7

Character Manipulation

While R is usually thought of as a language designed for numerical computation, it contains a full complement of functions which can manipulate character data. Combined with R's powerful vectorized operations, these functions can perform the same sorts of tasks that scripting languages like perl and python are often used for.

7.1 Basics of Character Data

Character values in R can be stored as scalars, vectors, or matrices, or they can be columns of a data frame or elements of a list. When applied to objects like this, the `length` function will report the number of character values in the object, not the number of characters in each string. To find the number of characters in a character value, the `nchar` function can be used. Like most functions in R, `nchar` is vectorized. For example, the names of the fifty states in the United States can be found in the vector `state.name` which is distributed as part of R. To find the lengths of the names of the states, `nchar` can be used:

```
> nchar(state.name)
[1]  7  6  7  8 10  8 11  8  7  7  6  5  8  7  4
[16]  6  8  9  5  8 13  8  9 11  8  7  8  6 13 10
[31] 10  8 14 12  4  8  6 12 12 14 12  9  5  4  7
[46]  8 10 13  9  7
```

7.2 Displaying and Concatenating Character Strings

Like other objects in R, character values will be displayed when their name is typed at the console or when they are passed to the `print` function. However, it is often more convenient to print or display these objects directly without the subscripts that the `print` function provides. The `cat` function will combine character values and print them to the screen or a file directly. The `cat` function coerces its arguments to character values, then concatenates and displays them. This makes the function ideal for printing messages and warnings from inside of functions:

```
> x = 7
> y = 10
> cat('x should be greater than y, but x=',x,'and y=',y,'\n')
x should be greater than y, but x= 7 and y= 10
```

Note the use of a newline (`\n`) in the argument list to insure that a complete line is displayed. `cat` will always print a newline when it encounters a newline character. When there are multiple strings passed to `cat`, or when the argument to `cat` is a vector of character strings, the `fill=` argument can be used to automatically insert newlines into the output string. If `fill=` is set to `TRUE`, the value of the system `width` option will be used to determine the linesize; if a numeric value is used, the output will be displayed using that width, although `cat` will not insert newlines into individual elements of its input:

```
> cat('Long strings can','be displayed over',
+     'several lines using','the fill= argument',
+     fill=40)
Long strings can be displayed over
several lines using the fill= argument
```

The `cat` function also accepts a `file=` argument to specify that its output should be directed to a file. When the `file=` argument is used, the `append=TRUE` argument can also be provided to have `cat` append its output to an already existing file.

For more control over the way that character values are concatenated, the `paste` function can be used. In its simplest usage, this function will accept an unlimited number of scalars, and join them together, separating each scalar with a space by default. To use a character string other than a space as a separator, the `sep=` argument can be used. If any object passed to `paste` is not of mode character, it is converted to character:

```
> paste('one',2,'three',4,'five')
[1] "one 2 three 4 five"
```

If a character vector is passed to `paste`, the `collapse=` argument can be used to specify a character string to place between each element of the vector:

```
> paste(c('one','two','three','four'),collapse=' ')
[1] "one two three four"
```

Note that the `collapse=` argument must be used in these cases, as `sep=` has no effect when applied to a vector.

When multiple arguments are passed to `paste`, it will vectorize the operation, recycling shorter elements when necessary. This makes it easy to generate variable names with a common prefix:

```
> paste('X',1:5,sep='')
[1] "X1" "X2" "X3" "X4" "X5"
> paste(c('X','Y'),1:5,sep='')
[1] "X1" "Y2" "X3" "Y4" "X5"
```

In cases like this, the `sep=` argument controls what is placed between each set of values that are combined, and the `collapse=` argument can be used to specify a value to use when joining those individual values to create a single string:

```
> paste(c('X','Y'),1:5,sep='_',collapse='|')
[1] "X_1|Y_2|X_3|Y_4|X_5"
```

The same sort of operations can be applied to multiple arguments to `paste`:

```
> paste(c('X','Y'),1:5,'^',c('a','b'),sep='_',collapse='|')
[1] "X_1_^_a|Y_2_^_b|X_3_^_a|Y_4_^_b|X_5_^_a"
```

By omitting the `collapse` argument, the individual pasted pieces are returned separately instead of being joined into a single string:

```
> paste(c('X','Y'),1:5,'^',c('a','b'),sep='_')
[1] "X_1_^_a" "Y_2_^_b" "X_3_^_a" "Y_4_^_b" "X_5_^_a"
```

