1.1 | Data Structures and Dimensionality

Having installed Program R and the RStudio Integrated Development Environment, the next step is to import and assess the data themselves. To do this, however, we first need a better understanding of the syntax of the R programming language and how it interacts with RStudio.

1.1.1 | Getting Started with R

When you open RStudio, you will see three windows. On the left is the **Console**, where you enter commands. Let's begin by using R as a simple calculator.

Try entering a few simple commands in the **Console** such as:

```
25 + 8
[1] 33

10 / 2 + 4
[1] 9

10 / (2 + 4)
[1] 1.666667
```

Note the difference between the last two expressions – just like in basic mathematics, order of operations matters in R! The number in brackets on the left side simply indicates the position of the output, in this case that the numbers are the first (and only) values being output. In future calculations, these brackets will become useful for quickly counting and finding the position of specific values in the output.

In addition to the basic operators, R has a large number of built-in *functions* that can perform specific tasks. For example, we can find the square root of a number using the `sqrt()` function:

```
sqrt(16)
[1] 4
```

On the right is a **Global Environment**, which will keep track of any *objects* or additional functions that we create along the way. We can name and store the results of calculation in objects using an arrow.

For example, suppose we want to create a one-dimensional array of numbers, known as a *vector*. We can use the `c()` function to combine or concatenate the numbers 1, 4, and 9 into a

single object, and then give this object a name such as `numbers`:

```
c(1, 4, 9)
[1] 1 4 9

numbers <- c(1, 4, 9)
```

To refer to the stored object, we can just type its name (note that capitalization matters):

```
numbers
[1] 1 4 9
```

We can also use `numbers` in future calculations. For example, we can add values or even apply operations or functions to each of the numbers in the vector:

```
numbers + 2
[1]  3  6 11

numbers ^ 2
[1]  1 16 81

sqrt(numbers)
[1] 1 2 3
```

As we create objects and use those objects to create additional objects, our **Global Environment** can quickly become inundated with object that we no longer need. To clean things up, we can use the `rm()` command to remove unused objects from our environment:

```
rm(numbers)
numbers
Error: object 'numbers' not found
```

Removing unused objects is a good habit to develop from the very beginning. We can even use the `rm(list = ls())` command to remove all objects before proceeding with a new analysis!

1.1.2 | Matrices in R

For more complex problems, you will typically want to write code in an R Script (a file that can be saved and the code later accessed) rather than simply typing commands into the **Console**.

To create a new script file, select the **File Menu**, then **New File -> R Script**. This will open a blank file where you can store code. Before doing anything else, save the file as "Script 1.R" in a directory that is easily accessible.

Type a command into the file, say `4 + 3`, highlight the code, and then select **Run** in the top right of the window. Doing so will run your code in the console window.

By using a script file, you can always go back and re-run your code. Let's continue to add code to the script file you have just created.

In addition to vectors, data analysis often makes use of *matrices*, which we can create and assign to objects using the `matrix()` function. But how exactly does this function work? To find out, we can use a `?` operator to call up its **Help File**, which will commonly appear on the bottom right of the screen below the **Global Environment**.

```
?matrix
```

R help files will likely be difficult to navigate at first. It helps, however, to know that all help files are structured similarly:

1. **Description**: Provides information about what the function does. Here, we learn that `matrix()` "creates a matrix from the given set of values." We also learn that there is also an `as.matrix()` and `is.matrix()` function that may come in useful.
2. **Usage**: Gives examples of how you would use the function. Here, we see that we need at least one argument (data), with additional arguments that are optional.
3. **Arguments**: Provides a list of options that you can supply to the function. Here, we see the argument `nrow` determines the "desired number of columns."
4. **Details:** More information about how the function works.
5. **Examples**: Examples illustrating the use of the code – these can be extremely helpful! You can highlight and run the examples to gain greater insight into the function.

