

# Writing your own functions

So far we have relied on the built-in functionality of R to carry out our analyses. We will cover:

- How to write your own functions
- How to use flow control mechanisms like if and for
- Debugging your code when something goes wrong
- The meaning of “environments” and “variable scope”
- Timing and speeding up your code

# Function we will write today

Convert a vector of measurements in inches into centimeters

## Writing your own functions in R

Think about the code we have been writing so far in R. It has been

- made up of a sequence of commands, one after another
- specific to the particular dataset we're working with.

Functions allow us to

- organize our code into tasks
- reuse the same code on different datasets by making the data an *argument* to the function.

Our task is to convert inches into centimeters for family height:

$$\text{family\$height} * 2.54$$

We can make a conversion function InToCm function that looks like:

`convert = function(x) where function(x) is  $x * 2.54$`

This function *encapsulates* the multiplication of each element by 2.54 for *any* x vector, not just the family\$height.

## Anatomy of a function

The syntax for writing a function is

```
function ( arguments ) body
```

Typically we assign the function to a particular name. This should describe what the function does.

```
myFunction = function (arguments) body
```

A function without a name is called an “orphan” function. These can be very powerfully used with the apply mechanism. Stay tuned....

```
function ( arguments ) body
```

The keyword `function` just tells R that you want to create a function.

Recall that the *parameters* to a function are its inputs, which may have default values.

```
> args(median)  
function (x, na.rm = FALSE)
```

Here, if we do not explicitly specify `na.rm` when we call `median`, it will be assigned the default value of `FALSE`.

A few notes on specifying the arguments:

When you're writing your own function, it's good practice to put the most important arguments first. Often these will not have default values.

This allows the user of your function to easily specify the arguments by position, eg. `plot(xvec, yvec)` rather than `plot(x = xvec, y = yvec)`.



Next we have the *body* of the function, which typically consists of expressions surrounded by curly brackets. Think of these as performing some operations on the input values given by the arguments.

```
{  
    expression 1  
    expression 2  
    return(value)  
}
```

The `return` expression hands control back to the caller of the function and returns a given `value`. If the function returns more than one thing, this is done using a named list, for example

```
return(list(total = sum(x), avg = mean(x))).
```

In the absence of a return expression, a function will return the *last* evaluated expression. This is particularly common if the function is short, for example:

```
convert = function(x) where function(x) is x*2.54
```

Here we don't need brackets {}, since there is only one expression. So:

```
convert = function(x) x*2.54
#but we could also write:
convert = function(x)
{
  x*2.54
}
```

A return expression anywhere in the function will cause the function to return control to the user *immediately*, without evaluating the rest of the function.

## Considerations when writing a function:

- What will the function do?
- What should we call it? (Relate the name to what it does, avoid overwriting other functions)
- What will be the arguments?
- Which arguments have default values and what are they?
- What (if anything) should the function return?

# Extend our function

Make `convert()` work in both directions  
(inches to cm or cm to inches)

# Control Flow

Computing in R consists of sequentially evaluating statements. *Flow control* structures allow us to control which statements are evaluated and in what order.

In R the primary ones consist of

- if/else statements and the ifelse function
- for and while loops

Expressions can be grouped together using curly braces “{” and “}”. A group of expressions is called a *block*. For today’s lecture, the word *statement* will refer to either a single expression or a block.

The basic syntax for an if/else statement is

```
if ( condition ) {  
    statement1  
} else {  
    statement2  
}
```

First, condition is evaluated. If the first element of the result is TRUE then statement1 is evaluated. If the first element of the result is FALSE then statement2 is evaluated. *Only the first element of the result is used.*

If the result is numeric, 0 is treated as FALSE and any other number as TRUE. If the result is not logical or numeric, or if it is NA, you will get an error.

When we discussed Boolean algebra before, we met the operators `&` (AND) and `|` (OR).

Recall that these are both *vectorized* operators.

If/else statements, on the other hand, are based on a single, “global” condition. So we often see constructions using `any` or `all` to express something related to the whole vector, like

```
if ( any(x < -1 | x > 1) )  
  warning("Value(s) in x outside the interval [-1,1]")
```

(We'll discuss error handling more later.)



The result of an if/else statement can be assigned. For example,

```
➤ if ( any(x <= 0) ) y = log(1+x) else y = log(x)
```

is the same as

```
> y = if ( any(x <= 0) ) log(1+x) else log(x)
```

Also, the else clause is optional. Another way to do the above is

```
> if( any(x <= 0) ) x = 1+x  
> y = log(x)
```

Note that this version this changes x as well.

If/else statements can be nested.

```
if (condition1 )  
    statement1  
else if (condition2)  
    statement2  
else if (condition3)  
    statement3  
else  
    statement4
```

The conditions are evaluated, in order, until one evaluates to TRUE. The the associated statement/block is evaluated. The statement in the final else clause is evaluated if none of the conditions evaluates to TRUE.

