Week 1 Module 2 – The Apply Functions

Let's clear the global computing environment:

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

Module Preview

Hello! And welcome to Module 2: The Apply Functions.

In this module, we'll study a family of functions that can apply a function to the elements of a list.

- In Section 1,

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

-

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

- `lapply()`
- `sapply()`
- `vapply()`
- `tapply()`

All right! Let's get started by learning how to apply a function to a list.

Section 2: Applying a Function to a List

Main Idea: *We can apply a function to the elements of a list*

In this section, we'll see how to apply a function to the elements of a list.

Often, we'll want to call a function on each column of a data frame.

For instance, consider the `iris()` data frame, which consists of four columns with numeric data and one column with categorical data.

Suppose we would like to use the `mean()` function to summarize the values in each of the numeric columns.

In this case, we will take each numeric column in turn and calculate the sample mean of their elements.

This is called "mapping" the `mean()` function over the columns of the data frame.

It's also called "applying" the function across the columns of the data frame.

Let's see this in action.

First, since we are focused on working on the numeric data in the `iris` data frame, let's select just the columns with numeric data and store these in a variable:

```
numeric.iris <-
    iris[ 1:4 ]

head( numeric.iris )
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3.0          1.4         0.2
## 3          4.7         3.2          1.3         0.2
## 4          4.6         3.1          1.5         0.2
## 5          5.0         3.6          1.4         0.2
## 6          5.4         3.9          1.7         0.4
```

We can calculate the sample mean of the elements of the first column:

```
mean( numeric.iris[[ 1 ]] )
```

```
## [1] 5.843333
```

Thus, we would like to construct a vector like this:

```
c(
    mean( numeric.iris[[ 1 ]] ),
    mean( numeric.iris[[ 2 ]] ),
    mean( numeric.iris[[ 3 ]] ),
    mean( numeric.iris[[ 4 ]] )
)
```

```
## [1] 5.843333 3.057333 3.758000 1.199333
```

This is not very efficient code, since we seem to be typing the same thing over and over again, with a slight variation.

Is there some way we can automate this process?

We might try something like this:

```
mean( iris[[ 1:4 ]] )
```

```
## Error in .subset2(x, i, exact = exact): recursive indexing failed at level 2
```

Unfortunately, the `mean()` function can take only numeric or logical vectors as its input argument, but we are trying to use a data frame as the input argument, so this generates an error.

So we can't map the `mean()` function across the columns of numeric data in the `iris` data frame using this approach.

However, there *is* a way to do this, but we have to use some special functions to do this.

The basic function for mapping a function across the columns of a data is called `lapply()`, and this takes two input arguments:

- The data frame, denoted X.
- The function to be mapped across the columns of X, denoted FUN.

The lapply() function then returns a list of the same length as the input data frame X, consisting of the output of the function FUN for each of the columns of X.

Notice that the two input argument names X and FUN use all-capital letters and thus do not conform to our standard naming conventions.

Let's see lapply() do this:

```
lapply(
    X = numeric.iris,
    FUN = mean
)
```

```
## $Sepal.Length
## [1] 5.843333
##
## $Sepal.Width
## [1] 3.057333
##
## $Petal.Length
## [1] 3.758
##
## $Petal.Width
## [1] 1.199333
```

Notice that lapply() returns the sample mean values in a named list.

That's the "l" part of "lapply()" – it means that the function returns a list.

There are two other functions that are variations on "lapply()".

The sapply() function will try to simply the return result:

```
sapply(
    X = numeric.iris,
    FUN = mean
)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

Here it was possible to return the return values in a vector (actually, a named vector), so that's what sapply() does.

If you're working interactively, it's often more convenient to use sapply() rather than lapply(), because it can be annoying to get the return values encased in a list structure.

However, there is a theoretical objection to sapply(), because the function determines how to simplify

There is a theoretical objection to sapply(): the function uses an internal algorithm to decide what "simple" means in any context, so it might return a vector in one case and a list in another.

Thus, there is another function called vapply(), where you supply a template data structure, and thus you can control exactly the data structure that is returned.

This is all very specialized, and usually `sapply()` is just fine.

Remember that ultimately everything is just a variation of `lapply()`.

You should be aware that `lapply()` is extremely flexible, and the first input argument can be any sort of list, so this is very general.

Of course, a data frame is a special case of a list, so that's why we can use `lapply()` with a data frame.

So that's how to apply a function to the elements of a list.

Now let's see an application of this idea.

Section 3: Application

Module Review