Introduction to R: **Functions**

Day 4, Part B

In this lecture

1. Viewing function code
2. Defining functions
3. Scoping
4. Lists
5. `apply()` and `lapply()`

A review of what we already know...

R functions are used to transform input into output in some way.

```
> log(x = 300, base = 10)
[1] 2.477121
```

1. Function name: `log()`
2. Argument name(s): `x, base`
3. Argument value(s): `300, 10`
4. Output: `2.4771213`

Function code

Most existing R functions are themselves written at least partially in R.

Function code

Most existing R functions are themselves written at least partially in R.

You can view the underlying source code either by typing the function name directly in the console:

```
> read.csv
function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
    fill = TRUE, comment.char = "", ...)
read.table(file = file, header = header, sep = sep, quote = quote,
    dec = dec, fill = fill, comment.char = comment.char, ...)
<bytecode: 0x0000000014d3c068>
<environment: namespace:utils>
```

or via the `View()` function:

```
> View(read.csv)
```

Functions as objects

These operations—typing the function name directly in the console, or using the `View()` function—work because functions are themselves objects.

Consequently, they are similarly defined using an assignment operator (= or <-) and, in this case, the `function()` function:

```
> plus_one <- function(x) {
+     y <- x + 1
+     return(y)
+ }
>
> plus_one(10)
[1] 11
>
> class(plus_one)
[1] "function"
```

User-defined functions

You can easily add your own functions! There's tons of reasons to do so, including:

1. Copy-and-paste is bad! If you find yourself doing essentially the same thing over and over, you should save yourself the trouble by writing a function. This has the added benefit of making it easier to update your code if needed, since you need only update in one place.

2. Functions are one way of breaking up your code into simpler parts that each (ideally) do one discrete task, making your code easier to read and easier to maintain.

3. Being able to define your own functions makes it possible to be much more efficient with certain other R functions (more on this later. . .)

Defining a function

Function definitions typically take the form:

```
> function_name <- function(arg_1, arg_2, ...) {
+     expression
+     return(value)
+ }
```

Defining a function

Function definitions typically take the form:

```
> function_name <- function(arg_1, arg_2, ...) {
+     expression
+     return(value)
+ }
```

For example, here we define a function named `plus_one` that takes a single argument `x`, includes a code block that defines `y` to be equal to `x + 1`, and then returns the value of `y`:

```
> plus_one <- function(x) {
+     y <- x + 1
+     return(y)
+ }
> plus_one(10)
[1] 11
```

Defining a function

It is not technically necessary to call `return()` explicitly at the end of the function. In this case, R will assume you want to return the output of the final line of code.

```
> plus_one <- function(x) {
+     y <- x + 1
+ }
> plus_one(10)
> print(plus_one(10))
[1] 11
```

```
> plus_one <- function(x) {
+     x + 1
+ }
> plus_one(10)
[1] 11
```

Defining a function

Very simple, one line functions can skip the brackets entirely.

```
> plus_one <- function(x) x + 1
> plus_one(10)
[1] 11
```

Defining a function

It is possible to supply defaults for some or all arguments using an = when listing the arguments.

```
> x_plus_y <- function(x, y) {
+     x + y
+ }
> x_plus_y()
Error in x_plus_y(): argument "x" is missing, with no default
```

```
> x_plus_y <- function(x = 5, y = 10) {
+     x + y
+ }
> x_plus_y()
[1] 15
> x_plus_y(x = 10)
[1] 20
> x_plus_y(x = 10, y = 20)
[1] 30
```

Scoping!

Scoping is the set of rules that link an object's value to its name.

Scoping!

Scoping is the set of rules that link an object's value to its name.

This can get quite complicated, but the important part when working with functions is that R looks first within the environment of the function, and if it doesn't find the object name it's looking for, it moves up a level (based on where the function is defined) and looks again.

```
> x_plus_y_minus_z <- function(x, y) {
+     x + y - z
+ }
>
> x <- 10
> y <- 10
> z <- 10
> x_plus_y_minus_z(1, 1)
[1] -8
```

Scoping!

Related to this, object/value pairs assigned within a function exist only in that function.

```
> x_plus_y_minus_z <- function(x, y) {
+     val1 <- x + y
+     val2 <- val1 - z
+     return(val2)
+ }
>
> x_plus_y_minus_z(1, 1)
[1] -8
> val1
Error in eval(expr, envir, enclos): object 'val1' not found
```

A quick(?) aside about lists

So far for data structures, we've talked about vectors, matrices, arrays, and data frames. There's one more basic data structure: lists.

	Homogeneous	Heterogeneous
1D	Vector*	List
2D	Matrix	Data frame
3D	Array	

A quick(?) aside about lists

So far for data structures, we've talked about vectors, matrices, arrays, and data frames. There's one more basic data structure: lists.

