---
title: "Manipulating Strings"
author: "M. Affouf"
date: "February 26, 2018"
output: html_document

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

Strings (character data) often need to be constructed or deconstructed to identify observations, preprocess text, combine information or satisfy any number of other needs. R offers functions for building strings, like paste and sprintf. It also provides a number of functions for using regular expressions and examining text data, although for those purposes it is better to use Hadley Wickham's stringr package.

##paste
The first function new R users reach for when putting together strings is paste. This function takes a series of strings, or expressions that evaluate to strings, and puts them together into one string. We start by putting together three simple strings.
```{r}
paste("Hello", "Jancy", "and others")
```

Notice that spaces were put between the strings. This is because paste has a third argument, sep, that determines what to put in between entries. This can be any valid text, including empty text ("").
```{r}
paste("Hello", "Jancy", "and others", sep = "/")
```

 Like many functions in R, paste is vectorized. This means each element can be a vector of data to be put together.
```{r}
paste(c("Hello", "Hey", "Howdy"), c("Jancy", "Maribel", "Yao"))
```

 In this case each vector had the same number of entries so they paired one-to-one. When the vectors do not have the same length they are recycled.
```{r}
paste("Hello", c("Jancy", "Maribel", "Yao"))
```

```{r}
paste("Hello", c("Jancy", "Maribel", "Yao"), c("Goodbye", "Seeya"))
```

paste also has the ability to collapse a vector of text into one vector containing all the elements with any arbitrary separator, using the collapse argument.

```{r}
vectorOfText <- c("Hello", "Everyone", "out there", ".")

paste(vectorOfText, collapse = " ")

paste(vectorOfText, collapse = "*")
```

##sprintf

While paste is convenient for putting together short bits of text, it can become unwieldy when piecing together long pieces of text, such as when inserting a number of variables into a long piece of text. For instance, we might have a lengthy sentence that has a few spots that require the insertion of special variables. An example is "Hello Jared, your party of eight will be seated in 25 minutes" where "Jared," "eight" and "25" could be replaced with other information.
Reforming this with paste can make reading the line in code difficult. To start, we make some variables to hold the information.
```{r}
person <- "Jancy"
partySize <- "eight"
waitTime <- 25
```

Now we build the paste expression.
```{r}
paste("Hello ", person, ", your party of ", partySize,
" will be seated in ", waitTime, " minutes.", sep="")
```

Making even a small change to this sentence would require putting the commas in just the right places.
A good alternative is the sprintf function. With this function we build one long string with special markers indicating where to insert values.
```{r}
sprintf("Hello %s, your party of %s will be seated in %s minutes",
person, partySize, waitTime)
```

Here, each %s was replaced with its corresponding variable. While the long sentence is easier to read in code, we must maintain the order of %s's and variables.
sprintf is also vectorized. Note that the vector lengths must be multiples of each other.
```{r}
sprintf("Hello %s, your party of %s will be seated in %s minutes",
c("Jancy", "Fatima"), c("eight", 16, "four", 10), waitTime)
```

##Extracting Text

Often text needs to be ripped apart to be made useful, and while R has a number of functions for doing so, the stringr package is much easier to use.
First we need some data, so we use the XML package to download a table of United States presidents from Wikipedia.
```{r}
require(XML)
```

 Then we use readHTMLTable to parse the table.
```{r}
#load("presidents.rdata")
theURL <- "http://www.loc.gov/rr/print/list/057_chron.html"
presidents <- readHTMLTable(theURL, which=3, as.data.frame=TRUE,
skip.rows=1, header=TRUE,       stringsAsFactors=FALSE)
```

Now we take a look at the data.
```{r}
head(presidents)
```

Examining it more closely, we see that the last few rows contain information we do not want, so we keep only the first 64 rows.
```{r}
tail(presidents$YEAR)
```

```{r}
presidents <- presidents[1:64, ]
```

To start, we create two new columns, one for the beginning of the term and one for the end of the term. To do this we need to split the Year column on the hyphen (-). The stringr package has the str split function that splits a string based on some value. It returns a list with an element for each element of the input vector. Each of these elements has as many elements as necessary for the split, in this case either two (a start and stop year) or one (when the president served less than one year).
```{r}
require(stringr)
# split the string
yearList <- str_split(string = presidents$YEAR, pattern = "-")
head(yearList)
```

```{r}
```

```
# combine them into one matrix
yearMatrix <- data.frame(Reduce(rbind, yearList))
head(yearMatrix)

# give the columns good names
names(yearMatrix) <- c("Start", "Stop")
# bind the new columns onto the data.frame
presidents <- cbind(presidents, yearMatrix)

# convert the start and stop columns into numeric
presidents$Start <- as.numeric(as.character(presidents$Start))

presidents$Stop <- as.numeric(as.character(presidents$Stop))

# view the changes

head(presidents)

tail(presidents)
```

