Functions in R

Quantitative Applications for Data Analylsis

Alexey Fedoseev

January 26, 2023

Recipe

Today we are making a raisin bread! It is simple

```
> cat("Whip the butter\n")
Whip the butter
> cat("Add sugar\n")
Add sugar
> cat("Add flour\n")
Add flour
> cat("Add raisins\n")
Add raisins
> cat("Mix everything well\n")
Mix everything well
> cat("Bake at 150C for 1 hour\n")
Bake at 150C for 1 hour
```

What if you want to bake one more cake? Let's automate it!

Defining a function

We can group all these steps into a procedure of making a raisin bread.

```
> make.raisin.bread <- function() {
+   cat("Whip the butter\n")
+   cat("Add sugar\n")
+   cat("Add flour\n")
+   cat("Add raisins\n")
+   cat("Mix everything well\n")
+   cat("Bake at 150C for 1 hour\n")
+ }
```

This is called a function. You simply combine several steps together and give it a distinctive name.

Note! The body of your function begins and ends with these symbols: `{  }` (knows as curly braces).

Calling a function

Next time you bake a new cake just call the function!

- If you ever do something more than once in your code, make a function to do it.
- By utilizing functions you can easily change any step within your function, for example if you want to use "brown sugar" instead of "sugar" you only change it in a function once.
- I cannot stress enough how important functions are.
- Your code will be much easier to read.
- Your functions could be used in your other programs.
- **Remember**: function names cannot start with a number.

```
> cat("I am making two cakes.\n")
I am making two cakes.
> make.raisin.bread()
Whip the butter
Add sugar
Add flour
Add raisins
Mix everything well
Bake at 150C for 1 hour
> make.raisin.bread()
Whip the butter
Add sugar
Add flour
Add raisins
Mix everything well
Bake at 150C for 1 hour
```

Calling a function

If you can’t remember what the function is, you can just type its name (without parenthesis) and its definition will be displayed in your R terminal.

```
> make.raisin.bread
function() {
  cat("Whip the butter\n")
  cat("Add sugar\n")
  cat("Add flour\n")
  cat("Add raisins\n")
  cat("Mix everything well\n")
  cat("Bake at 150C for 1 hour\n")
}
```

Structuring your code

You will note that this recipe requires a lot of mixing. We can put these steps in a function as well!

```
> mix.ingredients <- function() {
+   cat("Add sugar\n")
+   cat("Add flour\n")
+   cat("Add raisins\n")
+   cat("Mix everything well\n")
+ }
```

Now our baking procedure can use this function

```
> make.raisin.bread <- function() {
+   cat("Whip the butter\n")
+   mix.ingredients()
+   cat("Bake at 150C for 1 hour\n")
+ }
```

Arguments

What if we want to reuse the mixing function with other ingredients? You can specify what exactly you want to mix and in what order.

```
> mix.ingredients <- function(ingredient1, ingredient2, ingredient3) {
+   cat("Add ", ingredient1, "\n", sep="")
+   cat("Add ", ingredient2, "\n", sep="")
+   cat("Add ", ingredient3, "\n", sep="")
+   cat("Mix everything well\n")
+ }
> mix.ingredients("brown sugar", "flour", "raisins")
Add brown sugar
Add flour
Add raisins
Mix everything well
```

In programming the parameters of a function (in our example named `ingredient1`, `ingredient2`, `ingredient3`) are called **arguments of a function**.

Arguments

```
> mix.ingredients("chicken", "lettuce", "croutons")
Add chicken
Add lettuce
Add croutons
Mix everything well
```

R expects that you will provide exactly 3 arguments for the function `mix.ingredients`, unless you specify default values for certain arguments, which we will discuss later today.

```
> mix.ingredients("milk")
Add milk
Error in cat("Add ", ingredient2, "\n", sep = "") :
  argument "ingredient2" is missing, with no default
```

Arguments

Perhaps you need to mix 2 or 5 ingredients? Vectors can help us in this situation. A vector is a data structure that contains multiple elements of the same type.

```
> ingredients <- c("sugar","flour","raisins","vanilla","a pinch of salt")
> for (ing.index in seq_along(ingredients))
+   cat("Argument no.", ing.index, "is", ingredients[ing.index], "\n")
Argument no. 1 is sugar
Argument no. 2 is flour
Argument no. 3 is raisins
Argument no. 4 is vanilla
Argument no. 5 is a pinch of salt
```

Note: elements of a vector have to be of the **same type**. If your arguments have different types you need to use lists instead of vectors.

Arguments

If the number of parameters that your function needs can vary, define one argument as a vector containing as many ingredients as we need and work with each element of this vector as an argument.

```
> mix.ingredients <- function(ingredients) {
+   for (ingredient in ingredients)
+     cat("Add", ingredient, "\n")
+   cat("Mix everything well\n")
+ }
> ingredients <- c("sugar","flour","raisins","vanilla","a pinch of salt")
> mix.ingredients(ingredients)
Add sugar
Add flour
Add raisins
Add vanilla
Add a pinch of salt
Mix everything well
```

Local variables

Let us use our advanced mixing procedure to bake another cake.

```
> make.raisin.bread <- function() {
+   cat("Whip the butter\n")
+
+   myingredients <- c("sugar","flour","raisins","vanilla","a pinch of salt")
+   mix.ingredients(myingredients)
+
+   cat("Bake at 150C for 1 hour\n")
+ }
```

`myingredients` referenced above is a variable and is only seen inside of the function. It is not visible outside of the function, it is a ‘local variable’. Let us follow the example below.

