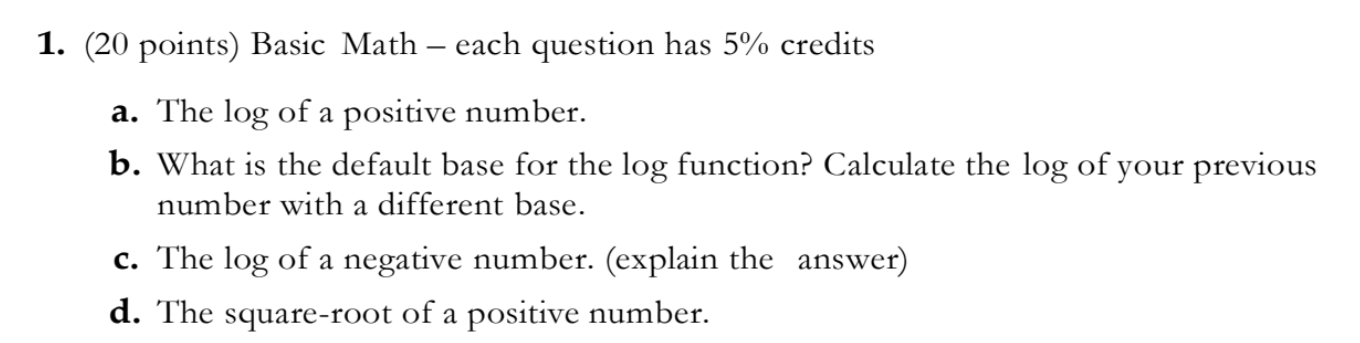
**Unit 1 Assignment**



> #1a

> #Log of 5

> log(5)

[1] 1.609438

>

> #1b.1

> #default base of log function is 10

>

> #1b.2

> #Log of 5 with base of 2

> log(5,2)

[1] 2.321928

>

> #1c

> #Calculate log of a negative number

> #natural log is only for numbers greater than 0, therefore you cannot log a negative number

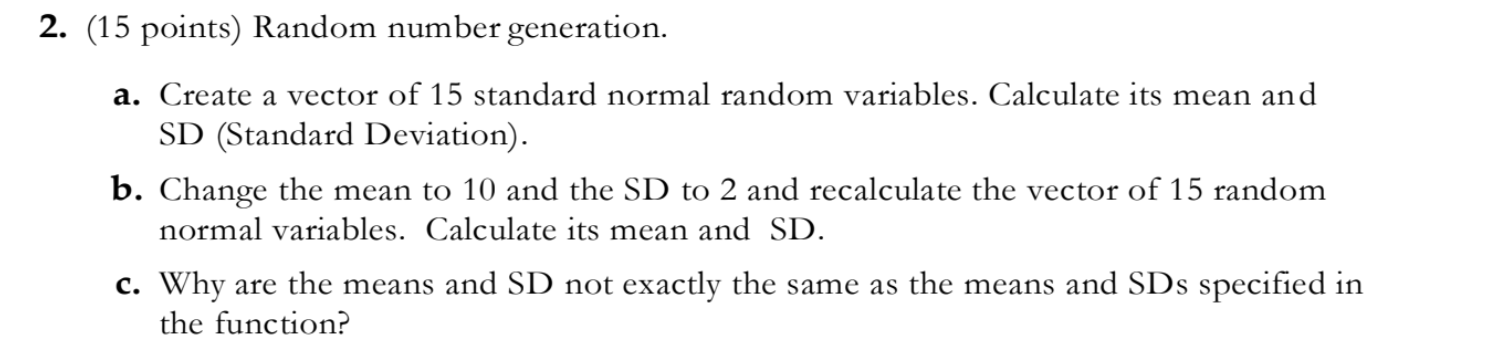
>

> #1d

> #Calculate log of square root of a positive number

> sqrt(4)

[1] 2



> #2a

> #Create a vector of 15 numbers

> RandomNumbers <- floor(runif(15, min=0, max=100))

>

> #2a.1

> #Calculate mean from previous

> mean(RandomNumbers)

[1] 41.86667

>

> #2a.2

> #Calculate SD from previous

> sd(RandomNumbers)

[1] 31.74647

>

> #2b Change the mean to 10 and SD to 2 recalculate vector of 15 random numbers

> RandomNumbersSetMeanSD <- rnorm(15, mean=10, sd=2)

> mean(RandomNumbersSetMeanSD)

[1] 10.25404

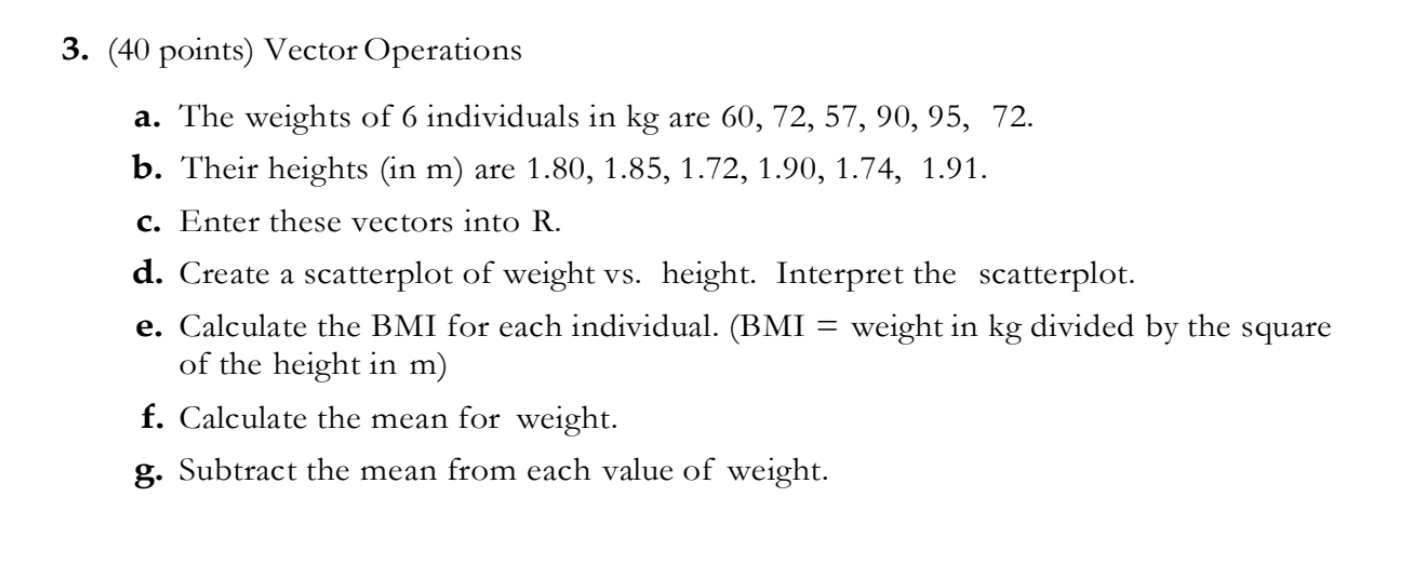
> sd(RandomNumbersSetMeanSD)

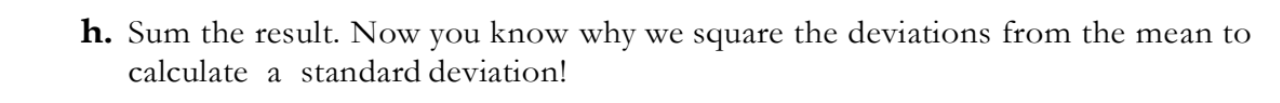
[1] 1.664765

>

> #2c Why are the means and and SD's not exactly the same as defined in the function?

