Selection: 12

| | 0%

| Whenever you're working with a new dataset, the first thing you

| should do is look at it! What is the format of the data? What are

| the dimensions? What are the variable names? How are the variables

| stored? Are there missing data? Are there any flaws in the data?

...

|== | 4%

| This lesson will teach you how to answer these questions and more

| using R's built-in functions. We'll be using a dataset constructed

| from the United States Department of Agriculture's PLANTS Database

| (http://plants.usda.gov/adv\_search.html).

...

|===== | 8%

| I've stored the data for you in a variable called plants. Type

| ls() to list the variables in your workspace, among which should

| be plants.

> ls(plants)

[1] "Active\_Growth\_Period" "Duration"

[3] "Foliage\_Color" "pH\_Max"

[5] "pH\_Min" "Precip\_Max"

[7] "Precip\_Min" "Scientific\_Name"

[9] "Shade\_Tolerance" "Temp\_Min\_F"

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

| more options.

| Use ls() to list the variables in your workspace.

> ls()

[1] "%p%" "boring\_function" "cls\_list"

[4] "cls\_vect" "evaluate" "flag\_colors"

[7] "flag\_shapes" "flags" "ints"

[10] "mad\_libs" "my\_mean" "ok"

[13] "plants" "remainder" "shape\_mat"

[16] "telegram" "Temperature" "unique\_vals"

[19] "viewinfo"

| You are quite good my friend!

|======= | 12%

| Let's begin by checking the class of the plants variable with

| class(plants). This will give us a clue as to the overall

| structure of the data.

> class(plants)

[1] "data.frame"

| Great job!

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

| It's very common for data to be stored in a data frame. It is the

| default class for data read into R using functions like read.csv()

| and read.table(), which you'll learn about in another lesson.

...

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

| Since the dataset is stored in a data frame, we know it is

| rectangular. In other words, it has two dimensions (rows and

| columns) and fits neatly into a table or spreadsheet. Use

| dim(plants) to see exactly how many rows and columns we're dealing

| with.

> dim(plants)

[1] 5166 10

| You are quite good my friend!

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

| The first number you see (5166) is the number of rows

| (observations) and the second number (10) is the number of columns

| (variables).

...

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

| You can also use nrow(plants) to see only the number of rows. Try

| it out.

> nrow(plants)

[1] 5166

| You're the best!

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

| ... And ncol(plants) to see only the number of columns.

> ncol(plants)

[1] 10

| You got it right!

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

| If you are curious as to how much space the dataset is occupying

| in memory, you can use object.size(plants).

> object.size(plants)

644232 bytes

| That's a job well done!

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

| Now that we have a sense of the shape and size of the dataset,

| let's get a feel for what's inside. names(plants) will return a

| character vector of column (i.e. variable) names. Give it a shot.

> names(plants)

[1] "Scientific\_Name" "Duration"

[3] "Active\_Growth\_Period" "Foliage\_Color"

[5] "pH\_Min" "pH\_Max"

[7] "Precip\_Min" "Precip\_Max"

[9] "Shade\_Tolerance" "Temp\_Min\_F"

| Excellent job!

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

| We've applied fairly descriptive variable names to this dataset,

| but that won't always be the case. A logical next step is to peek

| at the actual data. However, our dataset contains over 5000

| observations (rows), so it's impractical to view the whole thing

| all at once.

...

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

| The head() function allows you to preview the top of the dataset. Give it a try with only one

| argument.

> head(plants)

Scientific\_Name Duration Active\_Growth\_Period Foliage\_Color pH\_Min pH\_Max

1 Abelmoschus <NA> <NA> <NA> NA NA

2 Abelmoschus esculentus Annual, Perennial <NA> <NA> NA NA

3 Abies <NA> <NA> <NA> NA NA

4 Abies balsamea Perennial Spring and Summer Green 4 6

5 Abies balsamea var. balsamea Perennial <NA> <NA> NA NA

6 Abutilon <NA> <NA> <NA> NA NA

Precip\_Min Precip\_Max Shade\_Tolerance Temp\_Min\_F

1 NA NA <NA> NA

2 NA NA <NA> NA

3 NA NA <NA> NA

4 13 60 Tolerant -43

5 NA NA <NA> NA

6 NA NA <NA> NA

| You are doing so well!

