

# Functions

A **function** in R is a piece of code written to carry out a specified task. It takes inputs (arguments) and then applies some code to them (the body). From this it obtains an output (the return value).

R has a large number of in-built functions (sum, mean, sqrt, cos...) but the user can also create their own functions (called User Defined Functions (UDF)).

Functions are created using function().

**Basic syntax of an R function:**

*Be creative but consistent  
with function names.*

```
function_name <- function(arg1, arg2, ...) {  
  Function body  
}
```



## Example of a function

We will create a UDF which takes two numbers as input and returns their sum as output.

```
Add2Numbers <- function(x,y) { # body of function here
  z <- x + y
  return(z)
}
```

To call the function, we type:

```
> Add2Numbers(3,5)
[1] 8
```

Note: if you forget what the body of your function does, type the name of your function (without ()) and R will remind you.



# Function components

*This is how help files  
are structured, too.*

The different parts of a function are:

- ▶ **Function Name:** This is the actual name of the function. It is stored in R as an object with this name.  
Advice: Use an informative name for your function and make sure that the name that you choose is not an R reserved word.
- ▶ **Arguments:** These are the input variables in a function and they are placed in round brackets (arg1, arg2, ...).
- ▶ **Function Body:** The function body contains a collection of statements that defines what the function does. Statements are placed in curly braces {}. But if the body contains only a single statement then curly braces can be skipped. —
- ▶ **Return Value:** The return value of a function is the last expression in the function body to be evaluated.



# Calling a function with argument values

When a function is invoked, you pass a value to the argument. Note that:

- ▶ Arguments are optional (a function may contain no arguments).
- ▶ Arguments can have default values (see next slide).

Different equivalent ways to call a function:

```
> Add2Numbers(3,5)
> Add2Numbers(x=3,y=5) # use named arguments
> Add2Numbers(y=3,x=5) # with named arguments, the order does not matter.
> Add2Numbers(5,x=3) # use named and unnamed arguments in a single call.
```

When we use named and unnamed arguments in a single call, all the named arguments are matched first and then the remaining unnamed arguments are matched in a positional order.



## Calling a function with default arguments

We can assign default values to arguments in a function in R. This is done by providing an appropriate value to the formal argument in the function declaration.

```
Add2Numbers <- function(x,y=1) {  
  z <- x + y  
  return(z)  
}
```

Here, y is optional and will take the value 1 when not provided.

*Convenient for functions  
with A LOT OF arguments.*

```
> Add2Numbers(3)  
[1] 4  
> Add2Numbers(x=3,y=1)  
[1] 4  
> Add2Numbers(x=3,y=5)  
[1] 8
```



## Exercise

### Task

Write an R function `sum.of.squares` which:

1. Takes two arguments; the first argument is a vector named `x` and the second argument is variable named `location` with a default value of `mean(x)`.
2. Returns the sum of squares of deviations of `x` about the value of `location`.

For example, suppose your vector is  $x = (x_1, x_2, \dots, x_n)$  and you call your function with the default value for `location`. Then the function should return  $\sum_{i=1}^n (x_i - \text{mean}(x))^2$ .

Note: the default argument value is defined in terms of variables internal to the function.



## Exercise

### Task

The relationship between degrees celsius (°C) and (°F) is defined by:

$$T_f = \frac{9}{5} T_c + 32.$$

Write an R function `celsius.to.fahrenheit` which:

1. Takes one argument; a vector `x` of temperatures measured in °C.
2. Returns these temperatures in °F.

Find the boiling and freezing points of water in °F.



## Lazy evaluation and scoping

*Convenient for functions  
with A LOT OF arguments.*

```
Square <- function(a,b) {  
  s <- a^2  
  return(s)  
}
```

**Lazy evaluation:** R function arguments are not evaluated until the value of the argument is needed.

```
> Square(2) # the argument b is not actually used.  
[1] 4
```

**Scoping:** The scope of a variable is the region of code where that variable has meaning. In R, a local variable has meaning only within the function it is local to.

```
> s  
Error: object 's' not found
```





## Functions without return()

Unless told otherwise, a function returns the value of the last statement evaluated automatically. For example, the function `Square()` defined as:

```
Square <- function(a)  a^2
```

will return the value of  $a^2$ .

```
> Square(2)
[1] 4
```

We generally use explicit `return()` functions to return a value immediately from a function. But note that if we instead use:

```
Square <- function(a)  b <- a^2
```

then typing `Square(2)` will return no value. Why?




## Multiple returns

The return value of a function is the last expression in the function body to be evaluated.

```
f <- function(a,b) {  
  return(a)  
  return(b)  
}  
> f(2,3) # '2' got printed first before the function prematurely ended.  
[1] 2
```

The return() function can return only a single object. If we want to return multiple values in R, we can use a list (or other objects) and return it.

```
f <- function(a,b,c="Something else") {  
  d <- list(c(a,b),c)  
  return(d)  
}
```



# Exercise

## Task

Write an R function `multiple.return` which takes a vector `x` of length 4, and returns:

- ▶ the sum of the elements of `x`
- ▶ the product of the elements of `x`
- ▶ a  $2 \times 2$  matrix composed of the elements in `x`



## Some Principles of Good Coding

- ▶ **Keep it simple:** complicated logic for achieving a simple thing should be kept to a minimum
- ▶ **Keep it clean:** don't let a function have information it does not need.
- ▶ **Use meaningful names:** it makes it much easier to read the code.
- ▶ **Modularity:** Try to avoid re-writing the same lines of code to perform the same job. Create a function and call it whenever you need it. In general, try to break your code up into smaller, self-contained functions.
- ▶ **Commenting:** If you intend your code to be read by another programmer, leave comments so that they are able to understand what you were trying to achieve. At this stage, the most likely collaborator is yourself – so treat yourself nicely!

