

String Functions in R

1. Introduction to Strings
2. Understanding String Manipulation Functions
 - paste()
 - format()
 - substring()
 - sub() & gsub()
 - strsplit()
 - abbreviate()
 - tolower()
 - toupper()
 - casefold()
 - chartr()

Introduction to Strings

Any value written within a pair of single quote or double quotes in R is treated as a String.

`'A character string using single quotes'`

`"A character string using double quotes"`

Internally R stores every string within double quotes, even when you create them with single quote.

We can insert single quotes in a string with double quotes, and vice versa:

`"Welcome to 'Ask Analytics'"`

`'Welcome to "Ask Analytics"'`

We cannot insert single quotes in a string with single quotes, neither we can insert double quotes in a string with double quotes:

`"Welcome to "Ask" Analytics"`

`'Welcome to 'Ask' Analytics'`

Concatenating Strings - paste()

paste() is a very versatile function and is perhaps one of the most important functions that we can use to create and build strings.

paste() converts its arguments to character strings and concatenates (pastes) them to form one or several character strings.

```
a<-"Hello"  
b<-'How'  
c<-"are you? "
```

```
paste(a,b,c) ←  
[1] "Hello How are you?"
```

The default separator is a blank space (sep=" ")

```
paste(a,b,c,sep="-") ←  
[1] "Hello-How-are you?"
```

sep= takes a character string that is used as a separator. In this case it's "-".

```
paste("y",1:4,sep=".") ←  
[1] "y.1" "y.2" "y.3" "y.4"
```

single character "y" is pasted with the sequence 1:4 and separator sep="."

Concatenating Strings - paste()

If we want all the arguments to be collapsed into a single string, use the **collapse=** argument

```
# paste with collapsing
```

```
paste("y",1:4,sep=".",collapse="|")
```

```
[1] "y.1|y.2|y.3|y.4"
```

There is another function **paste0()** which is similar to **paste()**. The difference between them is that the argument sep by default is space (" ") in paste() and "" in paste0().

```
# paste0 function
```

```
paste0(a,b,c)
```

```
[1] "HelloHoware you? "
```



paste0() is faster than **paste()** if our objective is concatenate strings without spaces because we don't have to specify the argument **sep**.

Formatting Strings - format()

When you produce reports in R, you will want it to appear all nicely formatted to enhance the impact of your data and text on the viewer.

format() function formats numbers and strings to a specified style. It treats the elements of a vector as character strings using a common format. This is especially useful when printing numbers and quantities under different formats.

```
# default usage
```

```
format(45.4)
```

```
[1] "45.4"
```

Notice that the result is no longer a number but a character string.

This function takes a number of arguments to control the format of your result. We will see the use of some of the useful arguments like: `nsmall`, `digits`, `width`, `justify`

Formatting Strings - format()

```
format(19.7, nsmall=3)
```

```
[1] "19.700"
```

← nsmall= is used to specify the minimum number of digits after the decimal point.

```
format(13.567, digits=4)
```

```
[1] "13.57"
```

← digits= is used to specify how many significant digits of numeric values to show

```
format(c("I", "am", "fine"), width=5)
```

```
[1] "I   " "am  " "fine "
```

Here for each string the spaces are filled to make Length 5 for each string
For e.g. "I "

← width= is used to specify the minimum width of string
By default, format() fills the strings with spaces so that they are all of the same length.

```
format(c("I", "am", "fine"), width=5, justify="centre")
```

```
[1] "  I  " " am " "fine "
```

← justify= controls how the string is displayed. Takes the values "left", "right", "centre", "none"

Formatting Strings - format()

```
# Use of digits, widths and justify arguments
```

```
format(format(10/1:5,digits=2),width=6,justify="centre")
```

```
[1] " 10.0 " "  5.0 " "  3.3 " "  2.5 " "  2.0 "
```

For printing large numerical sequences we can use the arguments **big.mark**. For example, here is how we can print a number with sequences separated by a comma ","


```
format(145604500,big.mark=",")
```

```
[1] "145,604,500"
```


Extracting and Replacing - substring()

