Manejo de cadenas y expresiones regulares

27 d'octubre de 2007

1 Introduction

Los datos tipo caracter son muy importantes en algunos campos como la Bioinformática en donde se realizan a menudo acciones como localizar motivos ("subsecuencias fijas"), dentro de una secuencia, o realizar busquedas por coincidencia parcial o total en puede manejarlos utilizando una variedad de herramientas que se discutiran en este tema.

- Por un lado se encuentran las funciones propias de para el manejo y la manipulación de cadenas
- Dada la gran afinidad entre y Unix/linux no es de extrañar que se pueda trabajar con expresiones regulares con herramientas como grep o agrep.
- Finalmente hay paquetes orientados directamente al maejo de tipos específicos de cadenas como seqinR, annotate, matchprobes, Biostrings, geneR, aaMI.

2 Cadenas

Las cadenas en se representan mediante datos de tipo character.

Una cadena se obtiene situando algunos caracteres entre los delimitadores adecuados, ó bien ".

```
> (s1 <- "Hola mundo")

[1] "Hola mundo"

> (s2 <- "Hola mon")

[1] "Hola mon"

> (s3 <- "M'he perdut")

[1] "M'he perdut"
```

Las cadenas se escriben con `cat`. `print` solo muestra su representación.

```
> print(s1)

[1] "Hola mundo"

> cat(s1, "\n")

Hola mundo
```

Los caracteres especiales se indican utilizando secuencias de "escape" que se interpretan como "no trates la siguiente secuencia de manera especial".

Así se escribe `Esto es una contrabarra: \ }`.

```
> s = "Soy una contrabarra: \\"
> cat(s, "\n")

Soy una contrabarra: \
```

`\n }` es un código de control que hace que se realice unsalto de linea.

Las contrabarrras se utilizan para definir caminos del sistema. Esto sin embargo se hace de forma distinta en Windows que en Linux.

```
> dirName0 <- "c:dades\treballsmicroarrays"
> cat("Obviamente asi no es: ", "\n",
+     dirName0, "\n")

Obviamente asi no es:
 c:dades        reballsmicroarrays

> dirName1 <- "c:\\dades\\treballs\\microarrays"
> cat("Asi esta mejor: ", "\n", dirName1,
+     "\n")

Asi esta mejor:
 c:\dades\treballs\microarrays

> dirName2 <- "c:/dades/treballs/microarrays"
> cat("Este es el mejor: ", "\n",
+     dirName2, "\n", "¿Por qué?")

Este es el mejor:
 c:/dades/treballs/microarrays
 ¿Por qué?
```

Podemos cambiar entre ambas representaciones con las funciones chartr o gsub que, como no podía ser menos, funcionan de formas ligeramente distintas.

3 Funciones estándar para el manejo de Cadenas

Las cadenas son uno de los tipos predefinidos de

```
> s <- "Hola mon"
> class(s)

[1] "character"
```

Observese que una cadena es un vector de caracteres, "a la BASIC", es decir no se puede acceder directamente a sus elementos con la función [. Una cadena puede tener

```
> s[1]

[1] "Hola mon"

> s[2]

[1] NA

> length(s)

[1] 1

> nchar(s)

[1] 8

> y <- ""
> y

[1] ""

> length(y)

[1] 1

> nchar(y)

[1] 0

> z <- character(0)
> z

character(0)

> length(z)
```

```
[1] 0

> nchar(z)

integer(0)
```

El accceso a los elementos de una cadena se realiza con substr o substring. substr retorna un valor cuya longitud es como mucho la longitud del primer argumento. substring, en cambio devuelve un valor de longitud igual a la del mayor de los tres argumentos proporcionados.

```
> substr(s, 5, 8)

[1] " mon"

> substring(s, 1, 5)

[1] "Hola "

> substr(s, 5, rep(8, 3))

[1] " mon"

> substring(s, 5, rep(8, 3))

[1] " mon" " mon" " mon"
```

Estas funciones pueden tambien utilizarse en asignación, lo cual tiene el efecto colateral de canviar las cadenas originales.

```
> substr(s, 5, 8) <- "-"
> s

[1] "Hola-mon"

> t <- c("Hola mi amor", "Yo soy tu lobo")
> substr(t, nchar(t) - 4, nchar(t)) <- "*"
> t

[1] "Hola mi*amor"   "Yo soy tu*lobo"

> substring(s, 1, 5)

[1] "Hola-"

> substr(s, 5, rep(8, 3))

[1] "-mon"

> substring(s, 5, rep(8, 3))
```

```
[1] "-mon" "-mon" "-mon"
```