Local variables

If you try to use a variable, that is local to a function somewhere outside of this function, R will throw an error.

```
> make.raisin.bread()
Whip the butter
Add sugar
Add flour
Add raisins
Add vanilla
Add a pinch of salt
Mix everything well
Bake at 150C for 1 hour
> cat(myingredients)
Error in cat(myingredients) : object 'myingredients' not found
```

Global variables

Variables which are declared outside of functions are called ‘global variables’. You can have a variable outside of your function with the same name. However, it is advisable to come up with a different name for your variable, simply because you will not confuse them accidently.

```
> myingredients <- c("potatoes", "carrots", "peas")
> make.raisin.bread()
Whip the butter
Add sugar
Add flour
Add raisins
Add vanilla
Add a pinch of salt
Mix everything well
Bake at 150C for 1 hour
> cat(myingredients, "\n")
potatoes carrots peas
```

Global variables

You cannot rely on the variables outside of the function. Why?

```
> ingredients <- c("sugar", "flour", "raisins")
> make.raisin.bread <- function() {
+   cat("Whip the butter\n")
+
+   mix.ingredients(ingredients)
+
+   cat("Bake at 150C for 1 hour\n")
+ }
```

This code will work. However, what happens if you redefine your ingredients for another recipe?

Global variables vs Local variables

```
> ingredients <- c("onions", "garlic", "tomatoes", "herbs")
> make.raisin.bread()
Whip the butter
Add onions
Add garlic
Add tomatoes
Add herbs
Mix everything well
Bake at 150C for 1 hour
```

Oh no! It is not a raisin bread anymore! This could be a source of an error that is **really hard to trace**. Always specify any external to the function variables as arguments!

Use arguments instead of global variables

At some point you will modify your global variable. It will affect functions that rely on this global variable. A better alternative is to use arguments of the function to specify anything you need outside of that particular function to make your codes reliable.

```
> make.raisin.bread <- function(ingredients) {
+   cat("Whip the butter\n")
+   mix.ingredients(ingredients)
+   cat("Bake at 150C for 1 hour\n")
+ }
> ingredients <- c("sugar", "flour", "raisins")
> make.raisin.bread(ingredients)
Whip the butter
Add sugar
Add flour
Add raisins
Mix everything well
Bake at 150C for 1 hour
```

return statement

The result of the your function can be re-used later. For this purpose we need to tell what symbolizes the result of the function.

```
> make.raisin.bread <- function(ingredients) {
+   cat("Whip the butter\n")
+   mix.ingredients(ingredients)
+   cat("Bake at 150C for 1 hour\n")
+   ingredients.string <- paste(ingredients, collapse=", ")
+   result <- paste("Raisin bread made out of", ingredients.string)
+   return(result)
+ }
```

The function `paste` concatenates ingredients in one string. In order to add commas between the words we use the `collapse=", "` argument.

```
> paste(c("strawberry", "banana"), collapse=", ")
[1] "strawberry, banana"
```

return statement

```
> ingredients <- c("sugar", "flour", "raisins")
> my.bread <- make.raisin.bread(ingredients)
Whip the butter
Add sugar
Add flour
Add raisins
Mix everything well
Bake at 150C for 1 hour
> cat("What did you make today?\n")
What did you make today?
> cat(my.bread, "\n")
Raisin bread made out of sugar, flour, raisins
```

Default arguments

Now we want to make a bolognese sauce. We can serve it with different types of pasta, however it goes best with spaghetti. If we know the best value for our argument we can specify a default value.

```
> make.pasta.bolognese <- function(sauce.ingredients, pasta = "spaghetti") {
+   ingredients.string <- paste(sauce.ingredients, collapse=", ")
+   cat("Cook", ingredients.string, "together\n")
+   cat("Serve on top of", pasta, "\n")
+   return(paste("Bolognese sauce with", pasta))
+ }
> make.pasta.bolognese(c("onions", "ground beef", "tomatoes"))
Cook onions, ground beef, tomatoes together
Serve on top of spaghetti
[1] "Bolognese sauce with spaghetti"
```

Notice we did not specify the pasta argument, so by default it will use “spaghetti”.

Default arguments

Do you only have fettuccine? It will work as well

```
> make.pasta.bolognese(c("onions", "ground beef", "tomatoes"), "fettuccine")
Cook onions, ground beef, tomatoes together
Serve on top of fettuccine
[1] "Bolognese sauce with fettuccine"
```

Additionaly, you can specify the name and a value of an argument which makes your code well documented.

```
> make.pasta.bolognese(
+   sauce.ingredients = c("onions", "ground beef", "tomatoes", "herbs"),
+   pasta = "spaghettini")
Cook onions, ground beef, tomatoes, herbs together
Serve on top of spaghettini
[1] "Bolognese sauce with spaghettini"
```

Use scripts to save you work

When you exit R prompt, you have an option to save your workspace. It means that all your variables and functions will be available next time you open R prompt. However if you do not save your workspace, all your work will be lost.

If you continue using your workspace to save your work you expect certain functions and variables to be available all the time. It is easy to forget that a month ago you defined an important variable that you keep using. So you submit your code without defining this variable assuming others have it as well. But we don't! Consequently your code does not work on other computers and R throws an error `object not found`.

Writing your code in scripts preserves your work and allows to reuse your code on other computers.

In the next lecture we will discuss writing your scripts in depth.