As John Chambers, the creator of R once said:

|

| To understand computations in R, two slogans are helpful: 1. Everything that exists

| is an object. 2. Everything that happens is a function call.

**Notes from quiz 1 (for dataset visit week 1 folder)**

**Question 11**

In the dataset provided for this Quiz, what are the column names of the dataset?

**Answer**

Ozone, Solar.R, Wind, Temp, Month, Day

**Explanation**

You can get the column names of a data frame with the 'names()' function.

> hw1 = read.csv('hw1\_data.csv')

> names(hw1)

[1] "Ozone" "Solar.R" "Wind" "Temp" "Month" "Day"

**Question 12**

Extract the first 2 rows of the data frame and print them to the console. What does the output look like?

**Answer**

Ozone Solar.R Wind Temp Month Day

1 41 190 7.4 67 5 1

2 36 118 8.0 72 5 2

**Explantion**

You can extract the first two rows using the [ operator and an integer sequence to index the rows.

> hw1 = read.csv('hw1\_data.csv')

> hw1[c(1,2),]

Ozone Solar.R Wind Temp Month Day

1 41 190 7.4 67 5 1

2 36 118 8.0 72 5 2

**Question 13**

How many observations (i.e. rows) are in this data frame?

**Answer**

153

**Explanation**

You can use the 'nrows()' function to compute the number of rows in a data frame.

> hw1 = read.csv('hw1\_data.csv')

> nrow(hw1)

[1] 153

**Question 14**

Extract the last 2 rows of the data frame and print them to the console. What does the output look like?

**Answer**

Ozone Solar.R Wind Temp Month Day

152 18 131 8.0 76 9 29

153 20 223 11.5 68 9 30

**Explanation**

The 'tail()' function is an easy way to extract the last few elements of an R object.

> hw1 = read.csv('hw1\_data.csv')

> tail(hw1,2)

Ozone Solar.R Wind Temp Month Day

152 18 131 8.0 76 9 29

153 20 223 11.5 68 9 30

**Question 15**

What is the value of Ozone in the 47th row?

**Answer**

21

**Explanation**

The single bracket [ operator can be used to extract individual rows of a data frame.

> hw1 = read.csv('hw1\_data.csv')

> hw1[15,]

Ozone Solar.R Wind Temp Month Day

15 18 65 13.2 58 5 15

**Question 16**

How many missing values are in the Ozone column of this data frame?

**Answer**

37

**Explanation**

The 'is.na' function can be used to test for missing values.

> hw1 = read.csv('hw1\_data.csv')

> sub = subset(hw1, is.na(Ozone))

> nrow(sub)

[1] 37

**Question 17**

What is the mean of the Ozone column in this dataset? Exclude missing values (coded as NA) from this calculation.

**Answer**

42.1

**Explanation**

The 'mean' function can be used to calculate the mean.

> hw1 = read.csv('hw1\_data.csv')

> sub = subset(hw1, !is.na(Ozone), select = Ozone)

> apply(sub, 2, mean)

Ozone

42.12931

**Question 18**

Extract the subset of rows of the data frame where Ozone values are above 31 and Temp values are above 90. What is the mean of Solar.R in this subset?

**Answer**

212.8

**Explanation**

You need to construct a logical vector in R to match the question's requirements. Then use that logical vector to subset the data frame.

> hw1 = read.csv('hw1\_data.csv')

> sub = subset(hw1, Ozone > 31 & Temp > 90, select = Solar.R)

> apply(sub, 2, mean)

Solar.R

212.8

**Question 19**

What is the mean of "Temp" when "Month" is equal to 6?

**Answer**

79.1

**Explanation**

> hw1 = read.csv('hw1\_data.csv')

> sub = subset(hw1, Month == 6, select = Temp)

> apply(sub, 2, mean)

Temp

79.1

**Question 20**

What was the maximum ozone value in the month of May (i.e. Month = 5)?

**Answer**

115

**Explantion**

> hw1 = read.csv('hw1\_data.csv')

> sub = subset(hw1, Month == 5 & !is.na(Ozone), select = Ozone)

> apply(sub, 2, max)

Ozone

115

important functions from programming excercise of week 1.

my\_div<-z/my\_sqrt

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...

