Create account (/mw/index.php?title=Special:UserLogin&returnto=Literals%2FString&type=signup)
Log in (/mw/index.php?title=Special:UserLogin&returnto=Literals%2FString)

Search

(/wiki/Rosetta_Code)

Page (/wiki/Literals/String) Discussion (/wiki/Talk:Literals/String)
Edit (/mw/index.php?title=Literals/String&action=edit) History (/mw/index.php?title=Literals/String&action=history)

I'm working on modernizing Rosetta Code's infrastructure. Starting with communications. Please accept this time-limited open invite to RC's Slack. (https://join.slack.com/t/rosettacode/shared_invite/zt-glwmugtu-xpMPcqHs0u6MsK5zCmJF~Q). --Michael Mol (/wiki/User:Short_Circuit) (talk (/wiki/User_talk:Short_Circuit)) 20:59, 30 May 2020 (UTC)

Literals/String

Task

Show literal specification of characters and strings.

If supported, show how the following work:

- *verbatim strings* (quotes where escape sequences are quoted literally)
- *here-strings*

Also, discuss which quotes expand variables.

Related tasks

- Special characters (/wiki/Special_characters)
- Here document (/wiki/Here_document)

Other tasks related to string operations: [Expand]

(/wiki/Category:Solutions_by

Literals/String
You are encouraged to solve this task (/wiki/Rosetta_Code:Solve_a according to the task description, using any language you may know.

Contents

11l (/wiki/Category:11l)

Character literals:

```
V c = Char(‘a’)
```

Regular string literals are enclosed in double quotes, and use \ to delimit special characters:

```
"foo\nbar"
```

Raw string literals are enclosed in paired single quotation marks:

```
‘foo
bar’
```

If raw string literal should contains unpaired single quotation marks, then balancing of raw string should be performed:

```
'‘don’t’ // the same as "don’t"
```

Ada (/wiki/Category:Ada)

Single character literals require single quotes

```
ch : character := 'a';
```

String literals use double quotes

```
msg : string := "hello world";
empty : string := "";  -- an empty string
```

The length of a string in Ada is equal to the number of characters in the string. Ada does not employ a terminating null character like C. A string can have zero length, but zero length strings are not often used. Ada's string type is a fixed length string. It cannot be extended after it is created. If you need to extend the length of a string you neec to use either a *bounded string*, which has a pre-determined maximum length, similar to C strings, or an *unbounded string* which can expand or shrink to match the data it contains.

Aime (/wiki/Category:Aime)

Aime has no character representation, but it allows single quoted character constants. Their implied typed is integer.

```
integer c;
c = 'z';
```

String literals are double quoted.

```
text s;
s = "z";
```

ALGOL 68 (/wiki/Category:ALGOL_68)

In ALGOL 68 a single character (CHAR), character arrays ([]CHAR) and strings (STRING) are contained in double quotes. ALGOL 68 also has FORMAT strings which are contained between dollar ($) symbols.

```
CHAR charx = "z";
```

Strings are contained in double quotes.

```
[]CHAR charxyz = "xyz";
STRING stringxyz = "xyz";
FORMAT twonewlines = $ll$, threenewpages=$ppp$, fourbackspaces=$bbbb$;
```

Note: When only uppercase characters sets are available (eg on computers with only 6 bits per "byte") the single quote can used to denote a reserved word. eg

```
.PR QUOTE .PR
[]'CHAR' CHARXYZ = "XYZ";
```

The STRING type is simply a FLEX array of CHAR.

```
MODE STRING = FLEX[1:0]CHAR;
```

ALGOL 68 also has raw strings called BYTES, this type is a fixed width packed array of CHAR.

```
BYTES bytesabc = bytes pack("abc");
```

A string quote character is inserted in a string when two quotes are entered, eg:

```
STRING stringquote = """I'll be back."" - The Terminator";
```

A string can span lines, but cannot contain newlines. String literals are concatenated when compiled:

```
STRING linexyz := "line X;" +
 "line Y;" +
 "line Z;";
```

ALGOL 68 uses FORMATs for doing more advanced manipulations. For example given:

```
FILE linef; STRING line;
associate(linef, line);
```

Instead of using preprocessor macros ALGOL 68 can do FORMAT variable replacement within FORMATs at run time.

```
FORMAT my_symbol = $"SYMBOL"$;
FORMAT foo  = $"prefix_"f(my_symbol)"_suffix"$;
putf(linef ,foo);
```

In **standard** ALGOL 68 a "book" is a file. A book is composed of pages and lines and therefore a FORMAT be used for inserting backspaces, space, newlines and newpages into books.

```
INT pages=100, lines=25, characters=80;
FILE bookf; FLEX[pages]FLEX[lines]FLEX[characters]CHAR book;
associate(bookf, book);

# following putf inserts the string "    Line 4 indented 5" on page 3 #
putf(bookf, $3p"Page 3"4l5x"Line 4 indented 5"$)
```

Note: ALGOL 68G does not implement newpage and backspace.

