V05 – stringr

3. Mai 2021

Contents

1 Intro

Zur Manipulation von Strings dient das Paket `stringr`. Es ist Teil der Paketsammlung `tidyverse`.

```
library(stringr)
# oder
# library(tidyverse) # lädt stringr und ein paar weitere Pakete
```

Funktionen in `stringr` zur Manipulation von Strings beginnen mit `str_`.

2 Basics

Einige Funktionen aus Base-R haben ein Äquivalent in `stringr`.

```
# Base-R: nchar()
str_length(c("a", "abs", NA, ""))
## [1]  1  3 NA  0
```

```
# Base-R: paste0()
str_c("Apfel", "saft", "schorle", 3, TRUE, pi)
## [1] "Apfelsaftschorle3TRUE3.14159265358979"
str_c("Apfel", "saft", "schorle", 3, TRUE, pi, sep = ", ")
## [1] "Apfel, saft, schorle, 3, TRUE, 3.14159265358979"
```

`str_c()` ist vektorisiert, dh die Funktion wirkt auf Vektoren elementweise. Ggf findet Recycling statt.

```
str_c(1:3, LETTERS[1:3])
## [1] "1A" "2B" "3C"
str_c("prefix-", LETTERS[1:3], "-suffix")
## [1] "prefix-A-suffix" "prefix-B-suffix" "prefix-C-suffix"
str_c("prefix-", LETTERS[1:3], "-suffix", collapse = " * ")
```

```
## [1] "prefix-A-suffix * prefix-B-suffix * prefix-C-suffix"
str_c(1:3, LETTERS[1:3], T, collapse = " * ", sep=",")
## [1] "1,A,TRUE * 2,B,TRUE * 3,C,TRUE"
```

Zum Subsetting der Symbole eines Strings benötigt es eigene Funktionen, da `[[` und `[` schon eine andere Funktion haben (`character`-Vektoren subsetten). `stringr` bietet hierzu die Funktion `str_sub()`.

Die Symbole eines Strings `s` haben analog zu Vektoren die Indizes `1:str_length(s)`.

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, start=1, end=3)
## [1] "App" "Ban" "Pea"
str_sub(x, 1, 3)
## [1] "App" "Ban" "Pea"
str_sub(x, 3) # only start set => subset until end of string
## [1] "ple"  "nana" "ar"
str_sub(x, end=3) # only end set => subset from beginning of string
## [1] "App" "Ban" "Pea"
str_sub(x, -3, -1) # negativ numbers: count from end of string
## [1] "ple" "ana" "ear"
str_sub("ab", 1, 5) # values > str_length() act same as = str_length()
## [1] "ab"
```

`str_subs()` ist in jedem Argument vektorisiert. Ggf geschieht Recycling.

```
str_sub(x, 1:3, 3:5) # start and end are also vectorized
## [1] "App" "ana" "ar"
str_sub("123456", c(1,3), 6:3) # all arguments are repeated to match longest argument
## [1] "123456" "345"    "1234"   "3"
```

Analog zu `[<-` kann mit `str_sub<-` Subsetting mit Zuweisung durchgeführt werden.

```
str_sub(x, 1, 1) <- "_"
x
## [1] "_pple"  "_anana" "_ear"
str_sub(x, 1:3, 4) <- c("XX", "YY", "ZZ") # vectorized assignment with non-matching string lengths
x
## [1] "XXe"   "_YYna" "_eZZ"
```

2.1 Weiteres

```
# Base-R: trimws()
str_trim(c(" a  a", "b   b  ", " c    c  ")) # remove whitespace from beginning and end
## [1] "a  a"   "b   b"  "c    c"
str_squish(c(" a  a", "b   b  ", " c    c  ")) # also reduce repeated whitespce inside
## [1] "a a" "b b" "c c"
```

```
# Base-R: tolower(), toupper()
x <- c("Apple", "BANANA!", "pear")
str_to_upper(x)
## [1] "APPLE"   "BANANA!" "PEAR"
str_to_lower(x)
## [1] "apple"   "banana!" "pear"
str_to_title(x)
## [1] "Apple"   "Banana!" "Pear"
```

Was `rep(,times=)` für Vektoren ist, ist `str_dup()` für Strings.