One common task when working with strings is the extraction and replacement of some characters. For such tasks we have the function **substring()** which extracts or replaces substrings in a character vector.

substring(text, first, last)

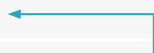


text is a character vector
first indicates the first element to be replaced
last indicates the last element to be replaced

```
# Extract 'THE'
```

```
substring("WELCOME TO THE WORLD OF R",12,14)
```

```
[1] "THE"
```



Characters from 12th to 14th
position are extracted

Extracting and Replacing - substring()

```
# Extract first 3 letters
```

```
Location<-c("Mumbai","Delhi","Mumbai","Kolkata","Delhi")  
substring(Location,1,3)
```

```
[1] "Mum" "Del" "Mum" "Kol" "Del"
```

```
# Extract each letter
```

```
substring("GOOD EVENING",1:12,1:12)
```

```
[1] "G" "O" "O" "D" " " " " "E" "V" "E" "N" "I" "N" "G"
```

```
# Replacing ':' with ' '
```

```
string=c("G1:E001","G2:E002","G3:E003")  
substring(string,3)<-" "  
string
```

```
[1] "G1 E001" "G2 E002" "G3 E003"
```

Get an Edge!

- **substr()** function, similar to **substring()** is used to extract and replace substrings in a character vector.
- Difference is **substr()** takes one start value and one end value, and returns one string. **substring()** takes n start values and the corresponding n end values, returning n strings, such that each string begins at n[i] and ends at n[i], where i is in 1:n.

The **substring()** function is a little more robust than **substr()**.

Replacing Substrings – sub() & gsub()

Within a string, if you want to replace one substring with another:

- Use **sub()** to replace the first instance of a substring

sub(old,new,string)

- Use **gsub()** to replace all instances of a substring. The g in **gsub()** stands for global.

gsub(old,new,string)

Using sub()

```
string<-"She is a data scientist. She works with an MNC"  
sub("She","Sharon",string)
```

```
[1] "Sharon is a data scientist. She works with an MNC"
```

sub() finds the first instance of the 'She' within string and replaces it with 'Sharon'

Replacing Substrings - sub() & gsub()

Replace a string with its first 3 letters

```
Location<-c("Mumbai","Delhi","Mumbai","Kolkata","Delhi")  
m<-sub("Mumbai","Mum",Location)  
d<-sub("Delhi","Del",m)  
k<-sub("Kolkata","Kol",d)  
k
```

```
[1] "Mum" "Del" "Mum" "Kol" "Del"
```

- ❑ Here we are replacing 'Mumbai' with 'Mum', 'Delhi' with 'Del' and 'Kolkata' with 'Kol'.
- ❑ Location is a vector of strings.
- ❑ sub() will find the first instance of the old substring in each string of Location and replaces it with the new substring.

Using gsub()

```
gsub("She","Sharon",string)
```

```
[1] "Sharon is a data scientist. Sharon works with an MNC"
```

gsub() does the same thing as sub(), but it replaces all instances of the substring(a global replace), not just the first

Splitting a String - strsplit()

Besides the tasks of extracting and replacing substrings, another common task is splitting a string based on a pattern. In R we have a function **strsplit()** which splits the elements of a character vector into substrings.

If we want to break a string into individual components (.i.e. words), we can use

strsplit().

For example:

```
sentence<-c("break a string into individual components")  
strsplit(sentence," ")
```

" " is a regular expression pattern used for splitting

```
[[1]]  
[1] "break"      "a"          "string"     "into"       "individual"  
[6] "components"
```

Splitting a String - strsplit()

split each date component

```
dates<-c("12-10-2014","01-05-2000","26-06-2015")  
strsplit(dates, "-")
```

```
[[1]]  
[1] "12"  "10"  "2014"
```

```
[[2]]  
[1] "01"  "05"  "2000"
```

```
[[3]]  
[1] "26"  "06"  "2015"
```

Here, date components joined by a dash "-", are split

Abbreviate Strings - abbreviate()

Another useful function for basic manipulation of character strings is **abbreviate()**.