ALGOL W (/wiki/Category:ALGOL_W)

```
begin
    % String literals are enclosed in double-quotes in Algol W.              %
    % There isn't a separate character type but strings of lenghth one can   %
    % be used instead.                                                       %
    % There are no escaping conventions used in string literals, except that %
    % in order to have a double-quote character in a string, two double      %
    % quotes must be used.                                                   %
    % Examples:                                                             %

    % write a single character                                              %
    write( "a" );

    % write a double-quote character                                        %
    write( """" );

    % write a multi-character string - note the "\" is not an escape        %
    % and a\nb will appear on the output, not a and b on separate lines     %
    write( "a\nb" );

end.
```

Output:

```
a
"
a\nb
```

ARM Assembly (/wiki/Category:ARM_Assembly)

Works with: as (/mw/index.php?title=As&action=edit&redlink=1) version Raspberry Pi

```
/* ARM assembly Raspberry PI  */
/*  program stringsEx.s   */

/* Constantes    */
.equ STDOUT, 1                          @ Linux output console
.equ EXIT,   1                          @ Linux syscall
.equ WRITE,  4                          @ Linux syscall

/* Initialized data */
.data
szMessString:            .asciz "String with final zero \n"
szMessString1:           .string "Other string with final zero \n"
sString:                 .ascii "String without final zero"
                         .byte 0             @ add final zero for display
sLineSpaces:             .byte '>'
                         .fill 10,1,' '      @ 10 spaces
                         .asciz "<\n"        @ add <, CR and final zero for display
sSpaces1:                .space 10,' '       @ other 10 spaces
                         .byte 0             @ add final zero for display
sCharA:                  .space 10,'A'       @ curious !! 10 A with space instruction
                         .asciz "\n"         @ add CR and final zero for display

cChar1:                  .byte 'A'           @ character A
cChar2:                  .byte 0x41          @ character A

szCarriageReturn:        .asciz "\n"

/* UnInitialized data */
.bss

/*  code section */
.text
.global main
main:
```

```
    ldr r0,iAdrszMessString
    bl affichageMess                        @ display message
    ldr r0,iAdrszMessString1
    bl affichageMess
    ldr r0,iAdrsString
    bl affichageMess
    ldr r0,iAdrszCarriageReturn
    bl affichageMess
    ldr r0,iAdrsLineSpaces
    bl affichageMess
    ldr r0,iAdrsCharA
    bl affichageMess

100:                                        @ standard end of the program
    mov r0, #0                              @ return code
    mov r7, #EXIT                           @ request to exit program
    svc 0                                   @ perform system call
iAdrszMessString:         .int szMessString
iAdrszMessString1:        .int szMessString1
iAdrsString:              .int sString
iAdrsLineSpaces:          .int sLineSpaces
iAdrszCarriageReturn:     .int szCarriageReturn
iAdrsCharA:               .int sCharA

/******************************************************************/
/*     display text with size calculation                         */
/******************************************************************/
/* r0 contains the address of the message */
affichageMess:
    push {r0,r1,r2,r7,lr}                   @ save  registers
    mov r2,#0                               @ counter length */
1:                                          @ loop length calculation
    ldrb r1,[r0,r2]                         @ read octet start position + index
    cmp r1,#0                               @ if 0 its over
    addne r2,r2,#1                          @ else add 1 in the length
    bne 1b                                  @ and loop
                                            @ so here r2 contains the length of the message
    mov r1,r0                               @ address message in r1
    mov r0,#STDOUT                          @ code to write to the standard output Linux
    mov r7, #WRITE                          @ code call system "write"
    svc #0                                  @ call system
    pop {r0,r1,r2,r7,lr}                    @ restaur registers
    bx lr                                   @ return
```

Arturo (/wiki/Category:Arturo)

```
str: "Hello world"

print [str "->" type str]

fullLineStr: « This is a full-line string

print [fullLineStr "->" type fullLineStr]

multiline: {
    This
    is a multi-line
    string
}

print [multiline "->" type multiline]

verbatim: {:
    This is
    a verbatim
    multi-line
    string
:}

print [verbatim "->" type verbatim]
```

Output:

```
Hello world -> :string
This is a full-line string -> :string
This
is a multi-line
string -> :string
This is
yet another
multi-line
string -> :string
```