```
str_dup(c("a", "bB"), 2)
## [1] "aa"   "bBbB"
str_dup(c("a", "bB"), 2:3)
## [1] "aa"     "bBbBbB"
```

3 Formatieren

Um Werte in Strings zu schreiben, setzt `stringr` auf das Paket `glue`, https://glue.tidyverse.org/, welches auch Teil des Tidyverse ist, aber nicht extra geladen werden muss (wenn `stringr` geladen ist).

Teile eines Strings, die von geschweiften Klammern `{ }` umschlossen sind, werden mittels `glue` als Ausdrücke in R ausgewertet.

```
x <- sample.int(100, 2)
str_glue("3 * 4 = {3*4}")
## 3 * 4 = 12
str_glue("Die Summe von {x[1]} und {x[2]} ist {sum(x)}.")
## Die Summe von 45 und 23 ist 68.
str_glue("Der glue-Text {'{'}'Wort'{'}'} erzeugt: {'Wort'}")
## Der glue-Text {'Wort'} erzeugt: Wort
```

Der Rückgabewert von `str_glue()` ist ein `character`-Vektor der S3-Klasse `glue`. Dies sorgt für eine etwas andere Ausgabe auf der Konsole.

```
word <- "blub"
word_glue <- str_glue(word)
word == word_glue
## [1] TRUE
identical(word, word_glue)
## [1] FALSE
word
## [1] "blub"
word_glue
## blub
typeof(word_glue)
## [1] "character"
attributes(word_glue)
## $class
## [1] "glue"      "character"
unclass(word_glue)
## [1] "blub"
```

Um die Ausgabe einer Dezimalzahl zu formatieren, nutze die Base-R-Funktion `format()`, siehe `?format` zusammen mit `str_glue()` oder die Base-R-Funktion `sprintf()`.

```
sprintf("Pi ist %f.", pi)
## [1] "Pi ist 3.141593."
sprintf("Pi ist %.2f.", pi)
## [1] "Pi ist 3.14."
str_glue("Pi ist {pi}.")
## Pi ist 3.14159265358979.
format(pi, digits=3)
## [1] "3.14"
str_glue("Pi ist {format(pi, digits=3)}.")
```

```
## Pi ist 3.14.
```

`str_glue()` ist vektorisiert.

```
descr <- c('gerade', 'ungerade')
num <- sample(5)
str_glue("Die Zahl {num} ist {descr[num %% 2 + 1]}.")
## Die Zahl 4 ist gerade.
## Die Zahl 3 ist ungerade.
## Die Zahl 1 ist ungerade.
## Die Zahl 2 ist gerade.
## Die Zahl 5 ist ungerade.
```

`str_glue_data()` wird an erster Position ein Tibble übergeben, dessen Spaltennamen als Variablen in den Ausdrücken des Strings genutzt werden können.

```
library(tibble)
tb <- tibble(x=1:3, y=letters[1:3])
str_glue_data(tb, "Variable x ist {x} und y ist {y}.")
## Variable x ist 1 und y ist a.
## Variable x ist 2 und y ist b.
## Variable x ist 3 und y ist c.
```

4 Reguläre Ausdrücke

Mit regulären Ausdrücken (*regular expressions*, **RegEx**) wird in Strings nach Mustern gesucht.

Mit `str_view()` wird die erste Übereinstimmung des Musters im String angezeigt, mit `str_view_all()` alle (nicht überlappend).

```
x <- c("apple", "banana", "mango")
str_view(x, "an")
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, pleas
```

In regulären Ausdrücken haben einige Symbole spezielle Bedeutung. Zu diesen sogenannten **Metazeichen** zählen `[ ] ( ) { } | ? + - * ^ $ \ . : ! < =`.

ZB steht `|` für *oder* und `.` für ein beliebiges Zeichen. Die genaue Funktionsweise dieser und anderer Metazeichen wird weiter unten erklärt.

```
x <- c("apple", "banana", "mango", "n|m ch|go |an")
str_view_all(x, "n|m")
```

Sollen die Metazeichen als normale Symbole interpretiert werden, müssen sie mit `\` markiert werden (*escaping*). Dabei ist `\` selbst ein Sonderzeichen für Strings in R. Um `\` in einem String zu schreiben, muss `"\\"` geschrieben werden.