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

| Take a minute to look through and understand the output above. Each row is labeled with the

| observation number and each column with the variable name. Your screen is probably not wide enough to

| view all 10 columns side-by-side, in which case R displays as many columns as it can on each line

| before continuing on the next.

...

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

| By default, head() shows you the first six rows of the data. You can alter this behavior by passing

| as a second argument the number of rows you'd like to view. Use head() to preview the first 10 rows

| of plants.

> head(plants, 10)

Scientific\_Name Duration Active\_Growth\_Period Foliage\_Color pH\_Min pH\_Max

1 Abelmoschus <NA> <NA> <NA> NA NA

2 Abelmoschus esculentus Annual, Perennial <NA> <NA> NA NA

3 Abies <NA> <NA> <NA> NA NA

4 Abies balsamea Perennial Spring and Summer Green 4 6.0

5 Abies balsamea var. balsamea Perennial <NA> <NA> NA NA

6 Abutilon <NA> <NA> <NA> NA NA

7 Abutilon theophrasti Annual <NA> <NA> NA NA

8 Acacia <NA> <NA> <NA> NA NA

9 Acacia constricta Perennial Spring and Summer Green 7 8.5

10 Acacia constricta var. constricta Perennial <NA> <NA> NA NA

Precip\_Min Precip\_Max Shade\_Tolerance Temp\_Min\_F

1 NA NA <NA> NA

2 NA NA <NA> NA

3 NA NA <NA> NA

4 13 60 Tolerant -43

5 NA NA <NA> NA

6 NA NA <NA> NA

7 NA NA <NA> NA

8 NA NA <NA> NA

9 4 20 Intolerant -13

10 NA NA <NA> NA

| All that practice is paying off!

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

| The same applies for using tail() to preview the end of the dataset. Use tail() to view the last 15

| rows.

> tail(plants, 15)

Scientific\_Name Duration Active\_Growth\_Period Foliage\_Color pH\_Min pH\_Max

5152 Zizania <NA> <NA> <NA> NA NA

5153 Zizania aquatica Annual Spring Green 6.4 7.4

5154 Zizania aquatica var. aquatica Annual <NA> <NA> NA NA

5155 Zizania palustris Annual <NA> <NA> NA NA

5156 Zizania palustris var. palustris Annual <NA> <NA> NA NA

5157 Zizaniopsis <NA> <NA> <NA> NA NA

5158 Zizaniopsis miliacea Perennial Spring and Summer Green 4.3 9.0

5159 Zizia <NA> <NA> <NA> NA NA

5160 Zizia aptera Perennial <NA> <NA> NA NA

5161 Zizia aurea Perennial <NA> <NA> NA NA

5162 Zizia trifoliata Perennial <NA> <NA> NA NA

5163 Zostera <NA> <NA> <NA> NA NA

5164 Zostera marina Perennial <NA> <NA> NA NA

5165 Zoysia <NA> <NA> <NA> NA NA

5166 Zoysia japonica Perennial <NA> <NA> NA NA

Precip\_Min Precip\_Max Shade\_Tolerance Temp\_Min\_F

5152 NA NA <NA> NA

5153 30 50 Intolerant 32

5154 NA NA <NA> NA

5155 NA NA <NA> NA

5156 NA NA <NA> NA

5157 NA NA <NA> NA

5158 35 70 Intolerant 12

5159 NA NA <NA> NA

5160 NA NA <NA> NA

5161 NA NA <NA> NA

5162 NA NA <NA> NA

5163 NA NA <NA> NA

5164 NA NA <NA> NA

5165 NA NA <NA> NA

5166 NA NA <NA> NA

| You are really on a roll!

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

| After previewing the top and bottom of the data, you probably noticed lots of NAs, which are R's

| placeholders for missing values. Use summary(plants) to get a better feel for how each variable is

| distributed and how much of the dataset is missing.

