MAP 2112 – Introdução à Lógica de Programação e Modelagem Computacional

1º Semestre - 2020

Prof. Dr. Luis Carlos de Castro Santos

lsantos@ime.usp.br

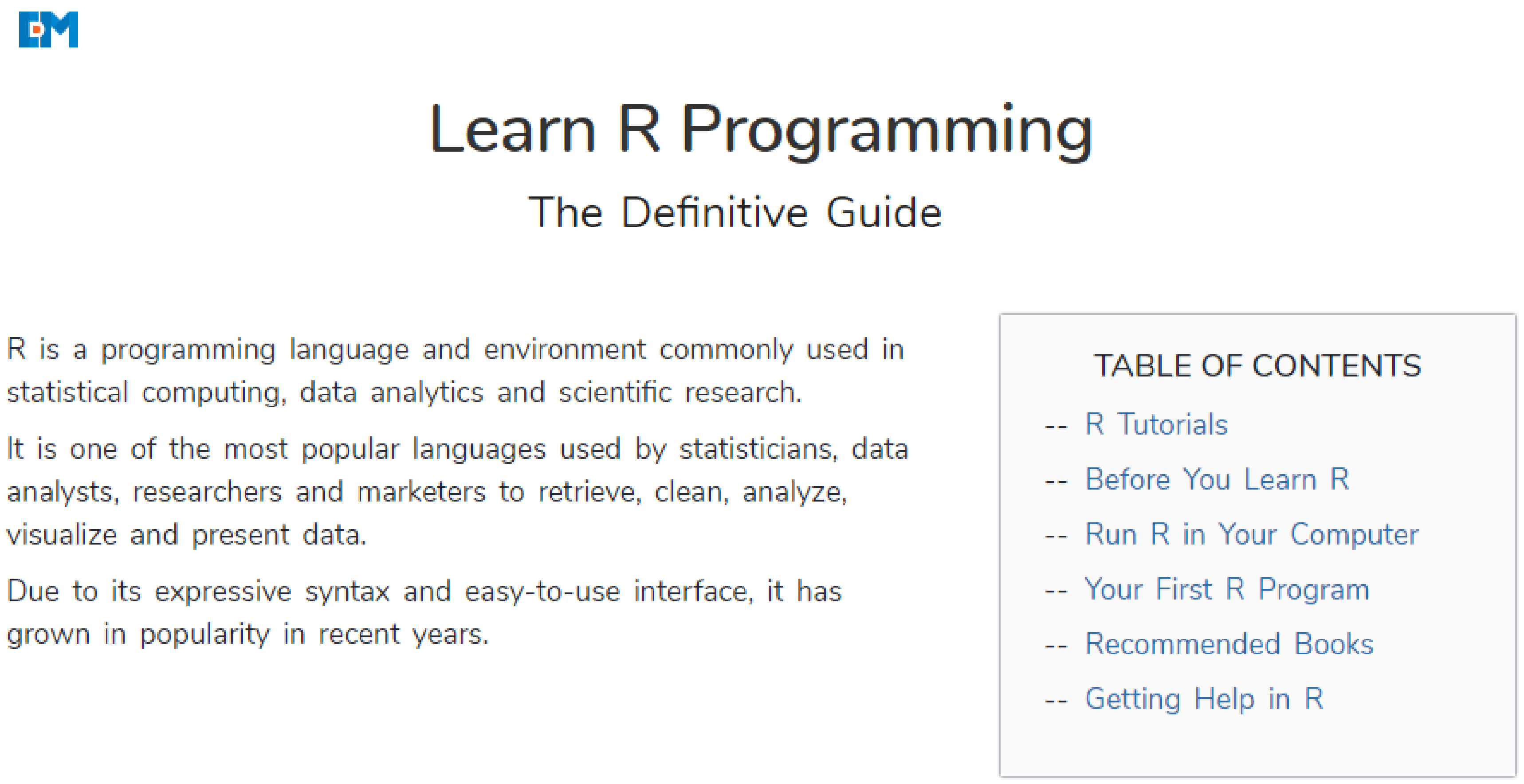
NÃO ESQUEÇA DE INICIAR A GRAVAÇÃO

MAP 2112 – Introdução à Lógica de Programação e Modelagem Computacional

1º Semestre - 2020

Prof. Dr. Luis Carlos de Castro Santos

lsantos@ime.usp.br



Learn R Programming

The Definitive Guide

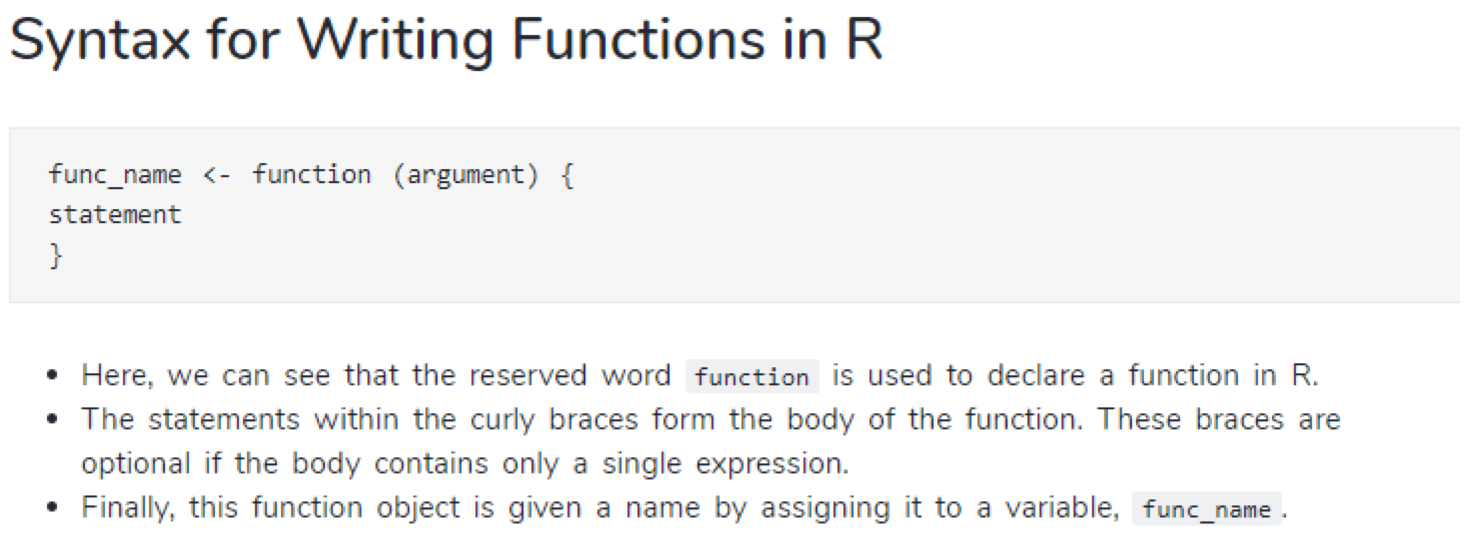
R is a programming language and environment commonly used in statistical computing, data analytics and scientific research.