7.3 Working with Parts of Character Values

Individual characters of character values are not accessible through ordinary subscripting. Instead, the `substring` function can be used either to extract parts of character strings, or to change the values of parts of character strings. In addition to the string being operated on, `substring` accepts a `first=` argument giving the first character of the desired substring, and a `last=` argument giving the last character. If not specified, `last=` defaults to a large number, so that specifying just a `start=` value will operate from that character to the end of the string. Like most functions in R, `substring` is vectorized, operating on multiple strings at once:

```
> substring(state.name,2,6)
 [1] "labam" "laska" "rizon" "rkans" "alifo" "olora" "onnec"
 [8] "elawa" "lorid" "eorgi" "awaii" "daho"  "llino" "ndian"
[15] "owa"   "ansas" "entuc" "ouisi" "aine"  "aryla" "assac"
[22] "ichig" "innes" "issis" "issou" "ontan" "ebras" "evada"
[29] "ew Ha" "ew Je" "ew Me" "ew Yo" "orth " "orth " "hio"
[36] "klaho" "regon" "ennsy" "hode " "outh " "outh " "ennes"
[43] "exas"  "tah"   "ermon" "irgin" "ashin" "est V" "iscon"
[50] "yomin"
```

Notice that in the case of strings that have fewer characters than specified in the `last=` argument (like `Ohio` or `Texas` in this example), `substring` returns as many characters as it finds with no padding provided. (The `sprintf` function can be used to pad a series of character values to a common size; see Section 2.13.)

Vectorization takes place for the `first=` and `last=` arguments as well as for character vectors passed to `subscript`. Although the `strsplit` function described in Section 7.6 can perform the operation automatically, a vector of character values can be created from a single string by `substring` as follows:

```
> mystring = 'dog cat duck'
> substring(mystring,c(1,5,9),c(3,7,12))
[1] "dog"  "cat"  "duck"
```

For finding locations of particular characters within a character string, the string first needs to be converted to a character vector containing individual characters. This can be done by passing a vector consisting of all the characters to be processed as both the `first=` and `last=` arguments, and then applying `which` to the result:

```
> state = 'Mississippi'
> ll = nchar(state)
> ltrs = substring(state,1:ll,1:ll)
> ltrs
 [1] "M" "i" "s" "s" "i" "s" "s" "i" "p" "p" "i"
> which(ltrs == 's')
[1] 3 4 6 7
```

The assignment form of `substring` allows replacement of selected portions of character strings, but `substring` will only replace parts of the string with values that have the same number of characters; if a string that's shorter than the implied substring is provided, none of the original string will be overwritten:

```
> mystring = 'dog cat duck'
> substring(mystring,5,7) = 'feline'
> mystring
[1] "dog fel duck"
> mystring = 'dog cat duck'
> substring(mystring,5,7) = 'a'
> mystring
[1] "dog aat duck"
```

7.4 Regular Expressions in R

Regular expressions are a method of expressing patterns in character values which can then be used to extract parts of strings or to modify those strings in some way. Regular expressions are supported in the R functions `strsplit`, `grep`, `sub`, and `gsub`, as well as in the `regexpr` and `gregexpr` functions which are the main tools for working with regular expressions in R.

Regular expression syntax varies depending on the particular implementation a program uses. R tries to provide a great deal of flexibility regarding the regular expressions it understands. By default, R uses a basic set of regular expressions similar to those used by UNIX utilities like `grep`. The `extended=TRUE` argument to R functions that support regular expressions extend the set of regular expressions to include those supported by the POSIX 1003.2 standard. To use regular expressions like those supported by scripting languages such as perl and python, the `perl=TRUE` argument can be used. Thus, if you're already familiar with a particular type of regular expressions, you can probably find an option that will make R work the way you expect it to.

The backslash character (\) is used in regular expressions to signal that certain characters with special meaning in regular expressions should be treated as normal characters. In R, this means that two backslash characters need to be entered into an input string anywhere that special characters need to be escaped. Although the double backslash will display when the string is printed, `nchar` or `cat` can verify that only a single backslash is actually included in the string. For example, in regular expressions, a period (`.`) is ordinarily matched by any single character. To create a regular expression that would match file names with an extension of "`.txt`", we could use a regular expression like

```
> expr = '.*\\.txt'
> nchar(expr)
[1] 7
> cat(expr,'\n')
.*\.txt
```

Single backslashes, like those which are part of a newline character (`\n`), will be interpreted correctly inside of regular expressions. One way to avoid the need for quotes or double backslashes is to use the `readline` function to enter your regular expressions into R. For example, we could use `readline` in the previous example as follows:

```
> expr = readline()
.*\.txt
> nchar(expr)
[1] 7
```