AutoHotkey (/wiki/Category:AutoHotkey)

unicode

```
"c" ; character
"text" ; string
hereString =   ; with interpolation of %variables%
(
"<>"
the time is %A_Now%
\!
)

hereString2 =  ; with same line comments allowed, without interpolation of variables
(Comments %
literal %A_Now%  ; no interpolation here
)
```

AWK (/wiki/Category:AWK)

In awk, strings are enclosed using doublequotes. Characters are just strings of length 1.

```
        c  = "x"
        str= "hello"
        s1 = "abcd"     # simple string
        s2 = "ab\"cd"   # string containing a double quote, escaped with backslash
        print s1
        print s2
```

Output:

Concatenation

```
$ awk 'BEGIN{c="x"; s="hello";s1 = "abcd"; s2 = "ab\"cd"; s=s c; print s; print s1; print s2}'
hellox
```

Axe (/wiki/Category:Axe)

Character literal:

```
'A'
```

String literal:

```
"ABC"
```

Note that string literals are only null-terminated if they are assigned to a variable (e.g. Str1).

BASIC (/wiki/Category:BASIC)

Traditional BASIC implementations do not use literal character notation within a string or here document notation. However, literal characters can be referenced using their character code and these can be added to strings as required. Here we use the ASCII code for doublequotes to get the characters into a string:

```
10 LET Q$=CHR$(34): REM DOUBLEQUOTES
20 LET D$=Q$+Q$: REM A PAIR OF DOUBLEQUOTES
30 LET S$=Q$+"THIS IS A QUOTED STRING"+Q$
40 PRINT Q$;"HELLO";Q$:REM ADD QUOTES DURING OUTPUT
```

Most modern BASIC implementations don't differentiate between characters and strings -- a character is just a string of length 1. Few (if any) BASIC implementations support variables inside strings; instead, they must be handled outside the quotes. Most BASICs don't support escaping inside the string, with the possible exception of the VB-style `""` for a single quotation mark (not supported by most BASICs). To insert otherwise-unprintable characters requires the use of `CHR$`. (One notable exception is FreeBASIC (/wiki/FreeBASIC), which supports C-style escaping with `OPTION ESCAPE`.)

Strings can optionally be declared as being a certain length, much like C strings.

```
DIM (http://www.qbasicnews.com/qboho/qckdim.shtml) c AS (http://www.qbasicnews.com/qboho/qckas.shtml) STRING (http://www.qbasicnews.com/qboho/qckstring.shtml) * 1, s AS (http://www.qbasicnews.com/qboho/qckas.shtml) STRING (http://www.qbasicnews.com/qboho/qckstring.shtml)

c = "char" 'everything after the first character is silently discarded
s = "string"
PRINT (http://www.qbasicnews.com/qboho/qckprint.shtml) CHR$ (http://www.qbasicnews.com/qboho/qckchr%24.shtml)(34); s; " data "; c; CHR$ (http://www.qbasicnews.com/qboho/qckchr%24.shtml)(34)
```

Output:

```
"string data c"
```

Applesoft BASIC (/wiki/Category:Applesoft_BASIC)

```
M$ = CHR$(13) : Q$ = CHR$(34)
A$ =      "THERE ARE" + M$
A$ = A$ + "NO " + Q$ + "HERE" + Q$ + " STRINGS."
? A$
```

IS-BASIC (/wiki/Category:IS-BASIC)

```
100 PRINT CHR$(34)
110 PRINT """"
120 PRINT "This is a ""quoted string""."
```

BASIC256 (/wiki/Category:BASIC256)

```
print "Hello, World."
print chr(34); "Hello, World." & chr(34)
print "Tom said," + "'The fox ran away.'"
```

ZX Spectrum Basic (/wiki/Category:ZX_Spectrum_Basic)

The ZX Spectrum supports the use of CHR$(34). Alternatively, it is possible to print the doublequotes, by adding an extra pair of doublequotes:

```
10 REM Print some quotes
20 PRINT CHR$(34)
30 REM Print some more doublequotes
40 PRINT """"
50 REM Output the word hello enclosed in doublequotes
60 PRINT """Hello"""
```

BBC BASIC (/wiki/Category:BBC_BASIC)

Quoted (literal) strings consist of 8-bit characters and support both ANSI and UTF-8 encodings; they may not contain 'control' characters (0x00 to 0x1F). The only special character is the double-quote " which must be escaped as "". There is no special representation for a single character (it is just a string of length one). 'Here strings' are not supported.

```
      PRINT "This is a ""quoted string"""
```

Output:

```
This is a "quoted string"
```

bc (/wiki/Category:Bc)

The double-quote " starts a literal string which ends at the next double-quote. Thus strings can span multiple lines and cannot contain a double-qoute (there is no escaping mechanism).