La instrucción paste puede utilizarse para combinar las cadenas de un vector de caracteres

```
> t <- c("Hola mi amor", "Yo soy tu lobo")
> paste(t)

[1] "Hola mi amor"   "Yo soy tu lobo"

> paste(t, collapse = ". ")

[1] "Hola mi amor. Yo soy tu lobo"
```

Las instrucciones strtrim y strwrap pueden utilizarse para formatear texto.

```
> x <- readLines(file.path(R.home(),
+     "COPYING"))
> x[1:10]

 [1] "\t\t    GNU GENERAL PUBLIC LICENSE"
 [2] "\t\t       Version 2, June 1991"
 [3] ""
 [4] " Copyright (C) 1989, 1991 Free Software Foundation, Inc."
 [5] "                          51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA"
 [6] " Everyone is permitted to copy and distribute verbatim copies"
 [7] " of this license document, but changing it is not allowed."
 [8] ""
 [9] "\t\t\t    Preamble"
[10] ""

> x <- paste(readLines(file.path(R.home(),
+     "COPYING")), collapse = "\n")
> writeLines(strwrap(x, 30, prefix = ": ")[1:10])

: GNU GENERAL PUBLIC LICENSE
: Version 2, June 1991
:
: Copyright (C) 1989, 1991
: Free Software Foundation,
: Inc.  51 Franklin St, Fifth
: Floor, Boston, MA
: 02110-1301 USA Everyone is
: permitted to copy and
: distribute verbatim copies

> writeLines(strwrap(x, 30, prefix = "")[1:10])
```

```
GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free
Software Foundation, Inc.  51
Franklin St, Fifth Floor,
Boston, MA 02110-1301 USA
Everyone is permitted to copy
and distribute verbatim
copies of this license
```

Las funciones chrtr y gsub pueden realizar substituciones dentro de una cadena de texto.

```
> randDNA = function(n) paste(sample(c("A",
+     "C", "T", "G"), n, replace = TRUE),
+     collapse = "")
> dna2rna <- function(inputStr) {
+     if (!is.character(inputStr))
+         stop("need character input")
+     is = toupper(inputStr)
+     chartr("T", "U", is)
+ }
> compSeq = function(x) chartr("ACTG",
+     "TGAC", x)
> (x <- c(randDNA(15), randDNA(12)))

[1] "TCCCGCTAGCGACCA" "ATCGTGGTTACC"

> (y <- dna2rna(x))

[1] "UCCCGCUAGCGACCA" "AUCGUGGUUACC"

> (cy <- compSeq(x))

[1] "AGGGCGATCGCTGGT" "TAGCACCAATGG"
```

3.1 Busqueda y coincidencia

Las funciones adecuadas son match, pmatch y charmatch además del operador %in%.

```
> mtable <- c("Intron", "Exon", "Example",
+     "Chromosome")
> match("Exon", mtable)

[1] 2
```

```
> "Example" %in% mtable

[1] TRUE
```

pmatch y charmatch permiten coincidencias parciales. La comparaciónn se realiza progresivamente por lo que puede fallar si dos cadenas empiezan igual

```
> pmatch("E", mtable)

[1] NA

> pmatch("Exo", mtable)

[1] 2

> pmatch("I", mtable)

[1] 1
```

3.2 Computación con el lenguaje y las funciones parse y deparse

Del manual *The R language*:

The parser is what converts the textual representation of code into an internal form which may then be passed to the evaluator which causes the specified instructions to be carried out. The internal form is itself an object and can be saved and otherwise manipulated within the R system.

El análisis ("parse") de una entrada es un proceso habitual en cualquier lenguaje y se encarga de convertir la entrada del usuario en codigo interpretable o compilable según el caso.

Esta capacidad se ha "externalizado" en de forma que un usuario puede utilizar parse para crear variables o expresiones a partir de literales, o la instrucción inversa, deparse, para convertir variables o expresiones en cadenas.

