Data Types

Author: Cole Brookson **Date:** 13 June 2022

Data Structures

To make use of these individual pieces of data we combine them together into data *structures*. There are 4 main data structures that we work with consistently:

1. Vectors
2. Matrices
3. Dataframes
4. Arrays

This helpful graphic may help you cement these types in your memory

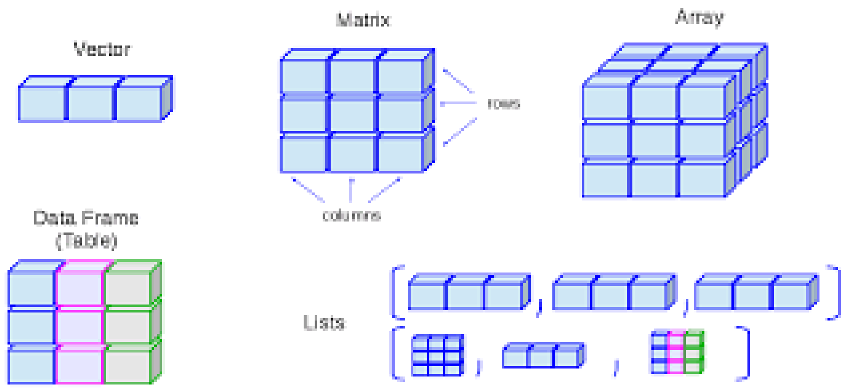

Figure 1: Types of data structures

Vectors

Vectors are subset into atomic vectors (collections of data of the same type) and lists (collections of different data types). Vectors are the workshorses of R and mosts other OOP languages use vectors or a very similar structure to hold data. From this point forward, we'll use 'vector' to refer to atomic vectors and call lists by name. We make vectors the same way we make variables, with an assignment statement.

Atomic Vectors Often we start by making an empty vector that we may decide to fill with values later. To do this, we use the `vector()` function.

```
x <- vector("character", length = 5)
```

v Here, the function `vector()` takes two required arguments. the first argument asks for the data *type* we would like to use, and the second is the number of elements (the *length*) that we want the vector to be.

When we already know the content we want included, we can use a simpler `c()` function (`c` is short for *combine*).

```
x <- c(1, 2, 3, 4)
```

Here, R automatically understands what type of data we're using and since we passed four values, the length is automatically four. We could actually shorten this using R's `:` notation:

```
y <- c(1:4)
```

And we can check this works as expected:

```r
x == y
```

```
## [1] TRUE TRUE TRUE TRUE
```

Note: One thing to understand is that previously when we assigned a variable a single data object (i.e. `x <- 10`), what we were actually doing is making a vector! This is just a special case vector of length 1.

We can determine what is and what is NOT an atomic vector with the following basic command `is.atomic()`. Recall that lists are *also* vectors, but not the kind we're working with now, so this checks if a vector is an atomic vector or not.

Vectors, especially string vectors, can change depending on how they're created. For example, here we have two vectors that spell out the same sentence, but are not equivalent:

```r
x  <- c("The quick brown fox jumped over the lazy dog")
y <- c("The", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog")
x == y
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

We can see by checking the length of each that they lengths are not equivalent. The second vector `y` is actually made up of 9 individual data pieces, while `x` is a singular datum:

```r
length(x)
```

```
## [1] 1
```

```r
length(y)
```

```
## [1] 9
```

Vectors are flexible in that we can bind them together easily:

```r
x1 <- c(x, "who was sunning himself")
y1 <- c(y, "who", "was", "sunning", "himself")
```

and also simply edit components of them:

```r
y[9] <- "cat"
y
```

```
## [1] "The"    "quick"  "brown"  "fox"    "jumped" "over"   "the"    "lazy"
## [9] "cat"
```

Note: We can only do this here because we createdv each word as it's own data point inside the vector.

The above is an example of *indexing* which we discuss elsewhere, but for now, it can be interpreted as selecting an element of the vector and changing it.

Lists Lists are the second type of vector we may want to use. They are more flexible in that they can contain multiple datatypes, whereas atomic vectors must all be of the same type. They are created much the same as vectors:

```r
l <- list(1, 2, "R is fun", c(1, 2, 3), TRUE)
l
```

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

```
## [[3]]
## [1] "R is fun"
##
## [[4]]
## [1] 1 2 3
##
## [[5]]
## [1] TRUE
```

Here we see a number of square brackets denoting the indexing of the list object `l`. This looks confusing but will be discussed in the *Indexing* section.

Instead of these numbers, we could name the list components. This is often handy for keeping data of variable lengths/types that can be related together. For example:

```
eeb_prof <- list("name" = "Dr. Agrawal",
                 "positions" = c("Distinguished Professor of Evolutionary Genetics",
                                 "Associate Chair, Graduate Studies"),
                 "concentrations" = c("Genetics, Genomics & Molecular Evolution",
                                      "Theoretical & Computational Biology"))

eeb_prof
```

```
## $name
## [1] "Dr. Agrawal"
##
## $positions
## [1] "Distinguished Professor of Evolutionary Genetics"
## [2] "Associate Chair, Graduate Studies"
##
## $concentrations
## [1] "Genetics, Genomics & Molecular Evolution"
## [2] "Theoretical & Computational Biology"
```

We now see we have three named list components, each of which gives a different piece of information about the professor at hand.

We will not work much with lists here as their primary utility does not become apparent until more advanced types of computations are required.

Matrices

Matrices are a logical extension of vectors as they can be thought of as a series of vectors bound together to form a 2D structure made up of rows and columns. Matrices have the same limit as a vector and must contain data of the same type (numeric or character). They are also constructed similarly:

```
m <- matrix(nrow = 2, ncol = 2)
m
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
```

This above is an empty matrix, we haven't told R to put anything in the matrix, but this empty matrix concept is used frequently to preallocate memory before being filled in later. We could make a matrix with some values like this:

```
m <- matrix(100, nrow = 2, ncol = 2)
```

And this will make a matrix with four values of 100. Multiple values are possible as well:

```
m <- matrix(c(100, 200, 300, 400), nrow = 2, ncol = 2)
```

Here we passed an atomic vector as the first argument to `matrix()`, and then the dimensions that we wanted to get for our matrix. Note that if the dimensions specified are not filled by the vector passed, an error will occur:

```
m <- matrix(c(100, 200, 300, 400), nrow = 3, ncol = 2)
```

```
## Warning in matrix(c(100, 200, 300, 400), nrow = 3, ncol = 2): data length [4] is
## not a sub-multiple or multiple of the number of rows [3]
```

We can also make matrices by combining existing vectors. Let's make two vectors and combine them into a matrix using two different approaches:

```
vec1 <- c(1, 2)
vec2 <- c(3, 4)
```

First, if I want `vec1` to be the first row and `vec2` to be the second row, I simply need to bind them together as *rows* with the aptly named `rbind()` function:

```
m1 <- rbind(vec1, vec2)
m1
```

```
##      [,1] [,2]
## vec1    1    2
## vec2    3    4
```

If instead I wanted `vec1` to be the first column and `vec2` to be the second column, I would bind them together as *columns* using the `cbind()` function:

```
m2 <- cbind(vec1, vec2)
m2
```

```
##      vec1 vec2
## [1,]    1    3
## [2,]    2    4
```

Dataframes

The darlings of R. These structures are the feature of R that make it so popular for data analysis and statistics. It is the easiest way to store, access, and perform operations on tabular data (the type of data we most often have in biology). Dataframes are actually a type of list, one wherein each element of the list has the same length, making it of dimension 2, which means we easily can look at rows and columns. They're usually created directly by reading in data (we'll get to this shortly), or by creation through the `data.frame` command:

```
c1 <- c(1:4)
c2 <- c("item1", "item2", "item3", "item4")
c3 <- c(11:14)

# we can combine these values as columns into a dataframe
d <- data.frame(c1, c2, c3)
d
```

```
##   c1    c2 c3
## 1  1 item1 11
## 2  2 item2 12
## 3  3 item3 13
## 4  4 item4 14
```

It's easy to add rows or columns using row binding:

```
c4 <- c("add1", "add2", "add3", "add4")
d1 <- cbind(d, c4)
d1
```

```
##   c1    c2 c3   c4
## 1  1 item1 11 add1
## 2  2 item2 12 add2
## 3  3 item3 13 add3
## 4  4 item4 14 add4
```

Or column binding:

```
r5 <- list(5L, "item5", 15L, "add5")
d2 <- rbind(d1, r5)
d2
```

```
##   c1    c2 c3   c4
## 1  1 item1 11 add1
## 2  2 item2 12 add2
## 3  3 item3 13 add3
## 4  4 item4 14 add4
## 5  5 item5 15 add5
```

Notice above to create row 5, we used `list()` instead of `c()`. Why? Recall that `c()` only works for atomic vectors, so if we used the same arguments but the command `c()`, R would think we wanted an atomic vector, so would actually change our integer values to characters. Let's look at it:

```
r5 <- c(5L, "item5", 15L, "add5")
str(r5)
```

```
##  chr [1:4] "5" "item5" "15" "add5"
```

We can see all the values of `r5` are now characters. Also, notice above that we force integer type with the `L` following the number. Why? Well when using `list()`, if we just pass the number `3`, `list()` would interpret that as a numeric type, and when we bound it to the dataframe, because each column must only have one type, to fix the fact that some values would be integers and some numerics, R would go behind the scenes and change the type of those columns to numerics. Let's see:

```
r5 <- list(5, "item5", 15, "add5")
str(r5)
```

```
## List of 4
##  $ : num 5
##  $ : chr "item5"
##  $ : num 15
##  $ : chr "add5"
```

Note that the first and third elements now have type `num` instead of type `int`!

```
d3 <- rbind(d1, r5)
d3
```

```
##   c1    c2 c3   c4
## 1  1 item1 11 add1
## 2  2 item2 12 add2
## 3  3 item3 13 add3
## 4  4 item4 14 add4
## 5  5 item5 15 add5
```

Now we can see in the output that columns `c1` and `c2` have type `<dbl>` which we recall from Data Types are equivalent. This isn't a big deal, but something to be aware of in case we really needed `c1` and `c2` to be integers.

Dataframes are ultra-flexible, and have a lot of underlying structure. The best way to inspect this structure is with the `str()` command:

```
str(d2)
```

```
## 'data.frame':    5 obs. of  4 variables:
##  $ c1: int  1 2 3 4 5
##  $ c2: chr  "item1" "item2" "item3" "item4" ...
##  $ c3: int  11 12 13 14 15
##  $ c4: chr  "add1" "add2" "add3" "add4" ...
```

We can also get the column names of a dataframe,

```
names(d2)
```

```
## [1] "c1" "c2" "c3" "c4"
```

the number of rows or columns,

```
nrow(d2)
```

```
## [1] 5
```

```
ncol(d2)
```

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

a summary of the dataframe,

```
summary(d2)
```

```
##        c1         c2                  c3          c4
##  Min.   :1   Length:5           Min.   :11   Length:5
##  1st Qu.:2   Class :character   1st Qu.:12   Class :character
##  Median :3   Mode  :character   Median :13   Mode  :character
##  Mean   :3                      Mean   :13
##  3rd Qu.:4                      3rd Qu.:14
##  Max.   :5                      Max.   :15
```

For the integer columns, we get a very helpful summary of the values in the column.

Factors We will briefly discuss factors here as their most common use is as columns in dataframes.

Factors are variables which can only take on a certain number of values (aka "levels"). They are often referred to as the "categorical" variables of R. They are of special importance in statistical modelling since categorical variables enter into statistical models in a different way than continuous variables may do.

Similar to other datatypes, they can be created with their own function:

```
fac <- factor(c("one", "two", "three", "four", "one", "two", "three", "four"))
fac
```