In the preceding example there was a quirk of R that can be frustrating at first pass. In order to convert the factor presidents$Start into a numeric, we first had to convert it into a character. That is because factors are simply labels on top of integers. So when applying as.numeric to a factor, it is converted to the underlying integers.
Just like in Excel, it is possible to select specified characters from text using str sub.

```{r}
# get the first 3 characters

str_sub(string = presidents$PRESIDENT, start = 1, end = 3)

# get the 4th through 8th characters
str_sub(string = presidents$PRESIDENT, start = 4, end = 8)

```

This is good for finding a president whose term started in a year ending in 1, which means he got elected in a year ending in 0, a preponderance of which ones died in office.

```{r}
presidents[str_sub(string = presidents$Start, start = 4,end = 4) == 1,
c("YEAR", "PRESIDENT", "Start", "Stop")]
```

##Regular Expressions

Sifting through text often requires searching for patterns, and usually these patterns have to be general and flexible. This is where regular expressions are very useful. We will not make an exhaustive lesson of regular expressions but will illustrate how to use them within R.

Let's say we want to find any president with "John" in his name, either first or last. Since we do not know where in the name "John" would occur, we cannot simply use str sub. Instead we use str detect.

```{r}
# returns TRUE/FALSE if John was found in the name

johnPos <- str_detect(string = presidents$PRESIDENT, pattern = "John")

presidents[johnPos, c("YEAR", "PRESIDENT", "Start", "Stop")]

```

This found John Adams, John Quincy Adams, John Tyler, Andrew Johnson, John F. Kennedy and Lyndon B. Johnson. Note that regular expressions are case sensitive, so to ignore case we have to put the pattern in ignore.case.
```{r}
badSearch <- str_detect(presidents$PRESIDENT, "john")

goodSearch <- str_detect(presidents$PRESIDENT, ignore.case("John"))

sum(badSearch)
sum(goodSearch)
```

To show off some more interesting regular expressions we will make use of yet another table from Wikipedia, the list of United States wars. Because we only care about one column, which has some encoding issues, we put an Rdata file of just that one column at http://www.jaredlander.com/data/warTimes.rdata. We load that file using load and we then see a new object in our session named warTimes.
For some odd reason, loading rdata files from a URL is not as straightforward as reading in a CSV file from a URL. A connection must first be made using url, then that connection is loaded with load, and then the connection must be closed with close.

```{r}
con <- url("http://www.jaredlander.com/data/warTimes.rdata")

load(con)

close(con)
```

This vector holds the starting and stopping dates of the wars. Sometimes it has just years, sometimes it also includes months and possibly days. There are instances where it has only one year. Because of this, it is a good dataset to comb through with various text functions. The first few entries follow.

```{r}
head(warTimes, 10)
```

We want to create a new column that contains information for the start of

the war. To get at this information we need to split the Time column. Thanks to Wikipedia's encoding, the separator is generally "ACAEA," which was originally "A ~$A^ CA^'" and converted to these characters to make life easier. There are two instances where the "-" appears, once as a separator and once to make a hyphenated word. This is seen in the following code.

```{r}
warTimes[str_detect(string = warTimes, pattern = "-")]
```

So when we are splitting our string, we need to search for either "ACAEA" or "-." In
str split the pattern argument can take a regular expression. In this case it will be "(ACAEA)|-," which tells the engine to search for either "(ACAEA)" or (denoted by the vertical pipe) "-" in the string. To avoid the instance, seen before, where the hyphen is used in "mid-July" we set the argument n to 2 so it returns at most only two pieces for each element of the input vector. The parentheses are not matched but rather act to group the characters "ACAEA" in the search.1 This grouping capability will prove important for advanced replacement of text, which will be demonstrated later in this section.

```{r}
theTimes <- str_split(string = warTimes, pattern = "(ACAEA)|-", n = 2)
head(theTimes)
```

Seeing that this worked for the first few entries, we also check on the two instances where a hyphen was the separator.

```{r}
which(str_detect(string = warTimes, pattern = "-"))

theTimes[[147]]

theTimes[[150]]
```

This looks correct, as the first entry shows "mid-July" still intact while the second entry shows the two dates split apart.

