Writing your own Functions

So far, we have used several functions in R. Summarized, a function takes some input variables, does something with them, and then returns one output variable. Inputs for functions are called arguments, while the output is called the return value. What types of arguments and return values a function has and what happens inside the function is described in the definition of the function. When using R you will run into situations where you want to do the same thing many times over in your program. In these situations, to avoid doing double work, we can define our own functions. As a bonus, the code looks nicer and becomes easier to read.

Defining functions To make a new function in R we use the function named function(). The simplest function we can make takes no arguments and always returns the same value. It looks something like this:

```
mysimplefunction <- function(){
return(0)
}
```

As you can see we use the assign ("<-") operator to store our new function in a variable named mysimplefunction. Once we have stored our function, we can call it. Same as with predefined function, a function call to our function consists of the name of the function, followed by the arguments enclosed in parentheses. Try it:

```
mysimplefunction()
```

```
[1]0
```

Obviously, such an empty function like this is not useful. To get more use out of our functions we can let our function allow arguments, and then do something with these inputs. For instance, we can use this function to multiply two inputs with one another:

```
mymultiply <- function(argument1, argument2){
product <- argument1 * argument2
return (product)
}
```

Note that in the second line we simply multiply the arguments the same way we would when we use the console directly. In effect, a function simply encapsulates a small part of your program. The same is true for the function arguments as well: we could have used any name and type that a variable is allowed to have. Now let's use our new program to multiply 253 and 255:

```
mymultiply(253, 255)
```

```
[1] 64515
```

Scope One pitfall in using functions is the scope. Scoping problems arise from the fact that everything in a function is encapsulated, meaning it is not defined outside the function. Take for example this code:

```
sq <- 2^2
square <- function(arg1){
sq <- arg1^2
return (sq)
}
```

We have defined a variable sq, equal to 4. In our definition for function square, we also have a variable called sq. Because the inside of a function is separate from the rest of the code, the variable sq at function scope is a new variable, and therefore our original variable sq at global scope does not exist change if we use the function square. Let's try it:

```
sq
```

```
[1] 4
```

```
square(7)
```

```
[1] 49
```

```
sq
```

```
[1] 4
```

We assigned 49 to sq within the function scope but sq at global scope is unchanged! In fact, the sq used by the function is automatically deleted after the function has finished running. Be careful with where you define your variables and where you use the! Remember that the inside of a function and the outside of a function are only connected through arguments and return values.

Editing functions

You might want to change a function that you have written in R. Knowing that functions are stored in a variable, we can simply assign a new function to that variable:

```
add <- function(a,b){
return (a+b)
}
```

```
add (2,3)
```

```
[1] 5
```

```
add <- function(a,b){
return (a+b+5)
}
```

```
add (2,3)
```

```
[1] 10
```

Often the definition of a function is done in the beginning of a script and them later on in your script you call a function. **Remember that any changes you make in your function definition will be actual only if you run the code that defines your function one more time.**

Examples Let us now give a few examples of simple functions to increase your understanding. One problem you often face is that some programs returns numeric values with commas, e.g. 33,400,570. These numbers will cause problems in your R code. So let us write a function to "decommafy" such numbers:

```
decommafy <- function(x){
y<- gsub(",","",x)
return(as.numeric(y))
}
```

This function uses an R function called gsub(). Gsub searches in a string patterns and replaces them with a new pattern. In this example our string is given in variable x, the pattern it is searching for is a comma: ',' and it replaces it with nothing, i.e., ''. We want this function to return a numeric value, and therefore we return y as numeric. This is how you can call this function:

```
gene_start <- "34,567,432"
decommafy(gene_start)
```

```
[1] 34567432
```

Another example is from the Bioinformatics reader, Chapter 6, question 5. The question asks us to write a a function that calculates alignment score between two sequences using the identity matrix, i.e., all amtaches are scored as 1 and mismatches are scored as 0. We will assume that both sequences are given as vectors, and the length of the sequences to be compared should be the same. So our inout can be something like:

```
seq1<- c("V", "W","E", "D","N","W","D", "D", "D")
seq2<- c("V", "S","E", "D","N","R","D", "D", "D")
```

Remember, we can easily compare two vectors of the same length in R by typing:

```
seq1==seq2
```

```
[1]  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

We can make use of the fact that in R TRUE is also defined as 1 and FALSE is defined as 0. Therefore the following command returns the number of positions that are equal between the two sequences:

```
sum(seq1==seq2)
```

```
[1]  7
```

Using an identity matrix, where every matching position scores as 1 and mismatches scores as 0, the number of the equal positions between the two sequences simply becomes the score of the alignment. Let us use this to write our function:

```
id_score<-function(seqX, seqY) {
return(sum(seqX==seqY))
}
id_score(seq1, seq2)
```

```
[1]  7
```

The identity matrix can also give punishment for mismatches. If every mismatch should cost -0.2 penalty, then we would update our function as the following:

```
id_score<-function(seqX, seqY) {
match = sum(seqX==seqY)
mismatch = length(seqX) - match
return(match-mismatch*0.2)
}
id_score(seq1, seq2)
```

```
[1]  6.6
```

We can make our function even more flexible by giving the mismatch penalty as an input:

```
id_score<-function(seqX, seqY, penalty=0) {
match = sum(seqX==seqY)
mismatch = length(seqX) - match
return(match-mismatch*penalty)
}
id_score(seq1, seq2, 0.4)
```

```
[1]  6.2
```

If the user does not provide any penalty value, by default a value of 0 will be taken:

```
id_score(seq1, seq2)
```

```
[1]  7
```

By giving a default value to the parameter *penalty* by setting $penalty = 0$, you do not force the user to always provide a value for the penalty of the mismatches. This function can calculate any two sequences of the same length, e.g.:

```
s1<- c("W", "D","N","W","D")
s2<- c("S", "D","N","R","D")
id_score(s1, s2, penalty=-1)
```