7.5 Basics of Regular Expressions

Regular expressions are composed of three components: literal characters, which are matched by a single character; character classes, which can be matched by any of a number of characters, and modifiers, which operate on literal characters or character classes. Since many punctuation marks are regular expression modifiers, the following characters must always be preceded by a backslash to retain their literal meaning:

```
. ^ $ + ? * ( ) [ ] { } | \
```

To form a character class, use square brackets (`[]`) surrounding the characters that you would like to match. For example, to create a character class that will be matched by either `a`, `b`, or `3`, use `[ab3]`. Dashes can be used inside of character classes to represent a range of values such as `[a-z]` or `[5-9]`. Because of this, if a dash is to be literally included in a character class, it should either be the first character in the class or it should be preceded by a backslash. Other special characters (except square brackets) do not need a backslash when used in a character class.

Using characters and character classes as basic building blocks, we can now construct regular expressions by understanding the modifiers that are part of the regular expression language. These operators are listed in Table 7.1.

Modifier	Meaning
`^`	anchors expression to beginning of target
`$`	anchors expression to end of target
`.`	matches any single character except newline
`|`	separates alternative patterns
`()`	groups patterns together
`*`	matches 0 or more occurrences of preceding entity
`?`	matches 0 or 1 occurrences of preceding entity
`+`	matches 1 or more occurrences of preceding entity
{*n*}	matches exactly *n* occurrences of preceding entity
{*n*,}	matches at least *n* occurrences of preceding entity
{*n*,*m*}	matches between *n* and *m* occurrences

Table 7.1. Modifiers for regular expressions

The modifiers operate on whatever entity they follow, using parentheses for grouping if necessary. As some simple examples, a string with two digits followed by one or more letters could be matched by the regular expression "`[0-9][0-9][a-zA-Z]+`"; three consecutive occurrences of the string "`abc`" could be matched by "`(abc){3}`"; a filename consisting of all letters, and ending in "`.jpg`" could be matched by "`^[a-zA-Z]+\\.jpg$`". (In the previous example, the double backslashes would be required if you entered the regular expression as a quoted string in R; if you used `readline` to enter the expression, you would use just a single backslash.)

Remember that regular expressions are simply character strings in R, so they can be manipulated like any other character strings. For example, the vertical bar (`|`) is used in regular expressions to express alternation. To create a regular expression that would be matched by several different strings, we can combine the strings using the bar as a separator:

```
> strs = c('chicken','dog','cat')
> expr = paste(strs,collapse='|')
> expr
[1] "chicken|dog|cat"
```

The variable `expr` could now be used as a regular expression to match any of the words in the original vector.

7.6 Breaking Apart Character Values

The `strsplit` function can use a character string or regular expression to divide up a character string into smaller pieces. The first argument to `strsplit` is the character string to break up, and the second argument is the character value or regular expression which should be used to break up the string into parts.

Like other functions that can return different numbers of elements from their inputs, `strsplit` returns its results as a list, even when its input is a single character string. For example, suppose we want to break up a simple sentence into individual words, by splitting the string wherever a blank occurs:

```
> sentence =
+ 'R is a free software environment for statistical computing'
> parts = strsplit(sentence,' ')
> parts
[[1]]
[1] "R"             "is"            "a"             "free"
[5] "software"      "environment" "for"           "statistical"
[9] "computing"
```

To access the results, the first element of the list must be used:

```
> length(parts)
[1] 1
> length(parts[[1]])
[1] 9
```

When the input to `strsplit` is a vector of character strings, `sapply` can be used to process the output to return results for each of the strings:

```
> more = c('R is a free software environment for statistical
+           computing', 'It compiles and runs on a wide
            variety of UNIX platforms')
> result = strsplit(more,' ')
> sapply(result,length)
[1]  9 11
```

Alternatively, if the structure of the output is not important, all of the split parts can be combined using `unlist`:

```
> allparts = unlist(result)
> allparts
 [1] "R"           "is"          "a"           "free"
 [5] "software"    "environment" "for"         "statistical"
 [9] "computing"   "It"          "compiles"    "and"
[13] "runs"        "on"          "a"           "wide"
[17] "variety"     "of"          "UNIX"        "platforms"
```

Because `strsplit` can accept regular expressions to decide where to split a character string, a wide variety of situations can be easily handled. For example, if there are multiple spaces in a string, and a space is used as the splitting character, extra empty strings may be returned:

```
> str = 'one  two   three four'
> strsplit(str,' ')
[[1]]
[1] "one"   ""      "two"   ""      ""      "three" "four"
```

By using a regular expression representing one or more blanks (using the + modifier), we can extract only nonempty strings:

```
> strsplit(str,' +')
[[1]]
[1] "one"   "two"   "three" "four"
```

Using an empty string as the splitting character, `strsplit` can return a list of individual characters from a vector of character strings:

```
> words = c('one two','three four')
> strsplit(words,'')
[[1]]
[1] "o" "n" "e" " " "t" "w" "o"

[[2]]
 [1] "t" "h" "r" "e" "e" " " "f" "o" "u" "r"
```

7.7 Using Regular Expressions in R

The `grep` function accepts a regular expression and a character string or vector of character strings, and returns the indices of those elements of the strings which are matched by the regular expression. If the `value=TRUE` argument is passed to `grep`, it will return the actual strings which matched the expression instead of the indices.