```
cat("I\\I\n")
## I\I
x <- "a.|.|.|b"
str_view(x, ".")
```

`\` ist sowohl für R-Strings als auch für RegEx ein Metazeichen und muss jeweils *escaped* werden, wenn nach dem Symbol `\` gesucht werden soll.

```
cat("I\\I\n")
## I\I
cat("I\\\\I\n")
```

```
## I\\I
x <- "a\\b"
cat(x, "\n")
## a\b
str_view(x, "\\")
## Error in stri_locate_first_regex(string, pattern, opts_regex = opts(pattern)): Unrecognized backslas
str_view(x, "\\\\")
```

R Version 4 bietet die Möglichkeit mit `r"(...)"` raw Strings zu schreiben, bei denen \ nicht als Metazeichen interpretiert wird.

```
x <- r"(a\b)"
cat(x, "\n")
## a\b
str_view(x, r"(\\)")
```

4.1 Funktionen zur Nutzung von RegEx

- `str_detect()`: Enthält der String das Muster?
- `str_subset()`: Gib alle Strings zurück, die das Muster enthalten.
- `str_which()`: Gib die Indizes der Strings zurück, die das Muster enthalten.
- `str_count()`: Wie oft passt das Muster (nicht-überlappenden) in den String?

```
x <- c("apple", "banana", "mango", "lime")
str_detect(x, ".n")
## [1] FALSE  TRUE  TRUE FALSE
str_subset(x, ".n")
## [1] "banana" "mango"
str_which(x, ".n")
## [1] 2 3
str_count(x, ".n")
## [1] 0 2 1 0
str_count(x, ".n.") # nicht überlappend
## [1] 0 1 1 0
```

- `str_extract()`: Gib die gefundenen Muster aus.
- `str_locate()`: Gib Start- und Endindex der gefundenen Muster aus.

Dazu gibt es die entsprechenden `_all`-Funktionen. *All* ist dabei immer nicht-überlappend gemeint.

```
x <- c("apple", "banana", "mango", "lime")
str_extract(x, "a.")
## [1] "ap" "an" "an" NA
str(str_extract_all(x, "a.") )
## List of 4
##  $ : chr "ap"
##  $ : chr [1:2] "an" "an"
##  $ : chr "an"
##  $ : chr(0)
str_locate(x, "a.")
##      start end
## [1,]     1   2
## [2,]     2   3
## [3,]     2   3
## [4,]    NA  NA
str_locate_all(x, "a.")
```

```
## [[1]]
##      start end
## [1,]     1   2
##
## [[2]]
##      start end
## [1,]     2   3
## [2,]     4   5
##
## [[3]]
##      start end
## [1,]     2   3
##
## [[4]]
##      start end
```

`str_split(str, pattern)` spaltet einen String `str` an allen Stellen, wo `pattern` gefunden wird.

```
x <- c("Kaffee, Schokolade, Kekse", "Orange, Melone")
str_split(x, ", ")
## [[1]]
## [1] "Kaffee"     "Schokolade" "Kekse"
##
## [[2]]
## [1] "Orange" "Melone"
x <- "asdf_7,jklö qwert"
str_split(x, "_|,| ") # split at _ and , and space
## [[1]]
## [1] "asdf"  "7"     "jklö"  "qwert"
```

Möchte man die Interpretation des Musters als RegEx abschalten, kann der String mit `fixed()` markiert werden.

```
str_split("ab.cde.f", ".")
## [[1]]
## [1] "" "" "" "" "" "" "" "" ""
str_split("ab.cde.f", fixed("."))
## [[1]]
## [1] "ab"  "cde" "f"
```

`fixed()` kann jedem `pattern`-Argument einer `stringr` Funktion angewendet werden.

