Programming in R

Author: Thomas Girke

Last update: 19 April, 2018

Overview

One of the main attractions of using the R (http://cran.at.r-project.org) environment is the ease with which users can write their own programs and custom functions. The R programming syntax is extremely easy to learn, even for users with no previous programming experience. Once the basic R programming control structures are understood, users can use the R language as a powerful environment to perform complex custom analyses of almost any type of data (Gentleman 2008).

Why Programming in R?

- Powerful statistical environment and programming language
- Facilitates reproducible research
- Efficient data structures make programming very easy
- Ease of implementing custom functions
- Powerful graphics
- Access to fast growing number of analysis packages
- Most widely used language in bioinformatics
- Is standard for data mining and biostatistical analysis
- Technical advantages: free, open-source, available for all OSs

R Basics

The previous Rbasics tutorial provides a general introduction to the usage of the R environment and its basic command syntax. More details can be found in the R & BioConductor manual here.

Code Editors for R

Several excellent code editors are available that provide functionalities like R syntax highlighting, auto code indenting and utilities to send code/functions to the R console.

- RStudio: GUI-based IDE for R
- Vim-R-Tmux: R working environment based on vim and tmux
- Emacs (ESS add-on package)
- gedit and Rgedit
- RKWard
- Eclipse
- Tinn-R
- Notepad++ (NppToR)

Programming in R using RStudio

Programming in R using Vim or Emacs

Finding Help

Reference list on R programming (selection)

- Advanced R, by Hadley Wickham
- R Programming for Bioinformatics, by Robert Gentleman
- S Programming, by W. N. Venables and B. D. Ripley
- Programming with Data, by John M. Chambers
- R Help & R Coding Conventions, Henrik Bengtsson, Lund University
- Programming in R (Vincent Zoonekynd)
- Peter's R Programming Pages, University of Warwick
- Rtips, Paul Johnsson, University of Kansas
- R for Programmers, Norm Matloff, UC Davis
- High-Performance R, Dirk Eddelbuettel tutorial presented at useR-2008
- C/C++ level programming for R, Gopi Goswami

Control Structures

Important Operators

Comparison operators

- `==` (equal)
- `!=` (not equal)
- `>` (greater than)
- `>=` (greater than or equal)
- `<` (less than)
- `<=` (less than or equal)

Logical operators

- `&` (and)
- `|` (or)
- `!` (not)

Conditional Executions: `if` Statements

An `if` statement operates on length-one logical vectors.

Syntax

```
if(TRUE) {
    statements_1
} else {
    statements_2
}
```

Example

```
if(1==0) {
    print(1)
} else {
```

```
    print(2)
}
```

```
## [1] 2
```

Conditional Executions: `ifelse` Statements

The `ifelse` statement operates on vectors.

Syntax

```
ifelse(test, true_value, false_value)
```

Example

```
x <- 1:10
ifelse(x<5, x, 0)
```

```
##  [1] 1 2 3 4 0 0 0 0 0 0
```

Loops

`for` loop

`for` loops iterate over elements of a looping vector.

Syntax

```
for(variable in sequence) {
    statements
}
```

Example

```
mydf <- iris
myve <- NULL
for(i in seq(along=mydf[,1])) {
    myve <- c(myve, mean(as.numeric(mydf[i,1:3])))
}
myve[1:8]
```

```
## [1] 3.333333 3.100000 3.066667 3.066667 3.333333 3.666667 3.133333 3.300000
```

Note: Inject into objecs is much faster than append approach with `c`, `cbind`, etc.

Example

```
myve <- numeric(length(mydf[,1]))
for(i in seq(along=myve)) {
    myve[i] <- mean(as.numeric(mydf[i,1:3]))
}
myve[1:8]
```

```
## [1] 3.333333 3.100000 3.066667 3.066667 3.333333 3.666667 3.133333 3.300000
```

Conditional Stop of Loops

The `stop` function can be used to break out of a loop (or a function) when a condition becomes `TRUE`. In addition, an error message will be printed.