> #FILL IN LATER





> #3 Vector Operations

> #3a, 3b, 3c Input the vectors into R

> weightINDIV\_kg <- c(60, 72, 57, 90, 95, 72)

> heightINDIV\_m <- c(1.80, 1.85, 1.72, 1.90, 1.74, 1.91)

> weightINDIV\_kg

[1] 60 72 57 90 95 72

> heightINDIV\_m

[1] 1.80 1.85 1.72 1.90 1.74 1.91

>

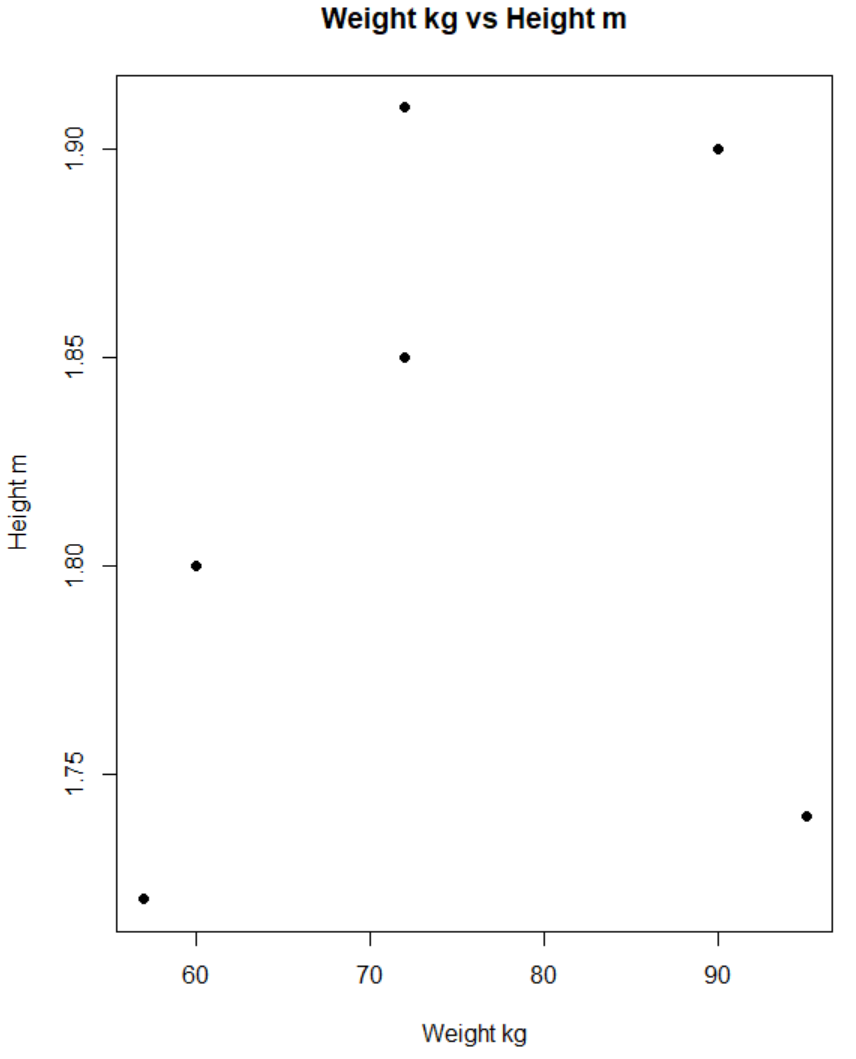
> #3d create scatter plot weight vs height

> plot(weightINDIV\_kg, heightINDIV\_m, main="Weight kg vs Height m",

+ xlab="Weight kg", ylab="Height m", pch=19)

> #Interpretation

> #As the weight of the individual increases so does the induvial height, with the exception of the induvial who weighs 95kg



> #3e calculate the BMI for each individual weight/height^2

> BMI <- weightINDIV\_kg/heightINDIV\_m^2

> BMI

[1] 18.51852 21.03725 19.26717 24.93075 31.37799 19.73630

>

> #3f Calcuate mean for weight

> meanWeight\_kg <- mean(weightINDIV\_kg)

> meanWeight\_kg

[1] 74.33333

>

> #3g Subtract mean weight from weight

> Weight\_Minuts\_MeanWeight <- weightINDIV\_kg-meanWeight\_kg

> Weight\_Minuts\_MeanWeight

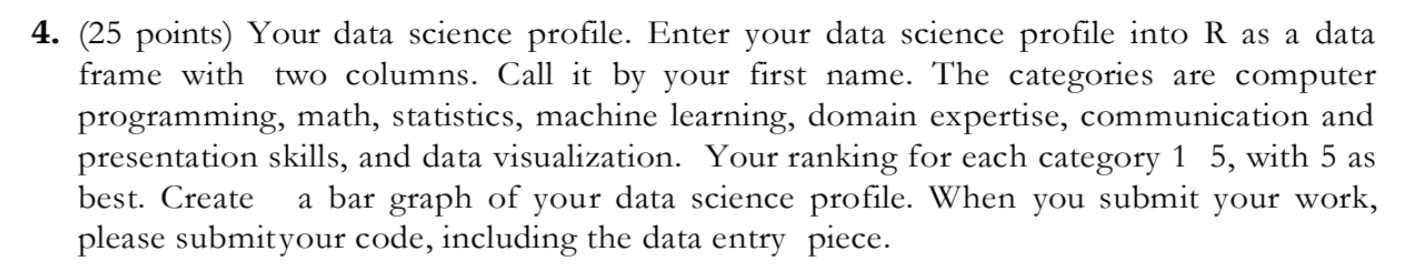
[1] -14.333333 -2.333333 -17.333333 15.666667 20.666667 -2.333333

>

> #3h Sum results from 3g

> sum(Weight\_Minuts\_MeanWeight)

[1] 2.842171e-14



> #4 Enter Data Science Profile as a data frame

> Category = c("Programming", "Math", "Statistics", "Machine Learning",

+ "Domain Expertise", "Communication and Presentation Skills",

+ "Data Visualization")

> Ranking = c(2, 2, 2, 1, 4, 4, 4)

>

> JamesVasquez = data.frame(Category, Ranking)