```
x <- fixed("asdf")
class(x)
## [1] "fixed"     "pattern"   "character"
attr(x, "options")
## $case_insensitive
## [1] FALSE
x <- fixed("asdf", ignore_case = TRUE)
attr(x, "options")
## $case_insensitive
## [1] TRUE

str_view_all("xa.bxA.bxa.BxA.Bxaobx", fixed("a.b", ignore_case=T))
```

4.2 RegEx Syntax

Die Funktionsweise regulärer Ausdrücke mit `stringr` wird gut zusammengefasst auf Seite 2 des offiziellen Cheatsheets: https://github.com/rstudio/cheatsheets/raw/master/strings.pdf.

4.2.1 Match Character

MATCH CHARACTERS see <- function(rx) str_view_all("abc ABC 123\t.!?\\(){}\n", r

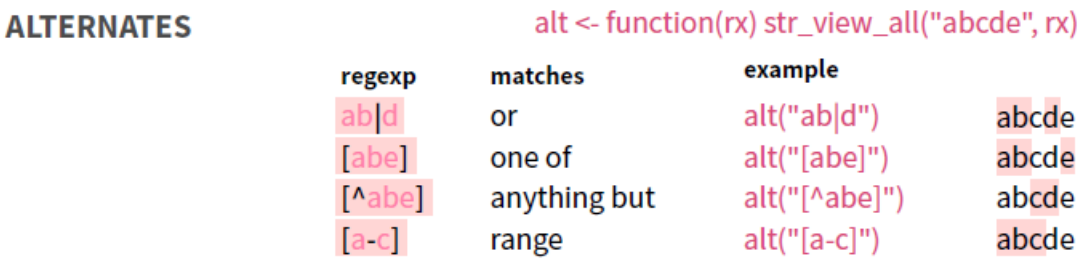
string (type this)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(
\\.	\.	.	see("\\.")	abc ABC 123 .!?\(
\\!	\!	!	see("\\!")	abc ABC 123 .!?\(
\\?	\?	?	see("\\?")	abc ABC 123 .!?\(
\\\\	\\	\	see("\\\\")	abc ABC 123 .!?\(
\\(	\(	(	see("\\(")	abc ABC 123 .!?\(
\\)	\)	)	see("\\)")	abc ABC 123 .!?\(
\\{	\{	{	see("\\{")	abc ABC 123 .!?\(
\\}	\}	}	see("\\}")	abc ABC 123 .!?\(
\\n	\n	new line (return)	see("\\n")	abc ABC 123 .!?\(
\\t	\t	tab	see("\\t")	abc ABC 123 .!?\(
\\s	\s	any whitespace (\S *for non-whitespaces*)	see("\\s")	abc ABC 123 .!?\(
\\d	\d	any digit (\D *for non-digits*)	see("\\d")	abc ABC 123 .!?\(
\\w	\w	any word character (\W *for non-word chars*)	see("\\w")	abc ABC 123 .!?\(
\\b	\b	word boundaries	see("\\b")	abc ABC 123 .!?\(
	[:digit:] [1]	digits	see("[:digit:]")	abc ABC 123 .!?\(
	[:alpha:] [1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(
	[:lower:] [1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(
	[:upper:] [1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(
	[:alnum:] [1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(
	[:punct:] [1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(
	[:graph:] [1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(
	[:space:] [1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(
	[:blank:] [1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(
	.	every character except a new line	see(".")	abc ABC 123 .!?\(

[1] Many base R functions require classes to be wrapped in a second set of [ ], e.g. [[:digit:]]

```
str_view_all("abc123ABC", "\\d\\D")
```

```
str_view_all("aÄá? ! :)", "[:alpha:]")
```

4.2.2 Alternates

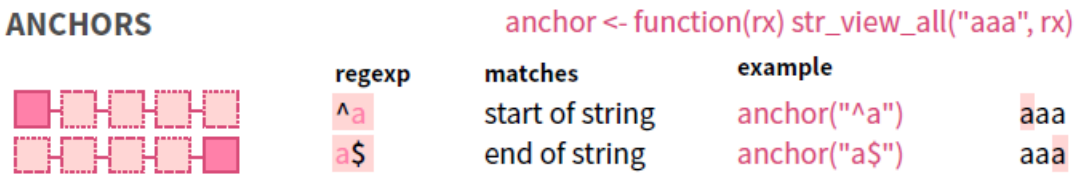



```
str_view_all(" a a b b", "a a|b ")
```

```
x <- c("apple", "banana", "mango", "lime")
str_view_all(x, "[aeiou][^aeiou][aeiou]") # Vokal, kein Vokal, Vokal
```