If the string to be matched should be interpreted literally (i.e., not as a regular expression), the `fixed=TRUE` argument should be used.

One important use of grep is to extract a set of variables from a data frame based on their names. For example, the LifeCycleSavings data frame contains two variables with information about the percentage of population less than 15 years old (pop15) or greater than 75 years old (pop75). Since both of these variables begin with the string "pop", we can find their indices or values using grep:

```
> grep('^pop',names(LifeCycleSavings))
[1] 2 3
> grep('^pop',names(LifeCycleSavings),value=TRUE)
[1] "pop15" "pop75"
```

To create a data frame with just these variables, we can use the output of grep as a subscript:

```
> head(LifeCycleSavings[,grep('^pop',names(LifeCycleSavings))])
          pop15 pop75
Australia 29.35  2.87
Austria   23.32  4.41
Belgium   23.80  4.43
Bolivia   41.89  1.67
Brazil    42.19  0.83
Canada    31.72  2.85
```

To find regular expressions without regard to the case (upper or lower) of the input, the ignore.case=TRUE argument can be used. To search for the string "dog" appearing as a word and ignoring case, we could use the following:

```
> inp = c('run dog run','work doggedly','CAT AND DOG')
> grep('\\<dog\\>',inp,ignore.case=TRUE)
[1] 1 3
```

Surrounding a string with escaped angle brackets (\\< and \\>) restricts matches to the case where the string is surrounded by either white space, punctuation, or a line ending or beginning.

If the regular expression passed to grep is not matched in any of its inputs, grep returns an empty numeric vector. Thus, the any function can be used to test if a regular expression occurs anywhere in a vector of strings:

```
> str1 = c('The R Foundation','is a not for profit
+           organization','working in the public interest')
> str2 = c(' It was founded by the members',
+          'of the R Core Team in order',
+          'to provide support for the R project')
> any(grep('profit',str1))
[1] TRUE
> any(grep('profit',str2))
[1] FALSE
```

While the grep function can be used to test for the presence of a regular expression, sometimes more details regarding the matches that are found are needed. In R, the regexpr and gregexpr functions can be used to pinpoint and possibly extract those parts of a string that were matched by a regular expression. The output from these functions is a vector of starting positions of the regular expressions which were found; if no match occurred, a value of -1 is returned. In addition, an attribute called match.length is associated with the vector of starting positions to provide information about exactly which characters were involved in the match. The regexpr function will only provide information about the first match in its input string(s), while the gregexpr function returns information about all matches found. The input arguments to regexpr and gregexpr are similar to those of grep; however, the ignore.case=TRUE argument is not available in versions of R earlier than version 2.6.

Since regexpr only reports the first match it finds, it will always return a vector, with -1 in those positions where no match was found. To extract the strings that actually matched, substr can be used, after calculating the ending position from the regexpr output and the match.length attribute:

```
> tst = c('one x7 two b1','three c5 four b9',
+          'five six seven','a8 eight nine')
> wh = regexpr('[a-z][0-9]',tst)
> wh
[1]  5  7 -1  1
attr(,"match.length")
[1]  2  2 -1  2
> res = substring(tst,wh,wh + attr(wh,'match.length') - 1)
> res
[1] "x7" "c5" ""   "a8"
```

In the case of the third string, which did not contain the regular expression, an empty string is returned, preserving the structure of the output relative to the input. If empty strings are not desired, they can be easily removed:

```
> res[res != '']
[1] "x7" "c5" "a8"
```

The output from gregexpr is similar to that of regexpr, but, like strsplit, gregexpr always returns its result in the form of a list. Continuing the previous example, but looking for all matches, we can call gregexpr as follows:

```
> wh1 = gregexpr('[a-z][0-9]',tst)
> wh1
[[1]]
[1]  5 12
attr(,"match.length")
[1] 2 2
```

```
[[2]]
[1]  7 15
attr(,"match.length")
[1] 2 2

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

[[4]]
[1] 1
attr(,"match.length")
[1] 2
```