> JamesVasquez

Category Ranking

1 Programming 2

2 Math 2

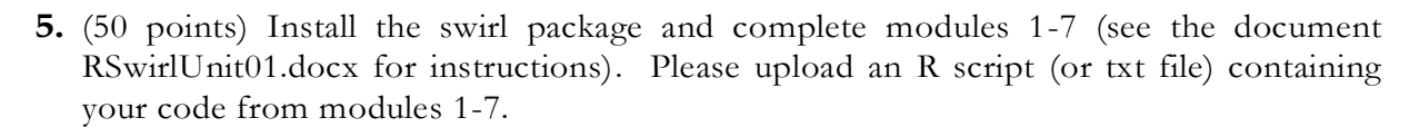
3 Statistics 2

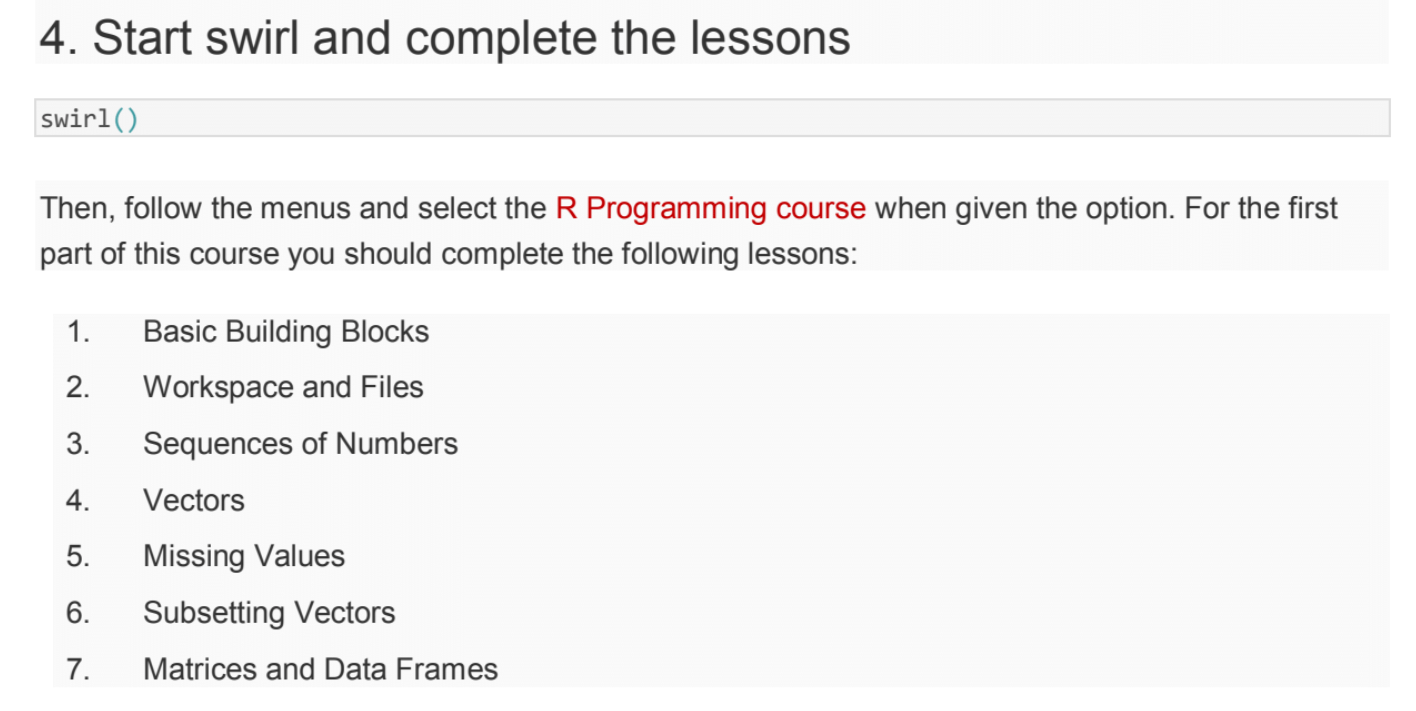
4 Machine Learning 1

5 Domain Expertise 4

6 Communication and Presentation Skills 4

7 Data Visualization 4







Selection: 1

| | 0%

| In this lesson, we will explore some basic building blocks of the R programming language.

...

|=== | 3%

| If at any point you'd like more information on a particular topic related to R, you can type help.start() at

| the prompt, which will open a menu of resources (either within RStudio or your default web browser, depending

| on your setup). Alternatively, a simple web search often yields the answer you're looking for.

...

|===== | 5%

| In its simplest form, R can be used as an interactive calculator. Type 5 + 7 and press Enter.

> 5+7

[1] 12

| That's a job well done!

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

| R simply prints the result of 12 by default. However, R is a programming language and often the reason we use

| a programming language as opposed to a calculator is to automate some process or avoid unnecessary repetition.

...

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

| In this case, we may want to use our result from above in a second calculation. Instead of retyping 5 + 7

| every time we need it, we can just create a new variable that stores the result.

...

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

| The way you assign a value to a variable in R is by using the assignment operator, which is just a 'less than'

| symbol followed by a 'minus' sign. It looks like this: <-

...

|================ | 16%

| Think of the assignment operator as an arrow. You are assigning the value on the right side of the arrow to

| the variable name on the left side of the arrow.

...

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

| To assign the result of 5 + 7 to a new variable called x, you type x <- 5 + 7. This can be read as 'x gets 5

| plus 7'. Give it a try now.

> x <- 5+7

| You are doing so well!

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

| You'll notice that R did not print the result of 12 this time. When you use the assignment operator, R assumes

| that you don't want to see the result immediately, but rather that you intend to use the result for something

| else later on.

...

|======================== | 24%

| To view the contents of the variable x, just type x and press Enter. Try it now.

> x

[1] 12

| You got it!

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

| Now, store the result of x - 3 in a new variable called y.

> y <- x-3

| You nailed it! Good job!

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

| What is the value of y? Type y to find out.

> y

[1] 9

| All that practice is paying off!

|================================= | 32%

| Now, let's create a small collection of numbers called a vector. Any object that contains data is called a

| data structure and numeric vectors are the simplest type of data structure in R. In fact, even a single number

| is considered a vector of length one.

...

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

| The easiest way to create a vector is with the c() function, which stands for 'concatenate' or 'combine'. To

| create a vector containing the numbers 1.1, 9, and 3.14, type c(1.1, 9, 3.14). Try it now and store the result

| in a variable called z.

> z <- c(1.1, 9, 3.14)

| You're the best!

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

| Anytime you have questions about a particular function, you can access R's built-in help files via the `?`

| command. For example, if you want more information on the c() function, type ?c without the parentheses that

| normally follow a function name. Give it a try.

> ?c

| You are quite good my friend!

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

| Type z to view its contents. Notice that there are no commas separating the values in the output.

> z

[1] 1.10 9.00 3.14

| That's a job well done!

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

| You can combine vectors to make a new vector. Create a new vector that contains z, 555, then z again in that

| order. Don't assign this vector to a new variable, so that we can just see the result immediately.

> c(z, 55)

[1] 1.10 9.00 3.14 55.00

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

| Type c(z, 555, z). Don't create a new variable. We just want to view the result.