```
str_view_all("aÄá", "[a-zA-Z]")
```

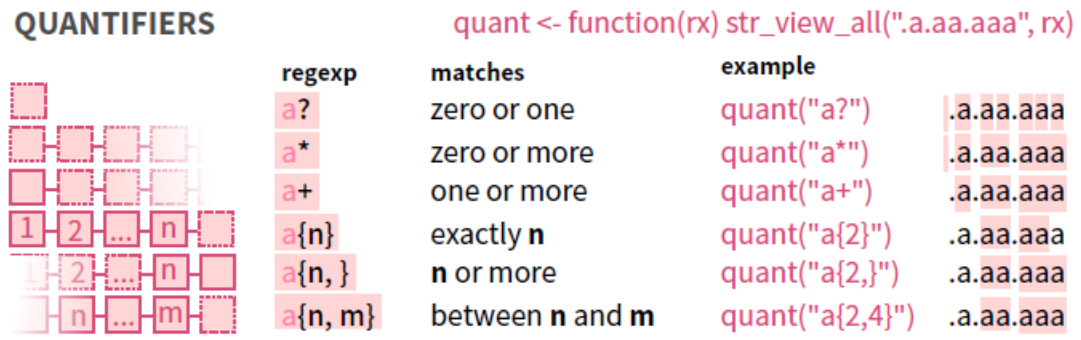
4.2.3 Anchors




```
str_view_all(c("aba", "bab ab"), "^ab")
```

Beachte: `^` hat innerhalb von `[, ]` eine andere Bedeutung als außerhalb.

4.2.4 Quantifiers



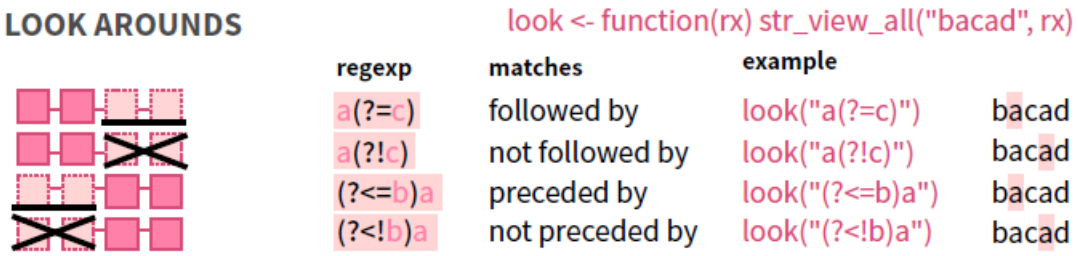
Quantoren geben an, wie oft der vorangegangene Ausdruck auftauchen muss.

```
x <- c("apple", "banana", "mango", "lime")
str_view(x, "p+")
```

Quantoren sind *gierig*, dh es wird ein Teil-String maximaler Länge ausgewählt. Wird einem Quantor ein `?` nachgestellt, ist der vorausgegangene Quantor *genügsam* (Teil-String minimaler Länge auswählen).

```
str_view(c("oABxxBxBo", "oAxBxBxBo"), "A.*B")
```

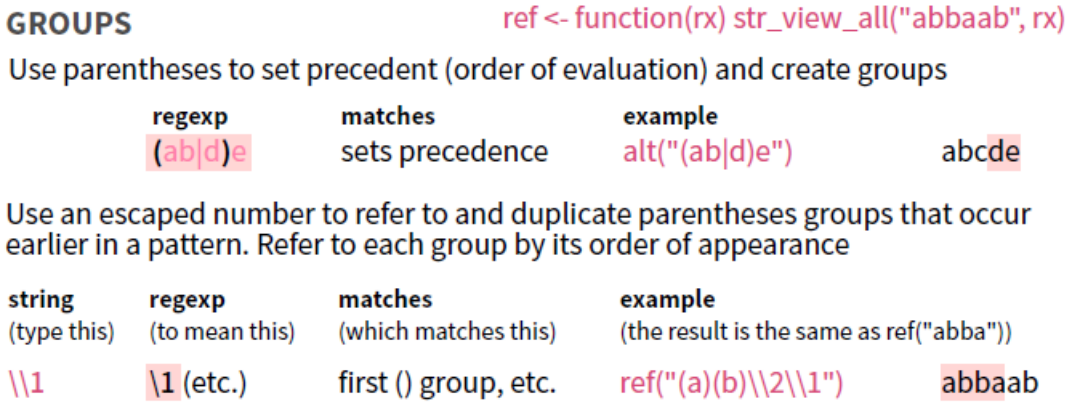
4.2.5 Look Arounds



Für **Look Arounds** müssen vor oder nach dem gesuchten Muster bestimmte Symbole oder Muster vorhanden sein. Diese sind aber nicht Teil des gefundenen Muster.