Using this information, we can now create a 2x3 matrix and populate it with the first six numbers of the Fibonacci Sequence (a sequence in which each number is the sum of the two preceding ones) using the `nrow` and `ncol` arguments to set the proper number of rows and columns:

```
m <- matrix(c(1, 1, 2, 3, 5, 8), nrow = 2, ncol = 3)
m
     [,1] [,2] [,3]
[1,]    1    2    5
[2,]    1    3    8
```

We can then access specific elements or data values in this matrix using simple row and column indices. For example, the data value in the second row, third column is:

```
m[2,3]
[1] 8
```

R also allows us to select values in the order in which they are stored internally using a single value:

```
m[6]
[1] 8
```

This index is based on a *column-major* storage order, wherein values are stored down the first column, then down the second, and so on.

We can also select entire rows or columns by leaving one of the indices blank:

```
m[2,]
[1] 1 3 8

m[,3]
[1] 5 8
```

We can even select sets of specific rows or columns by using the `c()` function to denote a single index:

```
m[,c(2,3)]
     [,1] [,2]
[1,]    2    5
[2,]    3    8
```

In addition to creating and indexing matrices, data analysis often involves combining matrices together, which can be accomplished by either binding rows together using `rbind()` or binding columns together using `cbind()`:

```
o <- matrix(c(3, 1, 4, 1, 5, 9), nrow = 2, ncol = 3)
o
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    1    1    9

rbind(m, o)
     [,1] [,2] [,3]
[1,]    1    2    5
[2,]    1    3    8
[3,]    3    4    5
[4,]    1    1    9

cbind(m, o)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    5    3    4    5
[2,]    1    3    8    1    1    9
```

1.1.3 | Data Frames

Thus far, we've discussed how to use R to perform various operations on numbers without considering what the numbers represent. Is the value in the second row, third column indicating that there were eight customers who purchased a Halloween costume or that the maximum temperature that day was 8°F? For this, we need to go a bit beyond numbers and discuss how data are commonly stored in R.

The data themselves are often represented in a spreadsheet using a **case-by-variable** format, wherein each row usually represents a single *case* or *observation*, and each column represents a single *variable*, *dimension*, or *attribute*.

Many data sets are stored in R as *data frames*, which we can create from scratch by using the `data.frame()` function. These are similar to matrices except that the values therein are not necessarily numeric.

In fact, we can easily turn a data matrix into a data frame using the `as.data.frame()` function:

```
d <- as.data.frame(m)
d
  V1 V2 V3
1  1  2  5
2  1  3  8
```

Note that because the original matrix did not have any names associated with each column, R automatically assigned each column a new name: V1, V2, and V3. We can use the `names()` command both to extract these column names and to rename them, as needed:

```
names(d)
[1] "V1" "V2" "V3"

names(d) = c("Month", "Day", "Maximum Temperature")
d
  Month Day Maximum Temperature
1     1   2                   5
2     1   3                   8
```

We can also convert a data frame to a matrix using the `as.matrix()` function. This is particularly useful if we are working only with numerical data and want to take advantage of certain functionalities that exist in matrices but are missing in data frames (e.g., single value indexing, matrix multiplication and inversion, etc.).

Given that the majority of examples that we will be illustrating and data sets that we will be analyzing contain mostly numerical dimensions, we will use data matrices as the default means of data storage throughout this text.

1.1.4 | More on Data Matrices

When creating data matrices from scratch, we can use the `dimnames` argument to assign proper row and column names from the very beginning, and the `byrow` argument to read in the data in (the often more natural) *row-major* storage order:

```
isleRoyale <- matrix(c(1300,  2,
                       1600,  2,
                       1475,  2,
                       2060, 15), nrow = 4, ncol = 2, byrow = T,
                     dimnames = list(c("2016", "2017", "2018", "2019"),
                                     c("Moose","Wolves")))
isleRoyale
     Moose Wolves
2016  1300      2
2017  1600      2
2018  1475      2
2019  2060     15
```

These data represent the number of wolves and moose counted from fixed wing aircraft each winter from 2016 to 2019 on Isle Royale in Michigan, USA (a more complete data set is explored in Exercise 1.1 and beyond).

Alternatively, we can import existing excel files as data frames by selecting the **File Menu**, then **Import Dataset -> From Excel** or **Import Dataset -> From Text (base)** if using a .csv file.

Once a data matrix is constructed and available in R, we can again refer to specific rows, columns, and elements using simple matrix notation. For example, we can extract the yearly counts of moose, which are contained in the first column, by referencing the second column.

```
isleRoyale[,1]
2016 2017 2018 2019
1300 1600 1475 2060
```

In addition to adding, subtracting, and otherwise altering values (much as we did with vectors), we can also make R to make comparisons to identify data values that are greater than (using the `>` operator), less than (using `<`), or equal to (using `==`) some other value. For example, we can find out whether any of the four years had more than 1500 moose.

```
isleRoyale[,1] > 1500
 2016  2017  2018  2019
FALSE  TRUE FALSE  TRUE
```

This output shows a kind of data different than the numerical data we have considered so far. Here, we've asked R to create *logical* data, where the values are either `TRUE` or `FALSE`, depending on if the moose count in each year was or was not greater than 1500, respectively.

We can see that there were only two years where the number of moose exceeded 1500. Although it is relatively easy to determine which years this occurred with such a small number of cases, it can become quite tedious with larger data sets. Instead, we can use the `which()` function to find the indices where the comparisons produced a value of `TRUE`.

```
which(isleRoyale[,1] > 1500)
2017 2019
   2    4
```

This indicates that the number of moose exceeded 1500 only during the second and fourth row in our data matrix, which correspond to the years 2017 and 2019. We can then use these indices to find how many wolves were on the island during this year, referencing either the second column numerically or by just using the `"Wolves"` name assigned to that column.

```
isleRoyale[c(2,4), "Wolves"]
2017 2019
   2   15
```

The first time that moose numbers exceeded 1500 (since 2010) coincided with one of the smallest number of wolves ever observed on the island – only two wolves were counted from fixed wing aircraft that winter. How wolf and moose numbers are related to one another is one of the main questions that researchers have used these data to investigate (and we will return to in Exercise 1.1 and beyond).

1.1.4 | The Dimensionality of Data

For a given data matrix, *dimensionality* refers to how many variables are in the data. In the Isle Royale data used above, we have two variables measured for each case (moose and wolves), indicating that our data have two dimensions. We can use the `dim()` function to quickly compute the number of cases and the corresponding dimensionality of any data set in R:

```
dim(isleRoyale)
[1] 4 2
```

The first number above indicates that there are four cases, each of which is measured on two different dimensions. Because these values are output as a single vector, we can always use vector notation and brackets to isolate one value or the other.

Data sets where the number of dimensions is so high that fitting classical statistical models (e.g., multiple linear regression) becomes difficult are referred to as *high dimensional*. With high dimensional data, the number of variables often exceeds the number of cases, requiring several modifications before more classical models can be applied.

Examples of high dimensional data are frequently found in financial information, such as when the number of features measured for each stock (e.g. PE Ratio, Market Cap, Trading Volume, Dividend Rate, etc.) is much greater than the number of individual stocks. They are also quite common in genomics, where the number of gene features (i.e., dimensions) for a given individual (i.e., case) can be massive. Other areas where variables or attributes exceed cases or observations include forestry, psychology, medical histories, high resolution imagery, and website analysis (e.g., advertising crawling, raking).

Recent and ongoing advances in remote sensing, data sharing, and social media have made it increasingly easy to gather and synthesize large amounts of relevant data, increasing the prevalence of high dimensional data throughout industry and research. Combined with relatively cheap options for data storage, this makes it very tempting to collect as much data as possible, even when you're unsure how you'll use it. For example, studies of black bears and other wildlife species require immense efforts to capture and sedate animals. Once they're captured, you might as well collect as much information as you could possibly think of!

Unfortunately, like most things in life, data collection is subject to the *law of diminishing returns*, an economic theory which states that after some optimal level of capacity is reached, adding an additional factor of production will actually result in smaller increases in output. In statistical modeling and data analysis, adding more dimensions to a data set may not only result in smaller increases of useable data, but it may actually make it more difficult to predict certain quantities or to identify significant or meaningful relationships between variables.