> c(z, 55, z)

[1] 1.10 9.00 3.14 55.00 1.10 9.00 3.14

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

| Type c(z, 555, z). Don't create a new variable. We just want to view the result.

> c(z, 555, z)

[1] 1.10 9.00 3.14 555.00 1.10 9.00 3.14

| You are quite good my friend!

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

| Numeric vectors can be used in arithmetic expressions. Type the following to see what happens: z \* 2 + 100.

> z\*2+100

[1] 102.20 118.00 106.28

| Excellent job!

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

| First, R multiplied each of the three elements in z by 2. Then it added 100 to each element to get the result

| you see above.

...

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

| Other common arithmetic operators are `+`, `-`, `/`, and `^` (where x^2 means 'x squared'). To take the square

| root, use the sqrt() function and to take the absolute value, use the abs() function.

...

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

| Take the square root of z - 1 and assign it to a new variable called my\_sqrt.

> my\_sqrt <- sqrt(z-1)

| Nice work!

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

| Before we view the contents of the my\_sqrt variable, what do you think it contains?

1: a vector of length 3

2: a single number (i.e a vector of length 1)

3: a vector of length 0 (i.e. an empty vector)

Selection: 2

| One more time. You can do it!

| Think about how R handled the other 'vectorized' operations: element-by-element.

1: a single number (i.e a vector of length 1)

2: a vector of length 3

3: a vector of length 0 (i.e. an empty vector)

Selection: 2

| You got it!

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

| Print the contents of my\_sqrt.

> my\_sqrt

[1] 0.3162278 2.8284271 1.4628739

| Nice work!

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

| As you may have guessed, R first subtracted 1 from each element of z, then took the square root of each

| element. This leaves you with a vector of the same length as the original vector z.

...

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

| Now, create a new variable called my\_div that gets the value of z divided by my\_sqrt.

> my\_div <- z/my\_sqrt

| All that practice is paying off!

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

| Which statement do you think is true?

1: my\_div is a single number (i.e a vector of length 1)

2: The first element of my\_div is equal to the first element of z divided by the first element of my\_sqrt, and so on...

3: my\_div is undefined

Selection: 2

| You are really on a roll!

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

| Go ahead and print the contents of my\_div.

> my\_div

[1] 3.478505 3.181981 2.146460

| Your dedication is inspiring!

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

| When given two vectors of the same length, R simply performs the specified arithmetic operation (`+`, `-`,

| `\*`, etc.) element-by-element. If the vectors are of different lengths, R 'recycles' the shorter vector until

| it is the same length as the longer vector.

...

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

| When we did z \* 2 + 100 in our earlier example, z was a vector of length 3, but technically 2 and 100 are each

| vectors of length 1.

...

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

| Behind the scenes, R is 'recycling' the 2 to make a vector of 2s and the 100 to make a vector of 100s. In

| other words, when you ask R to compute z \* 2 + 100, what it really computes is this: z \* c(2, 2, 2) + c(100,

| 100, 100).

...

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

| To see another example of how this vector 'recycling' works, try adding c(1, 2, 3, 4) and c(0, 10). Don't

| worry about saving the result in a new variable.

> c(1, 2, 3, 4) + c(0, 10)

[1] 1 12 3 14

| You are quite good my friend!

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

| If the length of the shorter vector does not divide evenly into the length of the longer vector, R will still

| apply the 'recycling' method, but will throw a warning to let you know something fishy might be going on.

...

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

| Try c(1, 2, 3, 4) + c(0, 10, 100) for an example.

> c(1, 2, 3, 4) + c(0, 10, 100)

[1] 1 12 103 4

Warning message:

In c(1, 2, 3, 4) + c(0, 10, 100) :

longer object length is not a multiple of shorter object length

| Excellent work!

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

| Before concluding this lesson, I'd like to show you a couple of time-saving tricks.

...

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

| Earlier in the lesson, you computed z \* 2 + 100. Let's pretend that you made a mistake and that you meant to

| add 1000 instead of 100. You could either re-type the expression, or...

...

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

| In many programming environments, the up arrow will cycle through previous commands. Try hitting the up arrow

| on your keyboard until you get to this command (z \* 2 + 100), then change 100 to 1000 and hit Enter. If the up

| arrow doesn't work for you, just type the corrected command.

>

>

> z\*2+100

[1] 102.20 118.00 106.28

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

| If your environment does not support the up arrow feature, then just type the corrected command to move on.

> z\*2+1000

[1] 1002.20 1018.00 1006.28

| Excellent job!

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

| Finally, let's pretend you'd like to view the contents of a variable that you created earlier, but you can't

| seem to remember if you named it my\_div or myDiv. You could try both and see what works, or...

...

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

| You can type the first two letters of the variable name, then hit the Tab key (possibly more than once). Most

| programming environments will provide a list of variables that you've created that begin with 'my'. This is

| called auto-completion and can be quite handy when you have many variables in your workspace. Give it a try.

| (If auto-completion doesn't work for you, just type my\_div and press Enter.)

>

> my\_div

[1] 3.478505 3.181981 2.146460

| You are quite good my friend!

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

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

1: Yes

2: No



Selection: 2

| | 0%

| In this lesson, you'll learn how to examine your local workspace in R and begin to explore the relationship

| between your workspace and the file system of your machine.

...

|=== | 3%

| Because different operating systems have different conventions with regards to things like file paths, the

| outputs of these commands may vary across machines.

...

|===== | 5%

| However it's important to note that R provides a common API (a common set of commands) for interacting with

| files, that way your code will work across different kinds of computers.

...

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

| Let's jump right in so you can get a feel for how these special functions work!

...

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

| Determine which directory your R session is using as its current working directory using getwd().

> getwd()

[1] "C:/Users/James/Documents"

| You are quite good my friend!

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

| List all the objects in your local workspace using ls().

> ls()

[1] "my\_div" "my\_sqrt" "x" "y" "z"

| You are doing so well!

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

| Some R commands are the same as their equivalents commands on Linux or on a Mac. Both Linux and Mac operating

| systems are based on an operating system called Unix. It's always a good idea to learn more about Unix!

...

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

| Assign 9 to x using x <- 9.

> x <- 9

| You are really on a roll!

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

| Now take a look at objects that are in your workspace using ls().

> ls()

[1] "my\_div" "my\_sqrt" "x" "y" "z"

| All that practice is paying off!

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

| List all the files in your working directory using list.files() or dir().

> dir()

[1] "Custom Office Templates" "desktop.ini" "My Music"

[4] "My Pictures" "My Videos" "R"

[7] "SMU" "SQL Server Management Studio" "Visual Studio 2015"

| That's correct!

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

| As we go through this lesson, you should be examining the help page for each new function. Check out the help

| page for list.files with the command ?list.files.

> ?list.files

| Perseverance, that's the answer.

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

| One of the most helpful parts of any R help file is the See Also section. Read that section for list.files.

| Some of these functions may be used in later portions of this lesson.

...

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

| Using the args() function on a function name is also a handy way to see what arguments a function can take.

...

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

| Use the args() function to determine the arguments to list.files().