> summary(plants)

Scientific\_Name Duration Active\_Growth\_Period

Abelmoschus : 1 Perennial :3031 Spring and Summer : 447

Abelmoschus esculentus : 1 Annual : 682 Spring : 144

Abies : 1 Annual, Perennial: 179 Spring, Summer, Fall: 95

Abies balsamea : 1 Annual, Biennial : 95 Summer : 92

Abies balsamea var. balsamea: 1 Biennial : 57 Summer and Fall : 24

Abutilon : 1 (Other) : 92 (Other) : 30

(Other) :5160 NA's :1030 NA's :4334

Foliage\_Color pH\_Min pH\_Max Precip\_Min Precip\_Max

Dark Green : 82 Min. :3.000 Min. : 5.100 Min. : 4.00 Min. : 16.00

Gray-Green : 25 1st Qu.:4.500 1st Qu.: 7.000 1st Qu.:16.75 1st Qu.: 55.00

Green : 692 Median :5.000 Median : 7.300 Median :28.00 Median : 60.00

Red : 4 Mean :4.997 Mean : 7.344 Mean :25.57 Mean : 58.73

White-Gray : 9 3rd Qu.:5.500 3rd Qu.: 7.800 3rd Qu.:32.00 3rd Qu.: 60.00

Yellow-Green: 20 Max. :7.000 Max. :10.000 Max. :60.00 Max. :200.00

NA's :4334 NA's :4327 NA's :4327 NA's :4338 NA's :4338

Shade\_Tolerance Temp\_Min\_F

Intermediate: 242 Min. :-79.00

Intolerant : 349 1st Qu.:-38.00

Tolerant : 246 Median :-33.00

NA's :4329 Mean :-22.53

3rd Qu.:-18.00

Max. : 52.00

NA's :4328

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

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

| summary() provides different output for each variable, depending on its class. For numeric data such

| as Precip\_Min, summary() displays the minimum, 1st quartile, median, mean, 3rd quartile, and maximum.

| These values help us understand how the data are distributed.

...

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

| For categorical variables (called 'factor' variables in R), summary() displays the number of times

| each value (or 'level') occurs in the data. For example, each value of Scientific\_Name only appears

| once, since it is unique to a specific plant. In contrast, the summary for Duration (also a factor

| variable) tells us that our dataset contains 3031 Perennial plants, 682 Annual plants, etc.

...

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

| You can see that R truncated the summary for Active\_Growth\_Period by including a catch-all category

| called 'Other'. Since it is a categorical/factor variable, we can see how many times each value

| actually occurs in the data with table(plants$Active\_Growth\_Period).

> table(plants$Active\_Growth\_Period)

Fall, Winter and Spring Spring Spring and Fall Spring and Summer

15 144 10 447

Spring, Summer, Fall Summer Summer and Fall Year Round

95 92 24 5

| Excellent job!

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

| Each of the functions we've introduced so far has its place in helping you to better understand the

| structure of your data. However, we've left the best for last....

...

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

| Perhaps the most useful and concise function for understanding the \*str\*ucture of your data is str().

| Give it a try now.

> str(plants)

'data.frame': 5166 obs. of 10 variables:

$ Scientific\_Name : Factor w/ 5166 levels "Abelmoschus",..: 1 2 3 4 5 6 7 8 9 10 ...

$ Duration : Factor w/ 8 levels "Annual","Annual, Biennial",..: NA 4 NA 7 7 NA 1 NA 7 7 ...

$ Active\_Growth\_Period: Factor w/ 8 levels "Fall, Winter and Spring",..: NA NA NA 4 NA NA NA NA 4 NA ...

$ Foliage\_Color : Factor w/ 6 levels "Dark Green","Gray-Green",..: NA NA NA 3 NA NA NA NA 3 NA ...

$ pH\_Min : num NA NA NA 4 NA NA NA NA 7 NA ...

$ pH\_Max : num NA NA NA 6 NA NA NA NA 8.5 NA ...

$ Precip\_Min : int NA NA NA 13 NA NA NA NA 4 NA ...

$ Precip\_Max : int NA NA NA 60 NA NA NA NA 20 NA ...

$ Shade\_Tolerance : Factor w/ 3 levels "Intermediate",..: NA NA NA 3 NA NA NA NA 2 NA ...