It is one of the most popular languages used by statisticians, data analysts, researchers and marketers to retrieve, clean, analyze, visualize and present data.

Due to its expressive syntax and easy-to-use interface, it has grown in popularity in recent years.

TABLE OF CONTENTS

https://www.datamentor.io/r-programming/function/



Syntax for Writing Functions in R

```
func_name <- function (argument) {
statement
}
```

- Here, we can see that the reserved word `function` is used to declare a function in R.
- The statements within the curly braces form the body of the function. These braces are optional if the body contains only a single expression.
- Finally, this function object is given a name by assigning it to a variable, `func_name`.

Funções são usadas para definir porções de código que podem ser reutilizadas e chamadas em pontos diferentes do código (ou aproveitadas entre códigos) simplificando a escrita.

Example of a Function

```
pow <- function(x, y) {
# function to print x raised to the power y
result <- x^y
print(paste(x,"raised to the power", y, "is", result))
}
```

Here, we created a function called `pow()`.

It takes two arguments, finds the first argument raised to the power of second argument and prints the result in appropriate format.

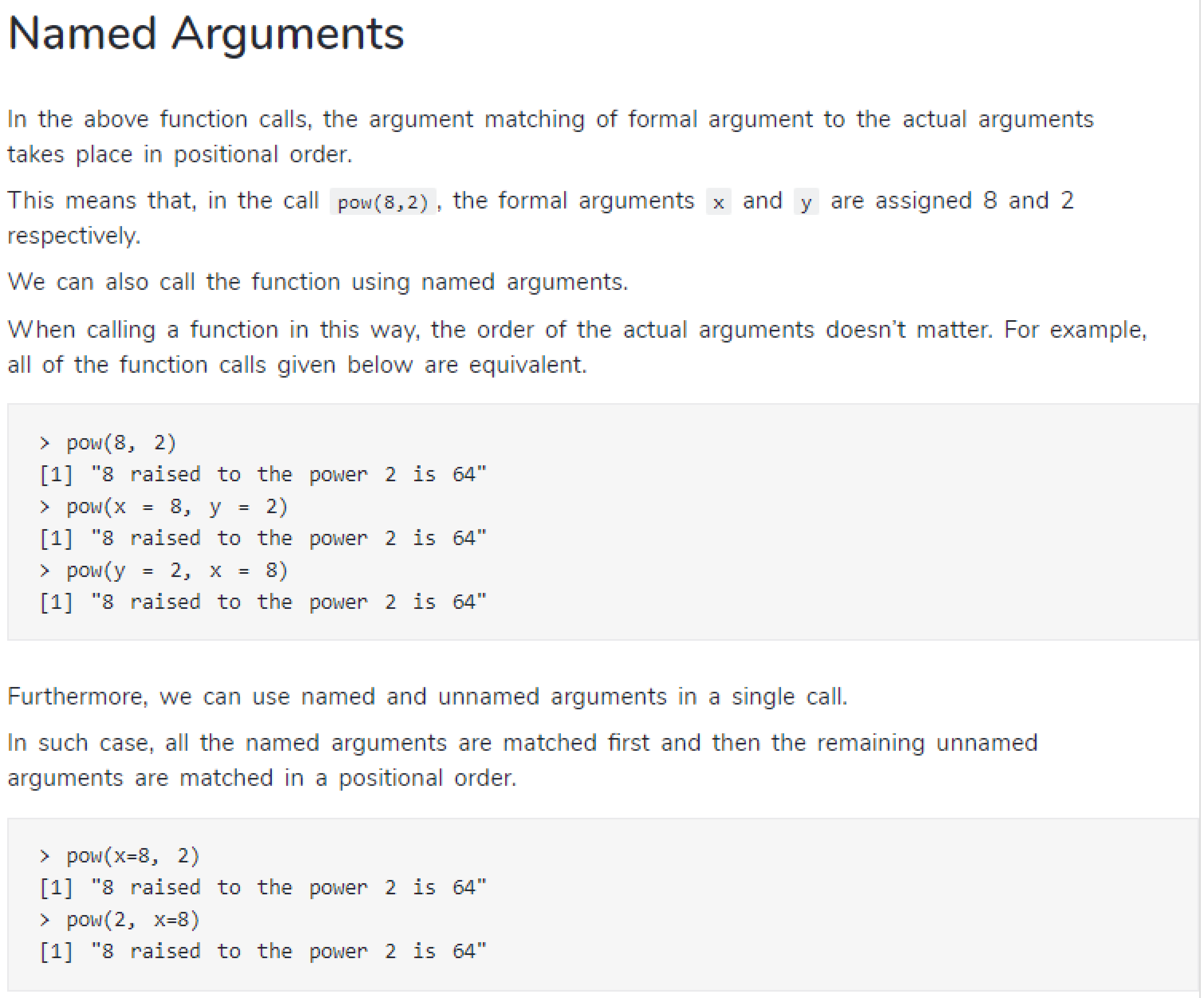
We have used a built-in function `paste()` which is used to concatenate strings.