> args(list.files)

function (path = ".", pattern = NULL, all.files = FALSE, full.names = FALSE,

recursive = FALSE, ignore.case = FALSE, include.dirs = FALSE,

no.. = FALSE)

NULL

| Your dedication is inspiring!

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

| Assign the value of the current working directory to a variable called "old.dir".

> old.dir <- getwd()

| You're the best!

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

| We will use old.dir at the end of this lesson to move back to the place that we started. A lot of query

| functions like getwd() have the useful property that they return the answer to the question as a result of the

| function.

...

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

| Use dir.create() to create a directory in the current working directory called "testdir".

> dir.create(testdir)

Error in dir.create(testdir) : object 'testdir' not found

> dir.create()

Error in dir.create() : argument "path" is missing, with no default

> dir.create(old.dir)

Warning message:

In dir.create(old.dir) : 'C:\Users\James\Documents' already exists

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

| Type dir.create("testdir") to create a directory in the current working directory called "testdir".

> dir.create("testdir")

| Nice work!

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

| We will do all our work in this new directory and then delete it after we are done. This is the R analog to

| "Take only pictures, leave only footprints."

...

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

| Set your working directory to "testdir" with the setwd() command.

> setwd("C:\Users\James\Documents\testdir")

Error: '\U' used without hex digits in character string starting ""C:\U"

> setwd("C:/Users/James/Documents/testdir")

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

| Use setwd("testdir") to set your working directory to "testdir".

> setwd("testdir")

Error in setwd("testdir") : cannot change working directory

> getwd()

[1] "C:/Users/James/Documents/testdir"

| Nice try, but that's not exactly what I was hoping for. Try again. Or, type info() for more options.

| Use setwd("testdir") to set your working directory to "testdir".

> setwd("testdir")

Error in setwd("testdir") : cannot change working directory

> setwd(old.dir)

| That's not the answer I was looking for, but try again. Or, type info() for more options.

| Use setwd("testdir") to set your working directory to "testdir".

> setwd("testdir")

| All that practice is paying off!

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

| In general, you will want your working directory to be someplace sensible, perhaps created for the specific

| project that you are working on. In fact, organizing your work in R packages using RStudio is an excellent

| option. Check out RStudio at http://www.rstudio.com/

...

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

| Create a file in your working directory called "mytest.R" using the file.create() function.

> file.create("mytest.R")

[1] TRUE

| That's the answer I was looking for.

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

| This should be the only file in this newly created directory. Let's check this by listing all the files in the

| current directory.

> ls()

[1] "my\_div" "my\_sqrt" "old.dir" "x" "y" "z"

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

| list.files() shows that the directory only contains mytest.R.

> dir()

[1] "mytest.R"

| Great job!

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

| Check to see if "mytest.R" exists in the working directory using the file.exists() function.

> file.exists(mytest.R)

Error in file.exists(mytest.R) : object 'mytest.R' not found

> file.exists("mytest.R")

[1] TRUE

| You're the best!

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

| These sorts of functions are excessive for interactive use. But, if you are running a program that loops

| through a series of files and does some processing on each one, you will want to check to see that each exists

| before you try to process it.

...

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

| Access information about the file "mytest.R" by using file.info().

> file.info("mytest.R")

size isdir mode mtime ctime atime exe

mytest.R 0 FALSE 666 2018-05-12 21:42:59 2018-05-12 21:42:59 2018-05-12 21:42:59 no

| You are really on a roll!

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

| You can use the $ operator --- e.g., file.info("mytest.R")$mode --- to grab specific items.

...

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

| Change the name of the file "mytest.R" to "mytest2.R" by using file.rename().

> file.rename("mytest.R", "mytest2.R")

[1] TRUE

| Your dedication is inspiring!

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

| Your operating system will provide simpler tools for these sorts of tasks, but having the ability to

| manipulate files programatically is useful. You might now try to delete mytest.R using

| file.remove('mytest.R'), but that won't work since mytest.R no longer exists. You have already renamed it.

...

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

| Make a copy of "mytest2.R" called "mytest3.R" using file.copy().

> file.copy("mytest2.R", "mytest3.R")

[1] TRUE

| You got it!

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

| You now have two files in the current directory. That may not seem very interesting. But what if you were

| working with dozens, or millions, of individual files? In that case, being able to programatically act on many

| files would be absolutely necessary. Don't forget that you can, temporarily, leave the lesson by typing play()

| and then return by typing nxt().

...

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

| Provide the relative path to the file "mytest3.R" by using file.path().

> ls()

[1] "my\_div" "my\_sqrt" "old.dir" "x" "y" "z"

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

| file.path("mytest3.R") works.

> file.path("mytest3.R")

[1] "mytest3.R"

| Excellent work!

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

| You can use file.path to construct file and directory paths that are independent of the operating system your

| R code is running on. Pass 'folder1' and 'folder2' as arguments to file.path to make a platform-independent

| pathname.

> file.path()

character(0)

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

| file.path("folder1", "folder2") works.

> file.path("folder1", "folder2")

[1] "folder1/folder2"

| You nailed it! Good job!

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

| Take a look at the documentation for dir.create by entering ?dir.create . Notice the 'recursive' argument. In

| order to create nested directories, 'recursive' must be set to TRUE.

> ?dir.create

| Excellent work!

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

| Create a directory in the current working directory called "testdir2" and a subdirectory for it called

| "testdir3", all in one command by using dir.create() and file.path().

> dir.create(file.path("testdir2", "testdir3"))

Warning message:

In dir.create(file.path("testdir2", "testdir3")) :

cannot create dir 'testdir2\testdir3', reason 'No such file or directory'

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

| dir.create(file.path('testdir2', 'testdir3'), recursive = TRUE) will do the trick. If you forgot the recursive argument, the command may have appeared to

| work, but it didn't create the nested directory.

> dir.create(file.path("testdir2", "testdir3"), recursive = TRUE)

| That's correct!

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

| Go back to your original working directory using setwd(). (Recall that we created the variable old.dir with the full path for the orginal working directory at

| the start of these questions.)

> setwd(old.dir)

| You are doing so well!

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

| It is often helpful to save the settings that you had before you began an analysis and then go back to them at the end. This trick is often used within

| functions; you save, say, the par() settings that you started with, mess around a bunch, and then set them back to the original values at the end. This isn't

| the same as what we have done here, but it seems similar enough to mention.

...

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

| After you finish this lesson delete the 'testdir' directory that you just left (and everything in it)

...

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

| Take nothing but results. Leave nothing but assumptions. That sounds like 'Take nothing but pictures. Leave nothing but footprints.' But it makes no sense!

| Surely our readers can come up with a better motto . . .

...

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

| In this lesson, you learned how to examine your R workspace and work with the file system of your machine from within R. Thanks for playing!

...

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

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

1: No

2: Yes

Selection: 1



Selection: 3

| | 0%

| In this lesson, you'll learn how to create sequences of numbers in R.

...

|======= | 4%

| The simplest way to create a sequence of numbers in R is by using the `:` operator. Type 1:20 to see how it works.

> 1:20

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

| That's the answer I was looking for.

