An introduction to R: Algorithmics in R (continued)

Noémie Becker, Benedikt Holtmann & Dirk Metzler

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

Winter semester 2016-17

1Special thanks to: Prof. Dr. Martin Hutzenthaler and Dr. Sonja Grath for course development
1. Writing your own functions
2. sapply() and tapply()
3. How to avoid slow R code
Contents

1 Writing your own functions

2 sapply() and tapply()

3 How to avoid slow R code
Writing your own functions

Basics

Syntax:

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

Example:

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

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

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

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

```r
GC("AATTCGCTTA")
[1] 0.3
```
Writing your own functions

Basics

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

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

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

GC("AATTCGCTTA")
[1] 0.3
Writing your own functions

Basics

Syntax:

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

Example:

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

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

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

```r
GC("AATTCGCTTA")
[1] 0.3
```
Writing your own functions

**Basics**

Syntax:

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

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

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

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

```r
GC("AATTCGCTTA")
[1] 0.3
```
Writing your own functions

Basics

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

Example:
We want to define a function that takes a DNA sequence as input and gives as output 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 gregexpr 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
Writing your own functions

Basics

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

Example:
We want to define a function that takes a DNA sequence as input and gives as output 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?

\texttt{GC("AATTAAATTA")}
Are we sure our function is correct?

```python
GC("AATTAATTA")
[1] 0.1
```

What happened?
Writing your own functions

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

```r
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)
}
```
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
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
```

How can we deal with this? What do we want our function to output in these cases? Find a solution collectively (answer below).
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?

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

```r
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.

```r
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.

```r
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.

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

```r
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

```r
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)
}
```
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:

```r
grep("[^GCAT]", dna)
integer(0)
grep("[^GCAT]", "fATCG")
[1] 1
```
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)
}
Most R functions have several arguments. You can see them listed in the help page.
Writing your own functions

Giving several arguments to a function

Most R functions 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.
Writing your own functions

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 several arguments to a function

```r
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
```
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

GC <- function(dna, AT = FALSE)

Test:
GC(dna) [1] 0.3
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:

```r
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`. 
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)
```
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
Writing your own functions

**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
```
Writing your own functions

Returning several values

To do so use a vector or a list.

```r
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
```
Returning several values

To do so use a vector or a list.

```r
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
```
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

1 Writing your own functions

2 sapply() and tapply()

3 How to avoid slow R code
You use `apply()` and its derivatives to apply the same function to each element of an object.

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

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

```r
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.

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

1. Writing your own functions
2. sapply() and tapply()
3. How to avoid slow R code
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.