How to call a function?

We can call the above function as follows.

```
>pow(8, 2)
[1] "8 raised to the power 2 is 64"
> pow(2, 8)
[1] "2 raised to the power 8 is 256"
```

Here, the arguments used in the function declaration (`x` and `y`) are called formal arguments and those used while calling the function are called actual arguments.



Named Arguments

In the above function calls, the argument matching of formal argument to the actual arguments takes place in positional order.

This means that, in the call `pow(8,2)`, the formal arguments `x` and `y` are assigned 8 and 2 respectively.

We can also call the function using named arguments.

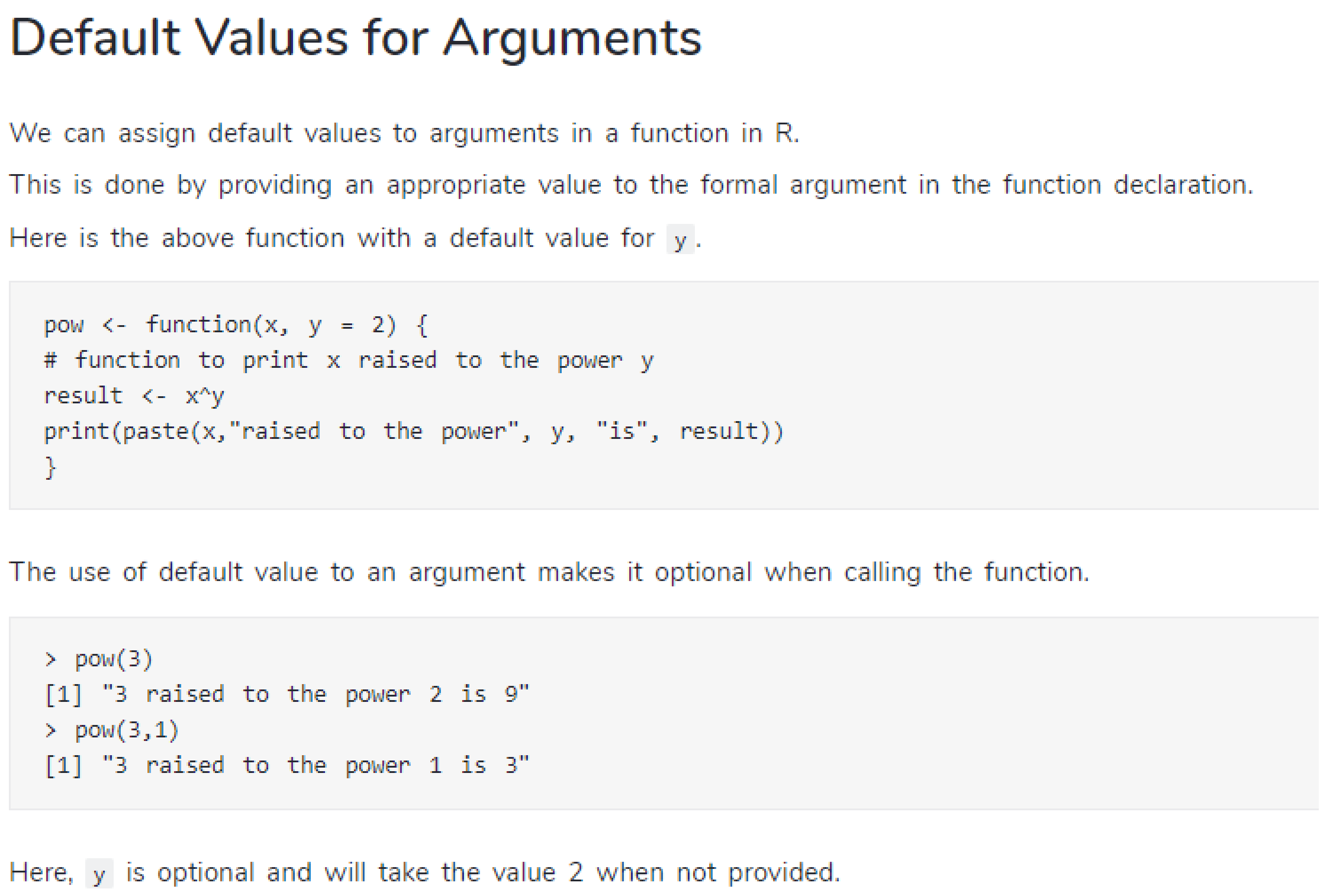
When calling a function in this way, the order of the actual arguments doesn't matter. For example, all of the function calls given below are equivalent.

```
> pow(8, 2)
[1] "8 raised to the power 2 is 64"
> pow(x = 8, y = 2)
[1] "8 raised to the power 2 is 64"
> pow(y = 2, x = 8)
[1] "8 raised to the power 2 is 64"
```

Furthermore, we can use named and unnamed arguments in a single call.

In such case, all the named arguments are matched first and then the remaining unnamed arguments are matched in a positional order.

```
> pow(x=8, 2)
[1] "8 raised to the power 2 is 64"
> pow(2, x=8)
[1] "8 raised to the power 2 is 64"
```



Default Values for 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.

