Strings and Regular ExpressionsStatistics 506

Working with Strings in R

In R, you create strings of type `character` using either single or double quotes. There is no difference (in R) between the two.

```
string1 <- "This is a string."
string2 <- 'This is a string.'
all.equal(string1, string2)
```

```
[1] TRUE
```

```
typeof(string1)
```

```
[1] "character"
```

This is not the case in all languages. For instance, as we discussed in SQL, single quotes always defines the begin and end of a string, where-as double quotes may not be defined for some variations of SQL.

In R, you can mix single and double quotes when you want to include one or the other within your string.

```
string_single <- "These are sometimes called 'scare quotes'."
print(string_single)
```

```
[1] "These are sometimes called 'scare quotes'."
```

```r
string_double <- 'Quoth the Raven, "Nevermore."'
print(string_double)
```

```
[1] "Quoth the Raven, \"Nevermore.\""
```

```r
cat(string_double, "\n")
```

```
Quoth the Raven, "Nevermore."
```

You can also include quotes within a string by escaping them:

```r
string_double <- "Quoth the Raven, \"Nevermore.\""
print(string_double)
```

```
[1] "Quoth the Raven, \"Nevermore.\""
```

```r
cat(string_double, "\n")
```

```
Quoth the Raven, "Nevermore."
```

Observe the difference between `print()` and `cat()` in terms of how the escaped characters are handled. Be aware also that because backslash plays this special role as an escape character, it itself needs to be escaped:

```r
backslash = "This is a backslash '\\', this is not '\ '."
writeLines(backslash)
```

```
This is a backslash '\', this is not ' '.
```

To have a consistent coding style, you should choose one of `"` or `'` to be the standard. I recommend `"`.

String operations

There are a number of functions in R that operate solely on strings. Here we will review some of them.

Concatenating strings

The function `paste` is used to join strings together.

Observe the difference between the `sep` and `collapse` arguments in `paste`.

```r
length(LETTERS)
```

```
[1] 26
```

```r
paste(LETTERS, collapse = "")
```

```
[1] "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```r
paste(1:26, LETTERS, sep = "-")
```

```
 [1] "1-A"  "2-B"  "3-C"  "4-D"  "5-E"  "6-F"  "7-G"  "8-H"  "9-I"  "10-J"
[11] "11-K" "12-L" "13-M" "14-N" "15-O" "16-P" "17-Q" "18-R" "19-S" "20-T"
[21] "21-U" "22-V" "23-W" "24-X" "25-Y" "26-Z"
```

```r
paste(1:26, LETTERS, sep = "-", collapse = ", ")
```

```
[1] "1-A, 2-B, 3-C, 4-D, 5-E, 6-F, 7-G, 8-H, 9-I, 10-J, 11-K, 12-L, 13-M, 14-N, 15-O, 16-P, 
```

Both `sep` and `collapse` control what goes in between strings, but:

- `sep` controls how multiple arguments are concatenated.
- `collapse` controls how a vector is concatenated.

The `paste0` function operates identically, but has the default argument `sep = ""`:

```r
paste("a", "b")
```

```
[1] "a b"
```

```r
paste0("a", "b")
```

```
[1] "ab"
```

Length of string

Recall that `length` returns the length of a vector. To get the length of a string use `nchar`:

```r
length(paste(LETTERS, collapse = "") )
```

```
[1] 1
```

```r
nchar(paste(LETTERS, collapse = "") )
```

```
[1] 26
```

Substrings

The `substr` functions extract sub-strings at given positions.

```r
substr("Strings",  3, 7)
```

```
[1] "rings"
```

Finding matches

Let's load up a list of superpowers from https://github.com/nolshru/SuperpowerRandSelector:

```r
powers <- readLines("https://raw.githubusercontent.com/nolshru/SuperpowerRandSelector/main
powers <- tolower(powers) # lowercase to avoid complexity
head(powers)
```

```
[1] "enhanced strength"    "enhanced speed"       "enhanced durability"
[4] "enhanced stamina"     "enhanced agility"     "enhanced flexibility"
```

The function `grep` returns the indices of all strings within a vector that contain the requested pattern. The `grepl` function behaves in the same way but returns a logical vector of the same length as the input `x`. (These actually use regular expression, which will be explored below.)

Note that these operate somewhat backwards - most R functions first argument is the object to be operated on, whereas in `grep`, the second argument is the object to be operated on.

```r
grep("arts", powers)
```

```
[1] 386 403 432 433 438 447 448 689 691
```

```r
head(grepl("arts", powers) )
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```r
powers[grepl("arts", powers)]
```

```
[1] "mixed martial arts intuition"   "secret arts"
[3] "animalistic martial arts"       "mystical martial arts"
[5] "ninpo arts"                     "fictional martial arts mastery"
[7] "martial arts mastery"           "dark arts"
[9] "white arts"
```

These functions are vectorized over the search vector but not the pattern.

```r
grep(c("arts", "dance"), powers)
```

```
Warning in grep(c("arts", "dance"), powers): argument 'pattern' has length > 1
and only the first element will be used
```

```
[1] 386 403 432 433 438 447 448 689 691
```

```r
sapply(c("arts", "dance"), grep, x = powers)
```

```
$arts
[1] 386 403 432 433 438 447 448 689 691

$dance
[1] 367 369 374 376 379 382 383
```

The `%in%` operator can be used to create a logical vector whether the left hand side exists in the right hand side. Order matters, as it is vectorized - it returns a logical vector of the same length as the left hand side. It does not do partial matching.

```r
"secret arts" %in% powers
```

```
[1] TRUE
```

```r
head(powers %in% "secret arts")
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```r
table(powers %in% "secret arts")
```

```

FALSE  TRUE 
  718     1 
```