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

| That gave us every integer between (and including) 1 and 20. We could also use it to create a sequence of real numbers. For example, try pi:10.

> pi:10

[1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593

| You are amazing!

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

| The result is a vector of real numbers starting with pi (3.142...) and increasing in increments of 1. The upper limit of 10 is never reached, since the next

| number in our sequence would be greater than 10.

...

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

| What happens if we do 15:1? Give it a try to find out.

> 15:1

[1] 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

| You got it!

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

| It counted backwards in increments of 1! It's unlikely we'd want this behavior, but nonetheless it's good to know how it could happen.

...

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

| Remember that if you have questions about a particular R function, you can access its documentation with a question mark followed by the function name:

| ?function\_name\_here. However, in the case of an operator like the colon used above, you must enclose the symbol in backticks like this: ?`:`. (NOTE: The

| backtick (`) key is generally located in the top left corner of a keyboard, above the Tab key. If you don't have a backtick key, you can use regular quotes.)

...

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

| Pull up the documentation for `:` now.

> ?":"

| You nailed it! Good job!

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

| Often, we'll desire more control over a sequence we're creating than what the `:` operator gives us. The seq() function serves this purpose.

...

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

| The most basic use of seq() does exactly the same thing as the `:` operator. Try seq(1, 20) to see this.

> seq(1,20)

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

| You nailed it! Good job!

|================================================================== | 43%

| This gives us the same output as 1:20. However, let's say that instead we want a vector of numbers ranging from 0 to 10, incremented by 0.5. seq(0, 10,

| by=0.5) does just that. Try it out.

> seq(0,10, by=0.5)

[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0

| You are really on a roll!

|======================================================================== | 48%

| Or maybe we don't care what the increment is and we just want a sequence of 30 numbers between 5 and 10. seq(5, 10, length=30) does the trick. Give it a shot

| now and store the result in a new variable called my\_seq.

> my\_seq <- seq(5, 10, length=30)

| You are doing so well!

|=============================================================================== | 52%

| To confirm that my\_seq has length 30, we can use the length() function. Try it now.

> length(my\_seq)

[1] 30

| You are doing so well!

|===================================================================================== | 57%

| Let's pretend we don't know the length of my\_seq, but we want to generate a sequence of integers from 1 to N, where N represents the length of the my\_seq

| vector. In other words, we want a new vector (1, 2, 3, ...) that is the same length as my\_seq.

...

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

| There are several ways we could do this. One possibility is to combine the `:` operator and the length() function like this: 1:length(my\_seq). Give that a

| try.

> 1:length(my\_seq)

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

| You nailed it! Good job!

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

| Another option is to use seq(along.with = my\_seq). Give that a try.

> seq(along.with=my\_seq)

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

| All that hard work is paying off!

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

| However, as is the case with many common tasks, R has a separate built-in function for this purpose called seq\_along(). Type seq\_along(my\_seq) to see it in

| action.

> seq\_along(my\_seq)

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

| Great job!

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

| There are often several approaches to solving the same problem, particularly in R. Simple approaches that involve less typing are generally best. It's also

| important for your code to be readable, so that you and others can figure out what's going on without too much hassle.

...

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

| If R has a built-in function for a particular task, it's likely that function is highly optimized for that purpose and is your best option. As you become a

| more advanced R programmer, you'll design your own functions to perform tasks when there are no better options. We'll explore writing your own functions in

| future lessons.

...

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

| One more function related to creating sequences of numbers is rep(), which stands for 'replicate'. Let's look at a few uses.

...

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

| If we're interested in creating a vector that contains 40 zeros, we can use rep(0, times = 40). Try it out.

> rep(0, times=40)

[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

| You are quite good my friend!

|========================================================================================================================================== | 91%

| If instead we want our vector to contain 10 repetitions of the vector (0, 1, 2), we can do rep(c(0, 1, 2), times = 10). Go ahead.

> rep(c(0,1,2), times=10)

[1] 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2

| Excellent job!

|================================================================================================================================================ | 96%

| Finally, let's say that rather than repeating the vector (0, 1, 2) over and over again, we want our vector to contain 10 zeros, then 10 ones, then 10 twos. We

| can do this with the `each` argument. Try rep(c(0, 1, 2), each = 10).

> rep(c(0,1,2), each=10)

[1] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2

| Excellent job!

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

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

1: Yes

2: No

Selection: 1



Selection: 4

| | 0%

| The simplest and most common data structure in R is the vector.

...

|==== | 3%

| Vectors come in two different flavors: atomic vectors and lists. An atomic vector contains exactly one data type, whereas a list may contain multiple data

| types. We'll explore atomic vectors further before we get to lists.

...

|======== | 5%

| In previous lessons, we dealt entirely with numeric vectors, which are one type of atomic vector. Other types of atomic vectors include logical, character,

| integer, and complex. In this lesson, we'll take a closer look at logical and character vectors.

...

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

| Logical vectors can contain the values TRUE, FALSE, and NA (for 'not available'). These values are generated as the result of logical 'conditions'. Let's

| experiment with some simple conditions.

...

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

| First, create a numeric vector num\_vect that contains the values 0.5, 55, -10, and 6.

> c(0.5, 55, -10, 6)

[1] 0.5 55.0 -10.0 6.0

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

| Recall that the c() function is used for creating a vector. If you forget how to use it, use ?c to access the help file. Don't forget to assign the result to

| a new variable called num\_vect.

> num\_vect <- c(0.5, 55, -10, 6)

| You are quite good my friend!

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

| Now, create a variable called tf that gets the result of num\_vect < 1, which is read as 'num\_vect is less than 1'.

> tf < c(num\_vect<1)

Error: object 'tf' not found

> tf <- c(num\_vect<1)

| One more time. You can do it! Or, type info() for more options.

| Use tf <- num\_vect < 1 to assign the result of num\_vect < 1 to a variable called tf.

> tf <- num\_vect<1

| Excellent work!

|======================== | 16%

| What do you think tf will look like?

1: a vector of 4 logical values

2: a single logical value

Selection: 2

| You almost had it, but not quite. Try again.

| Remember our lesson on vector arithmetic? The theme was that R performs many operations on an element-by-element basis. We called these 'vectorized'

| operations.

1: a single logical value

2: a vector of 4 logical values

Selection: 2

| You're the best!

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

| Print the contents of tf now.

> tf

[1] TRUE FALSE TRUE FALSE

| All that hard work is paying off!

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

| The statement num\_vect < 1 is a condition and tf tells us whether each corresponding element of our numeric vector num\_vect satisfies this condition.

...

|==================================== | 24%

| The first element of num\_vect is 0.5, which is less than 1 and therefore the statement 0.5 < 1 is TRUE. The second element of num\_vect is 55, which is greater

| than 1, so the statement 55 < 1 is FALSE. The same logic applies for the third and fourth elements.

...

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

| Let's try another. Type num\_vect >= 6 without assigning the result to a new variable.

> num\_vect >= 6

[1] FALSE TRUE FALSE TRUE

| You are amazing!