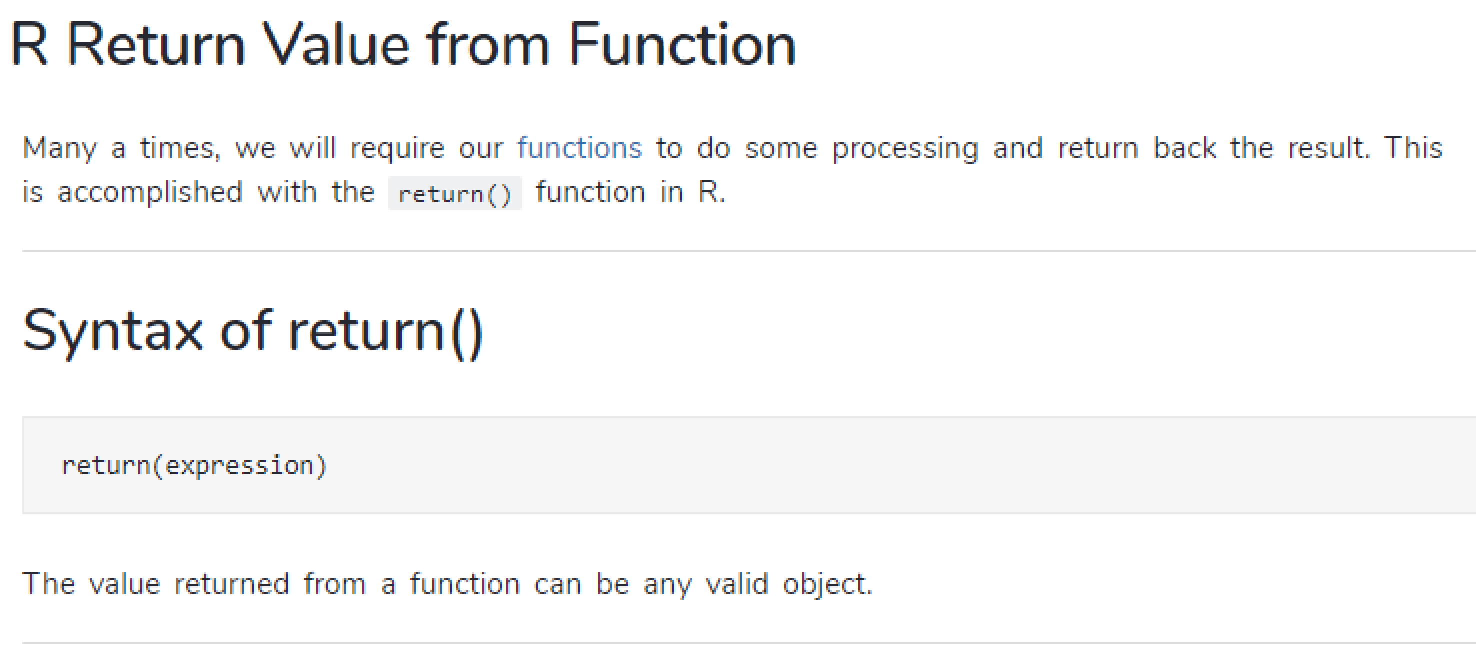
Here is the above function with a default value for `y`.

```
pow <- function(x, y = 2) {
# function to print x raised to the power y
result <- x^y
print(paste(x,"raised to the power", y, "is", result))
}
```

The use of default value to an argument makes it optional when calling the function.

```
> pow(3)
[1] "3 raised to the power 2 is 9"
> pow(3,1)
[1] "3 raised to the power 1 is 3"
```

Here, `y` is optional and will take the value 2 when not provided.



R Return Value from Function

Many a times, we will require our functions to do some processing and return back the result. This is accomplished with the `return()` function in R.

Syntax of return()

```
return(expression)
```

The value returned from a function can be any valid object.

Example: return()

Let us look at an example which will return whether a given number is positive, negative or zero.

```
check <- function(x) {
if (x > 0) {
result <- "Positive"
}
else if (x < 0) {
result <- "Negative"
}
else {
result <- "Zero"
}
return(result)
}
```

Here, are some sample runs.

```
> check(1)
[1] "Positive"
> check(-10)
[1] "Negative"
> check(0)
[1] "Zero"
```

Functions without return()

If there are no explicit returns from a function, the value of the last evaluated expression is returned automatically in R.

For example, the following is equivalent to the above function.