Find and replace

The `sub` (find and replace one time) and `gsub` (find and replace all occurences) operate similar to `grep` - the object to be operated on is the third argument.

```r
x <- "abc123abc"

# Replace first instance
sub("abc", "xyz", x)
```

```
[1] "xyz123abc"
```

```r
#replace all instance
gsub("abc", "xyz", x)
```

```
[1] "xyz123xyz"
```

You can also replace by position in a string using `substr`.

```r
substr(x, 1, 3) <- "XYZ"
x
```

```
[1] "XYZ123abc"
```

splitting strings

The function `strsplit` can be used to split a string into pieces based on a pattern. The example below finds all two-word super powers. Note that `strsplit` produces a `list`:

```r
power_list <- strsplit(powers, " ")
tail(power_list)
```

```
[[1]]
[1] "omnipresence"

[[2]]
[1] "omniscience"

[[3]]
[1] "perfection"

[[4]]
[1] "singularity"

[[5]]
[1] "totality"   "embodiment"

[[6]]
[1] "unity"
```

```r
two_words <- which(sapply(power_list, length) == 2)
power_two <- sapply(power_list[two_words], paste, collapse=" ")
head(power_two)
```

```
[1] "enhanced strength"    "enhanced speed"       "enhanced durability"
[4] "enhanced stamina"     "enhanced agility"     "enhanced flexibility"
```

```r
c(length(powers), length(power_two))
```

```
[1] 719 550
```

So 550 of the 719 total powers are two-worded.

Regular Expressions

Regular expressions ("regex" or "regexp") are a way to describe patterns in strings, often in an abstract way. There is a common regexp vocabulary though some details differ between implementations and standards. The basic idea is illustrated in the examples below using the power data.

```r
## find all powers with an "z" anywhere in the word
head(powers[grep("z", powers)])
```

```
[1] "zoolingualism"                  "puzzle mastery"
[3] "zither intuition"               "brazilian jiu-jitsu mastery"
[5] "zoological intuition"           "cryptozoology intuition"
```

```r
## find all powers starting with "z"
powers[grep("^z", powers)]
```

```
[1] "zoolingualism"        "zither intuition"     "zoological intuition"
```

```r
## find all powers ending with "z"
powers[grep("z$", powers)]
```

```
character(0)
```

```r
## find all powers starting with q or z
powers[grep("^[qz]", powers)]
```

```
[1] "zoolingualism"                  "quantum string manipulation"
[3] "zither intuition"               "qinggong mastery"
[5] "qigong mastery"                 "zoological intuition"
```

```r
## find all powers startingwith two consecutive vowels
powers[grep("^[aeiou]{2}", powers)]
```

```
[1] "authority manipulation" "earth manipulation"     "air manipulation"
[4] "aesthetic activity"     "aikido mastery"         "autopotence"
```

In the examples above, we return all strings matching a simple pattern.

We can specify that the pattern be found at the beginning `^a` or end `a$` using *anchors*. We can provide multiple options for the match within brackets `[]`. We can negate options within brackets using `^` in a different context. The curly braces ask for a specific number (or range `{min, max}`) of matches.

In the example below we use `.` to match any (single) character. We can ask for multiple matches by appending:

- `*` if we want 0 or more matches
- `+` if we want at least 1 match.
- `?` if we want exactly 0 or 1 match.

```r
## find all powers with two consecutive vowels twice, separated by any
## other character
head(powers[grep("[aeiou]{2}.[aeiou]{2}", powers)])
```

```
[1] "definition intuition"          "desire intuition"
[3] "etiquette intuition"           "experience intuition"
[5] "forensic psychology intuition" "furtive intuition"
```

```r
## find all powers with two consecutive vowels twice, separated by any
## number of other charactes
head(powers[grep("[aeiou]{2}.*[aeiou]{2}", powers)])
```

```
[1] "biological manipulation"    "blood manipulation"
[3] "body function manipulation" "hair manipulation"
[5] "360-degree vision"          "authority manipulation"
```

```r
## find all powers with exactly two consecutive vowels followed by at
## least four consonants
powers[grep("[aeiou]{2}[^aeiou ]{3}[^aeiou ]+", powers)]
```

```
[1] "therianthropology intuition"
```

To match an actual period (or other meta-character) we need to escape with a backslash. Thus, we use the regular expression `\\.`

```r
c(powers, "umich.edu")[grep("\\.", c(powers, "umich.edu"))]
```

```
[1] "umich.edu"
```

The double backslash is needed because the regular expression itself is passed as a string and strings also use backslash as an escape character. This is also important to remember when building file paths as strings on a Windows computer. In other languages, you generally only need a single backslash in your regular expression.

Matched values can be grouped using parentheses `()` and referred back to in the order they appear using a back reference `\\1`.

```
## find all powers with a repeated letter
head(powers[grep("(.)\\1", powers)])
```

```
[1] "enhanced speed"     "blood manipulation" "cell manipulation"
[4] "immutability"       "supernatural blood" "360-degree vision"
```

```
## find all powers with a repeated letter but exclude double e or o
head(powers[grep("([^eo])\\1", powers)])
```

```
[1] "cell manipulation"    "immutability"         "enhanced smell"
[4] "hyper awareness"      "immortality"          "contaminant immunity"
```

```
## find all powers that end with a repeated letter
powers[grep("(.)\\1$", powers)]
```

```
[1] "enhanced smell"               "hyper awareness"
[3] "accelerated thought process" "indomitable will"
[5] "bullet hell"                  "enhanced cannon skill"
[7] "enhanced chakram skill"
```

"stringr" package

The "stringr" re-implements these functions, and adds new functionality, trying to standardize them. If you are going to be manipulating a lot of strings, its worth considering using "stringr" instead of base R.