Vectors and indexing

R has a special data structure called a vector. A vector is a 1D set of the same type of object. Most often, a vector will simply be a sequence of numbers. We can create a sequence of numbers using the `:` operator.

```
numbers <- 1:10
numbers
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Note that vectors are treated the same way as a single element. Anything that works on a single number works the same way on a vector. This is called vectorization.

```
numbers + 5
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15
```

```
2 ^ numbers
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```
sin(numbers)
```

```
##  [1]  0.8414710  0.9092974  0.1411200 -0.7568025 -0.9589243 -0.2794155
##  [7]  0.6569866  0.9893582  0.4121185 -0.5440211
```

We can also create a vector with the `c()` function (`c` stands for concatenate, in case you are wondering).

```
concat <- c(4, 17, -1, 55, 2)
concat
```

```
## [1]  4 17 -1 55  2
```

Indexing

Often, we don't want to get the entire vector. Perhaps we only want a single element, or a set of specific elements. We do this by indexing (uses the `[]` brackets).

To get the first element of a vector, we could do the following. In R, array indexes start at 1 - the 1st element is at index 1. This is different than 0-based languages like C, Python, or Java where the first element is at index 0.

```
concat[1]
```

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

To get other elements, we could do the following:

```
concat[2]  # second element
```

```
## [1] 17
```

```
concat[length(concat)]  # last element
```

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

Notice that for the second example, we put a function inside the square brackets. In this case, `length()` is used to get the length of a vector, and since the lenght of the vector will be equal to the index of its last element, this is a nifty way of getting the last element of a vector.

We can actually put anything inside the square brackets. Putting another vector inside the brackets gives us multiple values, for instance.

```
concat[1:4]
```

```
## [1]  4 17 -1 55
```

```
concat[c(3, 5)]
```

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

We can use this technique to reassign certain values inside the vector. For instance, we could change the 3rd and 5th values to 76 with the following code:

```
concat[c(3, 5)] <- 76
concat
```

```
## [1]  4 17 76 55 76
```

It's even possible to index outside the bounds of a vector. Notice that R "fills in the blanks" with `NA` values. `NA` is R's placeholder for "no data" (since 0 often shows up in real data).

```
concat[10] <- 4.3
concat
```

```
##  [1]  4.0 17.0 76.0 55.0 76.0   NA   NA   NA   NA  4.3
```

DON'T EVER DO THIS. Though this is actually valid code in R (indexing outside of a vector's size is an error in most other languages), it comes at a performance cost. We will go more in-depth as to why later on.

Matrices

A matrix is a two-dimensional vector. Let's create a matrix with the `matrix()` function. One important note here: functions often have optional, "extra" arguments that are specified with `name=value` notation. In this case, we are creating a matrix with 2 rows and 5 columns.

```
mat <- matrix(1:10, nrow=2, ncol=5)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

All of the same operations that work on vectors also work on matrices.

```
mat + 20
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   21   23   25   27   29
## [2,]   22   24   26   28   30
```

```
dim(mat)  # grab dimensions of a matrix
```

```
## [1] 2 5
```

```
length(mat)  # number of elements in a matrix
```

```
## [1] 10
```

However, indexing a matrix is slightly different from indexing in a vector. We now have not only one, but two dimensions to choose from. When indexing using an object with multiple dimensions, we use a `,` to separate them. In R, rows are on the left side of the comma, and columns are on the right (a third dimension would be after the second comma, and so on…). Note that R is actually trying to help us out here. When we printed our matrix, the rows have a `[#,]` next to them, and the columns have a `[,#]`. This actually shows us the exact syntax we need to get each element.

So using the matrix output from above, `mat[1,]` should give us the first row, and `mat[,4]` should give us the fourth column. Let's check this:

```
mat[1,]
```

```
## [1] 1 3 5 7 9
```

```
mat[,4]
```

```
## [1] 7 8
```

We can index both rows and columns at the same time to get a specific element.

```
mat[1, 1]  # grab first row, first column
```

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

```
mat[2, 1:3]  # elements 1-3 of the second row
```

```
## [1] 2 4 6
```

Exercise - Grabbing unrelated elements:

Try getting the columns 1, 4, and 5 of the first row in one command.