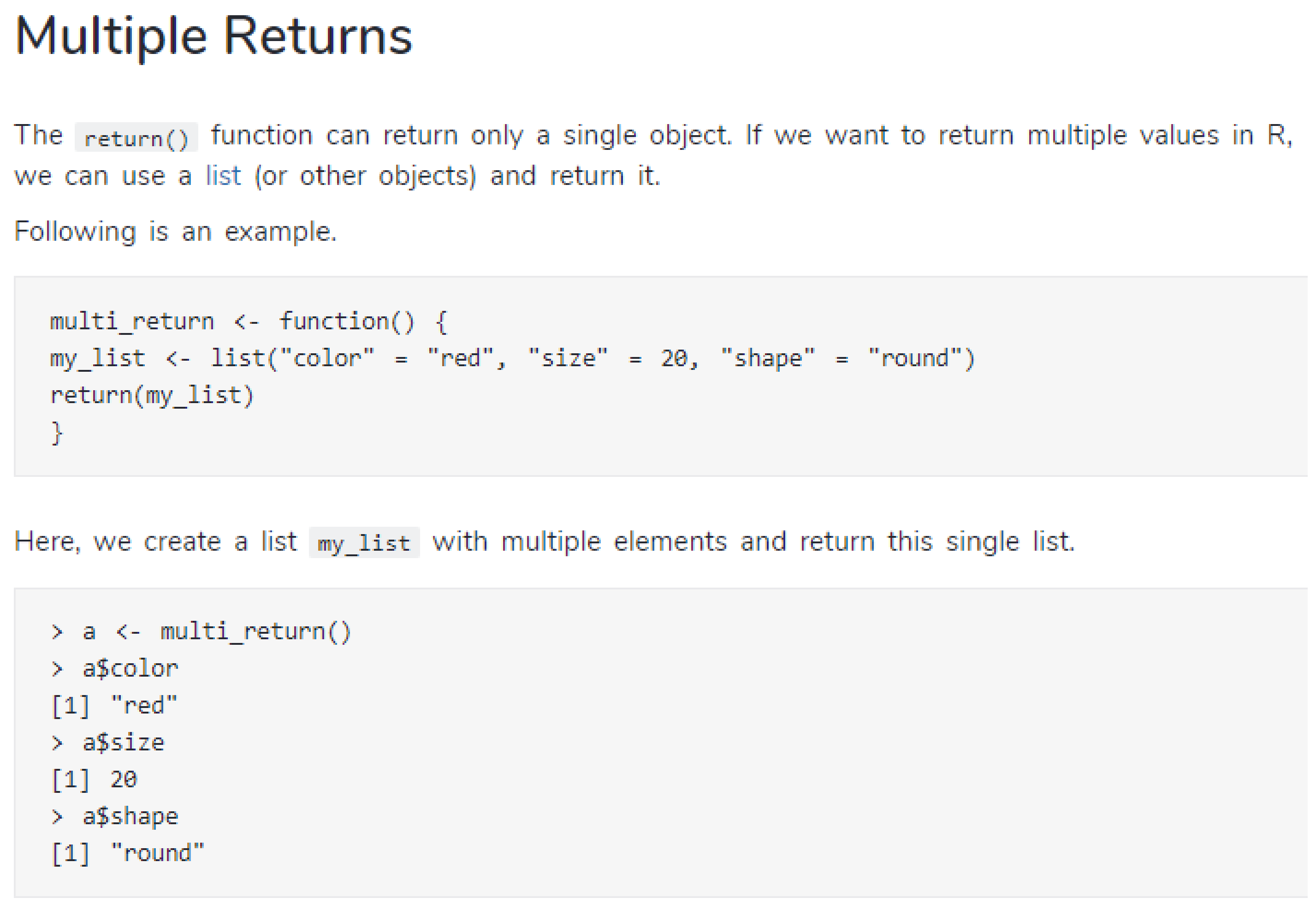
```
check <- function(x) {
if (x > 0) {
result <- "Positive"
}
else if (x < 0) {
result <- "Negative"
}
else {
result <- "Zero"
}
result
}
```

We generally use explicit `return()` functions to return a value immediately from a function.

If it is not the last statement of the function, it will prematurely end the function bringing the control to the place from which it was called.

```
check <- function(x) {
if (x>0) {
return("Positive")
}
else if (x<0) {
return("Negative")
}
else {
return("Zero")
}
}
```

In the above example, if `x > 0`, the function immediately returns `"Positive"` without evaluating rest of the body.



Multiple Returns

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.

Following is an example.

```
multi_return <- function() {
my_list <- list("color" = "red", "size" = 20, "shape" = "round")
return(my_list)
}
```

Here, we create a list `my_list` with multiple elements and return this single list.

```
> a <- multi_return()
> a$color
[1] "red"
> a$size
[1] 20
> a$shape
[1] "round"
```

Functions

Roger D. Peng, Associate Professor of Biostatistics
Johns Hopkins Bloomberg School of Public Health

Functions

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.

```
f <- function(<arguments>) {
        ## Do something interesting
}
```

Functions in R are “first class objects”, which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

Functions have *named arguments* which potentially have *default values*.

- The *formal arguments* are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be *missing* or might have default values

sd = standard deviation = desvio padrão

Argument Matching

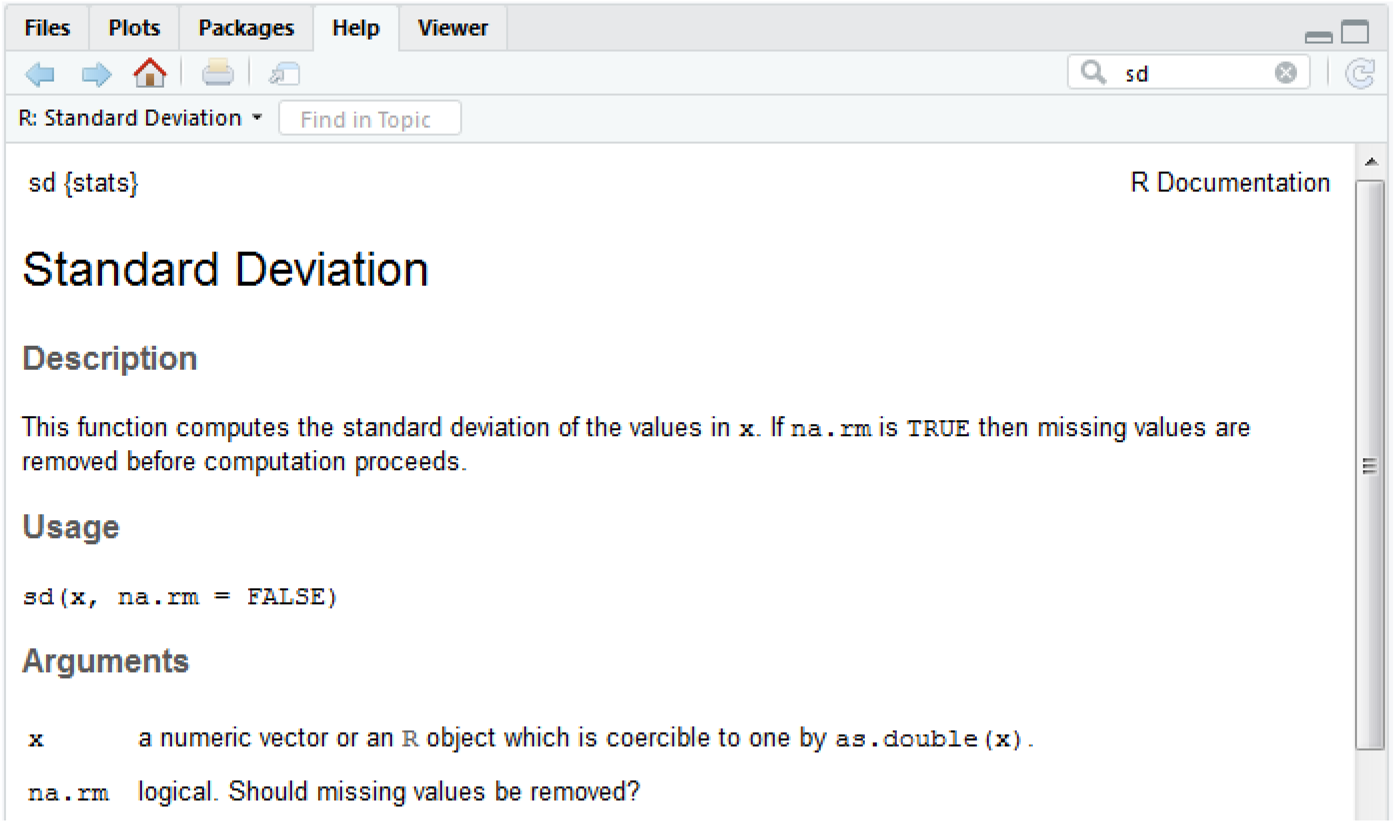
R functions arguments can be matched positionally or by name. So the following calls to `sd` are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

```
> formals(sd)
$x


$na.rm
[1] FALSE
```