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

| This time, we are asking whether each individual element of num\_vect is greater than OR equal to 6. Since only 55 and 6 are greater than or equal to 6, the

| second and fourth elements of the result are TRUE and the first and third elements are FALSE.

...

|================================================ | 32%

| The `<` and `>=` symbols in these examples are called 'logical operators'. Other logical operators include `>`, `<=`, `==` for exact equality, and `!=` for

| inequality.

...

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

| If we have two logical expressions, A and B, we can ask whether at least one is TRUE with A | B (logical 'or' a.k.a. 'union') or whether they are both TRUE

| with A & B (logical 'and' a.k.a. 'intersection'). Lastly, !A is the negation of A and is TRUE when A is FALSE and vice versa.

...

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

| It's a good idea to spend some time playing around with various combinations of these logical operators until you get comfortable with their use. We'll do a

| few examples here to get you started.

...

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

| Try your best to predict the result of each of the following statements. You can use pencil and paper to work them out if it's helpful. If you get stuck, just

| guess and you've got a 50% chance of getting the right answer!

...

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

| (3 > 5) & (4 == 4)

1: TRUE

2: FALSE

Selection: 2

| You're the best!

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

| (TRUE == TRUE) | (TRUE == FALSE)

1: FALSE

2: TRUE

Selection: 2

| You are amazing!

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

| ((111 >= 111) | !(TRUE)) & ((4 + 1) == 5)

1: FALSE

2: TRUE

Selection: 2

| You are amazing!

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

| Don't worry if you found these to be tricky. They're supposed to be. Working with logical statements in R takes practice, but your efforts will be rewarded in

| future lessons (e.g. subsetting and control structures).

...

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

| Character vectors are also very common in R. Double quotes are used to distinguish character objects, as in the following example.

...

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

| Create a character vector that contains the following words: "My", "name", "is". Remember to enclose each word in its own set of double quotes, so that R

| knows they are character strings. Store the vector in a variable called my\_char.

> my\_char <- c("My", "name", "is")

| You are doing so well!

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

| Print the contents of my\_char to see what it looks like.

> my\_char

[1] "My" "name" "is"

| You got it right!

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

| Right now, my\_char is a character vector of length 3. Let's say we want to join the elements of my\_char together into one continuous character string (i.e. a

| character vector of length 1). We can do this using the paste() function.

...

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

| Type paste(my\_char, collapse = " ") now. Make sure there's a space between the double quotes in the `collapse` argument. You'll see why in a second.

> paste(my\_char, collapse = " ")

[1] "My name is"

| That's correct!

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

| The `collapse` argument to the paste() function tells R that when we join together the elements of the my\_char character vector, we'd like to separate them

| with single spaces.

...

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

| It seems that we're missing something.... Ah, yes! Your name!

...

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

| To add (or 'concatenate') your name to the end of my\_char, use the c() function like this: c(my\_char, "your\_name\_here"). Place your name in double quotes

| where I've put "your\_name\_here". Try it now, storing the result in a new variable called my\_name.

> my\_name <- c(my\_name, "James")

Error: object 'my\_name' not found

> my\_name <- c(my\_char, "James")

| You got it!

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

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

> my\_name

[1] "My" "name" "is" "James"

| You are quite good my friend!

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

| 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\_name, collapse = " ")

[1] "My name is James"

| You are doing so well!

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

| 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.

...

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

| 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] "Helloworld!"

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

| Enter paste("Hello", "world!", sep = " ") to join the two words "Hello" and "world", separated by a single space. There should be a single space between the

| double quotes in the `sep` argument to the paste() function.

> paste("Hello", "world!", sep = " ")

[1] "Hello world!"

| You are amazing!

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

| 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(c(1:3), "X","Y","Z", sep="")

[1] "1XYZ" "2XYZ" "3XYZ"

| Keep trying! Or, type info() for more options.

| Use paste(1:3, c("X", "Y", "Z"), sep = "") to see what happens when we join two vectors of equal length using paste().

> paste(1:3), c("X","Y","Z"), sep="")

Error: unexpected ',' in "paste(1:3),"

> paste(1:3, c("X","Y","Z"), sep="")

[1] "1X" "2Y" "3Z"

| You are quite good my friend!

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

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

...

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

| 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(LETTERS, 1:4, spe = "-")

[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 -" "P 4 -" "Q 1 -" "R 2 -" "S 3 -"

[20] "T 4 -" "U 1 -" "V 2 -" "W 3 -" "X 4 -" "Y 1 -" "Z 2 -"

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

| Type paste(LETTERS, 1:4, sep = "-") to see how R recycles the vector 1:4 to match the length of LETTERS. Notice we are using `-` as our separator this time

| instead of a single space.

> 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" "P-4" "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2" "W-3" "X-4" "Y-1" "Z-2"

| Excellent job!

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

| 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.

...

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

| Also worth noting is that the numeric vector 1:4 gets 'coerced' into a character vector by the paste() function.

...

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

| 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".

...

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

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

1: No

2: Yes



Selection: 5

| | 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.

...

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

Error in C(44, NA, 5, NA) : object not interpretable as a factor

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

| That's a job well done!

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

| Now, let's multiply x by 3.

> x\*3

[1] 132 NA 15 NA

| That's the answer I was looking for.

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

| That's a job well done!

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

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

> z <- rep(NA, 1000)

| You're the best!

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

| Excellent work!

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

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

| Excellent job!

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

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

> my\_na

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

[27] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE

[53] FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE TRUE FALSE FALSE TRUE FALSE TRUE TRUE FALSE FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE

[79] TRUE TRUE TRUE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE

| Your dedication is inspiring!

|============================================================================ | 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 NA NA NA NA NA NA NA NA NA NA NA NA NA NA

[53] 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 NA NA NA NA NA NA NA NA NA NA

| That's correct!

|=========================================================================================== | 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] 38

| You nailed it! Good job!

|================================================================================================================================ | 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] 0.48878129 1.37855415 1.07666235 -2.05914129 0.58612459 NA -0.42092499 0.03979359 1.83698778 NA 0.60676678 NA 0.03728251

[14] 0.95104871 NA 0.52591743 -1.05124103 -0.41013093 NA NA 1.52747520 -0.36614163 -0.53019280 NA -0.83863327 -0.50249939

[27] -0.46934857 NA -0.51717981 NA -1.75471482 NA 0.91685844 NA NA 1.84055371 NA -0.04762558 1.22086530

[40] 1.43401269 -2.00297475 NA 0.77948029 1.37887209 1.74917018 NA NA NA NA 0.76435608 -0.56318875 1.33030550

[53] 0.86547346 NA 0.60614335 0.34008410 -1.02904595 2.64331519 NA 0.64203818 0.36168257 NA NA 0.22089032 -0.32830167

[66] NA 0.77046271 NA NA -0.07652923 0.59848766 -1.06800847 -2.08513764 NA NA NA 0.74314899 -1.53911564

[79] NA NA NA -0.42164442 -0.09315526 NA -1.23692332 NA NA NA -1.21426281 -0.90408706 0.74423071

[92] -0.43200397 0.12060483 NA 0.31848658 0.42565314 0.30949103 -1.18020340 NA NA

| Keep up the great work!