For our purposes we only care about the start date of the wars, so we need to build a function that extracts the first (in some cases only) element of each vector in the list.

```{r}
theStart <- sapply(theTimes, FUN = function(x) x[1])

head(theStart)
```

The original text sometimes had spaces around the separators and sometimes did not, meaning that some of our text has trailing white spaces. The easiest way to get rid of them is with the str trim function.

```{r}
theStart <- str_trim(theStart)
```

```
head(theStart)
```

To extract the word "January" wherever it might occur, use str extract. In places where it is not found will be NA.

```{r}
# pull out 'January' anywhere it's found, otherwise return NA

str_extract(string = theStart, pattern = "January")
```

To find elements that contain "January" and return the entire entry-not just "January"-use str detect and subset theStart with the results.

```{r}
# just return elements where 'January' was detected

theStart[str_detect(string = theStart, pattern = "January")]
```

To extract the year, we search for an occurrence of four numbers together. Because we do not know specific numbers, we have to use a pattern. In a regular expression search, "[0-9]" searches for any number. We use "[0-9][0-9][0-9][0-9]" to search for four consecutive numbers.

```{r}
# get incidents of 4 numeric digits in a row

head(str_extract(string = theStart, "[0-9][0-9][0-9][0-9]"), 20)
```

Writing "[0-9]" repeatedly is inefficient, especially when searching for many occurences of a number. Putting "4" in curly braces after "[0-9]" causes the engine to search for any set of four numbers.

```{r}
# a smarter way to search for four numbers

head(str_extract(string = theStart, "[0-9]{4}"), 20)
```

Even writing "[0-9]" can be inefficient, so there is a shortcut to denote any integer. In most other languages the shortcut is "\d" but in R there needs to be two backslashes ("\\d").

```{r}
# "\\d" is a shortcut for "[0-9]"

head(str_extract(string = theStart, "\\d{4}"), 20)
```

The curly braces offer even more functionality: for instance, searching for a number one to three times.

```{r}
# this looks for any digit that occurs either once, twice or thrice
str_extract(string = theStart, "\\d{1,3}")
```

Regular expressions can search for text with anchors indicating the beginning of a line ("^") and the end of a line ("$").

```{r}
# extract 4 digits at the beginning of the text

head(str_extract(string = theStart, pattern = "^\\d{4}"), 30)

# extract 4 digits at the end of the text
head(str_extract(string = theStart, pattern = "\\d{4}$"), 30)
```

```{r}
# extract 4 digits at the beginning AND the end of the text

head(str_extract(string = theStart, pattern = "^\\d{4}$"), 30)
```

Replacing text selectively is another powerful feature of regular expressions. We start by simply replacing numbers with a fixed value.

```{r}
# replace the first digit seen with "x"

head(str_replace(string=theStart, pattern="\\d", replacement="x"), 30)
```

```{r}
# replace all digits seen with "x"
# this means "7" -> "x" and "382" -> "xxx"

head(str_replace_all(string=theStart, pattern="\\d", replacement="x"), +
30)
```

```{r}
# replace any strings of digits from 1 to 4 in length with "x"
# this means "7" -> "x" and "382" -> "x"

head(str_replace_all(string=theStart, pattern="\\d{1,4}",
replacement="x"), 30)
```

Not only can regular expressions substitute fixed values into a string, they can also substitute part of the search pattern. To see this, we create a vector of some HTML commands.

```{r}
# create a vector of HTML commands
```

```
commands <- c("<a href=index.html>The Link is here</a>", "<b>This is bold text</b>")
```

Now we would like to extract the text between the HTML tags. The pattern is a set of opening and closing angle brackets with something in between ("<.+?>"), some text (".+?") and another set of opening and closing brackets ("<.+?>"). The "." indicates a search for anything, while the "+" means to search for it one or more times with the "?" meaning it is not a greedy search. Because we do not know what the text between the tags will be, and that is what we want to substitute back into the text, we group it inside parentheses and use a back reference to reinsert it using "\\1," which indicates use of the first grouping. Subsequent groupings are referenced using subsequent numerals, up to nine. In other languages a "$" is used instead of "\\."

```{r}
# get the text between the HTML tags
# the content in (.+?) is substituted using 1
str_replace(string=commands, pattern="<.+?>(.+?)<.+>", replacement="\\1")
```