sd {stats} R Documentation

Standard Deviation

Description

This function computes the standard deviation of the values in `x`. If `na.rm` is `TRUE` then missing values are removed before computation proceeds.

Usage

```
sd(x, na.rm = FALSE)
```

Arguments

`x`	a numeric vector or an R object which is coercible to one by `as.double(x)`.
`na.rm`	logical. Should missing values be removed?

Argument Matching

You can mix positional matching with matching by name. When an argument is matched by name, it is “taken out” of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action,
          method = "qr", model = TRUE, x = FALSE,
          y = FALSE, qr = TRUE, singular.ok = TRUE,
          contrasts = NULL, offset, ...)
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Argument Matching

- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
- Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Function arguments can also be *partially* matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

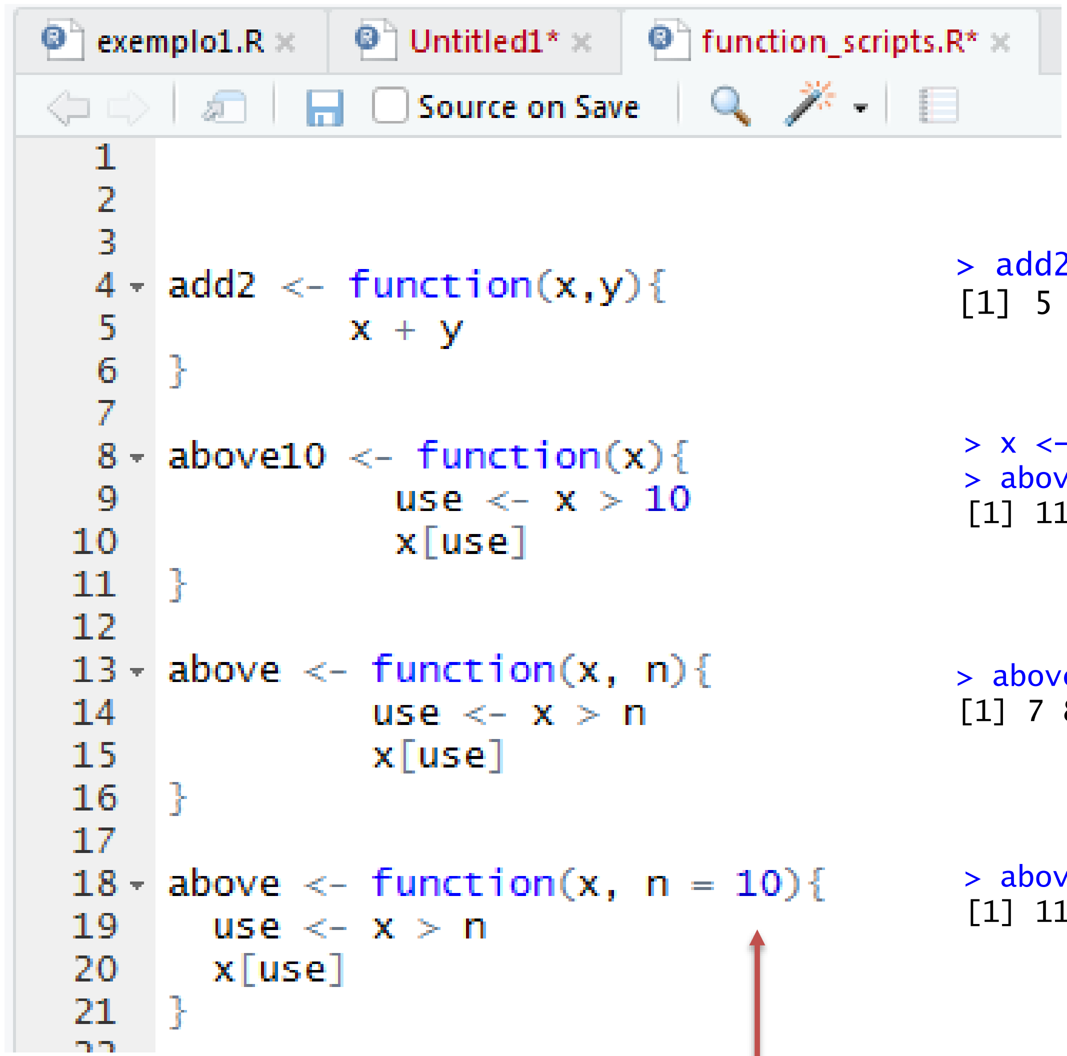
Defining a Function

```
f <- function(a, b = 1, c = 2, d = NULL) {


}
```

In addition to not specifying a default value, you can also set an argument value to `NULL`.