$ Temp\_Min\_F : int NA NA NA -43 NA NA NA NA -13 NA ...

| All that hard work is paying off!

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

| The beauty of str() is that it combines many of the features of the other functions you've already

| seen, all in a concise and readable format. At the very top, it tells us that the class of plants is

| 'data.frame' and that it has 5166 observations and 10 variables. It then gives us the name and class

| of each variable, as well as a preview of its contents.

...

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

| str() is actually a very general function that you can use on most objects in R. Any time you want to

| understand the structure of something (a dataset, function, etc.), str() is a good place to start.

...

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

| In this lesson, you learned how to get a feel for the structure and contents of a new dataset using a

| collection of simple and useful functions. Taking the time to do this upfront can save you time and

| frustration later on in your analysis.

...

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

Selection: 13

| | 0%

| One of the great advantages of using a statistical programming language like R is its vast collection

| of tools for simulating random numbers.

...

|=== | 3%

| This lesson assumes familiarity with a few common probability distributions, but these topics will

| only be discussed with respect to random number generation. Even if you have no prior experience with

| these concepts, you should be able to complete the lesson and understand the main ideas.

...

|====== | 6%

| The first function we'll use to generate random numbers is sample(). Use ?sample to pull up the

| documentation.

> ?sample

| All that practice is paying off!

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

| Let's simulate rolling four six-sided dice: sample(1:6, 4, replace = TRUE).

> sample(1:6, 4, replace = TRUE)

[1] 1 2 1 6

| You got it right!

|=========== | 12%

| Now repeat the command to see how your result differs. (The probability of rolling the exact same

| result is (1/6)^4 = 0.00077, which is pretty small!)

> sample(1:6, 4, replace = TRUE)

[1] 2 3 6 1

| Keep up the great work!

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

| sample(1:6, 4, replace = TRUE) instructs R to randomly select four numbers between 1 and 6, WITH

| replacement. Sampling with replacement simply means that each number is "replaced" after it is

| selected, so that the same number can show up more than once. This is what we want here, since what

| you roll on one die shouldn't affect what you roll on any of the others.

...

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

| Now sample 10 numbers between 1 and 20, WITHOUT replacement. To sample without replacement, simply

| leave off the 'replace' argument.

> sample(1:20, 10, repalce = FALSE)

Error in sample(1:20, 10, repalce = FALSE) :

unused argument (repalce = FALSE)

> sample(1:20, 10, replace = FALSE)

[1] 3 18 17 5 19 20 14 1 10 6

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

| Type sample(1:20, 10) to sample 10 numbers between 1 and 20, without replacement.

> sample(1:20, 10)

[1] 1 11 8 7 5 10 17 2 16 13

| You're the best!

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

| Since the last command sampled without replacement, no number appears more than once in the output.

...

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

| LETTERS is a predefined variable in R containing a vector of all 26 letters of the English alphabet.

| Take a look at it now.

> LETTERS

[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y"

[26] "Z"

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

|========================== | 27%

| The sample() function can also be used to permute, or rearrange, the elements of a vector. For

| example, try sample(LETTERS) to permute all 26 letters of the English alphabet.

> sample(LETTERS)

[1] "M" "E" "I" "Q" "A" "J" "P" "T" "R" "O" "N" "K" "G" "L" "D" "S" "X" "Y" "H" "F" "C" "W" "V" "Z" "B"

[26] "U"

| Excellent job!

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

| This is identical to taking a sample of size 26 from LETTERS, without replacement. When the 'size'

| argument to sample() is not specified, R takes a sample equal in size to the vector from which you

| are sampling.

...

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

| Now, suppose we want to simulate 100 flips of an unfair two-sided coin. This particular coin has a

| 0.3 probability of landing 'tails' and a 0.7 probability of landing 'heads'.

...

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

| Let the value 0 represent tails and the value 1 represent heads. Use sample() to draw a sample of

| size 100 from the vector c(0,1), with replacement. Since the coin is unfair, we must attach specific

| probabilities to the values 0 (tails) and 1 (heads) with a fourth argument, prob = c(0.3, 0.7).

| Assign the result to a new variable called flips.