	Homogeneous	Heterogeneous
1D	Vector*	List
2D	Matrix	Data frame
3D	Array	

*What we have called vectors up until now are more properly called 'atomic vectors', as lists are technically also vectors. That said, it's common practice to use vector synonymously with atomic vector, and to call a list a list.

(For a *much* deeper dive: http://adv-r.had.co.nz/Data-structures.html)

A quick(?) aside about lists

Lists are vectors (ordered collection of elements) where each element can be any data type*.

```
> num <- 1:10
> df <- data.frame(lower = letters, upper = LETTERS)
> mat <- matrix(rnorm(9), nrow = 3)
>
> my_list <- list(num, df, mat)
> str(my_list)
List of 3
 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ :'data.frame':   26 obs. of  2 variables:
  ..$ lower: Factor w/ 26 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10 ...
  ..$ upper: Factor w/ 26 levels "A","B","C","D",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ : num [1:3, 1:3] 1.361 -0.6 2.187 1.533 -0.236 ...
```

A quick(?) aside about lists

Lists are vectors (ordered collection of elements) where each element can be any data type*.

```
> num <- 1:10
> df <- data.frame(lower = letters, upper = LETTERS)
> mat <- matrix(rnorm(9), nrow = 3)
>
> my_list <- list(num, df, mat)
> str(my_list)
List of 3
 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ :'data.frame':   26 obs. of  2 variables:
  ..$ lower: Factor w/ 26 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10 ...
  ..$ upper: Factor w/ 26 levels "A","B","C","D",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ : num [1:3, 1:3] 1.361 -0.6 2.187 1.533 -0.236 ...
```

*Including lists! Nested lists are fun!!!

A quick(?) aside about lists

Specific elements of a list can be retrieved by index, using double brackets:

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

A quick(?) aside about lists

Specific elements of a list can be retrieved by index, using double brackets:

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

And if a list is named, using $ notation, similar to that used for columns in a data frames*:

```
> my_list <- list(numeric = num, data.frame = df, matrix = mat)
> my_list$matrix
           [,1]       [,2]       [,3]
[1,]  1.3606524  1.5326106 -0.7104066
[2,] -0.6002596 -0.2357004  0.2568837
[3,]  2.1873330 -1.0264209 -0.2466919
```

A quick(?) aside about lists

Specific elements of a list can be retrieved by index, using double brackets:

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

And if a list is named, using $ notation, similar to that used for columns in a data frames*:

```
> my_list <- list(numeric = num, data.frame = df, matrix = mat)
> my_list$matrix
           [,1]       [,2]       [,3]
[1,]  1.3606524  1.5326106 -0.7104066
[2,] -0.6002596 -0.2357004  0.2568837
[3,]  2.1873330 -1.0264209 -0.2466919
```

*data frames are technically also lists (each column is an element of the list), just highly structured ones, which is why the syntax is similar.

A quick(?) aside about lists

We care about lists in the context of a lecture on functions because an R function can only return a single object. So if you want to return lots of objects, that are potentially all different types and don't combine nicely, you stick them into a list and then return that list.

```
> random_fun <- function(n) {
+     one_dim <- rnorm(n)
+     two_dim <- matrix(rnorm(n * n), nrow = n)
+     all <- list(one_dim = one_dim, two_dim = two_dim)
+     return(all)
+ }
>
> random_fun(2)
$one_dim
[1] -0.3475426 -0.9516186

$two_dim
            [,1]       [,2]
[1,] -0.04502772 -1.6679419
[2,] -0.78490447 -0.3802265
```

A quick(?) aside about lists

You have already seen at least one function that does this (`lm()`) and this is extremely common for more complicated functions that have complex output.

```
> mod <- lm(price ~ carat, data = diamonds)
> class(mod)
[1] "lm"
> str(mod)
List of 12
 $ coefficients : Named num [1:2] -2256 7756
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "carat"
 $ residuals    : Named num [1:53940] 798 954 799 341 187 ...
  ..- attr(*, "names")= chr [1:53940] "1" "2" "3" "4" ...
 $ effects      : Named num [1:53940] -913392 853890 791 333 179 ...
  ..- attr(*, "names")= chr [1:53940] "(Intercept)" "carat" "" "" ...
 $ rank         : int 2
 $ fitted.values: Named num [1:53940] -472 -628 -472 -7 148 ...
  ..- attr(*, "names")= chr [1:53940] "1" "2" "3" "4" ...
 $ assign       : int [1:2] 0 1
 $ qr           :List of 5
  ..$ qr   : num [1:53940, 1:2] -2.32e+02 4.31e-03 4.31e-03 4.31e-03 4.31e-03 .
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:53940] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:2] "(Intercept)" "carat"
  .. ..- attr(*, "assign")= int [1:2] 0 1
  ..$ qraux: num [1:2] 1 1.01
  ..$ pivot: int [1:2] 1 2
```

The `apply` family

R contains a number of functions in the `apply` family that are intended primarily as a compact alternative to `for` loops.