Characters are just strings of length one.

Befunge (/wiki/Category:Befunge)

The double quote character (") enters a string literal mode, where ASCII values of characters encountered in the current instruction pointer direction up to the next quote are pushed onto the stack. Thus, any character may be used in a string except for a quote (ascii 34), which may be pushed using 57∗1−. Note: since you are pushing the string onto a stack, you usually want to define the string in reverse order so that the first character is on top.

```
"gnirts">:#,_@
```

Bracmat (/wiki/Category:Bracmat)

Strings of any length can always be surrounded by quotes. They *must* be surrounded by quotes if the string contains white space characters or one of the characters `=.,|&:+*^'$_;{}` or character sequences `\D` or `\L`. They must also be surrounded by quotes if they start with one or more characters from the set `[~/#<>%@?!-`. Inside strings the characters `"` and `\` must be escaped with a backslash `\`. White space characters for carriage return, newline and tabulator can be expressed as `\r`, `\n` and `\t`, respectively. Escape sequences need not be enclosed in quotes. A string expression prepended with `@` or `%` has no escape sequences: all characters except quotes are taken litterally. These are 10 examples of valid string expressions. (The last example is a multiline string.)

Examples:

```
string
"string"
stri\t-\tng\r\n
"-10"
"+10"
"{a*b}"
".,|&:+*^'$_"
"[~/#<>%@?!-"
string[~/#<>%@?!-
"str; ing
.,|&:+*^'$_

   "
```

C (/wiki/Category:C)

In C, single characters are contained in single quotes.

```
char ch = 'z';
```

Strings are contained in double quotes.

```
char str[] = "z";
```

This means that 'z' and "z" are different. The former is a character while the latter is a string, an array of two characters: the letter 'z' and the string-terminator null '\0'.

C has no raw string feature *(please define)*. C also has no built-in mechanism for expanding variables within strings.

A string can span lines. Newlines can be added via backslash escapes, and string literals are concatenated when compiled:

```
char lines[] = "line 1\n"
 "line 2\n"
 "line 3\n";
```

C can use library functions such as *sprintf* for doing formatted replacement within strings at run time, or preprocessor concatenation to build string literals at compile time:

```
#define FOO  "prefix_"##MY_SYMBOL##"_suffix"
```

C# (/wiki/Category:C_sharp)

C# uses single quotes for characters and double quotes for strings just like C.

C# supports verbatim strings. These begin with @" and end with ". Verbatim quotes may contain line breaks and so verbatim strings and here-strings overlap.

```
string path = @"C:\Windows\System32";
string multiline = @"Line 1.
Line 2.
Line 3.";
```

C++ (/wiki/Category:C%2B%2B)

Quoting is essentially the same in C and C++.

In C++11, it is also possible to use so-called "Raw Strings":

```
auto strA = R"(this is
a newline-separated
raw string)";
```

Clojure (/wiki/Category:Clojure)

Character literals are prefixed by a backslash:

```
[\h \e \l \l \o] ; a vector of characters
\uXXXX           ; where XXXX is some hex Unicode code point
\\               ; the backslash character literal
```

There are also identifiers for special characters:

```
\space
\newline
\tab
\formfeed
\return
\backspace
```

Clojure strings *are* Java Strings, and literals are written in the same manner:

```
"hello world\r\n"
```

COBOL (/wiki/Category:COBOL)

Strings can be enclosed in either single quotes or double quotes. There is no difference between them.

```
"This is a valid string."
'As is this.'
```

Character literals are strings of two-digit hexadecimal numbers preceded by an x.

```
X"00" *> Null character
X"48656C6C6F21" *> "Hello!"
```

There are also figurative constants which are equivalent to certain string literals:

```
HIGH-VALUE HIGH-VALUES *> Equivalent to (a string of) X"FF".
LOW-VALUE  LOW-VALUES  *>  "                       "  X"00".
NULL                   *>  "                       "  X"00".
QUOTE      QUOTES      *>  "                       "  double-quote character.
SPACE      SPACES      *>  "                       "  space.
ZERO   ZEROS   ZEROES  *>  "                       "  zero.
```

Common Lisp (/wiki/Category:Common_Lisp)

Character literals are referenced using a hash-backslash notation. Strings are arrays or sequences of characters and can be declared using double-quotes or constructed using other sequence commands.

```
(let ((colon #\:)
      (str "http://www.rosettacode.com/"))
   (format t "colon found at position ~d~%" (position colon str)))
```