It abbreviate strings to at least minimum length characters, such that they remain unique.

```
states<-c("New York","ALABAMA","ARKANSAS","ARIZONA")  
abbreviate(states,minlength=3)
```

New York	ALABAMA	ARKANSAS	ARIZONA
"NwY"	"ALA"	"ARK"	"ARI"

minlength= is used to specify the minimum length of the abbreviation

Other Basic String Manipulation Functions

#Converts any upper case characters into lower case

```
tolower(c("ASK ANALYTICS","data SCIENCE"))
```

```
[1] "ask analytics"
```

```
[2] "data science"
```

#Converts any lower case characters into upper case

```
toupper(c("ASK ANALYTICS","data SCIENCE"))
```

```
[1] "ASK ANALYTICS"
```

```
[2] "DATA SCIENCE"
```

#Case-folding function which is a wrapper for both `tolower()` and `toupper()`.

```
casefold("all characters in LOWER case")
```

```
[1] "all characters in lower case"
```

By default, it converts all characters to lower case, but argument `upper = TRUE` can be included to indicate the opposite (characters in upper case)

#`chartr` function which replaces a character

```
chartr("a","A","This is a string")
```

```
[1] "This is A string"
```

`chartr(old, new, x)`- translates each character in `x` that is specified in `old` to the corresponding character specified in `new`.

Quick Recap

In this session, we learnt how to manipulate strings with functions in base R.

Here is the quick recap:

String Manipulation Functions

- **paste()** converts its arguments to character strings and concatenates (pastes) them
- **format()** formats numbers and strings to a specified style
- **substring()** extracts or replaces substrings in a character vector.
- **sub()** replaces the first instance of a substring
- **gsub()** replaces all instances of a substring
- **strsplit()** which splits the elements of a character vector into substrings
- **abbreviate()** abbreviate strings to at least minimum length characters, such that they remain unique
- **toupper()** Converts any upper case characters into lower case
- **tolower()** Converts any lower case characters into upper case
- **casefold()** wrapper for both tolower() and toupper().
- **chartr()** stands for character translation.

Functions

In R, the operations that do all the work are called functions. R has a large number of in-built functions and the user can create their own functions. Most functions are in the following form:

`f(argument1, argument2,...)`

Where `f` is the name of the function, and `argument1`, `argument2`, ... are the arguments to the function.

In this tutorial we will be discussing General and Statistical Built-in Functions of R.

General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Absolute value of 'x'
```

```
abs(-4)
```

```
[1] 4
```

```
abs(c(-4,4.5,-10.5,6))
```

```
[1] 4.0 4.5 10.5 6.0
```

```
#Square Root of 'x'
```

```
sqrt(81)
```

```
[1] 9
```

```
#Rounds to the nearest integer that's larger than x
```

```
ceiling(445.67)
```

```
[1] 446
```

```
#Rounds to the nearest integer that's smaller than x
```

```
floor(445.67)
```

```
[1] 445
```

General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Rounds to the nearest integer toward 0.
```

```
trunc(445.67)
```

```
[1] 445
```

```
trunc(c(445.67,19.567,33.09))
```

```
[1] 445 19 33
```

```
#Rounds to the nearest possible value after mentioning ho many digits  
#to keep after decimal point.
```


```
round(44.5682,digits=2)
```

```
[1] 44.57
```

```
#specify the number of significant digits to be retained
```

```
signif(44.5681,digits=4)
```

```
[1] 44.57
```



Similar to round() function, signif() also specify number of significant digits regardless of the size of the number.

General Functions

Note: 'x' here is a numeral or a vector of numerals.

```
#Computes natural logarithms.
```

```
log(50)
```

```
[1] 3.912023
```

```
log(c(44,55))
```

```
[1] 3.784190 4.007333
```

```
#Computes binary (base 2) logarithm.
```

```
log2(8)
```

```
[1] 3
```

```
#Computes logarithm to the base 10.
```

```
log10(55)
```

```
[1] 1.740363
```

```
#Computes the exponential value , ex.
```

```
exp(6)
```

```
[1] 403.4288
```

Operators

An operator is a symbol which helps the user to command the compiler to perform specific mathematical or logical operations. R language is rich in built-in operators and are classified into the following categories:

- Assignment Operators
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Miscellaneous Operators

Assignment Operators

These operators are used to assign values to objects.

```
#Left Assignment (<-, <<-, =) can be done in 3 ways:
```

```
x1<-45
```

```
x1=45
```

```
x1<<-45
```

```
x1
```

```
[1] 45
```

```
#Right Assignment (->, ->>) can be done in 2 ways:
```

```
45->y1
```

```
45->>y1
```

```
y1
```

```
[1] 45
```

Different
assignments all
leading to same
output

Arithmetic Operators

These operators are used to perform mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

```
x<-7  
y<-18
```

```
#Addition
```

```
x+y  
[1] 25
```

```
#Subtraction
```

```
x-y  
[1] -11
```

```
#Multiplication
```

```
x*y  
[1] 126
```

Arithmetic Operators

#Division

y/x

[1] 2.571429

#Exponentiation

y^x

[1] 612220032

#Integer Division to get Remainder

$y\%x$

[1] 4

#Integer Division to get Quotient

$y\%/%x$

[1] 2

Relational Operators

These operators are used to compare values. The result of the comparison is the Boolean (True or False) value. Following table shows the relational operators available in R.

```
x<-7  
y<-18
```

```
#Less than
```

```
x<y  
[1] TRUE
```

```
#Greater than
```

```
x>y  
[1] FALSE
```

```
#Less than or equal to
```

```
x<=5  
[1] FALSE
```

Relational Operators

```
#Greater than or equal to
```

```
y>=20
```

```
[1] FALSE
```

```
#Equal to
```

```
y==16
```

```
[1] FALSE
```

```
#Not equal to
```

```
x!=5
```

```
[1] TRUE
```

Logical Operators

Logical Operators are applicable only to vectors of logical or numeric type. They compare each element of the first vector with the corresponding element of the second vector. Below table describes the logical operators available in R with different expressions & their respective outcomes :

Expression	Outcome	Expression	Outcome
true AND true	TRUE	true OR true	TRUE
true AND false	FALSE	true OR false	TRUE
false AND false	FALSE	false OR false	FALSE
true AND missing	missing	true OR missing	TRUE
missing AND missing	missing	missing OR missing	missing
false AND missing	FALSE	false OR missing	missing

0 means False & any number >0 is True

Logical Operators

Below table describes the logical operators available in R.

```
x<-c(FALSE,TRUE,2,5)
y<-c(TRUE,FALSE,FALSE,TRUE)
```

```
#Logical NOT
```

```
!x
[1] TRUE FALSE FALSE FALSE
```

```
#Element-wise logical AND
```

```
x&y
[1] FALSE FALSE FALSE TRUE
```

Logical Operators

`&&` and `||` examines only the first element of the vector resulting into single length logical vector.

```
#Logical AND  
x&& y  
[1] FALSE
```

```
#Element-wise logical OR  
x|y  
[1] TRUE TRUE TRUE TRUE
```

```
#Logical OR  
x||y  
[1] TRUE
```

Miscellaneous Operators

These operators are used for specific purpose and not general mathematical or logical operations.

#Colon operator.

```
x<-1:5
```

```
x
```

```
[1] 1 2 3 4 5
```

It creates simple integer sequences

```
x<-10
```

```
t<-1:8
```

```
x%in%t
```

```
[1] FALSE
```

This operator is used to identify if a value belongs to a vector or array

Quick Recap

In this session, we learnt different types of Functions and Operators in R. Here is a quick recap:

General Functions

- **abs()**, **sqrt()**, **ceiling()**, **floor()**, **trunc()**, **signif()**, **log()**, **log2()**, **log10()**, **exp()**

Operators

- Assignment Operators: **<-**, **<<-**, **=**, **->**, **->>**
- Arithmetic Operators: **+**, **-**, *****, **/**, **^**, **%%**, **/%**
- Relational Operators: **<**, **>**, **<=**, **>=**, **==**, **!=**
- Logical Operators: **!**, **&**, **&&**, **|**, **||**
- Miscellaneous Operators: **%in%**, **:**