Week 11 Module 1 – Lists

CSCI E-5a: Introduction to R

Let's clear the global computing environment:

```
rm( list = ls() )
```

Module Overview and Learning Outcomes

Hello! And welcome to Module 4: Lists

In this module, we'll meet another important compound data structure: the *list*.

- In Section 1, we'll define lists, and learn the basic properties of lists.
- In Section 2, we'll explore the relationship between lists and data frames.
- In Section 3, we'll see how to split a character string into substrings.

When you've completed this module, you should be able to:

- Define the concept of the list data structure, and explain its basic properties and methods.
- Explain the relationship between data frames and lists.
- Use the `strsplit()` function to split a string into a set of substrings.

There are five new built-in R functions in this module:

- `list()`
- `as.list()`
- `is.list()`
- `unlist()`
- `strsplit()`

All right! Let's get started by learning the basic concepts of lists.

Section 1: List Basics

> **Main Idea:** *The list is the most general compound data structure in R*

In this section, we'll define lists, and learn the basic properties of lists.

So far, we've studied two important compound data structures: vectors and data frames.

In addition to these two, R has a number of other compound data structures.

One of the most important of these other compound data structures is the *list*.

We won't do much with lists in our course, but I want to give you a quick overview of this important data structure.

Recall that a vector has two important properties:

- It's one dimensional, in the sense that we only need one positive integer to locate a particular element.
- All the values in the data structure must be of the same atomic data type.

The *list* data structure relaxes the second requirement.

That is, a list is still a one-dimensional object, but the values in it don't all have to be the same atomic data type.

For instance, a list could store a numeric value in the first position, a logical value in the second position, and a character value in the third position.

We can construct a list in R by using the `list()` function, which takes a sequence of R objects and constructs a list.

For instance, we can construct this simple list:

```
example.1.list <-
    list( 12, TRUE, "Hi Mom!" )

example.1.list
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] TRUE
##
## [[3]]
## [1] "Hi Mom!"
```

Notice the distinctive format when we directly display a list.

When we call the `class()` function with this list, it indicates that the class of the object is `list`:

```
class( example.1.list )
```

```
## [1] "list"
```

We can test an R object to see if it is a list by using the `is.list()` function:

```
is.list( example.1.list )
```

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

Notice that vectors are not lists:

```
is.list( 1:5 )
```

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

We can construct lists where all the elements have names in exactly the same way that we construct named vectors:

```
named.example.2.list <-
    list(
        first = 12,
        second = TRUE,
        third = "Hi Mom!"
    )

named.example.2.list
```

```
## $first
## [1] 12
##
## $second
## [1] TRUE
##
## $third
## [1] "Hi Mom!"
```

Another method for constructing a list is the `as.list()` function, which takes a vector as the input argument and constructs a list where each element of the list is just the corresponding element of the input argument:

```
example.3.list <-
    as.list( 1:5 )

example.3.list
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

In fact, we can store not just atomic values in lists, but also compound data structures.

Here's a list where the entries are all vectors:

```
example.4.list <-
    list(
        1:5,
        c(TRUE, TRUE, FALSE, TRUE),
        c("To", "Be", "Or", "Not", "To", "Be")
    )

example.4.list
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1]  TRUE  TRUE FALSE  TRUE
##
## [[3]]
## [1] "To"  "Be"  "Or"  "Not" "To"  "Be"
```

Notice that the vectors do not all have to be the same length.

The `unlist()` function "flattens" a list; that is, it creates a vector consisting of all the atomic values of the list.

For instance, let's create a list consisting of numeric vectors of different lengths:

```
example.5.list <-
    list(
        1:3,
        10:15,
        21:22
    )

example.5.list
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 10 11 12 13 14 15
##
## [[3]]
## [1] 21 22
```

Now we can flatten this list by using the `unlist()` function:

```
unlist( example.5.list )
```

```
##  [1]  1  2  3 10 11 12 13 14 15 21 22
```

Remember – when you flatten a list, all the elements are placed in a vector, and we know that all the values in a vector must be of the same atomic data type.

Thus, if the list contains an element that has a character string value, then all the values in the flattened vector will be coerced to character string values:

```
unlist( example.5.list )
```

```
##  [1]  1  2  3 10 11 12 13 14 15 21 22
```

In fact, a list can even contain another list.

We call this *nesting*, and the nested list itself can even contain other lists, so we can construct arbitrarily complicated data structures using lists:

```
complicated.list <-
    list(
        1:5,
        "Hi Mom!",
        list(
            3:9,
            c(TRUE, TRUE, FALSE, TRUE, FALSE),
            c("That", "is", "the", "question")
        )
    )

complicated.list
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] "Hi Mom!"
##
## [[3]]
## [[3]][[1]]
## [1] 3 4 5 6 7 8 9
##
## [[3]][[2]]
## [1]  TRUE  TRUE FALSE  TRUE FALSE
##
## [[3]][[3]]
## [1] "That"     "is"       "the"      "question"
```

This nesting feature makes lists very powerful, because they can be used to construct sophisticated data structures.

This is an advanced programming topic, and it's beyond the scope of our course, but I wanted to draw your attention to it.

Just as with vectors, there are a variety of methods for selecting elements from a list.

For instance, we can use positive integer indexing, with a single set of square brackets:

```
example.4.list[ 2 ]
```

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

Notice that when we use a single set of square brackets, the object that is returned is still a list.

```
class( example.4.list[ 2 ] )
```

```
## [1] "list"
```

If we want to extract the vector, we need to use double square brackets:

```
example.4.list[[ 2 ]]
```

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

You can tell that this object is a vector because of how it looks when we display it directly, but we can also check this explicitly:

```
class( example.4.list[[ 2 ]] )
```

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

We can also use techniques such as logical indexing or indexing with names.

For instance, here's a simple named list:

```
named.list <-
    list(
        numeric.vector = 1:5,
        logical.vector =
            c( TRUE, TRUE, FALSE, TRUE, FALSE ),
        character.vector =
            c( "a", "b", "c", "d", "e" )
    )

named.list
```

```
## $numeric.vector
## [1] 1 2 3 4 5
##
## $logical.vector
## [1]  TRUE  TRUE FALSE  TRUE FALSE
##
## $character.vector
## [1] "a" "b" "c" "d" "e"
```

Now for instance we can access columns using the dollar sign that we've seen with data frames:

```
named.list$character.vector
```

```
## [1] "a" "b" "c" "d" "e"
```

Let's check that this is a vector:

```
class( named.list$character.vector )
```

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

Lists are the most general data structure in R, because they can contain any other kind of R data structure, including other lists.

Thus, lists can be used to construct complex R objects that can be used to store rich data structures.

You've actually already seen an example of this, with linear models.

Let's construct a linear model object by using the `lm()` function:

```
linear.model.object <-
    lm( Sepal.Length ~ Sepal.Width, data = iris)
```

Let's take a look at the structure of this object:

```
str( linear.model.object )
```

```
## List of 12
##  $ coefficients : Named num [1:2] 6.526 -0.223
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "Sepal.Width"
##  $ residuals    : Named num [1:150] -0.644 -0.956 -1.111 -1.234 -0.722 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:150] -71.566 -1.188 -1.081 -1.187 -0.759 ...
##   ..- attr(*, "names")= chr [1:150] "(Intercept)" "Sepal.Width" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:150] 5.74 5.86 5.81 5.83 5.72 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:150, 1:2] -12.2474 0.0816 0.0816 0.0816 0.0816 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:150] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "Sepal.Width"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.08 1.02
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 148
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = Sepal.Length ~ Sepal.Width, data = iris)
##  $ terms        :Classes 'terms', 'formula'  language Sepal.Length ~ Sepal.Width
##   .. ..- attr(*, "variables")= language list(Sepal.Length, Sepal.Width)
```

```
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "Sepal.Length" "Sepal.Width"
##   .. .. .. ..$ : chr "Sepal.Width"
##   .. ..- attr(*, "term.labels")= chr "Sepal.Width"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(Sepal.Length, Sepal.Width)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "Sepal.Length" "Sepal.Width"
##  $ model        :'data.frame':   150 obs. of  2 variables:
##   ..$ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##   ..$ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language Sepal.Length ~ Sepal.Width
##   .. .. ..- attr(*, "variables")= language list(Sepal.Length, Sepal.Width)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "Sepal.Length" "Sepal.Width"
##   .. .. .. .. ..$ : chr "Sepal.Width"
##   .. .. ..- attr(*, "term.labels")= chr "Sepal.Width"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(Sepal.Length, Sepal.Width)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "Sepal.Length" "Sepal.Width"
##  - attr(*, "class")= chr "lm"
```

Wow! That's complicated.

Notice that at the very top of the output it says, "List of 12".

So that means that this linear model object is really a special kind of list.

Let's check this:

```
is.list( linear.model.object )
```

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

In fact, it's a named list, and so we can use the dollar sign notation to extract individual elements of the linear model.

For instance, we can select the named vector of coefficients like this:

```
linear.model.object$coefficients
```

```
## (Intercept) Sepal.Width
##   6.5262226  -0.2233611
```

We saw this code in Week 9, Module 4: Scatterplots, in Section 3: Regression Coefficients.

So that linear model object was really just a list all along!

So that's the basic idea of the list data structure.

Now let's explore the relationship between lists and data frames.

Section 2: Data Frames and Lists

> **Main Idea:** *Data frames are lists*

In this section, we'll explore the relationship between lists and data frames.

Here's an amazing fact: data frames are lists.

In fact, they are very special kinds of lists, and in order for a list to be a data frame it must satisfy three conditions:

- Every element of the list must be a vector or a factor.
- All the vectors and factors must be the same length.
- Each element of the list must have a unique name.

Let's check the class of the `iris` data frame:

```
class( iris )
```

```
## [1] "data.frame"
```

But also note that the `iris` data frame is also a list:

```
is.list( iris )
```

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

This explains some curious features of data frames, such as the distinction between single and double brackets when indexing.

For instance, we can select the `Sepal.Width` column from the `iris` data frame as a data frame by using single square brackets:

```
sepal.width.data.frame <-
    iris[ "Sepal.Width" ]

head( sepal.width.data.frame )
```

```
##   Sepal.Width
## 1         3.5
## 2         3.0
## 3         3.2
## 4         3.1
## 5         3.6
## 6         3.9
```

Let's check the class of this object:

```
class( sepal.width.data.frame )
```

```
## [1] "data.frame"
```

Thus, this R object is a data frame.

But it's also a list:

```
is.list( sepal.width.data.frame )
```

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

On the other hand, if we use double square bracket indexing, we have:

```
head( sepal.width.data.frame )
```

```
##   Sepal.Width
## 1         3.5
## 2         3.0
## 3         3.2
## 4         3.1
## 5         3.6
## 6         3.9
```

Now you can see that this is a numeric vector:

```
class( sepal.width.data.frame )
```

```
## [1] "data.frame"
```

Lists are an important and powerful tool in R programming, because they are the most general type of data structure.

We'll only scratch the surface of this important topic in CSCI E-5a, but it's something that you should investigate further on your own.

So that's the relationship between data frames and lists.

Now let's learn about an amazing string function that can be very useful.

Section 3: The `strsplit()` Function

> **Main Idea:** *We can split a string into substrings*

In this section, we'll see how to split a character string into substrings.

We've previously seen that the `substr()` function can select a substring from a character string value.

With the `substr()` function we must specify the starting and ending locations of the substring.

Thus, there had to be some sort of regular structure that we could exploit.

For instance, suppose we have a character string vector which consists of strings that have an 4-digit internal numeric code followed by a three-letter brand identifier:

```
id.brand.vector <-
    c(
        "4136-SBZ",
        "8725-HKT",
        "2119-SBZ"
    )
```

If we just want the three-letter brand identifiers, we can use the `substr()` function to select just these substrings:

```
substr(
    x = id.brand.vector,
    start = 6,
    stop = 8
)
```

```
## [1] "SBZ" "HKT" "SBZ"
```

This strategy works nicely in this situation, but it relied on the fact that the three-letter brand identifiers were always in the same position, starting in location 6 and ending in location 8.

Sometimes however you have data that doesn't fit such a nice pattern.

For instance, suppose both the internal numeric code and the brand identifier strings can have different lengths:

```
id.brand.vector.2 <-
    c(
        "112-Classic WiDgT",
        "487739-WiDgT 3k",
        "0304-Quadcore WiDgT",
        "61549873-Classic WiDgT",
        "96731-WiDgT 2.0"
    )
```

This is more difficult to work with, because the brand identifiers don't start or end in a consistent location, so it's hard to specify the `start` and `stop` values for the `substr()` function.

However, there *is* one regular structure: the internal numeric code and the brand identifier are always separated by a hypen character '-'.

Thus, we can obtain the brand identifiers by *splitting* the string on the hyphen character, which acts as a delimiter.

For instance, we want to take the first character string "112-Classic WiDgT" and split it into the substrings "112" and "Classic WiDgT".

The `strsplit()` function takes two input arguments:

- The first input argument, denoted `x`, is the data string, which is the main string that we want to split into substrings.
- The second input argument, denoted `split` is the splitting string, which is the string of characters that serves as the delimiter.

The `strsplit()` function then returns a vector consisting of the substrings delimited by the splitting string.

So, for this example, the string "112-Classic WiDgT" is the data string, and the string "-" is the splitting string.

When we run the `strsplit()` function, it returns the the substrings "112" and "Classic WiDgT".

There's a subtle point about the `strsplit()` function.

Recall that we were able to run the `substr()` function on a vector of character string values, and it returns a vector consisting of each substring:

```
substr(
    x = id.brand.vector,
    start = 6,
    stop = 8
)
```

```
## [1] "SBZ" "HKT" "SBZ"
```

This works, because the `substr()` function only returns a single atomic value for each element of the input data string, so we can combine these atomic values into a vector.

We can also run the `strsplit()` function on a vector of character string values, just like the `substr()` function.

However, the `strsplit()` function will in general return a vector of character strings for each element of the input data string.

So here's a puzzle for you: what class of data structure does `strsplit()` have to use to return its results?

Think about this for a little bit . . .

OK, time's up!

The answer is: `strsplit()` has to return its results by using a . . . LIST!!

Remember, for each element of the input string, `strsplit()` will return a character string vector, so it has to somehow bundle these vectors together into a single object.

The only R data structure that can combine vectors is a list.

So `strsplit()` HAS to return its result in a list structure.

I know this might seem a little abstract, so let's run some code and look at the output:

```
strsplit(
    x = id.brand.vector.2,
    split = "-"
)
```

```
## [[1]]
## [1] "112"           "Classic WiDgT"
##
## [[2]]
## [1] "487739"   "WiDgT 3k"
##
## [[3]]
## [1] "0304"           "Quadcore WiDgT"
##
```

```
## [[4]]
## [1] "61549873"      "Classic WiDgT"
##
## [[5]]
## [1] "96731"     "WiDgT 2.0"
```

And now you can see that we've split all the strings into their component substrings.

Sometimes lists in R feel a little abstract, and you might wonder if you would ever actually use them.

The `strsplit()` function is a great example of a situation where the structure of the output data requires that you use a list.

Even if you never actually use the `strsplit()` function in practice, it's a good idea to study it, because you'll learn a lot about lists with this simple example.

Finally, here's a little trick with the `strsplit()` function.

If the splitting string is actually the empty string " ", then the `strsplit()` function will split the data string into individual characters:

For instance, we want to split the string "Hi Mom!" into the 7 characters that are in this string.

To do this, we first use the `strsplit()` function with an empty `split` string, which will return a list consisting of the 7 characters of the string:

```
strsplit(
    x = "Hi Mom!",
    split = ""
)
```

```
## [[1]]
## [1] "H" "i" " " "M" "o" "m" "!"
```

This is nice, but it's returned to us in a list structure, and this might not be convenient for us.

Instead, we might just want this data as a vector instead of a list.

One way to fix this is to use the `unlist()` function:

```
unlist(
    strsplit(
        x = "Hi Mom!",
        split = ""
    )
)
```

```
## [1] "H" "i" " " "M" "o" "m" "!"
```

We can even put this into a little function:

```
split.string.to.chars <-
    function( data.string ) {

        return(
            unlist(
```

```
                strsplit(
                    x = data.string,
                    split = ""
                )
            )
        )
}
```

Let's try this out:

```
split.string.to.chars( "Hi Mom!" )
```

```
## [1] "H" "i" " " "M" "o" "m" "!"
```

Cute!

So that's how to split a string.

Now let's review what we've learned in this module.

Module Review

In this module, we met another important compound data structure: the *list*

- In Section 1, we defined lists, and learned the basic properties of lists.
- In Section 2, we explored the relationship between lists and data frames.
- In Section 3, we saw how to split a character string into substrings.

Now that you've completed this module, you should be able to:

- Define the concept of the list data structure, and explain its basic properties and methods.
- Explain the relationship between data frames and lists.
- Use the `strsplit()` function to split a string into a set of substrings.

There were five new built-in R functions in this module:

- `list()`
- `as.list()`
- `is.list()`
- `unlist()`
- `strsplit()`

Allright, that's it for Module 1: Lists.

Now let's move on to Module 2: The Apply Functions.