```
## [1] one   two   three four  one   two   three four
## Levels: four one three two
```

The output here shows the values we passed (those within the `c()` function), but the levels aren't maybe what we'd expect. We passed what we know to be numbers, but R can't actually tell that, it just knows they are string values, so has sorted them alphabetically. We can control the *order* of factors with an additional argument to the function `factor()`:

```r
fac1 <- factor(c("one", "two", "three", "four", "one", "two", "three", "four"),
               levels = c("one", "two", "three", "four"))
fac1
```

```
## [1] one   two   three four  one   two   three four
## Levels: one two three four
```

Now the levels are in a more logical order.

Factors are most commonly used in dataframes, so let's change one of our `chr` variables in a dataframe we've already made to a `factor`.

```r
d
```

```
##   c1    c2 c3
## 1  1 item1 11
## 2  2 item2 12
## 3  3 item3 13
## 4  4 item4 14
```

Currently `d2` has `c2` column of type `chr`, but let's make it a factor. We could do this by changing the way we define `c2` originally like this:

```r
c1 <- c(1:4)
c2 <- factor(c("item1", "item2", "item3", "item4"))
c3 <- c(11:14)

# we can combine these values as columns into a dataframe
d <- data.frame(c1, c2, c3)
d
```

```
##   c1    c2 c3
## 1  1 item1 11
## 2  2 item2 12
## 3  3 item3 13
## 4  4 item4 14
```

But often we want to perform "in-place" operations to dataframes already created, so let's create the dataframe as before then once created, only re-create that one column:

```r
c1 <- c(1:4)
c2 <- c("item1", "item2", "item3", "item4")
c3 <- c(11:14)

# we can combine these values as columns into a dataframe
d <- data.frame(c1, c2, c3)
d
```

```
##   c1    c2 c3
## 1  1 item1 11
## 2  2 item2 12
## 3  3 item3 13
## 4  4 item4 14
```

So currently it's a `chr`, but we can *index* our column of interest by name, using the `$` operator, and perform the change this way. To change the column to a factor we again can use the `factor()` column:

```
d$c2 <- factor(d$c2)
str(d)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ c1: int  1 2 3 4
##  $ c2: Factor w/ 4 levels "item1","item2",..: 1 2 3 4
##  $ c3: int  11 12 13 14
```

Now we can see the output has `c2` as a column. More on indexing later.

Arrays

Last we'll go over some arrays. Arrays are very useful but can be difficult to visualize since they can store data in >2 dimensions, but let's just focus on 3-dimensional arrays for now. Again, we can create an array with the `array()` function, combining multiple vectors.

```
vec1 <- c(1, 2, 3)
vec2 <- c(4, 5, 6)
vec3 <- c(7, 8, 9)
ar1 <- array(c(vec1, vec2, vec3), dim = c(3, 3, 2))
ar1
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

We can read the `dims` argument just as we would order mathematical dimensions, always starting with `x` the first dimension, `y`, the second, and then `z` the third one. So we can think of this as having three rows (the `x` coordinate), and three columns (the `y` coordinate), which make up a single matrix in each of the two `z` dimensions. It's helpful to visualize this:

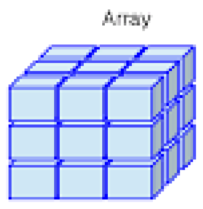

Figure 2: Arrays

It's often helpful to name each of our columns, rows, and matrices. To do this for the above array we might do the following:

```
row_names <- c("row1", "row2", "row3")
col_names <- c("col1", "col2", "col3")
```

```r
mat_names <- c("mat1", "mat2")

# now remake the array with the names
ar_named = array(c(vec1, vec2, vec3), dim = c(3, 3, 2),
                 dimnames = list(row_names, col_names, mat_names))
ar_named
```

```
## , , mat1
##
##      col1 col2 col3
## row1    1    4    7
## row2    2    5    8
## row3    3    6    9
##
## , , mat2
##
##      col1 col2 col3
## row1    1    4    7
## row2    2    5    8
## row3    3    6    9
```

With the rows, columns, and matrices now named it's easier to refer to them.