Strings in R

Strings: What They Are

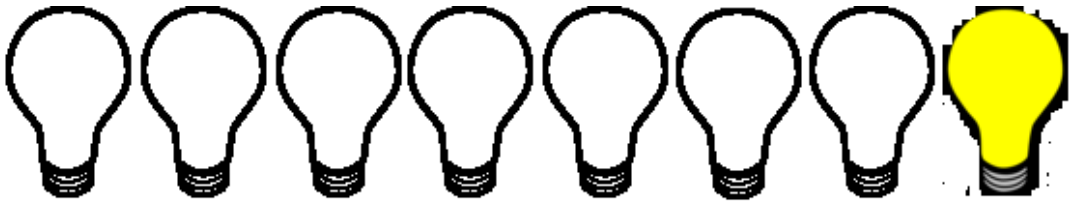
Strings, simply put, are a sequence of characters. Characters, in turn, are the symbols used in a natural language and they include letters, digits, punctuation as well as whitespace (regular spacing, indentations, returns, etc.).

In R, the term 'string' usually refers to a character vector of length 1.

Low-level background (optional reading)

In the modern computer's random-access memory (RAM), where program instructions are carried out, there are tiny addressable units. By addresses, we mean the smallest part of memory that is identified with a number. It is in this portion that data is created. This addressable unit is known as a **byte**.

The unit of information in modern microcomputers is the **bit** (short for "binary digit") and represents two values - 1 and 0, and each byte essentially consists of 8 bits, also known as an *octet*. Each of these bits is like a minuscule light that is either turned `ON` or `OFF`. The machine understands numbers only at the binary level. The eight bits are, by convention, numbered `0` to `7` and with the least-significant bit (LSB) on the far right and the most-significant bit (MSB) on the left.



00000001 = 1

11001000 = 200

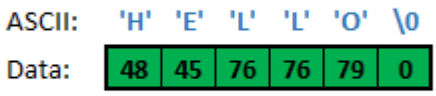
A character occupies 1 byte of data and this datum is created as a numerical value in base 2. Thus, for example the number 3 in base 2 is 11. At the level of a byte with 8 bits, this is represented as 00000011. Using the analogy of the light bulbs, this means that out of the 8 bulbs, the 2 on the far left are on while the rest are all off. This evaluates to 3 in base 2.

Each character has a numerical value to which it corresponds. This is what is known as encoding–the internal representation of characters. In the one of the common encoding systems used is the American Standard Code for Information Interchange (ASCII). The ASCII system caters for the language symbols in the English language only, and over time, more extended encoding systems have been developed and internationally accepted. The ASCII system, however is still very much relevant today.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

From the table, we see that the decimal number `32` is mapped to the letter `A`. At the machine-level, this number is in the binary form i.e. `00010000` (bit). So when we type in this letter as input this is what happens ultimately at the level of the processor. Then, contiguous portions of the memory, each with its own address, are then put together in an array to create a string. At the end of the string, the sequence is then closed of with a `NUL` character (`\0`). In high-level languages like R, we do not see this character at all, so we don't bother with it.



Although ASCII is considered the gold standard for character encoding, it is a legacy system. The default encoding for many of the text-related operations in R is **UTF-8**, which is part of the UNICODE standard as it allows for the use of characters outside the English lexicon. The work of the UNICODE Consortium has now broadened the consensus for language characters in use across the world, and some of these now allow for the use of more than one byte. ASCII system has only 127 characters, and the maximum number of possible characters that could fit into a single byte is $2^8 = 256$. The coming of 2-byte capabilities now makes

it possible to have a staggering $2^16 = 6.5536 \times 10^4$ characters!

Construction of strings

Making strings in R is simple and straightforward. Recalling that we have string literals, which are constants and identifiable by the R interpreter by quoting. All the operations that lead to the formation of the character sequence are carried out under the hood. However, to access the low-level values, one could use raw vectors (vectors whose elements are individual bytes)

```
greet <- "HELLO"
bytes <- charToRaw(greet)
bytes
```

```
## [1] 48 45 4c 4c 4f
```

The output of the above code can be compared with the ASCII lookup table. Note that the values are in hexadecimal (base 16). In this system the digits run from 0 to F, where A-F are the equivalent of 10-15 in the decimal. Hexadecimals are very useful in representing bytes because `F` is equal to 4 bits and `FF` is the same as `11111111` (256) when all the bits are turned on. So it is an effective shorthand for representing what is happening at the level of bytes. This knowledge is not only handy when it comes to dealing with strings, but also when dealing with colours and images (i.e. pixels). R gives us several functions to work with these.

```
as.integer(bytes)
```

```
## [1] 72 69 76 76 79
```

For day-to-day data analysis, there is no need to worry about low-level string representations and in English-speaking locales, there is rarely the need to worry about encoding. The locale is normally set during installation, but can be modified with the appropriate functions during an R session. The locale represents the broad application settings that influence how the system interprets some of these data e.g. language, time zones, currency symbols, etc. To check the current locale,

```
Sys.getlocale()
```

```
## [1] "LC_COLLATE=English_United Kingdom.1252;LC_CTYPE=English_United Kingdom.1252;LC_MONETARY=English_
```