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 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.)

getwd()

ls()//to show the variable in the current workspace

list.files () or dir()//to show the files in current working directory

Using the args() function on a function name is also a handy way to see what

| arguments a function can take.

Use dir.create() to create a directory in the current working directory

| called "testdir".

setwd()

Create a file in your working directory called "mytest.R" using the

| file.create() function.

Check to see if "mytest.R" exists in the working directory using the

| file.exists() function.

> file.exists("mytest.R")

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

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

> ?file.rename

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

> file.create("mytest.R")

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

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

file.path("mytest3.R")

[1] "mytest3.R"

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("folder1","folder2")

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'),recursive = TRUE)

pi:10

[1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593

15:1

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

seq()

> 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 are doing so well!

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

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)

| Great job!

rep()

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

[39] 0 0

| All that practice is paying off!

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

| You are quite good my friend!

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

| You are amazing!

paste(vector\_name,collapse = " ") //collapse for giving white spaces between strings.

] "My" "name" "is" "karan"

| Excellent work!

| |===================================================== | 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 karan"

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

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

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"

[13] "M-1" "N-2" "O-3" "P-4" "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2" "W-3" "X-4"

[25] "Y-1" "Z-2"

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

| Also worth noting is that the numeric vector 1:4 gets 'coerced' into a

| character vector by the paste() function.

missing values

In R, NA is used to represent any value that is 'not available' or 'missing'

| (in the statistical sense)

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

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)

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

[26] 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

[51] 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

[76] 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

| All that hard work is paying off!

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

.

| |==================================================== | 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] 55

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

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

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 working like that and you'll get there!

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

| Print y to the console.

> y

[1] -0.23468771 -0.97188433 -0.38571359 -1.15507480 -0.16498612 -1.68307471

[7] 0.26215325 1.51317723 -0.19436366 0.41928785 0.80530895 0.68205582

[13] 1.15062178 0.39005893 0.03958592 0.51485046 0.14150663 -0.64447390

[19] -0.81412446 -1.06368987

| You are really on a roll!

You might wonder why we didn't just start with x[x > 0] to isolate the

| positive elements of x. Try that now to see why.

> x[x>0]

[1] NA NA NA NA NA NA

[7] NA NA NA NA NA 0.26215325

[13] 1.51317723 0.41928785 NA 0.80530895 NA NA

[19] 0.68205582 1.15062178 0.39005893 NA 0.03958592 NA

[25] 0.51485046 0.14150663 NA NA NA NA

| You're the best!

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

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] 0.26215325 1.51317723 0.41928785 0.80530895 0.68205582 1.15062178

[7] 0.39005893 0.03958592 0.51485046 0.14150663

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.

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

| You got it!

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

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] -0.23468771 -0.97188433 NA NA NA NA

[7] NA NA NA NA NA -1.15507480

[13] -0.16498612 -1.68307471 NA 0.26215325 1.51317723 -0.19436366

[19] 0.41928785 NA 0.80530895 NA NA 0.68205582

[25] 1.15062178 0.39005893 NA 0.03958592 NA 0.51485046

[31] 0.14150663 NA NA -0.64447390 NA -0.81412446

[37] NA -1.06368987

| Keep up the great 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)]

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)

| You got it!

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

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

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

| that now.

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

| Excellent job!

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

| Then, we can add the `names` attribute to vect2 after the fact with

| names(vect2) <- c("foo", "bar", "norf"). Go ahead.

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

> vect["bar"]

bar

2

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

foo bar

11 2

dim(my\_vector)

[1] 4 5

| Excellent work!

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

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

| Try it now.

> attributes(my\_vect)

Error: object 'my\_vect' not found

> attributes(my\_vector)

$dim

[1] 4 5

ince 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".

> colnames(my\_data)<-cnames

| You are doing so well!

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

| That's the answer I was looking for.

funtion to calculate mean of all the colums of a matrix or dataframe and remove NA values by default

colmean<-function(x,removeNA = TRUE)//to remove na values by default,if false is passed then it wont //remove them

{

n<-ncol(x)

z<-1:n

for(i in seq\_len(ncol(x)))

{

y<-x[,i]

z[i]<-mean(y,na.rm = removeNA)

}

z

}

|  |
| --- |
| You can use the `&` operator to evaluate AND across a vector. The `&&`  | version of AND only evaluates the first member of a vector. Let's test both  | for practice. Type the expression TRUE & c(TRUE, FALSE, FALSE). |
|  |
| |  | | --- | |  | |

> TRUE & c(TRUE,FALSE,FALSE)

[1] TRUE FALSE FALSE

What happens in this case is that the left operand `TRUE` is recycled across

| every element in the vector of the right operand. This is the equivalent

| statement as c(TRUE, TRUE, TRUE) & c(TRUE, FALSE, FALSE).