Exercise - Reading documentation:

Create an 8x5 matrix using the numbers 1:40. See if you can get it to fill by row instead of by column.

Hint: you should check the documentation for `matrix()`.

Different types of data

In most programming languages, text is called a string. To create text in R, we simply surround it with double (`"`) or single (`'`) quotes. We've actually seen an example of a string already (`print('hello world!')`).

```
"this is a string"
```

```
## [1] "this is a string"
```

```
paste('we can combine strings', 'with the paste() function')
```

```
## [1] "we can combine strings with the paste() function"
```

What happens when we add a string to a vector of numbers?

```
numbers <- c(1, 4, 9, 10)
numbers
```

```
## [1]  1  4  9 10
```

```
numbers[3] <- "testing..."
numbers
```

```
## [1] "1"          "4"          "testing..." "10"
```

Our entire vector of numbers was turned into strings! This is an important property of vectors and matrices: they can only hold one type of data! If we try putting a different type into a vector, R will convert the entire vector to the new datatype.

Again, this conversion has a massive performance hit, especially for a large vector or matrix. R needs to create an entire new vector from scratch, and then copy over and convert every item.

So let's learn about R's different data types. There are a few more types that we're not covering here or cover later (like factors, for instance). These are simply the most common data types we will encounter.

Numeric

Numeric variables can hold any decimal number, positive or negative. In other languages these are called floats. Note that this is the default type of number in R.

To create a numeric value, all we have to do is type it normally.

```
1
```

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

```
-3.5
```

```
## [1] -3.5
```

We can convert another variable to numeric with the `as.numeric()` function. This only works with values that can be easily converted.

```
as.numeric("456")  # this works
```

```
## [1] 456
```

```
as.numeric("seven")  # this does not
```

```
## Warning: NAs introduced by coercion
```

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

Integers

Integers represent any whole number, positive or negative. They cannot hold decimal numbers.

To create a number explicitly as an integer, add an `L` after it.

```
15L
```

```
## [1] 15
```

We can convert a set of data to integer with the `as.integer()` function.

```
as.integer(c("65.3", "4"))
```

```
## [1] 65  4
```

Characters (strings)

As mentioned earlier, strings are sets of text. We can turn something into a string with the `as.character()` function.

```
as.character(TRUE)
```

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

Logical (boolean) values

There are just two logical values: `TRUE` and `FALSE`. These are used, quite literally, to represent whether or not a statement is true or false.

As usual, we can turn something into a logical/boolean value with the `as.logical()` function. One important thing to note is that this very nicely demonstrates what happens when we turn one data type into another. If one data type cannot hold extra information from another, that information is lost during conversion.

```
fifty <- as.logical(50)
fifty
```

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

```
as.numeric(fifty)
```

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

Determining data types

If something isn't working the way it's supposed to, a great technique is to check what type of data it is with the `class()` function:

```
class(14)
```

```
## [1] "numeric"
```

```
class(1L)
```

```
## [1] "integer"
```

```
class(TRUE)
```

```
## [1] "logical"
```

```
class("text")
```

```
## [1] "character"
```

```
class(NA)
```

```
## [1] "logical"
```

Note that `NA`'s can be of any type, and are often used as placeholders for missing data.

Exercise - Converting data types:

See if you can make the following expressions work:

`5 + "10"` - Get 15 as a result.

`3 + 3.7` - Get 6 as a result.

`1 + 1` - Your answer should equal "11".

`"4" + 10` - Get 5 as a result.

Exercise - Data sizes:

In many programming languages, different types of data take up different amounts of memory. Is this true in R?

You can check how big an object is in memory using the `object_size()` function from the `pryr` package. Are there any memory optimizations to be had from converting a vector of numberic values to a vector of integers? Why or why not?

Next section