Special characters

There are a few special characters that are used in R that are usually denoted with 'escapes'. Escaping is a concept where a character's meaning is no longer as-is, but altered by an adjunct character - in most modern languages this is the backslash (\) character. Escaping allows us to nest quotation marks when writing strings.

```
cat("We use quotation marks create a \"string\".")
```

```
## We use quotation marks create a "string".
```

```
cat("We use quotation marks\ncreate a \"string\".")
```

```
## We use quotation marks
## create a "string".
```

Capacity

We can check the size of a string using the `nchar` function

```
nchar(greet)
```

```
## [1] 5
```

And it is equally vectorized

```
greetings <- c(greet, "Hi", "Hey", "Whassuuup")
nchar(greetings)
```

```
## [1] 5 2 3 9
```

Some corner cases

The function `character()` creates a character vector. The first corner case is a character vector with no length and no quoted portions.

```
character()
```

```
## character(0)
```

This simply means there is nothing there. However, using this function is sometimes useful when we can to make sure a function argument receives only character vectors as input.

```
is.character(character())   # TRUE
```

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

```
length(character())
```

```
## [1] 0
```

```
myfun <- function(x = character()) {
  stopifnot(is.character(x))
  "The input is a character vector"
}

myfun()
```

```
## [1] "The input is a character vector"
```

```
myfun(42)
```

```
## Error in myfun(42): is.character(x) is not TRUE
```

```
myfun("What am I?")
```

```
## [1] "The input is a character vector"
```

String Operations

Concatenation

In R we can join 2 or more strings together using the `paste` and `paste0` functions (their defaults differ by how they handle spaces).

```
paste("Hello", "World")
```

```
## [1] "Hello World"
```

```
paste("The", "quick", "brown", "fox", "jumps")
```

```
## [1] "The quick brown fox jumps"
```

```
paste0("CV.", "docx")
```

```
## [1] "CV.docx"
```

and they follow the rules of vectorization and recycling:

```
oneTofive <- 1:5
AtoE <- LETTERS[oneTofive]

paste(AtoE, oneTofive)
```

```
## [1] "A 1" "B 2" "C 3" "D 4" "E 5"
```

```
paste0(AtoE, oneTofive)
```

```
## [1] "A1" "B2" "C3" "D4" "E5"
```

```
(qno <- paste0("(Ques. ", oneTofive, ") =>"))
```

```
## [1] "(Ques. 1) =>" "(Ques. 2) =>" "(Ques. 3) =>" "(Ques. 4) =>" "(Ques. 5) =>"
```

Replacement

Say we already have a string (or a collection of strings as a character vector with length > 1), there are ways that R allows us to make replace characters within a string For instance, to replace characters in a string, we can use the function `chartr`.

```
chartr(old = "_", new = "-", "holiday_inn_at_sunset")
```

```
## [1] "holiday-inn-at-sunset"
```

```
chartr("pp", "ff", "wapple")
```

```
## [1] "waffle"
```

It also does replacement along the length of a vector:

```
chartr("=", "-", qno)
```

```
## [1] "(Ques. 1) ->" "(Ques. 2) ->" "(Ques. 3) ->" "(Ques. 4) ->" "(Ques. 5) ->"
```

Checks

There are functions for checking prefixes and suffixes of strings - `startsWith` and `endsWith`.

```
song <- "Title: \"Blame It On The Rain\""
startsWith(x = song, "Title")
```

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

```
vars <- names(iris)
startsWith(x = vars, prefix = "Petal")
```

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

Extraction

A section of a string can be extracted using the `substr` and `substring` functions. The arguments are the string, the starting position and the ending as integers

```
substr(song, 1, 5)
```

```
## [1] "Title"
```

```
substr(song, 9, 13)
```

```
## [1] "Blame"
```

We can also use code to identify the starting and ending points, but that's the job of regular expressions (discussed below).

The assignment version of these functions can also be used to make text replacements. Note that this modifies the string in place i.e. we don't need to reassign its name.

```
substr(song, 1, 5) <- "TITLE"
song
```

```
## [1] "TITLE: \"Blame It On The Rain\""
```

Splitting

A string can be split into many pieces based on one (or a sequence) of its characters. Let say, for example we have a portion of text that has several sentences. We can create a collection of all the sentences, since a properly written sentence terminates with a period (`.`).

```
txt <- "Today is Friday. Some say \"Thank God it's Friday\", and they may have good reasons for saying s

strsplit(x = txt, split = '. ', fixed = TRUE)
```

```
## [[1]]
## [1] "Today is Friday"
## [2] "Some say \"Thank God it's Friday\", and they may have good reasons for saying so"
## [3] "However, there are those who actually enjoy their work so much that they rue weekends."
```

We have used the `fixed` argument to avoid the interpretation of `split` as a regular expression.

Regular expressions

In the preceding section, we had occasion to mention **regular expressions**. So, what are they? Regular expressions are a set of characters that are used to denote patterns in strings and these can be used to identify and modify them.