> flips <- sample(c(0,1), 100, prob = c(0.3, 0.7))

Error in sample.int(length(x), size, replace, prob) :

cannot take a sample larger than the population when 'replace = FALSE'

> flips <- sample(c(0,1), 100, replace = TRUE, prob = c(0.3, 0.7))

| That's a job well done!

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

| View the contents of the flips variable.

> flips

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

[50] 0 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1

[99] 1 0

| You are quite good my friend!

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

| Since we set the probability of landing heads on any given flip to be 0.7, we'd expect approximately

| 70 of our coin flips to have the value 1. Count the actual number of 1s contained in flips using the

| sum() function.

> sum(flips > 0 )

[1] 77

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

| sum(flips) will add up all the 1s and 0s, giving you the total number of 1s in flips.

> sum(flips)

[1] 77

| You're the best!

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

| A coin flip is a binary outcome (0 or 1) and we are performing 100 independent trials (coin flips),

| so we can use rbinom() to simulate a binomial random variable. Pull up the documentation for rbinom()

| using ?rbinom.

> ?rbinom

| All that hard work is paying off!

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

| Each probability distribution in R has an r\*\*\* function (for "random"), a d\*\*\* function (for

| "density"), a p\*\*\* (for "probability"), and q\*\*\* (for "quantile"). We are most interested in the r\*\*\*

| functions in this lesson, but I encourage you to explore the others on your own.

...

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

| A binomial random variable represents the number of 'successes' (heads) in a given number of

| independent 'trials' (coin flips). Therefore, we can generate a single random variable that

| represents the number of heads in 100 flips of our unfair coin using rbinom(1, size = 100, prob =

| 0.7). Note that you only specify the probability of 'success' (heads) and NOT the probability of

| 'failure' (tails). Try it now.

> rbinom(1, size = 100, prob = 0.7)

[1] 63

| That's the answer I was looking for.

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

| Equivalently, if we want to see all of the 0s and 1s, we can request 100 observations, each of size

| 1, with success probability of 0.7. Give it a try, assigning the result to a new variable called

| flips2.

> flips2 <- rbinom(100, size = 1, prob = 0.7)

| That's the answer I was looking for.

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

| View the contents of flips2.

> flips2

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

[50] 0 1 1 1 1 1 0 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 0 1 1 1 1 1 0 0 0 1 1 1 1 1

[99] 0 1

| You got it!

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

| Now use sum() to count the number of 1s (heads) in flips2. It should be close to 70!

> sum(flips2)

[1] 67

| All that hard work is paying off!

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

| Similar to rbinom(), we can use R to simulate random numbers from many other probability

| distributions. Pull up the documentation for rnorm() now.

> 0

[1] 0

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

| Type ?rnorm to view its help file.

> ?rnorm

| You are doing so well!

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

| The standard normal distribution has mean 0 and standard deviation 1. As you can see under the

| 'Usage' section in the documentation, the default values for the 'mean' and 'sd' arguments to rnorm()

| are 0 and 1, respectively. Thus, rnorm(10) will generate 10 random numbers from a standard normal

| distribution. Give it a try.

> rnorm(10)

[1] 0.3882533 -1.4618191 1.3195660 1.0066032 -1.2094389 -0.4953272 -0.6482444 1.1417324 0.2650164

[10] 1.1884557

| Nice work!

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

| Now do the same, except with a mean of 100 and a standard deviation of 25.

> rnorm(10, mean = 100, sd = 25)

[1] 132.32248 121.54032 148.75896 52.31213 55.11822 92.17108 178.52974 66.19502 69.70011 98.67914

| Nice work!

|==================================================================== | 73%

| Finally, what if we want to simulate 100 \*groups\* of random numbers, each containing 5 values

| generated from a Poisson distribution with mean 10? Let's start with one group of 5 numbers, then

| I'll show you how to repeat the operation 100 times in a convenient and compact way.

...

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

| Generate 5 random values from a Poisson distribution with mean 10. Check out the documentation for

| rpois() if you need help.

> ?rpois

>

> rpois(5, 10)

[1] 7 9 6 10 13

| You are amazing!