A First Example



```r


add2 <- function(x,y){
          x + y
}

above10 <- function(x){
             use <- x > 10
             x[use]
}

above <- function(x, n){
           use <- x > n
           x[use]
}

above <- function(x, n = 10){
    use <- x > n
    x[use]
}
```

default

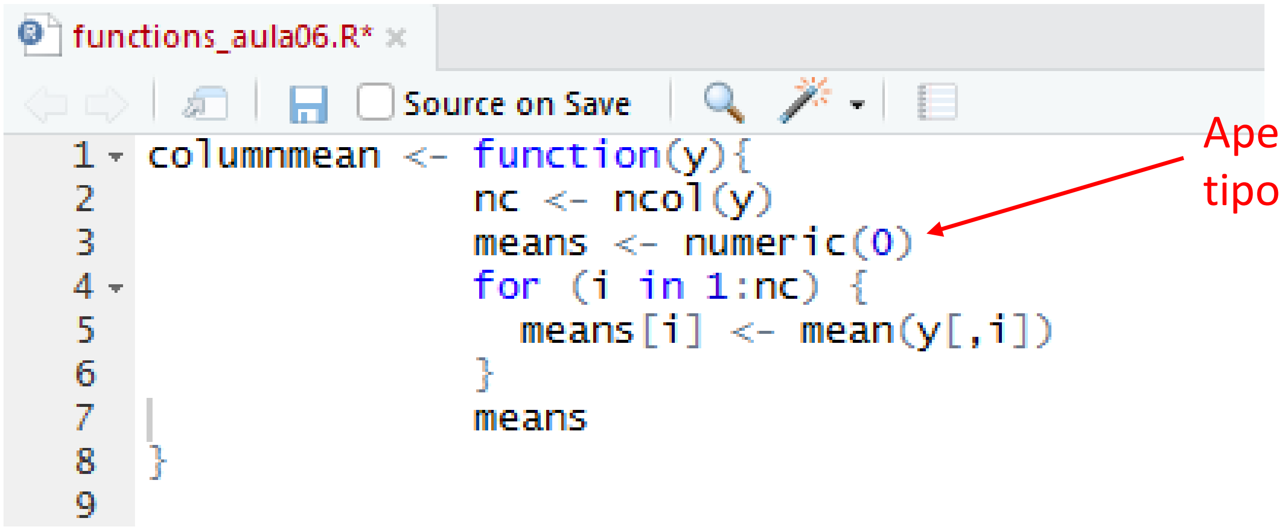
```
> add2(2,3)
[1] 5
```

```
> x <- 1:20
> above10(x)
[1] 11 12 13 14 15 16 17 18 19 20
```

```
> above(x,6)
[1] 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
> above(x)
[1] 11 12 13 14 15 16 17 18 19 20
```

A Second (more interesting) Example

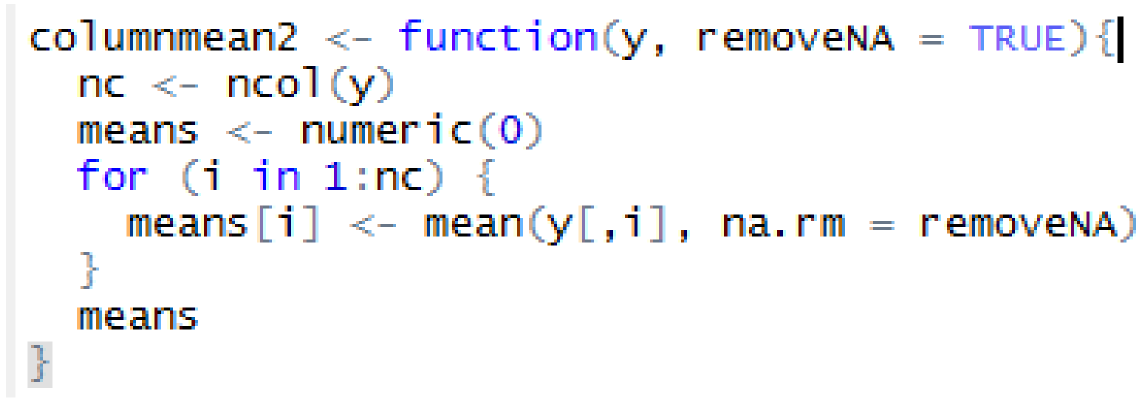


```
columnmean <- function(y){
                nc <- ncol(y)
                means <- numeric(0)
                for (i in 1:nc) {
                  means[i] <- mean(y[,i])
                }
                means
}
```

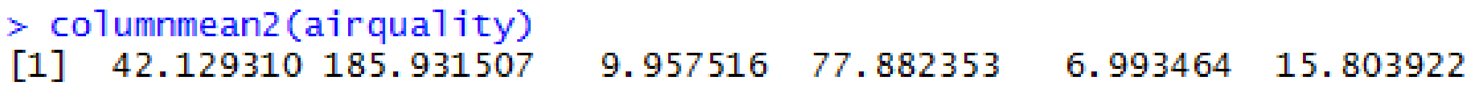
Apenas para definir o tipo do objeto

```
> columnmean(airquality)
[1]        NA        NA  9.957516 77.882353  6.993464 15.803922
```