Las expresiones creadas con parse pueden ser evaluadas con eval.

```
> distri1 <- "unif"
> distri2 <- "norm"
> rand1 <- function(distri, n) parse(text = paste("r",
+     distri, "(", n, ")", sep = ""))
> rand1(distri1, 10)

expression(runif(10))

> rand1(distri2, 15)

expression(rnorm(15))

> rand2 <- function(distri, n) eval(parse(text = paste("r",
+     distri, "(", n, ")", sep = "")))
> rand2(distri1, 10)
```

```
 [1] 0.42255554 0.92220198 0.59750842
 [4] 0.25947407 0.39814495 0.04770462
 [7] 0.26413265 0.09172816 0.14610684
[10] 0.83241952
```

```
> rand2(distri2, 15)
```

```
 [1] -2.1193240 -0.5042723 -0.8409443
 [4] -0.3813240 -1.3247041 -0.5502187
 [7] -0.4673854  0.4170305  0.2892733
[10] -1.6948306  0.8965090  0.2687821
[13]  1.1753084  2.0538172  1.7740624
```

La instruccion deparse realiza el proceso inverso.

```
> deparse(rand1(distri1, 10))
```

```
[1] "expression(runif(10))"
```

```
> deparse(substitute(rand1(distri1,
+     10)))
```

```
[1] "rand1(distri1, 10)"
```

```
> deparse(rand2(distri1, 10))
```

```
[1] "c(0.31225615600124, 0.457397691672668, 0.350731739308685, 0.456224514171481, "
[2] "0.44005275843665, 0.8065585878212, 0.219692165032029, 0.913406869862229, "
[3] "0.95179584901780, 0.510407668305561)"
```

Otro ejemplo de utilización de parse(eval(text=...)) Una situacion habitual entre usuarios de Bioconductor ...

```
> if (require(hgu133plus2)) {
+     hgu133plus2()
+     symbs <- unlist(as.list(hgu133plus2SYMBOL))
+     symbs[1:3]
+     envir <- paste("hgu133plus2",
+         "SYMBOL", sep = "")
+     envirenvir <- eval(parse(text = envir))
+     class(envirenvir)
+     mget(c(names(symbs)[1:3], "KK"),
+         env = envirenvir, ifnotfound = NA)
+ }
```

4 Expresiones regulares

A veces necesitamos encontrar algo concreto en un texto o cadena, o reemplazar un texto por otro, ya sea en una aplicación, o en un lenguaje de programación.

Por ejemplo si queremos buscar "tag" y reemplazarlo por "etiqueta" podemos utilizar chartr. La mayoría de aplicaciones o lenguajes tienen una función para realizar estas acciones de forma sencilla.

Pero a veces lo que queremos hacer es más complejo, porque puede que en vez de ser una palabra o parte de palabra simple, necesitemos hacer algo como "búscame todas las palabras que acaben en 'f' y que empiecen por un número del 2 al 71" o "reemplaza las palabras que contengan este grupo de letras por esto".

En estos casos podemos utilizar las *expresiones regulares* (que suelen llamrse "regex" o "regexp" de forma abreviada), que son como un lenguaje para poder definir exactamente qué es lo que queremos buscar o reemplazar.

Segun Wikipedia *una expresión regular, a menudo llamada también patrón, es una expresión que describe un conjunto de cadenas sin enumerar sus elementos.*

Por ejemplo, el grupo formado por las cadenas `Handel, Händel y Haendel` se describe mediante el patrón `H(a|ä|ae)ndel`.

```
> texto <- c("Handel", "Haendel",
+     "Händel", "Handemore", "Mendel",
+     "Handle")
> musicos <- grep("H[a|ä](e)?ndel",
+     texto)
> texto[musicos]
```

```
[1] "Handel"  "Haendel" "Händel"
```

A diferencia de otros lenguajes, en los que las expresiones regulares se encierran entre algun tipo especial de delimitadores (por ejemplo entre "/") en una expresión regular se representa ("es") como una cadena de texto.

```
> patron <- "H[a|ä](e)?ndel"
> musicos <- grep(patron, texto)
> texto[musicos]
```

```
[1] "Handel"  "Haendel" "Händel"
```

4.1 Tipos de expresiones regulares

considera tres tipos de expresiones regulares:

- **Expresiones regulares POSIX extendidas** El formato utilizado por defecto, y definido en el estándar "POSIX". Se construyen combinando caracteres y metacaracteres que son interpretados de manera especial al interpretar la expresión regular.