Example

```
x <- 1:10
z <- NULL
for(i in seq(along=x)) {
    if(x[i] < 5) {
        z <- c(z, x[i]-1)
    } else {
        stop("values need to be < 5")
    }
}
```

`while` loop

Iterates as long as a condition is true.

Syntax

```
while(condition) {
    statements
}
```

Example

```
z <- 0
while(z<5) {
    z <- z + 2
    print(z)
}
```

```
## [1] 2
## [1] 4
## [1] 6
```

The `apply` Function Family

`apply`

Syntax

```
apply(X, MARGIN, FUN, ARGs)
```

Arguments

- `X`: `array`, `matrix` or `data.frame`
- `MARGIN`: `1` for rows, `2` for columns
- `FUN`: one or more functions
- `ARGs`: possible arguments for functions

Example

```
apply(iris[1:8,1:3], 1, mean)
```

```
##        1        2        3        4        5        6        7        8
## 3.333333 3.100000 3.066667 3.066667 3.333333 3.666667 3.133333 3.300000
```

tapply

Applies a function to vector components that are defined by a factor.

Syntax

```
tapply(vector, factor, FUN)
```

Example

```
iris[1:2,]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
```

```
tapply(iris$Sepal.Length, iris$Species, mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

sapply and lapply

Both apply a function to vector or list objects. The `lapply` function always returns a list object, while `sapply` returns `vector` or `matrix` objects when it is possible.

Examples

```
l <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
lapply(l, mean)
```

```
## $a
## [1] 5.5
##
## $beta
## [1] 4.535125
##
## $logic
## [1] 0.5
```

```
sapply(l, mean)
```

```
##        a     beta    logic
## 5.500000 4.535125 0.500000
```

Often used in combination with a function definition:

```
lapply(names(l), function(x) mean(l[[x]]))
sapply(names(l), function(x) mean(l[[x]]))
```

Functions

Function Overview

A very useful feature of the R environment is the possibility to expand existing functions and to easily write custom functions. In fact, most of the R software can be viewed as a series of R functions.

Syntax to define function

```
myfct <- function(arg1, arg2, ...) {
    function_body
}
```

Syntax to call functions

```
myfct(arg1=..., arg2=...)
```

The value returned by a function is the value of the function body, which is usually an unassigned final expression, *e.g.*: `return()`

Function Syntax Rules

General

- Functions are defined by
  1. The assignment with the keyword `function`
  2. The declaration of arguments/variables (`arg1, arg2, ...`)
  3. The definition of operations (`function_body`) that perform computations on the provided arguments. A function name needs to be assigned to call the function.

Naming

- Function names can be almost anything. However, the usage of names of existing functions should be avoided.

Arguments

- It is often useful to provide default values for arguments (*e.g.*: `arg1=1:10`). This way they don't need to be provided in a function call. The argument list can also be left empty (`myfct <- function() { fct_body }`) if a function is expected to return always the same value(s). The argument `...` can be used to allow one function to pass on argument settings to another.

Body

- The actual expressions (commands/operations) are defined in the function body which should be enclosed by braces. The individual commands are separated by semicolons or new lines (preferred).

Usage

- Functions are called by their name followed by parentheses containing possible argument names. Empty parenthesis after the function name will result in an error message when a function requires certain arguments to be provided by the user. The function name alone will print the definition of a function.

Scope

- Variables created inside a function exist only for the life time of a function. Thus, they are not accessible outside of the function. To force variables in functions to exist globally, one can use the double assignment operator: `<<-`

Examples

Define sample function

```
myfct <- function(x1, x2=5) {
    z1 <- x1 / x1
    z2 <- x2 * x2
        myvec <- c(z1, z2)
        return(myvec)
}
```

Function usage

Apply function to values 2 and 5

```
myfct(x1=2, x2=5)
```

```
## [1]  1 25
```

Run without argument names

```
myfct(2, 5)
```

```
## [1]  1 25
```

Makes use of default value 5

```
myfct(x1=2)
```

```
## [1]  1 25
```

Print function definition (often unintended)

```
myfct
```

```
## function(x1, x2=5) {
##  z1 <- x1 / x1
##  z2 <- x2 * x2
##         myvec <- c(z1, z2)
##         return(myvec)
## }
```

Useful Utilities

Debugging Utilities

Several debugging utilities are available for R. They include:

- `traceback`
- `browser`
- `options(error=recover)`
- `options(error=NULL)`
- `debug`

The Debugging in R page provides an overview of the available resources.

Regular Expressions

R's regular expression utilities work similar as in other languages. To learn how to use them in R, one can consult the main help page on this topic with `?regexp`.

String matching with grep

The grep function can be used for finding patterns in strings, here letter `A` in vector `month.name`.

```
month.name[grep("A", month.name)]
```

```
## [1] "April"  "August"
```

String substitution with gsub

Example for using regular expressions to substitute a pattern by another one using a back reference. Remember: single escapes \ need to be double escaped \\ in R.

```
gsub('(i.*a)', 'xxx_\\1', "virginica", perl = TRUE)
```

```
## [1] "vxxx_irginica"
```

Interpreting a Character String as Expression

Some useful examples

Generates vector of object names in session

```
mylist <- ls()
mylist[1]
```

```
## [1] "i"
```

Executes 1st entry as expression

```
get(mylist[1])
```

```
## [1] 150
```

Alternative approach

```
eval(parse(text=mylist[1]))
```

```
## [1] 150
```

Replacement, Split and Paste Functions for Strings

Selected examples

Substitution with back reference which inserts in this example _ character

```
x <- gsub("(a)","\\1_", month.name[1], perl=T)
x
```

```
## [1] "Ja_nua_ry"
```

Split string on inserted character from above

```
strsplit(x,"_")
```

```
## [[1]]
## [1] "Ja"  "nua" "ry"
```

Reverse a character string by splitting first all characters into vector fields

```
paste(rev(unlist(strsplit(x, NULL))), collapse="")
```

```
## [1] "yr_aun_aJ"
```

Time, Date and Sleep

Selected examples

Return CPU (and other) times that an expression used (here ls)

```
system.time(ls())
```

```
##    user  system elapsed
##       0       0       0
```

Return the current system date and time

```
date()
```

```
## [1] "Sun Apr 16 08:11:14 2017"
```

Pause execution of R expressions for a given number of seconds (e.g. in loop)

```
Sys.sleep(1)
```

Example

Import of Specific File Lines with Regular Expression

The following example demonstrates the retrieval of specific lines from an external file with a regular expression. First, an external file is created with the `cat` function, all lines of this file are imported into a vector with `readLines`, the specific elements (lines) are then retieved with the `grep` function, and the resulting lines are split into vector fields with `strsplit`.

```
cat(month.name, file="zzz.txt", sep="\n")
x <- readLines("zzz.txt")
x[1:6]
```

```
## [1] "January"  "February" "March"    "April"    "May"      "June"
```

```
x <- x[c(grep("^J", as.character(x), perl = TRUE))]
t(as.data.frame(strsplit(x, "u")))
```

```
##                 [,1]  [,2]
## c..Jan....ary.. "Jan" "ary"
## c..J....ne..    "J"   "ne"
## c..J....ly..    "J"   "ly"
```

Calling External Software

External command-line software can be called with `system`. The following example calls `blastall` from R

```
system("blastall -p blastp -i seq.fasta -d uniprot -o seq.blastp")
```

Running R Scripts

Possibilities for Executing R Scripts

R console

```
source("my_script.R")
```

Command-line

```
Rscript my_script.R # or just ./myscript.R after making it executable
R CMD BATCH my_script.R # Alternative way 1
R --slave < my_script.R # Alternative way 2
```

Passing arguments from command-line to R

Create an R script named `test.R` with the following content:

```
myarg <- commandArgs()
print(iris[1:myarg[6], ])
```

Then run it from the command-line like this:

```
Rscript test.R 10
```

In the given example the number `10` is passed on from the command-line as an argument to the R script which is used to return to `STDOUT` the first 10 rows of the `iris` sample data. If several arguments are provided, they will be interpreted as one string and need to be split in R with the strsplit function. A more detailed example can be found here.

Building R Packages

Short Overview of Package Building Process

R packages can be built with the `package.skeleton` function. The given example will create a directory named `mypackage` containing the skeleton of the package for all functions, methods and classes defined in the R script(s) passed on to the `code_files` argument. The basic structure of the package directory is described here. The package directory will also contain a file named `Read-and-delete-me` with instructions for completing the package:

```
package.skeleton(name="mypackage", code_files=c("script1.R", "script2.R"))
```

Once a package skeleton is available one can build the package from the command-line (Linux/OS X). This will create a tarball of the package with its version number encoded in the file name. Subequently, the package tarball needs to be checked for errors with:

```
R CMD build mypackage
R CMD check mypackage_1.0.tar.gz
```

Install package from source

```
install.packages("mypackage_1.0.tar.gz", repos=NULL)
```

For more details see here

Programming Exercises

Exercise 1

`for` loop

Task 1.1: Compute the mean of each row in `myMA` by applying the mean function in a `for` loop.

```
myMA <- matrix(rnorm(500), 100, 5, dimnames=list(1:100, paste("C", 1:5, sep="")))
myve_for <- NULL
for(i in seq(along=myMA[,1])) {
    myve_for <- c(myve_for, mean(as.numeric(myMA[i, ])))
}
myResult <- cbind(myMA, mean_for=myve_for)
myResult[1:4, ]
```

```
##            C1         C2         C3         C4          C5   mean_for
## 1  0.07927221 -1.4428685  0.6975614 -1.1093279  0.44573774 -0.2659250
## 2  1.19742197 -0.4971164 -0.3432921  0.6728799 -0.03686824  0.1986050
## 3 -1.59735579 -2.3832667  0.7183594  0.3964477  1.33681336 -0.3058004
## 4  1.16939495 -1.2115285  0.1470573  0.7375473 -0.31301378  0.1058915
```

`while` loop

Task 1.2: Compute the mean of each row in `myMA` by applying the mean function in a `while` loop.

```
z <- 1
myve_while <- NULL
while(z <= length(myMA[,1])) {
    myve_while <- c(myve_while, mean(as.numeric(myMA[z, ])))
    z <- z + 1
}
myResult <- cbind(myMA, mean_for=myve_for, mean_while=myve_while)
myResult[1:4, -c(1,2)]
```

```
##           C3         C4          C5   mean_for mean_while
## 1  0.6975614 -1.1093279  0.44573774 -0.2659250 -0.2659250
## 2 -0.3432921  0.6728799 -0.03686824  0.1986050  0.1986050
## 3  0.7183594  0.3964477  1.33681336 -0.3058004 -0.3058004
## 4  0.1470573  0.7375473 -0.31301378  0.1058915  0.1058915
```

Task 1.3: Confirm that the results from both mean calculations are identical

```
all(myResult[,6] == myResult[,7])
```

```
## [1] TRUE
```

apply loop

Task 1.4: Compute the mean of each row in myMA by applying the mean function in an `apply` loop

```
myve_apply <- apply(myMA, 1, mean)
myResult <- cbind(myMA, mean_for=myve_for, mean_while=myve_while, mean_apply=myve_apply)
myResult[1:4, -c(1,2)]
```

```
##           C3         C4          C5   mean_for mean_while mean_apply
## 1  0.6975614 -1.1093279  0.44573774 -0.2659250 -0.2659250 -0.2659250
## 2 -0.3432921  0.6728799 -0.03686824  0.1986050  0.1986050  0.1986050
## 3  0.7183594  0.3964477  1.33681336 -0.3058004 -0.3058004 -0.3058004
## 4  0.1470573  0.7375473 -0.31301378  0.1058915  0.1058915  0.1058915
```

Avoiding loops

Task 1.5: When operating on large data sets it is much faster to use the rowMeans function

```
mymean <- rowMeans(myMA)
myResult <- cbind(myMA, mean_for=myve_for, mean_while=myve_while, mean_apply=myve_apply, mean_int=mymea
myResult[1:4, -c(1,2,3)]
```

```
##           C4          C5   mean_for mean_while mean_apply   mean_int
## 1 -1.1093279  0.44573774 -0.2659250 -0.2659250 -0.2659250 -0.2659250
## 2  0.6728799 -0.03686824  0.1986050  0.1986050  0.1986050  0.1986050
## 3  0.3964477  1.33681336 -0.3058004 -0.3058004 -0.3058004 -0.3058004
## 4  0.7375473 -0.31301378  0.1058915  0.1058915  0.1058915  0.1058915
```

Exercise 2

Custom functions

Task 2.1: Use the following code as basis to implement a function that allows the user to compute the mean for any combination of columns in a matrix or data frame. The first argument of this function should specify the input data set, the second the mathematical function to be passed on (*e.g.* `mean`, `sd`, `max`) and the third one should allow the selection of the columns by providing a grouping vector.

```
myMA <- matrix(rnorm(100000), 10000, 10, dimnames=list(1:10000, paste("C", 1:10, sep="")))
myMA[1:2,]
```

```
##          C1         C2          C3        C4        C5         C6         C7         C8         C9
## 1 1.0579551 -1.2770846 -1.32150140 0.6821067 0.3002011  0.6971941  0.9682067 -0.6422546  0.3912427
## 2 0.6294469 -0.9086788  0.05546482 0.8222621 0.1797109 -0.4354320 -0.4247061 -1.0789142 -0.1454791
##           C10
## 1 -0.04067505
## 2  0.43038546
```

```
myList <- tapply(colnames(myMA), c(1,1,1,2,2,2,3,3,4,4), list)
names(myList) <- sapply(myList, paste, collapse="_")
```

```
myMAmean <- sapply(myList, function(x) apply(myMA[,x], 1, mean))
myMAmean[1:4,]
```

```
##      C1_C2_C3   C4_C5_C6      C7_C8    C9_C10
## 1 -0.51354363  0.5598340  0.1629761 0.1752838
## 2 -0.07458905  0.1888470 -0.7518102 0.1424532
## 3 -0.11839169  0.8791726 -0.5360216 0.7488324
## 4 -0.34606523 -0.8315808 -0.9720177 0.2090652
```

Exercise 3

Nested loops to generate similarity matrices

Task 3.1: Create a sample list populated with character vectors of different lengths

```
setlist <- lapply(11:30, function(x) sample(letters, x, replace=TRUE))
names(setlist) <- paste("S", seq(along=setlist), sep="")
setlist[1:6]
```

```
## $S1
##  [1] "o" "b" "k" "a" "n" "x" "p" "w" "h" "n" "r"
##
## $S2
##  [1] "a" "j" "o" "m" "e" "a" "h" "i" "q" "p" "x" "s"
##
## $S3
##  [1] "z" "q" "n" "b" "n" "o" "u" "w" "s" "e" "j" "h" "f"
##
## $S4
##  [1] "n" "m" "h" "b" "h" "z" "v" "r" "x" "i" "f" "c" "x" "o"
##
## $S5
##  [1] "w" "v" "j" "b" "c" "z" "t" "p" "h" "g" "s" "e" "b" "o" "k"
##
## $S6
##  [1] "e" "z" "k" "i" "v" "b" "i" "q" "j" "o" "a" "h" "p" "b" "n" "s"
```

Task 3.2: Compute the length for all pairwise intersects of the vectors stored in `setlist`. The intersects can be determined with the `%in%` function like this: `sum(setlist[[1]] %in% setlist[[2]])`

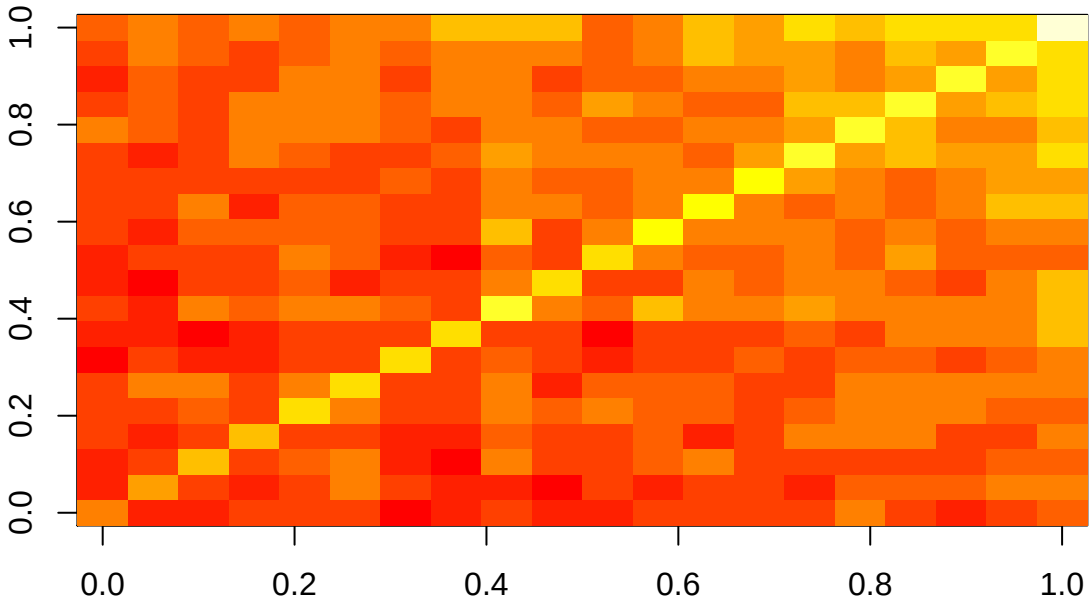
```
setlist <- sapply(setlist, unique)
olMA <- sapply(names(setlist), function(x) sapply(names(setlist),
              function(y) sum(setlist[[x]] %in% setlist[[y]])))
olMA[1:12,]
```