A Second (more interesting) Example (ii)



```
columnmean2 <- function(y, removeNA = TRUE){
  nc <- ncol(y)
  means <- numeric(0)
  for (i in 1:nc) {
    means[i] <- mean(y[,i], na.rm = removeNA)
  }
  means
}
```



```
> columnmean2(airquality)
[1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

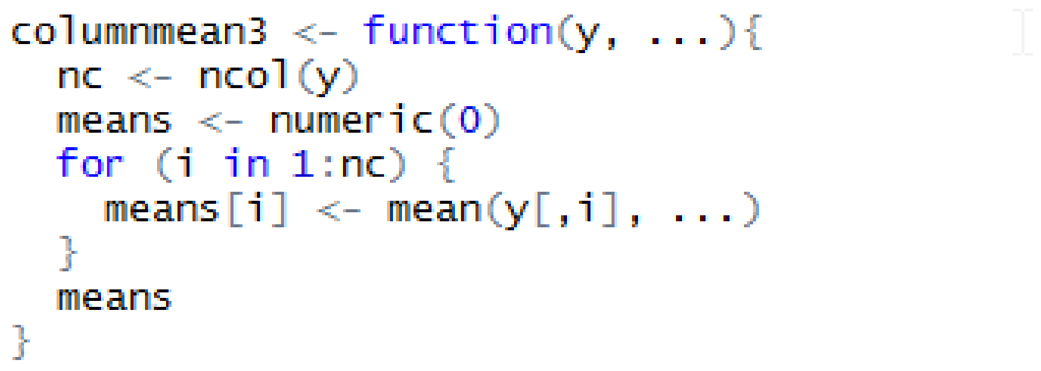
The "..." Argument

The ... argument indicate a variable number of arguments that are usually passed on to other functions.

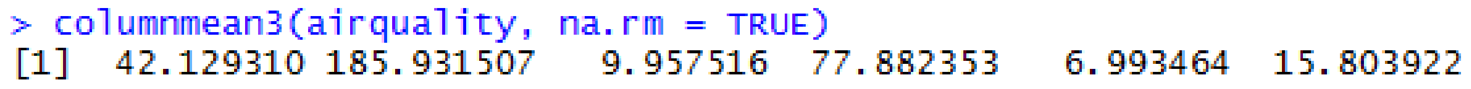
- ... is often used when extending another function and you don't want to copy the entire argument list of the original function

```
myplot <- function(x, y, type = "l", ...) {
        plot(x, y, type = type, ...)
}
```

A Second (more interesting) Example (iii)



```
columnmean3 <- function(y, ...){
  nc <- ncol(y)
  means <- numeric(0)
  for (i in 1:nc) {
    means[i] <- mean(y[,i], ...)
  }
  means
}
```



```
> columnmean3(airquality, na.rm = TRUE)
[1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

The “...” Argument

The ... argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> args(cat)
function (..., file = "", sep = " ", fill = FALSE,
    labels = NULL, append = FALSE)
```

Arguments Coming After the "..." Argument

One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> paste("a", "b", sep = ":")
[1] "a:b"

> paste("a", "b", se = ":")
[1] "a b :"
```

Aplicações

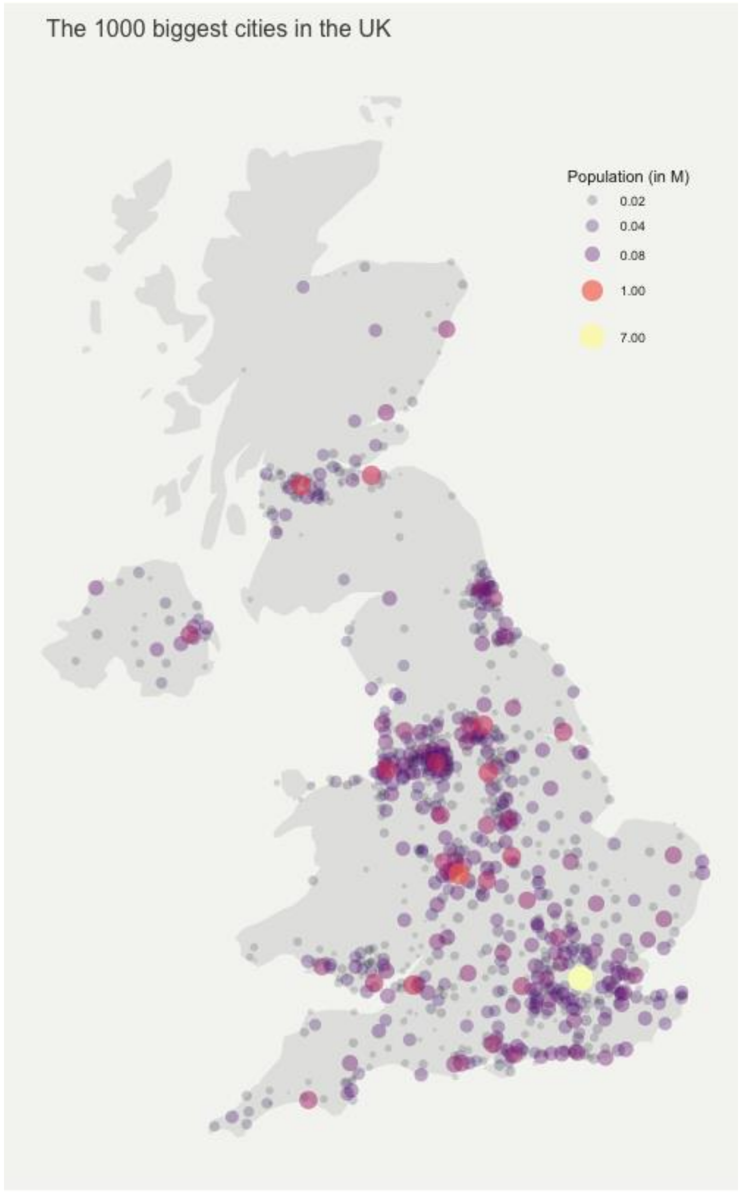
Condicionais
Laços
Funções

Problema 1

Escreva um código em R para encontrar as raízes de uma equação do 2º grau dados os coeficientes da equação.

$$ax^2 + bx + c = 0$$

para qualquer valores de *a*, *b* e *c*.



https://www.r-graph-gallery.com/330-bubble-map-with-ggplot2.html