

…

|======================================================== | 63% | 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.

…

|========================================================== | 66% | 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] 0.34810102 NA -1.33973993 NA 0.60510744 -0.61203694 0.19072382 [8] NA -1.00855756 NA NA -0.09567495 1.01203604 -0.80640531 [15] -1.27939940 NA NA NA NA -0.55819941 NA [22] NA NA 0.43595781 1.20779399 NA -0.46312613 NA [29] 0.82276140 NA NA NA -1.03353306 -0.16572335 NA [36] -0.21960928 -0.22751462 NA

Keep working like that and you’ll get there!

|============================================================ | 68% | 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] 0.34810102 NA -1.33973993 NA 0.60510744 -0.61203694 0.19072382 [8] NA -1.00855756 NA NA -0.09567495 1.01203604 -0.80640531 [15] -1.27939940 NA NA NA NA -0.55819941 NA [22] NA NA 0.43595781 1.20779399 NA -0.46312613 NA [29] 0.82276140 NA NA NA -1.03353306 -0.16572335 NA [36] -0.21960928 -0.22751462 NA

You are doing so well!

|=============================================================== | 71% | 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.

…

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

|=================================================================== | 76% | 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

That’s the answer I was looking for.

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

|======================================================================== | 82% | Alternatively, we can create an unnamed vector vect2 with c(11, 2, NA). Do that now.

vect2(11,2,NA) Error in vect2(11, 2, NA) : could not find function “vect2” vect2 <- c(11,2,NA)

Excellent work!

|========================================================================== | 84% | 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!

|============================================================================ | 87% | Now, let’s check that vect and vect2 are the same by passing them as arguments to the | identical() function.

identical(vect,vect2) [1] TRUE

All that practice is paying off!

|=============================================================================== | 89% | Indeed, vect and vect2 are identical named vectors.

…

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

All that practice is paying off!

|=================================================================================== | 95% | Now, try it out.

vect[“bar”] bar 2

Keep up the great work!

|====================================================================================== | 97% | 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 working like that and you’ll get there!

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

…

You’ve reached the end of this lesson! Returning to the main menu…

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

1: 14 310x Intro to R 2: Take me to the swirl course repository!

Selection: 1

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

1: Welcome 2: Basic Building Blocks  
3: Workspace and Files 4: Sequences of Numbers  
5: Vectors 6: Missing Values  
7: Subsetting Vectors 8: Matrices and Data Frames  
9: Looking at Data 10: Base Graphics  
11: Manipulating Data with dplyr 12: Getting and Cleaning Data  
13: Tidying Data with tidyr

Selection: 8

| | 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 <- 1:20

You are really on a roll!

|======== | 9% | 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

All that practice is paying off!

|========== | 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

All that hard work is paying off!

|============= | 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

Keep up the great work!

|=============== | 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)

That’s correct!

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

…

|==================== | 23% | Use dim(my\_vector) to confirm that we’ve set the dim attribute correctly.

dim(my\_vector) [1] 4 5

All that practice is paying off!

|======================= | 26% | Another way to see this is by calling the attributes() function on my\_vector. Try it now.

attributes(my\_vector) $dim [1] 4 5

Excellent work!

|========================= | 29% | 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

Your dedication is inspiring!

|============================== | 34% | 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”

That’s a job well done!

|================================= | 37% | 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 <- my\_vector

That’s a job well done!

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

…

|====================================== | 43% | Bring up the help file for the matrix() function now using the ? function.

?matrix

You are amazing!

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

my\_matrix2 <- matrix(data = my\_vector, nrow = 4, ncol = 5)

Keep working like that and you’ll get there!

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

All that hard work is paying off!

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

…

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

…

|================================================== | 57% | 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 <- c(“Bill”,“Gina”,“Kelly”,“Sean”)

Perseverance, that’s the answer.

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

Keep up the great work!

|======================================================= | 63% | 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.

…

|========================================================== | 66% | 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.

…

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

…

|=============================================================== | 71% | 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)

All that hard work is paying off!

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

Perseverance, that’s the answer.

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

…

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

…

|========================================================================= | 83% | Let’s confirm this by calling the class() function on our newly created data frame.

class(my\_data) [1] “data.frame”

You’re the best!

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

…

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

…

|================================================================================ | 91% | 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 are quite good my friend!

|=================================================================================== | 94% | 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) <- cnames

You got it!

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

Perseverance, that’s the answer.

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

…

You’ve reached the end of this lesson! Returning to the main menu…

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

1: 14 310x Intro to R 2: Take me to the swirl course repository!

Selection: 1

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

1: Welcome 2: Basic Building Blocks  
3: Workspace and Files 4: Sequences of Numbers  
5: Vectors 6: Missing Values  
7: Subsetting Vectors 8: Matrices and Data Frames  
9: Looking at Data 10: Base Graphics  
11: Manipulating Data with dplyr 12: Getting and Cleaning Data  
13: Tidying Data with tidyr

Selection: