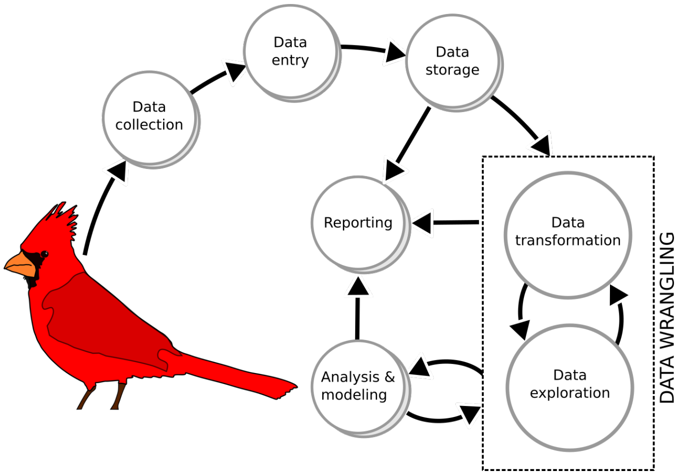
Lesson two: Objects

Brian S. Evans, Ph.D.

Introduction



Re-thinking the structure of data: How we have interacted with data in the past dictates how we structure data mentally. Many of us in science, myself included, have learned to use programs like Microsoft Excel (or iOS' numbers, libreOffice's Calc, Google Sheets, etc.) for viewing and managing data. Through learning and using spreadsheet programs, we tend to format and organize data in ways that are incompatible with environments outside of that system. In other words, there are ways that we interact with data in Excel, especially in regards to how we organize spreadsheets, which do not lend itself well to working with data outside of the spreadsheet environment. Under these conditions, using R (and other programming environments) becomes a frustrating exercise of constantly wrestling with datasets to perform even simple analyses. This process ends up taking most of our computer time.

In addition to increasing our stress level when working with R, the spreadsheet paradigm fails to provide us with fully reproducible data or adequate reporting. Ideally, our target audience should be able to reproduce each step of a data process, starting with the raw data itself.

In this lesson we will work to develop a strong understanding of **objects**. Objects are containers that hold values or other objects. Different classes of objects structure values in different ways.

Dimensions	**Homogeneous**	**Heterogeneous**
1-D	Atomic vector	List
2-D	Matrix	Data frame

For each object type, we'll address:

- The number of dimensions: One vs. two dimensions
- The classes of values within an object:

    - **Homogeneous**: Only one class of values per object
    - **Heterogeneous**: One or more classes of values per object
- **Attributes** of the object class: Names, dimensions, and metadata associated with an object
- **Indexing**: Subsetting an object by the location of a value or values within an object

Atomic vectors

An **atomic vector** is a **one-dimensional** collection of values. Values in an atomic vector are **homogeneous**. In other words, all values must be of the same class. We’ve already learned how to create an atomic vector – this is the type of object that we created using the combine function, `c()` .

Create an atomic vector with the numbers 1, 1, 2, and 3 and assign the name `numericVector` to the object.

To print the output, type the name assigned to the object:

The structure of atomic vectors

The structure of an object describes the class of object and the number of values of an object in each dimension. In the previous lesson we used `class()` and `typeof()` to observe object class and type. For atomic vectors, which are one-dimensional we can use `length()` to determine the number of values that make up the object.

Determine the class and type of values in the object `numericVector` .

Determine the number of values that make up the numericVector.

We can view the length and class of an object in one step using the function `str()` (structure).

Use `str()` to determine the class and number of values in the object `numericVector` in one step.

If the object is numeric, integer, or factor, we can view quantitative summary information of an object using the `summary()` function.

Use `summary()` to calculate summary statistics for the object `numericVector` .

To explore how atomic vectors are homogeneous, observe what happens when the values input into an atomic vector are heterogeneous. Create a vector with the values, 1, ‘one’, 2, and 3

Repeat the above and use `class()` to determine the class of the object

Repeat the above and use `str()` to determine the length and class of the object

Repeat the above and use `summary()` to calculate the summary statistics of the object (if possible!)

Attributes of Atomic vectors

You can determine the attributes of an object using the function `attributes()` . Vectors typically do not have attributes. Some attributes, however, can be added to vectors. For example, we might want to get or add names to a vector. We can both assess and add to the names attribute of a vector using the `names()` function.

Explore the code below, which demonstrates adding names attributes to the values a vector

```
# Adding attributes to a vector:

names(numericVector)

names(numericVector) <-
  c('orange', 'pear', 'apple', 'grape')

numericVector

attributes(numericVector)
```

Indexing atomic vectors

Each value in a vector has a both a value and a position, denoted by “[*x*]”. The position and value information for our `numericVector` could be written as:

[1]	[2]	[3]	[4]
1	1	2	3

The brackets above represent an index of a values location. We can refer to this index using the notation `object_name[x]` . This process is method to **filter** or **subset** a vector by position. Filtering or querying a data object is the process of reducing the number of values in that object to only those that match a given condition. It may be useful to consider the code snippet `object_name[x]` as `object_name` where (`[`) the position is equal to `x` . For example, we could subset the `numericVector` object to the second value by typing `numericVector[2]` .

Subset the atomic vector `numericVector` to the value at position 3:

The value, `3` , that we indexed above is actually a vector of one value, the number three. We can determine the values at multiple positions by providing a vector of multiple integer positions inside the brackets.

Subset the atomic vector `numericVector` to the values at positions 1, 2, and 3:

In addition to indexing a named vector by position, you can also index a named vector by name, by indexing a vector of names in quotes. For example, we might consider our vector above as:

[‘orange’]	[‘pear’]	[‘apple’]	[‘grape’]
1	1	2	3

Subset the atomic vector `numericVector` to the position in which the name of the value is “grape”:

```
# Indexing a vector by the name of the value:

names(numericVector) <-
  c('orange', 'pear', 'apple', 'grape')
```

You may also choose to subset the named vector by denoting position using a vector of names.

Subset the atomic vector `numericVector` to the position in which value is named “orange” or “grape”:

```
# Indexing a vector by the name of the value:

numericVector[c('orange', 'grape')]
```

We will be addressing indexing in depth in the next lesson, so be sure you have a thorough understanding of the above!

Matrix objects

A **matrix** is a two dimensional object – an atomic vector that is arranged into rows and columns (see `?matrix`). You can use the function `matrix()` to create a matrix. The first argument to supply to the matrix function is the data that you would like to express as a matrix. You can add arguments specifying the number of rows, `nrow` , or columns, `ncol` to change the shape of the resultant matrix. These arguments accept numeric values.

Convert the atomic vector `c(1, 1, 2, 3)` into a matrix object:

Repeat the above and use the `nrow` or `ncol` arguments to arrange the matrix into an object with two rows and two columns:

Note in the above that the matrix was created by arranging the atomic vector into columns. When you generated the matrix without an `nrow` or `ncol` argument, the data were arranged as a one column matrix, in the order of the values of the vector. Specifying `nrow` or `ncol` split the column, but the order of values remains the same. This order is known as “by column”. You can specify that you would like to arrange the values of the atomic vector in the matrix “by row” by adding a `byrow` argument to the matrix function. This argument accepts logical values (TRUE or FALSE).

Observe what happens when a matrix is generated with the argument `byrow = TRUE` :

```
# Convert c(1, 1, 2, 3) to a matrix:

matrix(
  c(1, 1, 2, 3),
  nrow = 2,
  byrow = FALSE
)

matrix(
  c(1, 1, 2, 3),
  nrow = 2,
  byrow = TRUE
)
```

When we use the `class()` function, the class of object is returned. The `typeof()` function provides the type of values that make up the object. Like the atomic vectors that matrices *secretly* are, all values in a matrix must be of the same type of values.

Generate a matrix using the values 1, 'one', 2, and 3.

Use `class` to determine the class of object that was generated

Use `typeof()` to determine the type of values that make up the matrix

Matrices have one attribute (`?attributes`) by default – the dimensions of the matrix. Additional attributes, such as row and column names, can be added using the `rownames()` and `colnames()` functions, respectively.

Generate a two-column matrix using the values 1, 1, 2, 3, 5, 8, 13, and 21 and assign the name `myMatrix` to the object.

No output? Type the name (and run the line of code) to see the object

Determine the attributes of the object `myMatrix` :

Add the column names "hello" and "world" to `myMatrix` and view the attributes of the resultant object:

Matrix objects: Indexing

Values in a matrix have a row (x) and column (y) position, denoted by "[*x*, *y*]". It may be useful for you to think of a value's position (also known as "address") as "[*row*, *column*]".

	[,1]	[,2]
[1,]	1	2
[2,]	1	3

Just as you would specify `vectorName[x]` to extract the value from the object `vectorName` at position `x`, you can use this notation to extract values from a matrix. To do so, you need to specify the `x` and `y` address using: `matrixName[x, y]` – this will return all values of the object `matrixName` where the row number is equal to x and the column number is equal to y.

What is the value of `myMatrix` at row 1, column 2?

As with atomic vectors, you can specify a range or subset of values by supplying a vector of numbers for either the `x` or `y` position. You can subset a matrix dimension by name by using the column (or row) name instead of the numeric position. You can specify all values for a row or column by leaving the `x` or `y` position blank (e.g., `[x,]` or `[,y]`).

Subset the object `myMatrix` using the following criteria:

- All values in column 1:
- The first **through** third values in column 2:
- The first **and** third values in column 2:

List objects

A **list** is a one dimensional object constructed by combining ANY objects with ANY dimensionality. Because of their flexibility, lists can be very powerful (and occasionally messy and/or dangerous!). A list is created using the `list()` function, the arguments of which are the items contained within the list.

Explore the following to observe the structure of lists:

- A homogeneous list of values:

```
# Homogeneous list of values:

list(1, 1, 2)
```

- A heterogeneous list of values:

```
# Heterogeneous list of values:

list(1, 'one', 2)
```

- A list of multidimensional objects:

```
# List of multidimensional objects:

list(numericVector, myMatrix)
```

- A list with a name assigned:

```
# Assigned list:

myList <-
  list(numericVector, myMatrix)

myList
```

Lists are vectors, just heterogeneous ones. We can view the structure of list objects in a similar way to atomic vectors.

Explore the structure of `myList` :

- Determine the class of the object:
- Determine the length of the object:
- View the overall structure of the object using `str()` :

Names attributes can be added to list items. To do so, you may name a pre-existing list using the convention `names(listName) <- c(namesVector)` . You may also specify names of list items when a list is created (e.g., `list(hello = 1, world = 2)`).

Explore the following to observe the naming of list items:

- Name objects when the list is contructed:

```
list(apple = 1, pear = 2)
```

- Specify the names of objects in a pre-existing list:

```
myList_named <-
  myList

names(myList_named) <-
  c('numericVector', 'myMatrix')

attributes(myList_named)
```

Lists are often used to inform functions. For example, we have thus far used `attributes()` to view the attributes of an object. `attributes()` can also be used to assign attributes to an object. This makes it a much more powerful function than it might have appeared.

Add column names to `myMatrix` using a list of attributes:

```
# Adding attributes to myMatrix:

attributes(myMatrix)

attributes(myMatrix) <-
  list(colnames = c('hello','world'))

attributes(myMatrix)
```

When it comes to adding attributes, the sky is the limit. For example, you can add metadata to an object using `attributes()`. This may add useful information without affecting the performance of the object.

Explore adding a description to an object by providing a list of attributes:

```
# Adding attributes to myMatrix:

attributes(myMatrix) <-
  list(
    colnames = c('hello','world'),
    author = 'Brian',
    reason = 'To show attribute lists'
    )

attributes(myMatrix)
```

List objects: Indexing

List position is denoted by [[*x*]]. To query a list, use the convention `listName[[x]]`.

[[1]]

[1]	[2]	[3]	[4]
1	1	2	3

[[2]]

	[,1]	[,2]
[1,]	1	2
[2,]	1	3

[[3]]

	[,1]	[,2]
[1,]	“1”	“2”
[2,]	“one”	“3”

The object `myList` is comprised of two objects. Use indexing to extract the second list object:

If names have been assigned to list items, lists can be indexed by these names. To do so, refer to the name of the list item, in quotes, formatted as `listName[['listItem_name']]`.

Subset the list object `myList_named` to the list item named `myMatrix`.

You can also use an operator, `$`, as a shorthand for indexing the named list item. To use this operator, instead of typing `listName[['listItem_name']]`, you would type `listName$listItem_name`. Note that the name of the list item is not quoted.

Use the `$` operator to subset the list object `myList_named` to the list item named `myMatrix`.

Data frames

A **data frame** is the object class that is most often used for storing data. Like a matrix, a data frame has two dimensions with data arranged into rows, *x*, and columns, *y*. This is where the similarity between data frames and matrices pretty much ends, however – a data frame is actually list of atomic vectors in which the length of each vector is equal. As data frames are lists, they are heterogeneous – each atomic vector that makes up the object (columns) may be of a different type.

One method for arranging values into data frames is to use the `data.frame()` function. The arguments of this function are the atomic vectors that make up the object (`data.frame(vector1, vector2)`).

Explore the code below to observe the generation of data frames:

```
# Make a data frame from vectors:

data.frame(
  c('hello', 'world'),
  1:2
)
```

Note in the above that the vectors that make up the data frame may be of different classes. Individual vectors, however, will be coerced into the same class.

Generate a data frame comprised of the vectors `c('hello', 1)` and `1:2`:

As described above, the vectors that make up a data frame must be of the same length. Before making data frames, make sure to explore their length – making data frames of unequal length vectors may result in errors or lead to unexpected results.

Explore what happens when the vectors that make up a data frame are of unequal lengths

```
data.frame(
  1:2,
  1:4
  )
```

```
data.frame(
  1,
  1:4
  )
```

When generating data frames, the columns (vectors) are named by default, but the assigned name may not be meaningful. You can assign the name of the columns when you create the data frame, using the convenction `data.frame(vector1_name = vector1, vector2_name = vector2)`.

Generate a data frame using two vectors, one containing the words “hello” and “world” and the other the integers 1 and 2. Assign the names “factors” and “integers” to the columns.

If a data frame is generated using pre-defined vector objects, the column titles will be the names assigned to each object.

Data frame structure and attributes

Data frames contain a rich set of structural components and attributes that are worth exploring.

Run the code below to observe the structure and attributes of data frames.

- Generate the data frame below and assign the name ‘myDataFrame’:

```
data.frame(
    words = c('hello', 'hello', 'world' , 'world'),
    integers = 1:4
  )
```

- `class()` tells us that the object arranges values into a data frame:

```
class(myDataFrame)
```

- `names()` provides the names of the data frame columns (the names of vectors that make up the object):

```
names(myDataFrame)
```

- `attributes()` provides the names of columns and rows, as well as the class of the object:

```
attributes(myDataFrame)
```

- `length()` provides the number of vectors that make up a data frame:

```
length(myDataFrame)
```

- `nrow()` and `ncol()` provides the number of rows and columns of the data frame, respectively:

```
## [1] 4
## [1] 2
```

- `dim()` provides the number of rows and columns of the data frame:

```
dim(myDataFrame)
```

- `str()` provides structural information on the data frame, including the types of values of the composite vectors:

```
str(myDataFrame)
```

- `summary()` provides statistical summary information of a data frame:

```
summary(myDataFrame)
```

Indexing data frames

Similar to matrices, values in a data frame have a row (x) and column (y) position, denoted by "[*x*, *y*]".

	[,1]	[,2]
[1,]	hello	1
[2,]	hello	2
[3,]	world	3
[4,]	world	4

	[,'factors']	[,'integers']
[1,]	hello	1
[2,]	hello	2
[3,]	world	3
[4,]	world	4

Data frames can be indexed using the convention `dataFrameName[x,y]`, `dataFrameName[,'columnName']`, or `dataFrameName$columnName`.

Complete the following indexing operations on the object `myDataFrame`:

- Use the numeric address (`[x,y]`) to extract the value in the first row (x) and column (y):
- Use the numeric address (`[x,y]`) to extract all values in the first row (x):
- Use the numeric address (`[x,y]`) to extract all values in the second column (y):
- Use the numeric address (`[x,y]`) to extract the values in the first and third rows (x) of the first column (y):
- Use the numeric address for the row (`[x,]`) and the name of the column (`[,'columnName']`) to extract the values in the first and third rows (x) of the first column (y):
- Use the convention `dataFrameName$columnName` to extract the vector associated with column 2 of the data frame:

Tibbles!

The final object that we will look at is an object class provided by the package “tidyverse”. *Note: You must run the “tidyverse” library (`library(tidyverse)`) before you can tibbles*! A **tibble** is a special type of data frame. You can make a tibble using the `tibble()` function. The arguments for this function are the same as that of `data.frame()`.

Generate a data frame and a tibble and observe any differences between the two:

- A data frame:

```
data.frame(
  words = c('hello', 'hello', 'world' , 'world'),
  integers = 1:4
)
```

- A tibble:

```
tibble(
  words = c('hello', 'hello', 'world' , 'world'),
  integers = 1:4
)
```

Notice that, in addition to printing the data frame, the tibble prints some additional information about the object. By default, tibbles will also only print 10 lines at once for long data frames. This is very useful when working with large datasets. Another thing to notice is the type of object of the “words” column is different – `data.frame()` produced a factor vector for that column whereas `tibble()` produced a character vector. This is incredibly useful, as factors can cause lots of problems when they are generated unintentionally.

Data frames can also be converted to tibbles using the `as_tibble()` function. This will not, however, convert factor to character columns.

Tibbles behave quite differently from data frames in meaningful ways beyond what I have described above. Advanced R users who are new to tibbles may want to read the tibble vignette (`vignette('tibble')`).

Use `as_tibble()` to convert `myDataFrame` to a tibble:

```
as_tibble(myDataFrame)
```

Term review and glossary

Functions

- `as_tibble` Convert a data frame to a tibble
- `attributes` Print or set the attributes of an object (e.g., names)
- `colnames` Add column names to a matrix or data frame
- `data.frame` Arrange equal length vectors into a data frame data structure
- `dim` Determine the number of rows and columns in a data frame or matrix
- `length` Determine the length of an object. For atomic vectors and matrices, this returns the number of values. For lists and data frames, this returns the number of component objects.
- `list` Structure component objects into a list data structure
- `matrix` Arrange an atomic vector as a matrix data structure
- `names` Add names to an object’s values or component objects
- `ncol` Determine the number of columns in a data frame or matrix
- `nrow` Determine the number of rows in a data frame or matrix
- `rownames` Add row names to a matrix or data frame
- `str` Determine the structure of an object, including the length and class of component objects
- `summary` Print summary statistical information for an object
- `tibble` Structure component objects into a tibble data structure

Operators

- `listN[[x]]` Use to subset `listN` to the values or component objects at position `x`
- `vectorN[x]` Use to subset `vectorN` to the values at position `x`
- `matrixN[x, y]` Use to subset `matrixN` to the values at position `x,y` (row, column)
- `dataFrameN[x, y]` Use to subset `dataFrameN` to the values at position `x,y` (row, column)
- `dataFrameN$colName` Use to subset `dataFrameN` (or a tibble) to the column named `colName`

Vocabulary

- **Atomic vector**: A set of values of the same class (*homogeneous*) arranged in one dimension.
- **Attributes**: Names, dimensions, and metadata associated with an object.
- **Data frame**: A special type of list that contains equal-length atomic vectors. Data frames are *heterogenous* and have two dimensions.
- **Heterogeneous object**: a container that holds values of one or more object classes
- **Homogeneous object**: a container that holds values of only one class
- **Indexing**: Subsetting an object by the location of a value or values within an object
- **List**: A storage container that holds objects of any class. Lists are *heterogenous* and have one dimension.
- **Matrix**: An atomic vector that is arranged into rows and columns. Matrices are *homogeneous* and have two dimensions.
- **Object**: a container that holds values or other objects
- **Tibble**: A special type of data frame that prints informative metadata about a data frame and does not coerce character vectors to factor.

End of lesson 2!

SMITHSONIAN'S NATIONAL ZOO
& CONSERVATION BIOLOGY INSTITUTE