Introduction to R

Afternoon

Afternoon Session

- The afternoon is to be a more informal session, either continuing the main tutorial or looking at other tutorial topics.
- We've found some of the tutorial chapters to be a bit confusing.
  - This short presentation tries to make some of those topics a bit clearer.

Functions

The definition of an R function looks like this:

```
some_function <- function(arg1, arg2 = 2) {
    out <- arg1 + arg2 * 7
    return(out)
}
```

Functions

The definition of an R function looks like this:

```
some_function <- function(arg1, arg2 = 2) {
    out <- arg1 + arg2 * 7
    return(out)
}
```

This is the name of the function. Once you have made it you can call it like any other function:

```
> some_function(3)
```

Functions

The definition of an R function looks like this:

```
some_function <- function(arg1, arg2 = 2) {
    out <- arg1 + arg2 * 7
    return(out)
}
```

This portion tells R that we are creating a function. We are assigning it to a variable, like we would anything else, but in this case the object we are making is a function.

Functions

The definition of an R function looks like this:

```
some_function <- function(arg1, arg2 = 2) {
    out <- arg1 + arg2 * 7
    return(out)
}
```

These are the arguments to the function. Each of these is one argument the function can be given when you use it. The second argument, here, is given a default value. If you don't give a second argument when using this function, it uses the default.

Functions

The definition of an R function looks like this:

```
some_function <- function(arg1, arg2 = 2) {
    out <- arg1 + arg2 * 7
    return(out)
}
```

The curly brackets go around the code for the function, and end the function definition.

Functions

The definition of an R function looks like this:

```
some_function <- function(arg1, arg2 = 2) {
    out <- arg1 + arg2 * 7
    return(out)
}
```

Inside those braces you can put any R code you need to. However, any assignment you do inside the function only applies for the rest of the function. Here, 'out' will not be assigned any value in our main R session.

Functions

The definition of an R function looks like this:

```
some_function <- function(arg1, arg2 = 2) {
    out <- arg1 + arg2 * 7
    return(out)
}
```

We usually want the function to return a value. This is done using a return statement.

Functions

- Some functions are built-in, or part of an R library package.
- Other functions (such as ones you've written yourself) are stored alongside the data objects and show up when you use objects().
  - If you write function with the same name as a built-in function, the built-in function will be masked, which you probably don't want.
  - Accidentally masking important functions like c() and list() can break R in all kinds of exciting ways.
  - You can undo the damage by deleting the problem function, and built-in functions cannot be removed in this way.

Blocks

- A block is a chunk of R code, contained between { and }.
  - The body of a function is a block.
  - When you define a block, R doesn't process any of the commands until the end of the block.
  - Blocks are also used for other programming features:
    - Conditional statements
    - Loops

Controlling Flow

- Conditional statements are used to send the code down different paths depending on the data.

```
if( value < 5 ) {
    value <- 0
    my.mode <- "small"
} else {
    my.mode <- "large"
}
```

Controlling Flow

```
if( value < 5 ) {
    value <- 0
    my.mode <- "small"
} else {
    my.mode <- "large"
}
```

This defines the main 'if' block. The 'if' statement looks like a bit like a function, but it isn't one.

Controlling Flow

```
if( value < 5 ) {
    value <- 0
    my.mode <- "small"
} else {
    my.mode <- "large"
}
```

This is a logical (TRUE or FALSE) value. You can put anything here that evaluates to a logical value.

Controlling Flow

```
if( value < 5 ) {
    value <- 0
    my.mode <- "small"
} else {
    my.mode <- "large"
}
```

This is the code that will be executed if and only if the condition inside the brackets is TRUE.

Controlling Flow

```
if( value < 5 ) {
    value <- 0
    my.mode <- "small"
} else {
    my.mode <- "large"
}
```

The 'else' statement contains the code that is executed if and only if the condition was FALSE. This part is optional.

Controlling Flow

```
if( value < 5 ) {
    value <- 0
    my.mode <- "small"
} else {
    my.mode <- "large"
}
```

Here is the code for the 'else' block.

Loops

- Loops allow you do do the same (or a similar) thing several times.

```
cumulative <- 0
for( n in some.vec ) {
    cumulative <- cumulative + n^2
}
```

Loops

```
cumulative <- 0
for( n in some.vec ) {
    cumulative <- cumulative + n^2
}
```

This is how a 'for' block is set up. The 'in' is a special keyword that R recognises.

Loops

```
cumulative <- 0
for( n in some.vec ) {
    cumulative <- cumulative + n^2
}
```

This is a vector values, of any length. The length of the vector determines how many times the loop will run.

Loops

```
cumulative <- 0
for( n in some.vec ) {
    cumulative <- cumulative + n^2
}
```

This is a variable name, which can be anything we want. Each time we go through the loop, this variable will have a different value. It steps through each of the values in *some.vec*.

Loops

```
cumulative <- 0
for( n in some.vec ) {
    cumulative <- cumulative + n^2
}
```

This is the code that is executed each time we go through the loop. Remember that *n* is a variable containing the current value from *some.vec*. When we assign inside the loop, we replace whatever value *cumulative* held before.

Loops

- Vectors in R provide a way of performing the same action on an entire vector. This can eliminate the need to use a for loop completely. The example can be done in a single line:

```
cumulative <- sum(some.vec^2)
```

- Although using vectors in this way is quicker, it is often harder to understand. Worrying about vectorising calculations is for advanced R users.

Afternoon Session

- Thank you for listening!