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

| Now use replicate(100, rpois(5, 10)) to perform this operation 100 times. Store the result in a new

| variable called my\_pois.

> my\_pois <- replicate(100, rpois(5, 10))

| Great job!

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

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

> my\_pois

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

[1,] 5 13 6 9 13 8 8 8 14 10 3 7 10 12 6 10 9 6

[2,] 5 10 12 10 7 12 10 9 10 12 9 8 10 12 11 9 8 8

[3,] 8 9 10 6 10 10 7 9 22 5 16 8 8 9 14 11 8 10

[4,] 10 15 5 8 9 7 8 17 8 5 9 9 10 20 6 16 8 10

[5,] 9 8 12 19 9 8 14 11 10 10 10 7 10 4 9 10 5 9

[,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26] [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34]

[1,] 6 9 10 12 15 5 7 6 12 15 7 11 9 12 10 8

[2,] 14 10 9 10 15 12 16 11 15 15 7 9 14 5 13 5

[3,] 10 10 12 9 12 11 17 8 6 9 12 17 10 7 13 11

[4,] 10 11 9 12 14 9 14 9 9 8 10 14 6 6 16 6

[5,] 11 18 8 9 11 12 13 14 10 11 7 13 7 8 6 6

[,35] [,36] [,37] [,38] [,39] [,40] [,41] [,42] [,43] [,44] [,45] [,46] [,47] [,48] [,49] [,50]

[1,] 13 7 10 6 4 10 11 9 9 5 10 7 6 8 11 11

[2,] 10 14 9 10 10 6 6 9 7 7 13 10 11 5 9 9

[3,] 9 10 7 12 8 14 10 8 13 8 15 11 10 16 9 9

[4,] 9 16 6 7 7 13 14 11 13 12 8 9 10 11 11 23

[5,] 6 8 8 7 11 8 8 7 15 11 13 13 5 12 15 10

[,51] [,52] [,53] [,54] [,55] [,56] [,57] [,58] [,59] [,60] [,61] [,62] [,63] [,64] [,65] [,66]

[1,] 7 5 15 8 15 18 16 6 10 10 8 13 9 6 10 16

[2,] 14 10 12 11 12 4 11 5 9 12 7 16 8 9 13 6

[3,] 14 12 5 10 8 8 13 11 15 10 8 8 6 9 9 10

[4,] 12 15 10 8 14 11 7 12 9 13 9 8 9 9 7 6

[5,] 9 10 9 6 10 8 8 11 10 10 9 9 5 8 7 7

[,67] [,68] [,69] [,70] [,71] [,72] [,73] [,74] [,75] [,76] [,77] [,78] [,79] [,80] [,81] [,82]

[1,] 7 5 7 10 4 4 10 13 8 11 12 5 7 13 9 14

[2,] 5 12 14 7 13 13 9 9 9 8 15 11 8 7 12 11

[3,] 6 8 10 11 14 8 4 8 16 6 14 9 9 14 10 10

[4,] 8 9 9 11 5 12 13 10 8 10 15 13 10 10 16 8

[5,] 10 14 7 13 7 12 11 9 10 5 16 11 12 4 19 12

[,83] [,84] [,85] [,86] [,87] [,88] [,89] [,90] [,91] [,92] [,93] [,94] [,95] [,96] [,97] [,98]

[1,] 13 7 9 5 11 13 11 7 11 8 7 13 16 5 11 17

[2,] 10 13 8 11 9 12 7 12 11 10 11 8 12 9 10 12

[3,] 14 14 14 12 7 8 14 6 3 11 11 12 13 16 11 15

[4,] 5 17 2 11 10 10 12 18 8 9 11 10 6 7 13 10

[5,] 13 5 15 8 8 12 13 6 7 9 13 13 9 10 10 8

[,99] [,100]

[1,] 14 5

[2,] 14 11

[3,] 11 13

[4,] 15 11

[5,] 6 7

| Excellent work!

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

| replicate() created a matrix, each column of which contains 5 random numbers generated from a Poisson

| distribution with mean 10. Now we can find the mean of each column in my\_pois using the colMeans()

| function. Store the result in a variable called cm.

> cm <- colMeans(my\_pois)

| Keep up the great work!