How regex works

Let's look at the word "apple". It has 5 letters "a", "p", "p", "l" and "e". However it has only 4 patterns i.e. **kinds of characters** – `a`, `p`, `l` and `e` since the letter "p" occurs twice. Thus, we use the pattern to ask a question like "Does the letter 'p' exist in the word 'apple' "? The answer of course is "Yes".

One of the key functions that uses regex in R is `grep`; by default it applies a pattern to a character vector and returns the index for the element that matches that pattern. So, using R code to ask the question above, with

```
grep(pattern = 'p', x = 'apple')
```

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

we find that the vector `x` matches the pattern "p" at position `1L`. For clarity, we will add more elements to the input vector.

```
chr <- c("triplet", "home", "turf", "holy", "apple", "clothes")
grep('p', chr)
```

```
## [1] 1 5
```

The result says that the vector `chr` has 2 elements that contain the letter "p" at positions `1` and `5`.

`grep` has other arguments such as `value` and `ignore.case` to give additional capabilities.

```
grep('p', chr, value = TRUE)
```

```
## [1] "triplet" "apple"
```

So, instead of the index, it is often more useful to know what exactly matches the pattern.

Regular expressions are case-sensitive, so `grep` gives us the option of overriding this

```
# Match with a capital 'P'
grep('P', chr)
```

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

```
grep('P', chr, ignore.case = TRUE)
```

```
## [1] 1 5
```

Then we have the variant of `grepl` that just checks and returns a logical result – also extremely useful during analyses.

```
grepl('p', chr)
```

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

Meta-characters

We have already seen that all regular characters i.e. letters and digits (in ASCII mode) match themselves. In addition to these we have the meta-characters `. \ | ( ) [ { ^ $ * + ?` that singly or in combination have can have various meanings depending on how they are used.

Anchors The characters `^` and `$` are used to represent the beginning or end of a string, respectively. Thus, using our vector `chr`we can check for all the strings that contain a 't' as follows

```
grep('t', chr, value = T)
```

```
## [1] "triplet" "turf"    "clothes"
```

but if we want to check only strings that **start** with 't', we will use the `^` character as our anchor

```
grep("^t", chr, value = T)
```

```
## [1] "triplet" "turf"
```

and if we add more characters, we can fine-tune the search

```
grep('^tu', chr, value = T)
```

```
## [1] "turf"
```

The strings "triple" and "turf" both have 't' and 'r' in them, but their order of use differs and regular expressions are strict when it comes to order

```
grep('^tr', chr, value = TRUE)
```

```
## [1] "triplet"
```

The regex `^tr` is interpreted as "*a string that starts with a 't' followed by an 'r', both of which are lower-case letters*".

Use of the termination anchor `$` is similar

```
grep("e", chr, value = TRUE)
```

```
## [1] "triplet" "home"    "apple"   "clothes"
```

```
grep("e$", chr, value = TRUE)
```

```
## [1] "home"  "apple"
```

```
grep("me$", chr, value = TRUE)
```

```
## [1] "home"
```

Quantifiers The meta-characters `* ? + {n} {n, } {n,m}` are used to state how many times a character is to be matched in sequence. This is how they are interpreted for any character `X`:

Meta	Matches
`+`	One or more times
`*`	Zero or more times
`?`	At most once i.e. it is optional
`{n}`	Exactly `n` times
`{n,}`	At least `n` times
`{n,m}`	Between `n` and `m` times (inclusively)

```
chr2 <- c(chr, "plain", "ball", "Boil", "people")
chr2
```

```
##  [1] "triplet" "home"    "turf"    "holy"    "apple"   "clothes" "plain"
##  [8] "ball"    "Boil"    "people"
```

```
grep("p", chr2, value = T)
```

```
## [1] "triplet" "apple"   "plain"   "people"
```

```
grep("pp", chr2, value = T)
```

```
## [1] "apple"
```

```
grep("p{2,}", chr2, value = T)
```

```
## [1] "apple"
```

Character classes

The regex engine provides for the use of character classes, which are constructs that broadly define certain groupings of characters. So, instead of doing this

```
numbers <- c("zero", "1" , "2", "three", "Four", "FIVE")
grep("[0-9]", numbers, value = TRUE)
```

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

```
grep("[a-z]", numbers, value = T)
```

```
## [1] "zero"  "three" "Four"
```

```
grep("[A-Z]", numbers, value = T)
```

```
## [1] "Four" "FIVE"
```

```
grep("[[:alpha:]]", numbers, value = TRUE)
```

```
## [1] "zero"  "three" "Four"  "FIVE"
```

```
grep("[[:digit:]]", numbers, value = T)
```

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

```
grep("[[:alnum:]]", numbers, value = T)
```

```
## [1] "zero"  "1"     "2"     "three" "Four"  "FIVE"
```

Back-references

We can use parentheses to demarcate part of a string and then use back-references as placeholders to refer to those portions of the string.

```
sub(pattern = "s(ho)r(t)", replacement = "\\1\\2", "A short sentence")
```

```
## [1] "A hot sentence"
```