An introduction to R: **Algorithmics in R (continued)**

Noémie Becker, Benedikt Holtmann & Dirk Metzler [1]

nbecker@bio.lmu.de - holtmann@bio.lmu.de

Winter semester 2016-17

[1] Special thanks to: Prof. Dr. Martin Hutzenthaler and Dr. Sonja Grath for course development

Contents

Basics

Syntax:

```
myfun <- function (arg1, arg2, . . .) { commands }
```

Basics

Syntax:

```
myfun <- function (arg1, arg2, . . .) { commands }
```

Example:
We want to define a function that takes a DNA sequence as input and gives as ouptut the GC content (proportion of G and C in the sequence).

Basics

Syntax:

```
myfun <- function (arg1, arg2, . . .) { commands }
```

Example:
We want to define a function that takes a DNA sequence as input and gives as ouptut the GC content (proportion of G and C in the sequence).

```
?gc # oops there is already a function named gc
```

Basics

Syntax:

```
myfun <- function (arg1, arg2, . . .) { commands }
```

Example:
We want to define a function that takes a DNA sequence as input and gives as ouptut the GC content (proportion of G and C in the sequence).

```
?gc # oops there is already a function named gc
?GC # ok this time
```

Basics

Syntax:

```
myfun <- function (arg1, arg2, . . .) { commands }
```

Example:
We want to define a function that takes a DNA sequence as input and gives as ouptut the GC content (proportion of G and C in the sequence).

```
?gc # oops there is already a function named gc
?GC # ok this time
```

We will use the function `gregexp` for regular expressions.

```
?gregexpr
```

```
GC <- function(dna) {
    gc.cont <- length(gregexpr("C|G",dna)[[1]])/nchar(dna)
    return(gc.cont)
}
```

Basics

Syntax:

```
myfun <- function (arg1, arg2, . . .) { commands }
```

Example:
We want to define a function that takes a DNA sequence as input and gives as ouptut the GC content (proportion of G and C in the sequence).

```
?gc # oops there is already a function named gc
?GC # ok this time
```

We will use the function `gregexp` for regular expressions.

```
?gregexpr

GC <- function(dna) {
    gc.cont <- length(gregexpr("C|G",dna)[[1]])/nchar(dna)
    return(gc.cont)
}
GC("AATTCGCTTA")
[1] 0.3
```

Are we sure our function is correct?

Are we sure our function is correct?

```
GC("AATTAAATTA")
```

Are we sure our function is correct?

```
GC("AATTAAATTA")
[1] 0.1
```

What happened?

Are we sure our function is correct?

```
GC("AATTAAATTA")
[1] 0.1
```

What happened?

A function should always be tested with several inputs.

Better version of the function

```
GC <- function(dna) {
    gc1 <- gregexpr("C|G",dna)[[1]]
    if (length(gc1)>1){
        gc.cont <- length(gc1)/nchar(dna)
    } else {
        if (gc1>0){
            gc.cont <- 1/nchar(dna)
        } else {
            gc.cont <- 0
        }
    }
    return(gc.cont)
}
```

Deal with wrong arguments

So far we assumed that the input was a chain of characters with only A, T, C and G.
What happens if we try another type of argument?

```
GC("23")
[1] 0
```

Deal with wrong arguments

So far we assumed that the input was a chain of characters with only A, T, C and G.
What happens if we try another type of argument?

```
GC("23")
[1] 0
GC(TRUE)
[1] 0
```

Deal with wrong arguments

So far we assumed that the input was a chain of characters with only A, T, C and G.
What happens if we try another type of argument?

```
GC("23")
[1] 0
GC(TRUE)
[1] 0
GC("notDNA")
[1] 0
```

Deal with wrong arguments

So far we assumed that the input was a chain of characters with only A, T, C and G.
What happens if we try another type of argument?

```
GC("23")
[1] 0
GC(TRUE)
[1] 0
GC("notDNA")
[1] 0
GC("Cool")
[1] 0.25
```

Deal with wrong arguments

So far we assumed that the input was a chain of characters with only A, T, C and G.
What happens if we try another type of argument?

```
GC("23")
[1] 0
GC(TRUE)
[1] 0
GC("notDNA")
[1] 0
GC("Cool")
[1] 0.25
```

How can we deal with this?
What do we want our function to output in these cases?

