Vectors and Lists

There are two basic object types in R for storing ordered data:

Vectors are fairly strict, with all elements being of the same type (i.e., integers or characters). We have created these using the c() function. They are used internally to store the columns of a tibble.

Lists are much more flexible. Nearly anything can be put into each slot of a list: vectors of different lengths and data types, other lists, and even other R objects.

As we have seen, vectors are usually fairly nice and can be put directly into a data table. Lists require more work.

Accessing Vector Elements

If we want to access a single element of a vector, we can use a single square bracket with the desired element number inside (Note: R starts numbering at 1):

```
> example <- c(1, 1, 2, 3, 5, 8)
> example[3]
[1]  2
```

List Example

Here is an example of a small list in R with three elements (don’t worry about how it is created; we will see this soon):

```
> example
[[1]]
[1] 1 2 3

[[2]]
[1]  6  7  8  9 10

[[3]]
 [1] 0.26550866 0.37212390 0.57285336 0.90820779
 [5] 0.20168193 0.89838968 0.94467527 0.66079779
 [9] 0.62911404 0.06178627 0.20597457 0.17655675
[13] 0.68702285 0.38410372 0.76984142 0.49769924
[17] 0.71761851 0.99190609 0.38003518 0.77744522
```

The first contains three integers, the second five integers, and the last has twenty random values between 0 and 1.

List Example With Names

The elements of an R list can, optionally, contain names. Here is the same list with names:

```
> example_name
$alpha
[1] 1 2 3

$beta
[1]  6  7  8  9 10

$gamma
 [1] 0.26550866 0.37212390 0.57285336 0.90820779
 [5] 0.20168193 0.89838968 0.94467527 0.66079779
 [9] 0.62911404 0.06178627 0.20597457 0.17655675
[13] 0.68702285 0.38410372 0.76984142 0.49769924
[17] 0.71761851 0.99190609 0.38003518 0.77744522
```

Accessing List Elements

There are two ways to get a single list element: (1) by using the position of the element or (2) using the name, when it exists.

To get a particular element by number, use double square brackets along with the element number (Note: R starts with element 1, not zero). Here, we get the second element of the example list:

```
> example[[2]]
[1]  6  7  8  9 10
```

To get a particular element by name, use a dollar sign followed by the element name:

```
> example_name$beta
 [1]  6  7  8  9 10
```

Map

Typically, we will want to automate the process of cycling through the elements of a list. We can do this with a map function, which applies a function to each element of a list. For example, let’s say we want to determine the length of each element of our list, we could map our example list using the function **length**:

```
> map(example, length)
[[1]]
[1] 3

[[2]]
[1] 5

[[3]]
[1] 20
```

Map

The output of the map function is another list. If we want to return a vector, we need to specify the data type of the vector as a postfix (remember, vectors only have one data type).

```
> map_int(example, length)
[1]  3  5 20
```

The data type is quite flexible and R will do its best to create the desired output. For example, we could instead return a character vector, and R will return character values without a problem:

```
> map_chr(example, length)
[1] "3"  "5"  "20"
```

Other postfix options for the map function include dbl (double) and lgl (logical).

Inline Functions

Often we will want to define a custom function to apply to each element of a list. To do this, we can create an anonomous function using the tilda operator ~ (you'll need to find it on your keyboard). You define what to do to each list by writing code relative to a variable that is named **.x** which is easier to show than explain.

Here is the use of an inline function to grab the third element of each sub-part of the example list:

```
> map_dbl(example, ~  .x[3])
[1] 3.0000000 8.0000000 0.5728534
```

The way to think about this is that we are writing a short-hand that grabs these three elements all at once:

example[[1]][3]

example[[2]][3]

example[[3]][3]

The parts in bold orange are what we call **.x** in the code above.

Two More List Functions

There are two other things that are helpful to know when extracting data from lists. First, we can get a vector of the names of a list using the function **names()**

```
> names(example_name)
[1] "alpha" "beta"  "gamma"
```

Secondly, we can unravel the elements of a list into a single vector by using the flatten functions. It has a postfix like the map functions:

```
> flatten_dbl(example)
[1]   1.00000000  2.00000000  3.00000000  6.00000000
[5]   7.00000000  8.00000000  9.00000000 10.00000000
[9]   0.26550866  0.37212390  0.57285336  0.90820779
[13]  0.20168193  0.89838968  0.94467527  0.66079779
[17]  0.62911404  0.06178627  0.20597457  0.17655675
[21]  0.68702285  0.38410372  0.76984142  0.49769924
[25]  0.71761851  0.99190609  0.38003518  0.77744522
```