```
str_view(x, "[aeiou][^aeiou].*$")
```

4.2.6 Groups



Einzelne Bestandteile eines regulären Ausdrucks können mit `( )` zu einer Einheit **gruppiert** werden.

```
x <- c("apple", "banana", "mango", "lime")
str_view(x, "(an)+")
```

`str_match()` und `str_match_all()` geben neben dem String-Teil, der dem gesamten Muster entspricht, auch diejenigen Teile aus, die den einzelnen Gruppen entsprechen.

```
x <- c("apple", "banana", "mango", "lime")
str_match(x, "(a)(.*)([aeiou])")
##      [,1]    [,2] [,3]  [,4]
## [1,] "apple" "a"  "ppl" "e"
## [2,] "anana" "a"  "nan" "a"
## [3,] "ango"  "a"  "ng"  "o"
## [4,] NA      NA   NA    NA
```

Diese Mechanik wird beim Ersetzen mit RegEx genutzt. `str_replace()` und `str_replace_all()` ersetzen gefundene Muster mit neuen Werten. Diese Ersetzungen können mittels `\1`, `\2`, `\3`, ... auf die Werte der gefundenen Gruppen verweisen.

```
str_replace("banana", "a", "o")
## [1] "bonana"
str_replace_all("banana", "a", "o")
## [1] "bonono"
str_replace_all("banana", "a(.)", "a\\1\\1")
## [1] "bannanna"
```

```r
str_replace("1 * (2 + 3); (-:", "\\(([^\\)]*)\\)", "[\\1]")
## [1] "1 * [2 + 3]; (-:"
```

Als `replacement` kann `str_replace()` auch eine Funktionen übergeben werden, die den gefundenen Teil-String in seine Ersetzung umwandelt.

```r
colours <- str_c("\\b", colors(), "\\b", collapse="|")
col2hex <- function(col) {
  rgb <- col2rgb(col)
  rgb(rgb["red", ], rgb["green", ], rgb["blue", ], max = 255)
}

cat(str_sub(colours, end=50))
## \bwhite\b|\baliceblue\b|\bantiquewhite\b|\bantique
col2hex("orange")
## [1] "#FFA500"

x <- c(
  "Roses are red, violets are blue.",
  "My favorite color is green.")
str_replace_all(x, colours, col2hex)
## [1] "Roses are #FF0000, violets are #0000FF."
## [2] "My favorite color is #00FF00."
```

5 Weiteres

CheatSheet https://github.com/rstudio/cheatsheets/raw/master/strings.pdf

Auf der Seite https://regexr.com/ lassen sich RegEx ausprobieren. Leider werden nicht alle oben genannten Möglichkeiten unterstützt.

Viele Texteditoren und IDEs (zB RStudio) unterstützen RegEx für Such- und Ersetz-Funktion.

https://xkcd.com/208/

https://xkcd.com/1638/


\ BACKSLASH
\\ REAL BACKSLASH
\\\ *REAL* REAL BACKSLASH
\\\\ ACTUAL BACKSLASH, FOR REAL THIS TIME
\\\\\ ELDER BACKSLASH
\\\\\\ BACKSLASH WHICH ESCAPES THE SCREEN AND ENTERS YOUR BRAIN
\\\\\\\ BACKSLASH SO REAL IT TRANSCENDS TIME AND SPACE
\\\\\\\\ BACKSLASH TO END ALL OTHER TEXT
\\\\\\\\\\... THE TRUE NAME OF BA'AL, THE SOUL-EATER