Find a solution collectively (answer below).

Error and warning

There are two types of error messages in R:

- Error message stops execution and returns no value.
- Warning message continues execution.

Error and warning

There are two types of error messages in R:

- Error message stops execution and returns no value.
- Warning message continues execution.

```
x <- sum("hello")
Error in sum("hello") :  invalid 'type' (character) of argument
```

Error and warning

There are two types of error messages in R:

- Error message stops execution and returns no value.
- Warning message continues execution.

```
x <- sum("hello")
Error in sum("hello") :  invalid 'type' (character) of argument
x <- mean("hello")
Warning message:
In mean.default("hello") :  argument is not numeric or logical:
returning NA
```

Error and warning

There are two types of error messages in R:

- Error message stops execution and returns no value.
- Warning message continues execution.

```
x <- sum("hello")
Error in sum("hello") :  invalid 'type' (character) of argument
x <- mean("hello")
Warning message:
In mean.default("hello") :  argument is not numeric or logical:
returning NA
```

We can define such messages with the functions `stop()` and `warning()`.

In our example:

- Error when argument not character
- Warning if character argument not DNA.

Deal with non character arguments

```
GC <- function(dna) {
    if (!is.character(dna)){
        stop("The argument must be of type character.")
    }
    gc1 <- gregexpr("C|G",dna)[[1]]
    if (length(gc1)>1){
        gc.cont <- length(gc1)/nchar(dna)
    } else {
        if (gc1>0){
            gc.cont <- 1/nchar(dna)
        } else {
            gc.cont <- 0
        }
    }
    return(gc.cont)
}
```

Deal with non DNA character

We define as non DNA any character different from A, C, T, G.
If there is another character we compute the value but issue a warning.

Deal with non DNA character

We define as non DNA any character different from A, C, T, G.
If there is another character we compute the value but issue a warning.

We can use the function `grep` as follows:

```
grep("[^GCAT]", dna)
integer(0)
grep("[^GCAT]", "fATCG")
[1] 1
```

Deal with non DNA character

```
GC <- function(dna) {
    if (!is.character(dna)){
        stop("The argument must be of type character.")
    }
    if (length(grep("[^GCAT]", dna))>0){
        warning("The input contains characters other than A, C, T, G - value
should not be trusted!")
    }
    gc1 <- gregexpr("C|G",dna)[[1]]
    if (length(gc1)>1){
        gc.cont <- length(gc1)/nchar(dna)
    } else {
        if (gc1>0){
            gc.cont <- 1/nchar(dna)
        } else {
            gc.cont <- 0
        }
    }
    return(gc.cont)
}
```

Giving several arguments to a function

Most R fucntions have several arguments. You can see them listed in the help page.

Giving several arguments to a function

Most R fucntions have several arguments. You can see them listed in the help page.

A frequent argument in R functions is `na.rm` that removes `NA` values from vectors if it is set to `TRUE`.

```
mean(c(1,2,NA))
[1] NA
mean(c(1,2,NA), na.rm=TRUE)
[1] 1.5
```

We could give our function a second argument to output the AT content instead of GC.

Giving several arguments to a function

```
GC <- function(dna,AT ) {
    gc1 <- gregexpr("C|G",dna)[[1]]
    if (length(gc1)>1){
        gc.cont <- length(gc1)/nchar(dna)
    } else {
        if (gc1>0){
            gc.cont <- 1/nchar(dna)
        } else {
            gc.cont <- 0
        }
    }
    if (AT==TRUE){
        return(1-gc.cont)
    } else {
        return(gc.cont)
    }
}
```

Giving several arguments to a function

```
GC <- function(dna,AT ) {
    gc1 <- gregexpr("C|G",dna)[[1]]
    if (length(gc1)>1){
        gc.cont <- length(gc1)/nchar(dna)
    } else {
        if (gc1>0){
            gc.cont <- 1/nchar(dna)
        } else {
            gc.cont <- 0
        }
    }
    if (AT==TRUE){
        return(1-gc.cont)
    } else {
        return(gc.cont)
    }
}
```