```
##     S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17 S18 S19 S20
## S1  10  5  5  6  6  7  3  5  6   5   5   6   6   6   7   9   7   5   6   8
## S2   5 11  6  5  6  9  7  5  5   4   6   5   7   6   5   8   8   8   9   9
## S3   5  6 12  6  8  9  5  3  9   7   7   8   9   7   6   7   6   6   8   8
## S4   6  5  6 12  6  7  5  5  8   7   6   8   5   6   9   9   9   6   6  10
## S5   6  6  8  6 14 10  7  7 10   8   9   8   8   7   8  10  10  10   8   8
## S6   7  9  9  7 10 14  7  6  9   5   8   8   8   7   6   9  10   9  10   9
## S7   3  7  5  5  7  7 13  6  8   7   5   6   7   8   7   8   8   7   8   9
## S8   5  5  3  5  7  6  6 13  7   7   4   7   7   7   8   7   9  10   9  12
## S9   6  5  9  8 10  9  8  7 17   9   8  12  10   9  11  10  10  10   9  12
```

```
## S10  5  4  7  7  8  5  7  7  9  14   6   7  10   8  10   9   8   7   9  12
## S11  5  6  7  6  9  8  5  4  8   6  13  10   8   8   9   8  11   8   8   8
## S12  6  5  8  8  8  8  6  7 12   7  10  15  10   9  10   8  10   8   9  10
```

Task 3.3 Plot the resulting intersect matrix as heat map. The `image` or the `heatmap.2` function from the `gplots` library can be used for this.

```
image(olMA)
```



Exercise 4

Build your own R package

Task 4.1: Save one or more of your functions to a file called `script.R` and build the package with the `package.skeleton` function.

```
package.skeleton(name="mypackage", code_files=c("script1.R"))
```

Task 4.2: Build tarball of the package

```
system("R CMD build mypackage")
```

Task 4.3: Install and use package

```
install.packages("mypackage_1.0.tar.gz", repos=NULL, type="source")
library(mypackage)
?myMAcomp # Opens help for function defined by mypackage
```

Homework 5

Reverse and complement of DNA

Task 1: Write a `RevComp` function that returns the reverse and complement of a DNA sequence string. Include an argument that will allow to return only the reversed sequence, the complemented sequence or the reversed and complemented sequence. The following R functions will be useful for the implementation:

```
x <- c("ATGCATTGGACGTTAG")
x
```

```
## [1] "ATGCATTGGACGTTAG"
```

```
x <- substring(x, 1:nchar(x), 1:nchar(x))
x
```

```
##  [1] "A" "T" "G" "C" "A" "T" "T" "G" "G" "A" "C" "G" "T" "T" "A" "G"
```

```
x <- rev(x)
x
```

```
##  [1] "G" "A" "T" "T" "G" "C" "A" "G" "G" "T" "T" "A" "C" "G" "T" "A"
```

```
x <- paste(x, collapse="")
x
```

```
## [1] "GATTGCAGGTTACGTA"
```

```
chartr("ATGC", "TACG", x)
```

```
## [1] "CTAACGTCCAATGCAT"
```

Task 2: Write a function that applies the `RevComp` function to many sequences stored in a vector.

Translate DNA into Protein

Task 3: Write a function that will translate one or many DNA sequences in all three reading frames into proteins. The following commands will simplify this task:

```
AAdf <- read.table(file="http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/My_R_Scripts/AA.txt", heade
AAdf[1:4,]
```

```
##   Codon AA_1 AA_3 AA_Full AntiCodon
## 1   TCA    S  Ser  Serine       TGA
## 2   TCG    S  Ser  Serine       CGA
## 3   TCC    S  Ser  Serine       GGA
## 4   TCT    S  Ser  Serine       AGA
```

```
AAv <- as.character(AAdf[,2])
names(AAv) <- AAdf[,1]
AAv
```

```
## TCA TCG TCC TCT TTT TTC TTA TTG TAT TAC TAA TAG TGT TGC TGA TGG CTA CTG CTC CTT CCA CCG CCC CCT CAT
## "S" "S" "S" "S" "F" "F" "L" "L" "Y" "Y" "*" "*" "C" "C" "*" "W" "L" "L" "L" "L" "P" "P" "P" "P" "H"
## CAC CAA CAG CGA CGG CGC CGT ATT ATC ATA ATG ACA ACG ACC ACT AAT AAC AAA AAG AGT AGC AGA AGG GTA GTG
## "H" "Q" "Q" "R" "R" "R" "R" "I" "I" "I" "M" "T" "T" "T" "T" "N" "N" "K" "K" "S" "S" "R" "R" "V" "V"
## GTC GTT GCA GCG GCC GCT GAT GAC GAA GAG GGA GGG GGC GGT
## "V" "V" "A" "A" "A" "A" "D" "D" "E" "E" "G" "G" "G" "G"
```

```
y <- gsub("(...)", "\\1_", x)
y <- unlist(strsplit(y, "_"))
y <- y[grep("^...$", y)]
AAv[y]
```

```
## GAT TGC AGG TTA CGT
## "D" "C" "R" "L" "R"
```

Homework submission

Submit the 3 functions in one well structured and annotated R script to the instructor. The script should include instructions on how to use the functions.

Due date

This homework is due on Thu, April 26th at 6:00 PM.

Homework Solutions

Session Info

```
sessionInfo()
```

```
## R version 3.3.3 (2017-03-06)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 14.04.5 LTS
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C               LC_TIME=en_US.UTF-8
##  [4] LC_COLLATE=en_US.UTF-8     LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C
## [10] LC_TELEPHONE=C             LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  utils     datasets  grDevices base
##
## other attached packages:
## [1] ggplot2_2.1.0   limma_3.30.0    BiocStyle_2.2.0
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.7      codetools_0.2-15 digest_0.6.10    assertthat_0.1   plyr_1.8.4
##  [6] grid_3.3.3       gtable_0.2.0     formatR_1.4      magrittr_1.5     scales_0.4.0
## [11] evaluate_0.10    stringi_1.1.2    rmarkdown_1.1    tools_3.3.3      stringr_1.1.0
## [16] munsell_0.4.3    yaml_2.1.13      colorspace_1.2-7 htmltools_0.3.5  knitr_1.14
## [21] methods_3.3.3    tibble_1.2
```

References

Gentleman, Robert. 2008. *R Programming for Bioinformatics (Chapman & Hall/CRC Computer Science & Data Analysis)*. 1 edition. Chapman; Hall/CRC. http://www.amazon.com/Programming-Bioinformatics-Chapman-Computer-Analysis/dp/1420063677.