This *curse of dimensionality* refers to various issues that arise as more and more dimensions are added to a data set, which do not occur in low-dimensional settings. One of these issues is that as dimensionality increases, the volume of space represented by the data increases at an alarming rate. Consider modeling the home range of one of the black bears we discussed earlier. Quantifying and predicting such a home range in two-dimensional space would require location data throughout an area of roughly 900 square kilometers, which can be represented as a 30 km by 30 km square. These data are commonly acquired using radio telemetry and GPS tracking. Imagine adding just one more dimension to this model, thereby transforming the space from a two-dimensional square to a three-dimensional cube. The predictive space increases exponentially, from 900 square kilometers to 27,000 cubic kilometers.

When you add more dimensions, it makes sense that the computations burden also increases. From a more statistical viewpoint, the number of observations required to accurately parameterize such models increases as well. This phenomenon is referred to as the *statistical curse of dimensionality*, wherein the required sample size n grows exponentially with the number of dimensions d that a data set has. Even if you were able to collect data on enough cases, high dimensional data are often *sparse*, suffering from issues related to missing values (e.g., blank or invalid responses) that make determining statistical significance problematic.

Despite these challenges, high dimensional data are becoming more and more prevalent in today's world, requiring that we as data scientists and statisticians adapt accordingly.

Exercises for Section 1.1

1.1 Wolves and Moose on Isle Royale. The wolves and moose of Isle Royale, MI, USA have been studied for more than five decades and represents the longest continuous study of any predator-prey system in the world. An excerpt of these data is below (additional data and information are available online at https://isleroyalewolf.org/data/data/home.html):

	Wolves	Moose	Kill Rate	Predation Rate	Moose Recruitment Rate
1959	20	538	NA	NA	20
1960	22	564	NA	NA	14.3
1961	22	572	NA	NA	19.5
1962	23	579	NA	NA	16.5
1963	20	596	NA	NA	21.2
1964	26	620	NA	NA	15.9
1965	28	634	NA	NA	13.2
1966	26	661	NA	NA	18
1967	22	766	NA	NA	21
1968	22	848	NA	NA	20
1969	17	1041	NA	NA	16.5
1970	18	1045	NA	NA	16.1
1971	20	1183	0.615	0.062	11.4
1972	23	1243	0.819	0.091	10.7
1973	24	1215	0.760	0.090	15.1
1974	31	1203	0.599	0.093	14.7
1975	41	1139	0.645	0.139	12.3
1976	44	1070	0.563	0.139	10.3
1977	34	949	0.298	0.064	6.1
1978	40	845	0.507	0.144	10.2
1979	43	857	0.387	0.117	13
1980	50	788	0.330	0.126	12
1981	30	767	0.217	0.051	23.7
1982	14	780	0.869	0.094	20.7
1983	23	830	0.394	0.065	16.4
1984	24	927	0.440	0.068	14.5
1985	22	976	0.457	0.062	13.1
1986	20	1014	0.670	0.079	16
1987	16	1046	0.549	0.050	16
1988	12	1116	0.864	0.056	15.2
1989	12	1260	0.810	0.046	13.4
1990	15	1315	0.857	0.059	14.8
1991	12	1496	1.029	0.050	13.9
1992	12	1697	1.428	0.061	10.9
1993	13	1784	0.877	0.038	14
1994	17	2017	0.792	0.040	12
1995	16	2117	1.391	0.063	7
1996	22	2398	1.274	0.070	3

The most important events have been essentially unpredictable, including a crash in the wolf population in 1981 caused by humans inadvertently introducing canine-parvovirus to the island and a collapse of the moose population after 1996 during the most severe winter on record.

a. Use the matrix() function to construct this data matrix in R and use the dimnames argument to assign to it the proper row and column names (because typing each value in manually is incredibly time consuming, you can always copy and paste the values and then add commas where appropriate)
b. Find the number of cases and the dimensionality of these data
c. Use the head() function to display the first three cases in these data (because the default is six, you will need to use the ? operator to determine what argument needs to be change to only display the first three cases) – this comes in handy if the number of cases in a data set is too large to be displayed in its entirety
d. Use matrix notation to find the number of Wolves during the 7th year of data collection
e. Use matrix notation to find the number of Moose during the 13th year of data collection
f. Use matrix notation to find the number of Wolves and the number of Moose (but not the other three dimensions) in 1981 after the introduction of the canine-parvovirus.