  Los metacaracteres principales son: `. \\ | ( ) [  * + ?'`

- **Expresiones regulares básicas** Es un subconjuntop de las anteriores, que, entre otras características, se caracteriza por disponer de un subconjunto distinto de metacaracteres. En concreto los metacaracters '?', '+', '{', '|', '(', y ')' pierden su significado especial. Para conseguir el mismo efecto debe utilizarse la versión con contrabarra: '\?'}, \verb", '\', '\|'}, \verb"', y'\)'. *Aspuesenestecasolosmetacaracteresson* : '. \ [^ $ *'\}.

- **Expresiones regulares de Perl** Utilizan, como su nombre indica la sintaxis de Perl, salvo por algunas excepciones.

Las instrucciones que usan expresiones regulares disponen de dos parámetros, `extended`, que por defecto se encuentra a `TRUE` y `perl`, que por defecto se encuenntra a `FALSE`. Si deseamos trabajar con expresiones regulares básicas pondremos `extended`, a `FALSE` y para trabajar con las de perl pondremos `perl` a `TRUE`.

Mientras no se especifique lo contrario, las explicaciones que se dan a continuación se basan en expresiones regulares extendidas.

```
> texto <- c("Handel", "Haendel",
+     "Händel", "Handemore", "Mendel",
+     "Handle")
> sinMusicos <- grep("H[a|ä](e)?ndel",
+     texto, extended = F)
> sinMusicos

integer(0)
```

4.2 Instrucciones para el manejo de expresiones regulares

Las expresiones regulares no se utilizan "sin más". Habitualmente se pasan como argumentos de una función, que utiliza el patron representado por ellas para realizar alguna tarea como búsqueda o sustitución.

Las instrucciones que trabajan con expresiones regulares son las siguientes:

```
grep(pattern, x)grep(pattern, x)(pattern, x, ignore.case = FALSE, extended = TRUE,
     perl = FALSE, value = FALSE, fixed = FALSE, useBytes = FALSE)

regexpr(pattern, text, ignore.case = FALSE, extended = TRUE,
        perl = FALSE, fixed = FALSE, useBytes = FALSE)

gregexpr(pattern, text, ignore.case = FALSE, extended = TRUE,
         perl = FALSE, fixed = FALSE, useBytes = FALSE)

sub(pattern, replacement, x,
    ignore.case = FALSE, extended = TRUE, perl = FALSE,
    fixed = FALSE, useBytes = FALSE)
```

```
gsub(pattern, replacement, x,
     ignore.case = FALSE, extended = TRUE, perl = FALSE,
     fixed = FALSE, useBytes = FALSE)
```

Las funcione grep, regexp, gregexp localizan, de formas distintas, las cadenas que verifican el patron definido por la expresion regular. grep retorna el indice en un vector de aquellas cadenas que verifican la ER. regexp, gregexp devuelven el valor =1 si la cadena verifica la ER y -1 si no lo verifica. Ambas funciones devuelven ademas la longitud de la cadena.

```
> texto <- c("Handel", "Haendel",
+     "Händel", "Handemore", "Mendel",
+     "Handle")
> patron <- "H[a|ä](e)?ndel"
> grep(patron, texto)

[1] 1 2 3

> regexpr(patron, texto)

[1]  1  1  1 -1 -1 -1
attr(,"match.length")
[1]  6  7  6 -1 -1 -1

> gregexpr(patron, texto)

[[1]]
[1] 1
attr(,"match.length")
[1] 6

[[2]]
[1] 1
attr(,"match.length")
[1] 7

[[3]]
[1] 1
attr(,"match.length")
[1] 6

[[4]]
[1] -1
attr(,"match.length")
[1] -1

[[5]]
```

```
[1] -1
attr(,"match.length")
[1] -1

[[6]]
[1] -1
attr(,"match.length")
[1] -1
```

```
> txt <- c("arm", "foot", "lefroo",
+     "bafoobar")
> if (any(i <- grep("foo", txt))) cat("'foo' appears at least once in\n\t",
+     txt, "\n")
```

```
'foo' appears at least once in
         arm foot lefroo bafoobar
```

```
> i
```

```
[1] 2 4
```

```
> txt[i]
```

```
[1] "foot"     "bafoobar"
```