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

| And let's take a look at the distribution of our column means by plotting a histogram with hist(cm).

> histogram(cm)

Error in histogram(cm) : could not find function "histogram"

> hist(cm)

| Great job!

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

| Looks like our column means are almost normally distributed, right? That's the Central Limit Theorem

| at work, but that's a lesson for another day!

...

|======================================================================================== | 94%

| All of the standard probability distributions are built into R, including exponential (rexp()),

| chi-squared (rchisq()), gamma (rgamma()), .... Well, you see the pattern.

...

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

| Simulation is practically a field of its own and we've only skimmed the surface of what's possible. I

| encourage you to explore these and other functions further on your own.

...

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

Selection: 14

| | 0%

| R has a special way of representing dates and times, which can be helpful if you're working with data that

| show how something changes over time (i.e. time-series data) or if your data contain some other temporal

| information, like dates of birth.

...

|=== | 3%

| Dates are represented by the 'Date' class and times are represented by the 'POSIXct' and 'POSIXlt'

| classes. Internally, dates are stored as the number of days since 1970-01-01 and times are stored as

| either the number of seconds since 1970-01-01 (for 'POSIXct') or a list of seconds, minutes, hours, etc.

| (for 'POSIXlt').

...

|====== | 6%

| Let's start by using d1 <- Sys.Date() to get the current date and store it in the variable d1. (That's the

| letter 'd' and the number 1.)

> d1 <- Sys.Date()

| That's a job well done!

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

| Use the class() function to confirm d1 is a Date object.

> class(d1)

[1] "Date"

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

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

| We can use the unclass() function to see what d1 looks like internally. Try it out.

> unclass(d1)

[1] 17560

| All that practice is paying off!

|============== | 14%

| That's the exact number of days since 1970-01-01!

...

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

| However, if you print d1 to the console, you'll get today's date -- YEAR-MONTH-DAY. Give it a try.

> d1

[1] "2018-01-29"

| Excellent job!

|=================== | 19%

| What if we need to reference a date prior to 1970-01-01? Create a variable d2 containing

| as.Date("1969-01-01").

> d2 <- as.Date("1969-01-01")

| Keep up the great work!

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

| Now use unclass() again to see what d2 looks like internally.

> unclass(d2)

[1] -365

| You are amazing!

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

| As you may have anticipated, you get a negative number. In this case, it's -365, since 1969-01-01 is

| exactly one calendar year (i.e. 365 days) BEFORE 1970-01-01.

...

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

| Now, let's take a look at how R stores times. You can access the current date and time using the

| Sys.time() function with no arguments. Do this and store the result in a variable called t1.

> t1 <- Sys.time()

| All that practice is paying off!

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

| View the contents of t1.

> t1

[1] "2018-01-29 21:11:23 CST"

| That's a job well done!

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

| And check the class() of t1.

> class(t1)

[1] "POSIXct" "POSIXt"

| You are doing so well!

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

| As mentioned earlier, POSIXct is just one of two ways that R represents time information. (You can ignore

| the second value above, POSIXt, which just functions as a common language between POSIXct and POSIXlt.)

| Use unclass() to see what t1 looks like internally -- the (large) number of seconds since the beginning of

| 1970.

> unclass(t1)

[1] 1517281883

| That's a job well done!

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

| By default, Sys.time() returns an object of class POSIXct, but we can coerce the result to POSIXlt with

| as.POSIXlt(Sys.time()). Give it a try and store the result in t2.

> t2 <- as.POSIXlt(Sys.time())

| You are doing so well!

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

| Check the class of t2.

> class(t2)

[1] "POSIXlt" "POSIXt"

| Nice work!

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

| Now view its contents.

> t2

[1] "2018-01-29 21:15:58 CST"

| All that hard work is paying off!

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

| The printed format of t2 is identical to that of t1. Now unclass() t2 to see how it is different

| internally.

> unclass(t2)

$sec

[1] 58.7454

$min

[1] 15

$hour

[1] 21

$mday

[1] 29

$mon

[1] 0

$year

[1] 118

$wday

[1] 1

$yday

[1] 28

$isdst

[1] 0

$zone

[1] "CST"

$gmtoff

[1] -21600

attr(,"tzone")

[1] "" "CST" "CDT"

| Excellent job!