1.2 The Multidimensional Poverty Measure. The World Bank Group is an international financial institution that, among other things, compiles data on local, national, and global indicators of poverty and inequality using the Multidimensional Poverty Measure (MPM). This measure seeks to understand poverty beyond monetary deprivations by including access to education and basic infrastructure along with the monetary headcount ratio at the $1.90 international poverty line. An excerpt of these data for countries in South America (for whom data were available) is below:

	Water	Electricity	Sanitation	Education
Argentina	0.3	0.0	0.4	1.5
Bolivia	7.4	4.9	16.3	13.2
Brazil	1.7	0.2	34.2	16.0
Chile	0.1	0.3	0.6	4.0
Colombia	2.4	1.3	8.2	5.1
Ecuador	4.3	1.4	3.6	3.9
Paraguay	2.1	0.3	9.0	6.3
Peru	6.2	4.1	12.1	5.4
Uruguay	0.5	0.1	1.0	2.0

The values above denote what percent of each country's population did not have access to the designated dimension of poverty in 2019.

a. Use the matrix() function to construct this data matrix in R and use the dimnames argument to assign to it the proper row and column names
b. Find the dimensionality of this data matrix
c. Use matrix notation to extract only the dimensions corresponding to a lack of access to Electricity and Education for all countries
d. Use matrix notation to change the percent of Chile's population without access to Sanitation from 0.6 to 2.4 – this comes in handy if there is an error in the data
e. Use the which() function to identify all instances where any measure is greater than 10
f. Use matrix notation to change all measures greater than 10 to 10 – this comes in handy if data need to be trimmed to specific maximum prior to analysis

1.3 Magic Squares. In recreational mathematics, a *magic square* is a matrix of positive integers arranged so that the sum of each row, column, and main diagonal (the elements that run from the top left to the bottom right) is the same. Four excerpts from a 3x3 magic square are below:

$$\boldsymbol{A} = [\mathbf{4} \quad \mathbf{3} \quad \mathbf{8}] \quad \boldsymbol{B} = [\mathbf{9}] \quad \boldsymbol{C} = \begin{bmatrix} \mathbf{7} & \mathbf{6} \\ \mathbf{5} & \mathbf{1} \end{bmatrix} \quad \boldsymbol{D} = [\mathbf{2}]$$

Accurately constructing such a square is a popular puzzle that exists in 3x3, 4x4, and even 5x5 and beyond sizes.

a. Use the `matrix()` function to construct each of these matrices in R
b. Use the `cbind()` and `rbind()` functions to combine them into a single matrix that meets the criterion for a magic square
c. Use the `rowSums()` and `colSums()` functions to confirm that the total of each row and column is the same
d. Use the `sum()` and `diag()` functions to confirm that the total of the main diagonal is the same as that of each row and column

1.4 Presidential Approval Ratings and Margin of Victory. Historically, a sitting president's prospects of winning a second term in office has been closely tied to their job approval rating. Since 1950, all incumbents with an approval rating of 50% or higher leading up to an election have won, whereas almost all presidents with approval ratings lower than 50% have lost:

	Year	Approval	Margin
Eisenhower	1956	68	15.4
Johnson	1964	74	22.6
Nixon	1972	59	23.2
Ford	1976	45	-2.1
Carter	1980	37	-0.7
Reagan	1984	58	18.2
BushSr	1992	35	-5.5
Clinton	1996	54	8.5
BushJr	2004	48	2.4
Obama	2012	52	3.9
Trump	2020	45	-4.4

The data above include the margin of victory (or defeat) in the popular vote (which often, but not always, matches the results of the electoral vote).

a. Use the `matrix()` function to construct this data matrix in R and use the `dimnames` argument to assign to it the proper row and column names
b. Use the `which()` function to identify all presidents who had an approval rating below 50
c. Use the `which()` function to identify all presidents who had a negative margin of victory, indicating that they lost the popular vote
d. Describe any differences between these two lists of presidents