Lecture 11. String Manipulation

R and Data Visualization

BIG2006, Hanyang University, Fall 2022

String Manipulation

It is true that R is a statistical language with numerical vectors and matrices playing a central role, but character strings are also quite important.

Ranging from birth dates stored in medical research data files to text-mining applications, character data arises quite frequently in R.

R has a number of string-manipulation utilities!

String-Manipulation Functions

grep()

- `grep(pattern, x)`: searches for a specified substring `pattern` in a vector `x` of strings
- If `x` contains n strings, `grep(pattern, x)` will return a vector of length up to n. Each element will be index in `x` at which a match of `pattern` as a substring of `x[i]` was found.

```
grep("Pole",c("Equator","North Pole","South Pole"))
```

```
## [1] 2 3
```

```
grep("pole",c("Equator","North Pole","South Pole"))
```

```
## integer(0)
```

nchar()

- `nchar(x)`: finds the length of string `x`

```
nchar("South Pole")
```

```
## [1] 10
```

```
nchar(NA)
```

```
## [1] NA
```

```
# nchar(factor("abc"))
```

Note: Here, 10 characters were found. There is no NULL character terminating R strings. Also, the result of `nchar()` will be unpredictable if `x` is not in character mode.

For more consistent results on nonstring object, use `stringr` package.

paste()

- `paste(...)`: concatenates several strings, returning the result in one long string
- The optional argument `sep` can be used to put something other than a space between the pieces being sliced together.

```
paste("North", "Pole")
```

```
## [1] "North Pole"
```

```
paste("North", "Pole", sep="")
```

```
## [1] "NorthPole"
```

```
paste("North", "Pole", sep=".")
```

```
## [1] "North.Pole"
```

```
paste("North", 1:3, "Pole", rep(0,3))
```

```
## [1] "North 1 Pole 0" "North 2 Pole 0" "North 3 Pole 0"
```

sprintf()

- `sprintf(...)`: assembles a string from parts in a formatted manner

```
i <- 8
s <- sprintf("the square of %d is %d", i, i^2)
s
```

```
## [1] "the square of 8 is 64"
```

Note: %d (decimal) means in the base-10 number system.

If you need a decimal point in the result, then use %f or %g (The %g format eliminates the superfluous zeros).

substr()

- `substr(x,start,stop)`: returns the substring in the given character position range `start:stop` in the given string `x`

```
substring("Hanyang University", 3, 7)
```

```
## [1] "nyang"
```

strsplit()

- `strsplit(x,split)`: splits a string `x` into an R list of substrings based on another string `split` in `x`

```
strsplit("10-13-2022", split = "-")
```

```
## [[1]]
## [1] "10"   "13"   "2022"
```

regexpr()

- `regexpr(pattern,text)`: finds the character position of the first instance of `pattern` within `text`

```
regexpr("nyang","Hanyang")
```

```
## [1] 3
## attr(,"match.length")
## [1] 5
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

gregexpr()

- `gregexpr(pattern,text)`: is the same as `regexpr()`, but it finds all instances of `pattern`

```
gregexpr("data","Big data science deals with Big data")
```

```
## [[1]]
## [1]  5 33
## attr(,"match.length")
## [1] 4 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

Note: This finds that "data" appears twice in the text, starting at character positions 5 and 33.

Regular Expressions

You must pay attention when using the string functions `grep()`, `grepl()`, `regexpr()`, `gregexpr()`, `sub()`, `gsub()`, and `strsplit()`.

A regular expression is a kind of wild card. It is shorthand to specify broad classes of strings.

Some examples of the `regular expression`

- `"[au]"`: refers to any string that contains either of the letters a or u
- A period `.`: represents any single character

```
grep("[au]",c("Equator","North Pole","South Pole"))
```

```
## [1] 1 3
```

```
grep("o.e",c("Equator","North Pole","South Pole"))
```

```
## [1] 2 3
```

```
grep("N..t",c("Equator","North Pole","South Pole"))
```

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

Some examples of the `metacharacter`

- A metacharacter is a character that is not to be taken literally
- If a period (`.`) appears in the first argument of `grep()`, it does not actually mean a period; it means any character.

```
grep(".",c("Hanyang","University","Hanyang.Uni"))
```

```
## [1] 1 2 3
```

```
grep("\\.",c("Hanyang","University","Hanyang.Uni"))
```

```
## [1] 3
```

Note: The first call failed because periods are metacharacters. You need to **escape** the metacharacter nature of the period, which is done via backslash.

Example: Testing a Filename for a Given Suffix

- We want to test for a specified suffix in a filename. For example, find all HTML files (those with suffix `.html`, `.htm`, and so on).

```
testsuffix <- function(fn,suff) {
  parts <- strsplit(fn,".",fixed=TRUE)
  nparts <- length(parts[[1]])
  return(parts[[1]][nparts] == suff)
}
testsuffix("Hanyang.Univ","Univ")
```

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

```
testsuffix("Korea.Hanyang.Univ","Uni")
```

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

```r
# Another function with a good illustration
testsuffix <- function(fn,suff) {
  ncf <- nchar(fn) # nchar() gives the string length
  # determine where the period would start
  # if suff is the suffix in fn
  dotpos <- ncf - nchar(suff) + 1
  # now check that suff is there
  return(substr(fn,dotpos,ncf)==suff)
}
testsuffix("Hanyang.Univ","Univ")
```

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

```r
testsuffix("Korea.Hanyang.Univ","Uni")
```

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

Example: Forming Filenames

- We want to create five files, `q1.pdf` through `q5.pdf`, consisting of histograms of 100 random $N(0, i^2)$ variates.

```
for(i in 1:5){
  fname <- paste("q",i,".pdf",sep="")
  #fname <- paste("q%d.pdf",i)
  pdf(fname)
  hist(rnorm(100,sd=i))
  dev.off()
}
```

Use of String Utilities in Debugging a Program

Programmers often find that they spend more time debugging a program than actually writing it.

Good debugging skills are invaluable and there are useful debugging tool in R.

You refer to Sections 11.3 and 13 of "The Art of R Programming" for more details.

- Fixing a buggy program is a process of confirming, one by one, that the many things you believe to be true and the code actually are true.
- When you find that one of your assumption is not true, you have found a clue to the location (if not the exact nature) of a bug.

```
x <- y^2 + 3*g(z,2)
w <- 28
if (w+q > 0) u <- 1 else v <- 10
```

Questions: Do you think the value of your variable `x` should be 3 after `x` is assigned? Do you think the `else` will be executed, not the `if` on that third line?

Simple approach: You can answer the previous questions by temporarily inserting `print` statements into the code and returning the program to see what printed out.

```
x <- y^2 + 3*g(z,2)
cat("x =",x,"\n")
w <- 28
if (w+q > 0) {
  u <- 1
  print("the 'if' was done")
} else {
  v <- 10
  print("the 'else' was done")
}
```

Note: We would return the program and inspect the feedback printed out. Then we remove the `print` statements and put in new ones to track down the next bug.

Reference

- Matloff, N. The Art of R Programming: A Tour of Statistical Software Design. No Starch Press. Chapter 11.