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

[1] NaN

| You are really on a roll!

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

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

1: No

2: Yes



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] 0.51139087 0.34193562 NA 1.20520731 0.00372116 NA NA -0.29752650 -0.24037701 NA NA 2.69513024 NA

[14] NA -0.13120991 -1.05470311 0.06218725 NA -1.62898629 NA -1.00760108 NA -0.02580717 NA NA 0.28733798

[27] NA 0.50763600 NA -1.93138827 NA NA NA NA -1.98688919 -2.42953825 NA NA -0.26815808

[40] 0.67526207

| You are doing so well!

|============ | 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] 0.51139087 0.34193562 NA 1.20520731 0.00372116 NA NA -0.29752650 -0.24037701 NA

| All that hard work 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 TRUEs and FALSEs

2: A vector with no NAs

3: A vector of all NAs

4: A vector of length 0

Selection: 3

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

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

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

> x[is,na(x)]

Error in na(x) : could not find function "na"

> 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 got it!

|=================================== | 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)]

| Keep up the great work!

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

| Print y to the console.

> y

[1] 0.51139087 0.34193562 1.20520731 0.00372116 -0.29752650 -0.24037701 2.69513024 -0.13120991 -1.05470311 0.06218725 -1.62898629 -1.00760108 -0.02580717

[14] 0.28733798 0.50763600 -1.93138827 -1.98688919 -2.42953825 -0.26815808 0.67526207

| All that hard work is paying off!

|=========================================== | 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 the negative elements of y

2: A vector of TRUEs and FALSEs

3: A vector of all NAs

4: A vector of all the positive elements of y

5: A vector of length 0

Selection: 5

| Almost! Try again.

| The logical expression y > 0 will give us TRUE for each element of y that is positive. Based on that, what do you think y[y > 0] will return?

1: A vector of all the positive elements of y

2: A vector of all NAs

3: A vector of all the negative elements of y

4: A vector of length 0

5: A vector of TRUEs and FALSEs

Selection: 1

| You are doing so well!

|================================================== | 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] 0.51139087 0.34193562 1.20520731 0.00372116 2.69513024 0.06218725 0.28733798 0.50763600 0.67526207

| You are amazing!

|====================================================== | 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] 0.51139087 0.34193562 NA 1.20520731 0.00372116 NA NA NA NA 2.69513024 NA NA 0.06218725 NA

[15] NA NA NA NA 0.28733798 NA 0.50763600 NA NA NA NA NA NA NA

[29] 0.67526207

| All that hard work is paying off!

|========================================================== | 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] 0.51139087 0.34193562 1.20520731 0.00372116 2.69513024 0.06218725 0.28733798 0.50763600 0.67526207

| Nice work!

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

> c(x)-c(3,5,7)

[1] -2.488609 -4.658064 NA -1.794793 -4.996279 NA NA -5.297526 -7.240377 NA NA -4.304870 NA NA -7.131210

[16] -4.054703 -4.937813 NA -4.628986 NA -8.007601 NA -5.025807 NA NA -4.712662 NA -2.492364 NA -8.931388

[31] NA NA NA NA -6.986889 -9.429538 NA NA -7.268158 -2.324738

Warning message:

In c(x) - c(3, 5, 7) :

longer object length is not a multiple of shorter object length

| Not quite! 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.

> evect <- c(3,5,7)

| 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(x) - c(evect)

[1] -2.488609 -4.658064 NA -1.794793 -4.996279 NA NA -5.297526 -7.240377 NA NA -4.304870 NA NA -7.131210

[16] -4.054703 -4.937813 NA -4.628986 NA -8.007601 NA -5.025807 NA NA -4.712662 NA -2.492364 NA -8.931388

[31] NA NA NA NA -6.986889 -9.429538 NA NA -7.268158 -2.324738

Warning message:

In c(x) - c(evect) :

longer object length is not a multiple of shorter object length

| You almost had it, but not quite. 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)]

Error: unexpected '[' in "["

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

Error: unexpected '[' in "c(x) - ["

>

> [evect]

Error: unexpected '[' in "["

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

[1] NA 0.00372116 NA

| Not quite right, but keep trying. 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 0.00372116 NA

| You are quite good my friend!

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

| Excellent work!

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

| All that hard work is paying off!

|========================================================================================= | 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] 0.51139087 NA 1.20520731 0.00372116 NA NA -0.29752650 -0.24037701 NA 2.69513024 NA NA -0.13120991

[14] -1.05470311 0.06218725 NA -1.62898629 NA -1.00760108 NA -0.02580717 NA NA 0.28733798 NA 0.50763600

[27] NA -1.93138827 NA NA NA NA -1.98688919 -2.42953825 NA NA -0.26815808 0.67526207

| Excellent work!

|===================================================================================================== | 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] 0.51139087 NA 1.20520731 0.00372116 NA NA -0.29752650 -0.24037701 NA 2.69513024 NA NA -0.13120991

[14] -1.05470311 0.06218725 NA -1.62898629 NA -1.00760108 NA -0.02580717 NA NA 0.28733798 NA 0.50763600

[27] NA -1.93138827 NA NA NA NA -1.98688919 -2.42953825 NA NA -0.26815808 0.67526207

| Excellent job!

|========================================================================================================= | 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 up the great work!

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

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

|==================================================================================================================== | 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!

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

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

> c(11, 2, NA)

[1] 11 2 NA

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

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

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

| You're the best!

|============================================================================================================================ | 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 doing so well!

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

| 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

| That's the answer I was looking for.

|==================================================================================================================================== | 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: 3

| Not quite right, but keep trying.

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

1: vect[bar]

2: vect["2"]

3: vect["bar"]

Selection: 3

| Keep up the great work!

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

| Now, try it out.

> vect["bar"]

bar

2

| You are quite good my friend!

|=============================================================================================================================================== | 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 are amazing!

|=================================================================================================================================================== | 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: No

2: Yes



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)

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

| You are amazing!

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

| That's correct!

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

| Perseverance, that's the answer.

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

| You got it!

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

| You nailed it! Good job!

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

| You got it!

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

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

|================================================== | 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"

| That's a job well done!

|======================================================= | 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 <- my\_vector

| You are really on a roll!

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

| All that practice is paying off!

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

> my\_matrix2 <- as.matrix(1:20)

| Not quite, but you're learning! 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).

> play()

| Entering play mode. Experiment as you please, then type nxt() when you are ready to resume the lesson.

> nxt()

| Resuming lesson...

| 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(1:20, nrow = 4, ncol = 5, byrow = FALSE)

| Keep up the great work!

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

| Nice 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 <- c("Bill", "Gina", "Kelly", "Sean")

| You got it right!

|======================================================================================== | 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 are really on a roll!

|============================================================================================ | 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 nailed it! Good job!

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

| You are really on a roll!

|================================================================================================================= | 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"

| All that hard work is paying off!

|============================================================================================================================== | 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 are really on a roll!

|========================================================================================================================================== | 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) <- cnames

| Perseverance, that's the answer.

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

| Great job!

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