...

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

| Now we'll type the same expression except we'll use the `&&` operator. Type

| the expression TRUE && c(TRUE, FALSE, FALSE).

> TRUE && c(TRUE,FALSE,FALSE)

[1] TRUE

The OR operator follows a similar set of rules. The `|` version of OR

| evaluates OR across an entire vector, while the `||` version of OR only

| evaluates the first member of a vector.

.

| The function identical() will return TRUE if the two R objects passed to it

| as arguments are identical. Try out the identical() function by typing:

| identical('twins', 'twins')

> identical("twins","twins")

[1] TRUE

You should also be aware of the xor() function, which takes two arguments.

| The xor() function stands for exclusive OR. If one argument evaluates to TRUE

| and one argument evaluates to FALSE, then this function will return TRUE,

| otherwise it will return FALSE. Try out the xor() function by typing: xor(5

| == 6, !FALSE)

> xor(5==6,!FALSE)

[1] TRUE

reate this vector by typing: ints <- sample(10)

> ints<- sample(10)

The vector `ints` is a random sampling of integers from 1 to 10 without

| replacement.

Use the which() function to find the indices of ints that are greater than 7.

> which(ints>7)

[1] 7 9 10

Like the which() function, the functions any() and all() take logical vectors

| as their argument. The any() function will return TRUE if one or more of the

| elements in the logical vector is TRUE. The all() function will return TRUE

| if every element in the logical vector is TRUE.

...

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

| Use the any() function to see if any of the elements of ints are less than

| zero.

> any(ints<0)

[1] FALSE

Use the all() function to see if all of the elements of ints are greater than

| zero.

> all(ints>0)

[1] TRUE

If you want to see the source code for any function, just type the function name

| without any arguments or parentheses. Let's try this out with the function you just

| created. Type: boring\_function to view its source code.

#1: sum() returns the sum of a vector.

# Ex: sum(c(1, 2, 3)) evaluates to 6

#1: You can use the modulus operator %% to find the remainder.

# Ex: 7 %% 4 evaluates to 3.

#

With all of this talk about arguments, you may be wondering if there is a way you

| can see a function's arguments (besides looking at the documentation). Thankfully,

| you can use the args() function! Type: args(remainder) to examine the arguments for

| the remainder function.

> args(remainder)

function (num, divisor = 2)

You may be surprised to learn that you can pass a function as an argument without

| first defining the passed function. Functions that are not named are appropriately

| known as anonymous functions.

As you can see the first argument of paste() is `...` which is referred to as an

| ellipsis or simply dot-dot-dot. The ellipsis allows an indefinite number of

| arguments to be passed into a function. In the case of paste() any number of

| strings can be passed as arguments and paste() will return all of the strings

| combined into one string.

paste (..., sep = " ", collapse = NULL)

#

# Notice that the ellipses is the first argument, and all other arguments after

# the ellipses have default values. This is a strict rule in R programming: all

# arguments after an ellipses must have default values.

|  |
| --- |
| telegram<-function(...)  + {  + paste("START",...,"STOP")  + }  > telegram("chal chaiya chaiya")  [1] "START chal chaiya chaiya STOP" |
|  |
| |  | | --- | | > | |

The syntax for creating new binary operators in R is unlike anything else in

# R, but it allows you to define a new syntax for your function. I would only

# recommend making your own binary operator if you plan on using it often!

#

# User-defined binary operators have the following syntax:

# %[whatever]%

# where [whatever] represents any valid variable name.

#

# Let's say I wanted to define a binary operator that multiplied two numbers and

# then added one to the product. An implementation of that operator is below:

#

# "%mult\_add\_one%" <- function(left, right){ # Notice the quotation marks!

# left \* right + 1

# }

#

# I could then use this binary operator like `4 %mult\_add\_one% 5` which would

# evaluate to 21.