> file.create("mytest.R")

[1] TRUE

| Your dedication is inspiring!

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

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

| files in the current directory.

> list()

list()

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

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

> list.files()

[1] "mytest.R"

| Great job!

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

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

> skip()

| Entering the following correct answer for you...

> file.exists("mytest.R")

[1] TRUE

| You got it right!

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

| 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.

...

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

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

> skip(

+

+ skip()

+

+

+

+

+

+ skip()

Error: unexpected symbol in:

"

skip"

> skip()

| Entering the following correct answer for you...

> file.info("mytest.R")

size isdir mode mtime ctime atime exe

mytest.R 0 FALSE 666 2016-04-12 14:17:50 2016-04-12 14:17:50 2016-04-12 14:17:50 no

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

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

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

...

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

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

> skip()

| Entering the following correct answer for you...

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

[1] TRUE

| You are doing so well!

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

| 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.

...

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

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

> skip()

| Entering the following correct answer for you...

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

[1] TRUE

| Your dedication is inspiring!

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

| 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().

...s

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

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

> skip()

| Entering the following correct answer for you...

> file.path("mytest3.R")

[1] "mytest3.R"

| You are doing so well!

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

| 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.

> skip()

| Entering the following correct answer for you...

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

[1] "folder1/folder2"

| You are amazing!

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

| 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.

> skip()

| Entering the following correct answer for you...

> ?dir.create

| All that hard work is paying off!

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

| 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().

> skip()

| Entering the following correct answer for you...

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

| That's the answer I was looking for.

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

| To delete a directory you need to use the recursive = TRUE argument with the function unlink(). If

| you don't use recursive = TRUE, R is concerned that you're unaware that you're deleting a directory

| and all of its contents. R reasons that, if you don't specify that recursive equals TRUE, you don't

| know that something is in the directory you're trying to delete. R tries to prevent you from making

| a mistake.

...skip()

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

| Delete the "testdir2" directory that you created by using unlink().

> skip(0)

Error in skip(0) : unused argument (0)

> skip()

| Entering the following correct answer for you...

> unlink("testdir2", recursive = TRUE)

| That's correct!

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

| Why is this command named "unlink" rather than something more sensible like "dir.delete" or

| "dir.remove"? Mainly, history. unlink is the traditional Unix command for removing directories.

...

|================================================================================== | 88%

| 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

Error in setwd() <- old.dir : invalid (NULL) left side of assignment

> skip()

| Entering the following correct answer for you...

> setwd(old.dir)

| Keep up the great work!

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

...

|====================================================================================== | 93%

| Delete the 'testdir' directory that you just left (and everything in it)

> unlink("testdir",recursive = TRUE)

| Great job!

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

...

|=========================================================================================== | 98%

| 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: 2

What is your email address? tanishnoah@gmail.com

What is your assignment token? DWBc0Epjv52nZ6c6

Grade submission succeeded!

| Great job!

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

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

1: R Programming

2: Take me to the swirl course repository!

Selection:

Enter an item from the menu, or 0 to exit

Selection: 1

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

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

4: Vectors 5: Missing Values 6: Subsetting Vectors

7: Matrices and Data Frames 8: Logic 9: Functions

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

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

Selection: 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 quite good my friend!

|============ | 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're the best!

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

> `:`

.Primitive(":")

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

| In order to view the documentation for a symbol like the colon operator, you have to use backticks

| (or quotes). This is so R knows you are not attempting to use the symbol in the command. Here's what

| it looks like: ?`:`. Don't forget the question mark out front.

> ?`:`

| All that practice is paying off!

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

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

| Type seq(1, 20) and press Enter. The space after the comma is not required, but is recommended as it

| tends to make your code appear less cluttered.

> seq(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 correct!

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

[20] 9.5 10.0

| You are quite good my friend!