Las funciones sub, gsub realizan substituciones sobre las secuencias que verifican el patron.

```
> gsub("([ab])", "\\1_\\1_", "abc and ABC")
```

```
[1] "a_a_b_b_c a_a_nd ABC"
```

```
> str <- "Now is the time      "
> sub(" +$", "", str)
```

```
[1] "Now is the time"
```

```
> sub("[[:space:]]+$", "", str)
```

```
[1] "Now is the time"
```

```
> sub("\\s+$", "", str, perl = TRUE)
```

```
[1] "Now is the time"
```

```
> gsub("(\\w)(\\w*)", "\\U\\1\\L\\2",
+     "a test of capitalizing", perl = TRUE)
```

```
[1] "A Test Of Capitalizing"
```

```
> gsub("\\b(\\w)", "\\U\\1", "a test of capitalizing",
+     perl = TRUE)
```

```
[1] "A Test Of Capitalizing"
```

4.3 Creacion de expresiones regulares

Existen multitud de tutoriales sobre expresiones regulares en internet. En estos tutoriales se explican los conceptos presentados aqui, y ademas, se expone con un cierto detalle la sintaxis necesaria para crear expresiones regulares.

ALgunos de estos tutoriales son:

http://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

http://www.wellho.net/solutions/php-pattern-matching-a-primer-on-regular-expressions.html

http://www.monografias.com/trabajos29/introduccion-expresiones-regulares/introduccion-expresiones-regulares.shtml

4.4 Ejemplos

4.4.1 Algunos ejemplos de la ayuda de

```
> txt <- c("The", "licenses", "for",
+     "most", "software", "are",
+     "designed", "to", "take", "away",
+     "your", "freedom", "to", "share",
+     "and", "change", "it.", "",
+     "By", "contrast,", "the", "GNU",
+     "General", "Public", "License",
+     "is", "intended", "to", "guarantee",
+     "your", "freedom", "to", "share",
+     "and", "change", "free", "software",
+     "--", "to", "make", "sure",
+     "the", "software", "is", "free",
+     "for", "all", "its", "users")
> (i <- grep("[gu]", txt))

[1]  7 11 16 24 29 30 35 41 49

> stopifnot(txt[i] == grep("[gu]",
+     txt, value = TRUE))
> (ot <- sub("[b-e]", ".", txt))

 [1] "Th."       "li.enses"  "for"
 [4] "most"      "softwar."  "ar."
 [7] ".esigned"  "to"        "tak."
[10] "away"      "your"      "fr.edom"
[13] "to"        "shar."     "an."
[16] ".hange"    "it."       ""
[19] ".y"        ".ontrast," "th."
[22] "GNU"       "G.neral"   "Pu.lic"
[25] "Li.ense"   "is"        "int.nded"
[28] "to"        "guarant.e" "your"
```

```
[31] "fr.edom"   "to"        "shar."
[34] "an."       ".hange"    "fr.e"
[37] "softwar."  "--"        "to"
[40] "mak."      "sur."      "th."
[43] "softwar."  "is"        "fr.e"
[46] "for"       "all"       "its"
[49] "us.rs"

> txt[ot != gsub("[b-e]", ".", txt)]

 [1] "licenses"  "designed"  "freedom"
 [4] "change"    "General"   "Public"
 [7] "License"   "intended"  "guarantee"
[10] "freedom"   "change"    "free"
[13] "free"

> txt[gsub("g", "#", txt) != gsub("g",
+     "#", txt, ignore.case = TRUE)]

[1] "GNU"     "General"

> regexpr("en", txt)

 [1] -1  4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[13] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  2 -1
[25]  4 -1  4 -1 -1 -1 -1 -1 -1 -1 -1 -1
[37] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[49] -1
attr(,"match.length")
 [1] -1  2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[13] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  2 -1
[25]  2 -1  2 -1 -1 -1 -1 -1 -1 -1 -1 -1
[37] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[49] -1
```

4.4.2 Aplicaciones a la Bioinformática

Las expresiones regulares pueden ser de gran utilidad en análisis de secuencias en donde se realiza un gran número de tareas relacionadas con la búsqueda y la sustitución o extracción de cadenas de un texto.

La web de Thomas Girke contiene algunos interesantes ejemplos de análisis de secuencias con Por ejemplo desde la dirección:

`http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/R_Programming.html#Seq_analysis`

se puede acceder al script `sequenceanalysis.txt` `http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/My_R_Scripts/sequenceAnalysis.txt` que realiza algunos analisis basicos que utilizan expresiones regulares.