To further process the results from `gregexpr`, we need to call the `substring` function for each element of the output list. One way to do it is with a loop:

```
> res1 = list()
> for(i in 1:length(wh1))
+          res1[[i]] = substring(tst[i],wh1[[i]],
+                          wh1[[i]] +
+                          attr(wh1[[i]],'match.length') -1)
> res1
[[1]]
[1] "x7" "b1"

[[2]]
[1] "c5" "b9"

[[3]]
[1] ""

[[4]]
[1] "a8"
```

Another possibility for processing the output is to use `mapply`. The first argument to `mapply` is a function that accepts multiple arguments; the remaining arguments are vectors of equal lengths (like the text input and the output from `gregexpr`), whose elements will be passed to that function one at a time. The same technique used in the previous example can be encapsulated into a function as follows:

```
> getexpr = function(str,greg)substring(str,greg,
+                           greg + attr(greg,'match.length') - 1)
```

Now `mapply` can be called with the two vectors of interest:

```
> res2 = mapply(getexpr,tst,wh1)
> res2
$"one x7 two b1"
[1] "x7" "b1"

$"three c5 four b9"
[1] "c5" "b9"

$"five six seven"
[1] ""

$"a8 eight nine"
[1] "a8"
```

One advantage of this approach is that it automatically creates an appropriate object to hold the output; in addition, `mapply` uses the input strings as names in the output, which may or may not be desirable.

7.8 Substitutions and Tagging

For substituting text based on regular expressions, R provides two functions: `sub` and `gsub`. Each of these functions accepts a regular expression, a string containing what will be substituted for the regular expression, and the string or strings to operate on. The `sub` function changes only the first occurrence of the regular expression, while the `gsub` function performs the substitution on all occurrences within the string.

One important use of these functions concerns numeric data which is read from text sources like web pages or financial reports, and which may contain commas or dollar signs. For example, suppose we've input a vector of values from a financial report as follows:

```
> values = c('$11,317.35','$11,234.51','$11,275.89',
+            '$11,278.93','$11,294.94')
```

To use these values as numbers, the commas and dollar signs need to be removed before `as.numeric` can be used. A regular expression to find either commas or dollar signs can be composed using a character class, and this can be passed to `gsub` with an empty substitution pattern, providing values which can be converted to numbers:

```
> as.numeric(gsub('[$,]','',values))
[1] 11317.35 11234.51 11275.89 11278.93 11294.94
```

When using this technique, avoid using `as.numeric` on anything less than an entire vector of values, since the mode of an individual element of a matrix or a single value in a data frame cannot be changed.

When using the substitution functions, a powerful feature known as tagging of regular expressions is available. When part of a regular expression is surrounded by (unescaped) parentheses, that part can be used in a substitution pattern by representing it as a backslash followed by a number. The first tagged pattern is represented by `\\1`, the second by `\\2`, and so on. A common practice in financial reports is to surround values that represent negative numbers with parentheses; these parentheses will prevent R from properly interpreting such values as numbers. We can tag the number inside the parentheses using a regular expression, and substitute the value by preceding it with a minus sign. Note the difference between the literal parentheses (preceded by two backslashes) and the parentheses used for tagging:

```
> values = c('75.99','(20.30)','55.20')
> as.numeric(gsub('\\(([0-9.]+)\\)','-\\1',values))
[1]  75.99 -20.30  55.20
```

To extract just the tagged pattern from a regular expression, one possibility is to use the regular expression beginning and end anchor characters (`^` and `$`, respectively) to account for all the nontagged characters in the string, and specify just the tagged expression for the substitution string. For example, suppose we are trying to extract a value preceded by the string `value=` from a longer string. Simply substituting the regular expression for the tagged part will retain all the other parts of the string:

```
> str = 'report: 17 value=12 time=2:00'
> sub('value=([^ ]+)','\\1',str)
[1] "report: 17 12 time=2:00"
```

(The regular expression `[^ ]+` is interpreted as one or more occurrences of a character that is not a blank.) By expanding the regular expression to include all the unwanted parts, the substitution will extract just what we want:

```
> sub('^.*value=([^ ]+).*$','\\1',str)
[1] "12"
```

Another strategy is to use `regexpr` or `gregexpr` to find the location of the match, and apply `sub` or `gsub` to the extracted parts:

```
> str = 'report: 17 value=12 time=2:00'
> greg = gregexpr('value=[^ ]+',str)[[1]]
> sub('value=([^ ]+)','\\1',
+      substring(str,greg,greg
                     + attr(greg, 'match.length') - 1))
[1] "12"
```