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

> seq(5,10,length=30)

[1] 5.000000 5.172414 5.344828 5.517241 5.689655 5.862069 6.034483 6.206897 6.379310

[10] 6.551724 6.724138 6.896552 7.068966 7.241379 7.413793 7.586207 7.758621 7.931034

[19] 8.103448 8.275862 8.448276 8.620690 8.793103 8.965517 9.137931 9.310345 9.482759

[28] 9.655172 9.827586 10.000000

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

| You're using the same function here, but changing its arguments for different results. Be sure to

| store the result in a new variable called my\_seq, like this: my\_seq <- seq(5, 10, length=30).

> my\_seqseq(5,10,length=30)

Error: could not find function "my\_seqseq"

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

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

| You're using the same function here, but changing its arguments for different results. Be sure to

| store the result in a new variable called my\_seq, like this: my\_seq <- seq(5, 10, length=30).

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

| All that practice is paying off!

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

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

> length(my\_seq)

[1] 30

| Your dedication is inspiring!

|===================================================== | 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 got it!

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

| That's a job well done!

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

| Your dedication is inspiring!

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

...1:length(my\_seq)

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

| Your dedication is inspiring!

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

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

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

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

1: No

2: Yes

Selection: 2

What is your email address? tanishnoah@gmail.com

What is your assignment token? n5xQXS9N6zga1tBk

Grade submission succeeded!

| Nice work!

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

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

1: R Programming

2: Take me to the swirl course repository!

Selection:

Enter an item from the menu, or 0 to exit

Selection: 1

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

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

4: Vectors 5: Missing Values 6: Subsetting Vectors

7: Matrices and Data Frames 8: Logic 9: Functions

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

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

Selection: 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.

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

| That's the answer I was looking for.

|============ | 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<-num\_vect<1

| You nailed it! Good job!

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

| What do you think tf will look like?

1: a single logical value

2: a vector of 4 logical values

Selection: 2

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

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

| Print the contents of tf now.

> tf

[1] TRUE FALSE TRUE FALSE

| That's correct!

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

| Your dedication is inspiring!

|=========================== | 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: FALSE

2: TRUE

Selection: 1

| You got it right!

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

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

1: FALSE

2: TRUE

Selection: 2

| You are really on a roll!

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

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

1: TRUE

2: FALSE

Selection: 1

| Great job!

|============================================== | 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")

| All that hard work is paying off!

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

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

> my\_char

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

| You're the best!

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

| You got it!

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

> c(my\_char, "Tanish")

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

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

| Tack your name on to the end of the my\_char vector using the c() function. Be sure to assign the

| result to a new variable called my\_name. If your name was "Swirl", you would type my\_name <-

| c(my\_char, "Swirl").

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

| You are quite good my friend!

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

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

> my\_name

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

| You're the best!

|======================================================================= | 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] "MynameisTanish"

| You're close...I can feel it! Try it 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 Tanish"

| That's correct!

|========================================================================= | 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] "Hello world!"

| That's correct!

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

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

| Nice work!

|================================================================================= | 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, 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"

[17] "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2" "W-3" "X-4" "Y-1" "Z-2"

| All that practice is paying off!

|====================================================================================== | 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 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: 2

What is your email address? tanishnoah@gmail.com

What is your assignment token? rzLzWszz1IYeB8K3

Grade submission succeeded!

| Keep up the great work!

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

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

1: R Programming

2: Take me to the swirl course repository!

Selection:

Enter an item from the menu, or 0 to exit

Selection: 1

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

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

4: Vectors 5: Missing Values 6: Subsetting Vectors

7: Matrices and Data Frames 8: Logic 9: Functions

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

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

Selection: 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.

...save.image("C:/Users/Tanish/Desktop/Swirl\_assignments.RData")

|===== | 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.

...save.image("~/Swirl\_assignments.RData")

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

> save.image("C:/Users/Tanish/Desktop/swirl.r.RData")

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

| Assign the vector c(44, NA, 5, NA) to a variable x. The NA must uppercase.

> rm(list=ls())

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

| Assign the vector c(44, NA, 5, NA) to a variable x. The NA must uppercase.

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

| You are amazing!

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

| Now, let's multiply x by 3.

> x\*3

[1] 132 NA 15 NA

| You got it!

|=================== | 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).