Test:

```
GC(dna,AT=TRUE) [1] 0.7
```

Giving a default value to an argument

In the current version of the function, there will be an error if you forget to specify the value of `AT`.
Test:

```
GC(dna) Error in GC(dna) :  argument "AT" is missing,
with no default
```

Giving a default value to an argument

In the current version of the function, there will be an error if you forget to specify the value of `AT`.
Test:

```
GC(dna) Error in GC(dna) :  argument "AT" is missing,
with no default
```

We should give the value `FALSE` per default to `AT` and it will be changed only if the user specifies `AT = TRUE`.

Giving a default value to an argument

In the current version of the function, there will be an error if you forget to specify the value of `AT`.
Test:

```
GC(dna) Error in GC(dna) :  argument "AT" is missing,
with no default
```

We should give the value `FALSE` per default to `AT` and it will be changed only if the user specifies `AT = TRUE`.

`GC <- function(dna,AT = FALSE )` etc

Giving a default value to an argument

In the current version of the function, there will be an error if you forget to specify the value of `AT`.
Test:

```
GC(dna) Error in GC(dna) :  argument "AT" is missing,
with no default
```

We should give the value `FALSE` per default to `AT` and it will be changed only if the user specifies `AT = TRUE`.

`GC <- function(dna,AT = FALSE )` etc

Test:

```
GC(dna) [1] 0.3
```

Giving a default value to an argument

In the current version of the function, there will be an error if you forget to specify the value of `AT`.
Test:

```
GC(dna) Error in GC(dna) :  argument "AT" is missing,
with no default
```

We should give the value `FALSE` per default to `AT` and it will be changed only if the user specifies `AT = TRUE`.

`GC <- function(dna,AT = FALSE )` etc

Test:

```
GC(dna) [1] 0.3
GC(dna,AT=TRUE) [1] 0.7
```

Returning several values

To do so use a vector or a list.

Returning several values

To do so use a vector or a list.

```
ci.norm <- function(x,conf=0.95)
{
    q <- qnorm(1-(1-conf)/2)
    return(
list(lower=mean(x)-q*se(x),upper=mean(x)+q*se(x)))
}
```

Returning several values

To do so use a vector or a list.

```
ci.norm <- function(x,conf=0.95)
{
    q <- qnorm(1-(1-conf)/2)
    return(
list(lower=mean(x)-q*se(x),upper=mean(x)+q*se(x)))
}

ci.norm(rnorm(100))
$lower [1] -0.1499551
$upper [1] 0.2754680

ci.norm(rnorm(100,conf=0.99))
$lower [1] -0.1673693
$upper [1] 0.2443276
```

Contents

sapply() and tapply()

You use apply() and its derivatives to apply the same function to each element of an object.

```
v <- 1:4
sapply(v,factorial)
# returns a vector, lapply() would return a list
[1] 1 2 6 24
```

sapply() and tapply()

You use apply() and its derivatives to apply the same function to each element of an object.

```
v <- 1:4
sapply(v,factorial)
# returns a vector, lapply() would return a list
[1] 1 2 6 24
```

tapply() is used for data frames.

sapply() and tapply()

You use apply() and its derivatives to apply the same function to each element of an object.

```
v <- 1:4
sapply(v,factorial)
# returns a vector, lapply() would return a list
[1] 1 2 6 24
```

tapply() is used for data frames.
Example: data frame containing lifespan for people from 3 classes of weight. You want the mean lifespan for each class.

```
tapply(lifespan,weightcls,mean)
1 2 3
69 61 53
```

Contents

How to avoid slow R code

- R has to interpret your commands each time you run a script and it takes time to determine the type of your variables.
- So avoid using loops and calling functions again and again if possible
- When you use loops, avoid increasing the size of an object (vector ...) at each iteration but rather define it with full size before.
- Think in whole objects such as vectors or lists and apply operations to the whole object instead of looping through all elements.