- `apply()` – apply function to margins of a matrix/data frame
- `lapply()` – apply function to the elements of a vector or list
- `sapply()` – same as `lapply()`, but with simplified output
- `tapply()` – apply a function to a vector, stratified by a second vector
- `mapply()` – apply a function to multiple vectors or lists of the same length
- etc.

apply()

The `apply()` function allows for repeating the same set of operations along the margins of a matrix or data frame.

This essentially the same as using a `for` loop over rows or columns:

```
> VADeaths
      Rural Male Rural Female Urban Male Urban Female
50-54       11.7          8.7       15.4          8.4
55-59       18.1         11.7       24.3         13.6
60-64       26.9         20.3       37.0         19.3
65-69       41.0         30.9       54.6         35.1
70-74       66.0         54.3       71.1         50.0
> colsums <- rep(NA, nrow(VADeaths))
> for (r in 1:nrow(VADeaths)) {
+     colsums[r] <- sum(VADeaths[r, ])
+ }
> colsums
[1]  44.2  67.7 103.5 161.6 241.4
```

But `apply()` is often far more compact:

```
> colsums <- apply(VADeaths, 1, sum)
> colsums
50-54 55-59 60-64 65-69 70-74
 44.2  67.7 103.5 161.6 241.4
```

lapply()

The `lapply()` function allows for repeating the same set of operations on each element of a vector or list, and returns a list.

In this example, we start with a vector and end with a list of the same length:

```
> lapply(1:3, log)
[[1]]
[1] 0

[[2]]
[1] 0.6931472

[[3]]
[1] 1.098612
```

lapply()

And in this example, we start with a list and end with another list of the same length:

```
> data_list <- list(diamonds, economics, presidential)
> lapply(data_list, names)
[[1]]
 [1] "carat"   "cut"     "color"   "clarity" "depth"
 [6] "table"   "price"   "x"       "y"       "z"

[[2]]
[1] "date"     "pce"      "pop"      "psavert"  "uempmed"
[6] "unemploy"

[[3]]
[1] "name"  "start" "end"   "party"
```

User-defined functions in apply family functions

User-defined functions make apply family functions much more powerful, because you can do essentially anything, rather than being restricted to operations for which a single simple function already exists.

```
> cv <- function(x) sd(x)/mean(x)
> apply(VADeaths, 2, cv)
  Rural Male Rural Female   Urban Male Urban Female
   0.6596253    0.7317007    0.5578668    0.6749732
```

```
> summarize_dims <- function(df) {
+     data.frame(nrow = nrow(df), ncol = ncol(df), NAs = sum(is.na(df)))
+ }
> lapply(data_list, summarize_dims)
[[1]]
   nrow ncol NAs
1 53940   10   0

[[2]]
  nrow ncol NAs
1  574    6   0

[[3]]
  nrow ncol NAs
1   11    4   0
```

User-defined functions in apply family functions

Conveniently, user-defined functions can be defined **within** apply family functions.

```
> apply(VADeaths, 2, function(x) sd(x)/mean(x))
  Rural Male Rural Female   Urban Male Urban Female 
   0.6596253    0.7317007    0.5578668    0.6749732 
```

```
> lapply(data_list, function(df) {
+     data.frame(nrow = nrow(df), ncol = ncol(df), NAs = sum(is.na(df)))
+ })
[[1]]
   nrow ncol NAs
1 53940   10   0

[[2]]
  nrow ncol NAs
1  574    6   0

[[3]]
  nrow ncol NAs
1   11    4   0
```

User-defined functions in other functions

User-defined functions can also be supplied to any function that takes a function as an argument (e.g., `dcast()`)

In general, there is nothing special about user-defined functions—they can be used in all the same situations and manners that functions from base R or various packages are used.

do.call()

It is very common to use lapply() to either load or otherwise transform data such that the output is a list of similar data frames.

```
> data_list <- lapply(data_list, function(df) {
+     data.frame(nrow = nrow(df), ncol = ncol(df), NAs = sum(is.na(df)))
+ })
> data_list
[[1]]
   nrow ncol NAs
1 53940   10   0

[[2]]
  nrow ncol NAs
1  574    6   0

[[3]]
  nrow ncol NAs
1   11    4   0
```

do.call()

The goal is then usually to combine everything into one big data frame via rbind(). This can be accomplished with the do.call() function:

```
> data_list <- do.call(rbind, data_list)
> data_list
   nrow ncol NAs
1 53940   10   0
2   574    6   0
3    11    4   0
```

In general, do.call() is used to feed a list of arguments to a function. This is most often useful with functions that have ... as the first argument, which means that it takes an arbitrary number of the first argument (e.g., rbind(), cbind(), grid.arrange(), etc.).