## A note about formatting if/else statements:

When the if statement is not in a block, the else (if present) must appear on the same line as `statement1` or immediately following the closing brace. For example,

```
if (condition) {statement1}  
else {statement2}
```

will be an error if not part of a larger block and/or function. I suggest using the format

```
if (condition) {  
    statement1  
} else {  
    statement2  
}
```

## Some common uses of if/else clauses

### 1. With logical arguments to tell a function what to do

```
corplot = function(x, y, plotit = TRUE){  
  if ( plotit == TRUE ) plot(x, y)  
  cor(x,y)  
}
```

### 2. To verify that the arguments of a function are as expected

```
if ( !is.matrix(m) )  
  stop("m must be a matrix")
```

### 3. To handle common numerical errors

```
ratio = if ( x!=0 ) y/x else NA
```


### 4. In general, to control which block of code is executed

```
normt = function(n, dist){  
  if ( dist == "normal" ){  
    return( rnorm(n) )  
  } else if (dist == "t"){  
    return(rt(n, df = 1, ncp = 0))  
  } else stop("distribution not implemented")  
}
```

These if/else constructions are useful for global tests, not tests applied to individual elements of a vector.

However, there is a vectorized function called `ifelse`.

```
> args(ifelse)  
function (test, yes, no)
```



R object that can be coerced to logical

R objects of the same size as test

The diagram consists of two arrows. One arrow originates from the text 'R object that can be coerced to logical' and points to the argument 'test' in the function signature. The other arrow originates from the text 'R objects of the same size as test' and points to the argument 'no' in the function signature.

For each element of `test`, the corresponding element of `yes` is returned if the element is `TRUE`, and the corresponding element of `no` is returned if it is `FALSE`.

## Some examples of ifelse

```
ratio = ifelse(x!=0, y/x, NA) # (Compare with earlier)
```

```
US.indicator = ifelse(country == "USA", 1, 0)
```

```
plot(Income, Donations,  
      col = ifelse(party == "Republican", "red", "blue"))
```

# Your turn

- Write a function that makes a line plot for  $x$  and  $y$  inputs.
- Extend the function to optionally fill the region between the line and the  $x$  axis with color
- Extend the plot to add vertical lines at the positions supplied, if any are supplied



# Anonymous functions

# Recall

Here is an expression to calculate the number of years that a weather station had been in operation

```
> length(unique(floor(rain[[1]])))
```

We can wrap this expression into a function:

```
numYears = function(y) {  
  length(unique(floor(y)))  
}
```

# Apply this function to `rain`

```
apply(rain, numYears)
```

We don't actually have to go through the hassle of writing a function definition.

We can use an anonymous function:

```
apply(rain, function(y)  
  length(unique(floor(y))))
```

# Recall

Here is the distance from a vector of  $x$ s and a vector of  $y$ s to a point with the expression:

```
sqrt( (x - pt[1])^2 + (y - pt[2])^2 )
```

Can you wrap this into a function?

```
distToPoint =  
  function(x, y, pt = c(0, 0)){  
    sqrt( (x - pt[1])^2 + (y - pt[2])^2 )  
  }
```

# Apply this function

Now we have two input argument that we would like to apply the function too

We can do this with mapply:

```
mapply(distToPoint, 1:10, 10:1)
```

But, what if we want to specify the parameter pt?

```
mapply(distToPoint, 1:10, 10:1,  
       MoreArgs = list(pt = c(1, 0)))
```

# The for statement

*Looping* is the repeated evaluation of a statement or block of statements.

Much of what is handled using loops in other languages can be more efficiently handled in R using vectorized calculations or one of the apply mechanisms.

However, certain algorithms, such as those requiring recursion, can only be handled by loops.

There are two main looping constructs in R: `for` and `while`.

## For loops

A *for loop* repeats a statement or block of statements a predefined number of times.

The syntax in R is

```
for ( var in vector ){  
  statement  
}
```

For each element in **vector**, the variable **var** is set to the value of that element and **statement** is evaluated.

**vector** often contains integers, but can be any valid type.



## While loops

A *while loop* repeats a statement or block of statements for as many times as a particular condition is TRUE.

The syntax in R is

```
while (condition){  
  statement  
}
```

`condition` is evaluated, and if it is `TRUE`, statement is evaluated. This process continues until condition evaluates to `FALSE`.

Exercise:

The expression

```
sample(1:0, size = 1, prob = c(p, 1-p))
```

simulates a random coin flip, where the coin has probability  $p$  of coming up heads, represented by a 1.

Write a function that simulates flipping a coin until a fixed number of heads are obtained. It should take the probability  $p$  and the total number of heads `total` and return the trial on which the final head was obtained. This produces a single sample from the *negative binomial distribution*.

The break statement causes a loop to exit. This is particularly useful with while loops, which, if we're not careful, might loop indefinitely (or until we kill R).

```
myRNG = function(total, p = 0.5){  
  # Simulate number of tosses to get 10 Heads  
  
  x = 0  
  steps = 0  
  max.iter = 1000  
  
  while(x < total){  
    x = x + sample(1:0, size = 1, prob = c(p, 1-p))  
    steps = steps + 1  
    if(steps >= max.iter){  
      warning("Maximum iteration reached")  
      break  
    }  
  }  
  return(steps)  
}
```