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

| t2, like all POSIXlt objects, is just a list of values that make up the date and time. Use

| str(unclass(t2)) to have a more compact view.

> str(unclass(t2))

List of 11

$ sec : num 58.7

$ min : int 15

$ hour : int 21

$ mday : int 29

$ mon : int 0

$ year : int 118

$ wday : int 1

$ yday : int 28

$ isdst : int 0

$ zone : chr "CST"

$ gmtoff: int -21600

- attr(\*, "tzone")= chr [1:3] "" "CST" "CDT"

| That's correct!

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

| If, for example, we want just the minutes from the time stored in t2, we can access them with t2$min. Give

| it a try.

> t2$min

[1] 15

| That's a job well done!

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

| Now that we have explored all three types of date and time objects, let's look at a few functions that

| extract useful information from any of these objects -- weekdays(), months(), and quarters().

...

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

| The weekdays() function will return the day of week from any date or time object. Try it out on d1, which

| is the Date object that contains today's date.

> weekdays(d1)

[1] "Monday"

| Great job!

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

| The months() function also works on any date or time object. Try it on t1, which is the POSIXct object

| that contains the current time (well, it was the current time when you created it).

> month(t1)

Error in month(t1) : could not find function "month"

> months(t1)

[1] "January"

| Perseverance, that's the answer.

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

| The quarters() function returns the quarter of the year (Q1-Q4) from any date or time object. Try it on

| t2, which is the POSIXlt object that contains the time at which you created it.

> quarters(t2)

[1] "Q1"

| That's the answer I was looking for.

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

| Often, the dates and times in a dataset will be in a format that R does not recognize. The strptime()

| function can be helpful in this situation.

...

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

| strptime() converts character vectors to POSIXlt. In that sense, it is similar to as.POSIXlt(), except

| that the input doesn't have to be in a particular format (YYYY-MM-DD).

...

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

| To see how it works, store the following character string in a variable called t3: "October 17, 1986

| 08:24" (with the quotes).

> t3 <- "October 17, 1986 08:24"

| You nailed it! Good job!

|========================================================================== | 75%

| Now, use strptime(t3, "%B %d, %Y %H:%M") to help R convert our date/time object to a format that it

| understands. Assign the result to a new variable called t4. (You should pull up the documentation for

| strptime() if you'd like to know more about how it works.)

> t4 <- strptime(t3, "%B %d, %Y %H:%M")

| Great job!

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

| Print the contents of t4.

> t4

[1] "1986-10-17 08:24:00 CDT"

| You are really on a roll!

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

| That's the format we've come to expect. Now, let's check its class().

> class(t4)

[1] "POSIXlt" "POSIXt"

| Excellent work!

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

| Finally, there are a number of operations that you can perform on dates and times, including arithmetic

| operations (+ and -) and comparisons (<, ==, etc.)

...

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

| The variable t1 contains the time at which you created it (recall you used Sys.time()). Confirm that some

| time has passed since you created t1 by using the 'greater than' operator to compare it to the current

| time: Sys.time() > t1

> Sys.time() > t1

[1] TRUE

| That's a job well done!

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

| So we know that some time has passed, but how much? Try subtracting t1 from the current time using

| Sys.time() - t1. Don't forget the parentheses at the end of Sys.time(), since it is a function.

> Sys.time() - t1

Time difference of 16.54665 mins

| Your dedication is inspiring!

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

| The same line of thinking applies to addition and the other comparison operators. If you want more control

| over the units when finding the above difference in times, you can use difftime(), which allows you to

| specify a 'units' parameter.

...

|============================================================================================== | 94%

| Use difftime(Sys.time(), t1, units = 'days') to find the amount of time in DAYS that has passed since you

| created t1.

> difftime(Sys.time(), t1, units = 'days')

Time difference of 0.01236421 days

| You are really on a roll!

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

| In this lesson, you learned how to work with dates and times in R. While it is important to understand the

| basics, if you find yourself working with dates and times often, you may want to check out the lubridate

| package by Hadley Wickham.

...

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