string manipulation in R

There is a substantial amount of **string manipulation** functions in R that are at the disposal of a programmer. The following discusses only a few of these functions and will additionally omit many optional arguments.

The **grep( )** function searches for a specified substring pattern in a vector of **x** strings. If **x** has **n** elements—contains **n** strings—the **grep( )** function will return a vector of length up to **n**. Each vector element is an index of **x** where the substring pattern of **x[i]** is found.

```
> grep(pattern, x, ...)                                    #function syntax
> grep("Well", c("Well Bore", "Well Head", "BOP", "Valve"))
[1] 1 2

> grep("well", c("Well Bore", "Well Head", "BOP", "Valve"))
integer(0)
```

Note the **pattern** argument in the above example is case-sensitive.

The **nchar( )** function returns the length of string **x** as follows:

```
nchar(x, type = "chars", allowNA = FALSE, keepNA = NA)    #function syntax
> nchar("Well Head")
[1] 9
```

The **nchar( )** function exhibited difficulties dealing with **NA** values and non-character classes; note that the above function accommodates **NA** values and passing a noncharacter class results in a return error.

The **paste( )** function concatenates multiple strings, returning a single string:

```
> paste("Well", "Head")
[1] "Well Head"

> paste("Well", "Head", sep = "")
[1] "WellHead"

> paste("Well", "Head", sep = ".")
[1] "Well.Head"

> paste("Production", "Well", "Head")
[1] "Production Well Head"
```

The **sep =** argument in the **paste( )** can be used to specify a character between the strings.

The **sprintf( )** function 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"
```

Concretely, the function name is intended to indicate the ***string print*** to a **string**, opposed to the screen. The function states to print the string **"the square of"** follows by the decimal value (base 10) of **i**.

The **substr( )** function returns the substring in a given character position range **stop:stop** as passed through.

```
> substring("Well Head", 6, 9)
[1] "Head"
```

The **strsplit( )** function splits a string **x** into an R list of substrings based on *another string split* in **x**.

```
> strsplit("11-01-2016", split = "-")
[[1]]
[1] "11"   "01"   "2016"
```

The **regexpr( )** finds the character position of the first instance of the specified **pattern** within the **text**.

```
> regexpr("Head", "Well Head")
[1] 6
attr(,"match.length")
[1] 4
attr(,"useBytes")
[1] TRUE
```

The above illustration reports that **"Head"** appeared in **"Well Head"**, beginning at character position **6**.

The **gregexpr( )** function is synonymous to the **regexpr( )** with the addition of finding ***all*** string instances.

```
> gregexpr("ing","Cementing and Casing")
[[1]]
[1]  7 18
attr(,"match.length")
[1] 3 3
attr(,"useBytes")
[1] TRUE
```

The above illustration finds **"ing"** twice in **"Cementing and Casing"**, starting at character position 7 and 18.

regular expressions [T] in R

programming languages occasionally reference ***regular expression*** in the context of **string manipulation**. The latter becomes relevant when using any of the below string functions in R:

The following functions should be carefully used with regular expressions in R:

`grep()` `grepl()` `regexpr()` `gregexpr()` `sub()` `gsub()` `strsplit()`

A ***regular expression*** is shorthand to specify broad classes of strings. For example, the expression **"[We]"** refers to any string the contains ***either*** of the letters **W** or **e**; illustrated as follows:

```
> grep("[We]", c("Production", "Well", "Head"))
[1] 2 3
```

The above illustration indicates the vector elements **2** and **3** contain either a **"W"** or a **"e"** (case sensitive).

```
> grep("W.l", c("Production", "Well", "Head"))
[1] 2

> grep("w.l", c("Production", "Well", "Head"))              #case sensitive!
integer(0)
```

The above illustration indicates the vector element **2** contains a **"W"** followed by any character before an **"l"**.

```
> grep("P..d", c("Production", "Well", "Head"))
[1] 1
```

The above illustration indicates the vector element **1** contains a **"P"** followed by any 2-characters before **"d"**.

In the case where an actual period **"."** is to be searched, the following displays the importance of syntax:

```
> grep(".", c("Prod.", "Well", "Hd."))
[1] 1 2 3

> grep("\\.", c("Prod.", "Well", "Hd."))
[1] 1 3
```

Note that specifying a period **"."** Chooses elements **1**, **2**, and **3**; periods are ***metacharacters***. Consequentially, the ***metacharacter*** nature of the period must be ***escaped*** prior to calling the function. The latter is achieved by using a backslash **"\"**; note, **two** are required due to the first backslash needing escape.

string utilities in the edtdbg debugging tool in R

The code within the **edtdbg** debugging tool (discussed later) utilizes string utilities. The **dgbsendeditcmd( )** functions is an illustration of the latter:

```
> dgbsendeditcmd                                    #send command to editor
function(cmd) {
    syscmd <- paste("vim --remote-send ", cmd, " iiservername ", vimserver, sep = "")
}
```

The main concept of the above illustration is that the **edtdbg** sends remote commands to the **Vim** text editor. For example, the following instructs **Vim** on server **168** to move the cursor to line **12** (typed into a terminal *shell* window):

```
vim --remote-send 12G --servername 168
```

Understanding the above illustration, it is shown how the **dgbsendeditcmd( )** function incorporates R programming to deliver the same results:

```
> paste("vim --remote-send ",cmd," --servername ",vimserver,sep="")
vim --remote-send 12G --servername 168
```

An additional core element in the operation of **edtdbg** is an arrangement to record the file ***dbgsink*** containing most R debugger data through a call to the **sink( )** function. The information captured includes the line numbers and cursor position in the source file as it is stepped through with R's debugger:

```
debug at cities.r#16: {
```

The following illustration continues to discuss other core functions of the **editdbg** tool in R that extensively demonstrates the usefulness of many R ***string manipulation*** functions. Details on the below code snippets can be found on pages 258-259 of ***The Art of R Programming*** by Norman Matloff:

```
> linenumstart <- regexpr("#",debugline) + 1
> buffname <- substr(debugline,10,linenumstart-2)
> colon <- regexpr(":",debugline)
> linenum <- substr(debugline,linenumstart,colon-1)
```

```
> substr(debugline,10,linenumstart-2)
```

```
> kbdin <- readline(prompt="enter number(s) of fns you wish to toggle dbg: ")
enter number(s) of fns you wish to toggle dbg:
```

```
1 4 5
```

```
> tognums <- as.integer(strsplit(kbdin,split=" ")[[1]])
```