D (/wiki/Category:D)

Character literals:

```
char c = 'a';
```

Regular strings support C-style escape sequences.

```
auto str = "hello";   // UTF-8
auto str2 = "hello"c;  // UTF-8
auto str3 = "hello"w; // UTF-16
auto str4 = "hello"d; // UTF-32
```

Literal string (escape sequences are not interpreted):

```
auto str = `"Hello," he said.`;
auto str2 = r"\n is slash-n";
```

Specified delimiter string:

```
// Any character is allowed after the first quote;
// the string ends with that same character followed
// by a quote.
auto str = q"$"Hello?" he enquired.$";
```

```
// If you include a newline, you get a heredoc string:
auto otherStr = q"EOS
This is part of the string.
    So is this.
EOS";
```

Token string:

```
// The contents of a token string must be valid code fragments.
auto str = q{int i = 5;};
// The contents here isn't a legal token in D, so it's an error:
auto illegal = q{@?};
```

Hex string:

```
// assigns value 'hello' to str
auto str = x"68 65 6c 6c 6f";
```

Delphi (/wiki/Category:Delphi)

```
var
  lChar: Char;
  lLine: string;
  lMultiLine: string;
begin
  lChar := 'a';
  lLine := 'some text';
  lMultiLine := 'some text' + #13#10 + 'on two lines';
```

DWScript (/wiki/Category:DWScript)

Strings are either single or double quote delimited, if you want to include the delimiter in the string, you just double it. Specific character codes (Unicode) can be specified vi
(outside of the string).

```
const s1 := 'quoted "word" in string';
const s2 := "quoted ""word"" in string"; // sames as s1, shows the doubling of the delimiter
const s2 := 'first line'#13#10'second line';   // CR+LF in the middle
```

Dyalect (/wiki/Category:Dyalect)

Strings in Dyalect are double quote delimited (and characters are single quote delimited). Both support escape codes:

```
let c = '\u0020' //a character
let str = "A string\non several lines!\sAnd you can incorporate expressions: \(c)!"
```

Dyalect also supports multiline strings:

```
let long_str = <[first line
second line
third line]>
```

Multiline strings do not support escape codes.

Déjà Vu (/wiki/Category:D%C3%A9j%C3%A0_Vu)

```
local :s "String literal"
local :s2 "newline \n carriage return \r tab \t"
!print "backslash \\ quote \q decimal character \{8364}"
```

Output:

```
backslash \ quote " decimal character €
```

E (/wiki/Category:E)

E has three sorts of quotes: *strings*, *characters*, and *quasiliterals*.

```
'T'                          # character
"The quick brown fox"        # string
`The $kind brown fox`        # "simple" quasiliteral
term`the($adjectives*, fox)` # "term" quasiliteral
```

Strings and characters use syntax similar to Java; double and single quotes, respectively, and common backslash escapes.

Quasiliterals are a user-extensible mechanism for writing any type of object "literally" in source, with "holes" for interpolation or pattern-matching. The fourth example above represents a Term object (terms are a tree language like XML (/wiki/XML) or JSON (/wiki/JSON)), with the items from the variable *adjectives* spliced in.

Quasiliterals can be used for strings as well. The third example above is the built-in simple interpolator, which also supports pattern matching. There is also a regular-expression quasi-pattern:

```
? if (http://wiki.erights.org/wiki/if) ("<abc,def>" =~ `<@a,@b>`) { [a, b] } else (http://wiki.erights.org/wiki/else) { null (http://wiki.erights.org/wiki/null) }
# value: ["abc", "def"]

? if (http://wiki.erights.org/wiki/if) (" >abc, def< " =~ rx`\W*(@a\w+)\W+(@b\w+)\W*`) { [a, b] } else (http://wiki.erights.org/wiki/else) { null (http://wiki.erights.org/wiki/null) }
# value: ["abc", "def"]
```

Ela (/wiki/Category:Ela)

Ela has both characters:

```
c = 'c'
```

and strings:

```
str = "Hello, world!"
```

Both support C-style escape codes:

```
c = '\t'
str = "first line\nsecond line\nthird line"
```

Also Ela supports verbatim strings with the following syntax:

```
vs = <[This is a
  verbatim string]>
```

Elena (/wiki/Category:Elena)

ELENA 4.x :

```
   var c := $65; // character
   var s := "some text"; // UTF-8 literal
   var w := "some wide text"w; // UTF-16 literal
   var s2 := "text with ""quotes"" and
two lines";
```

Elixir (/wiki/Category:Elixir)

String

Strings are between double quotes; they're represented internally as utf-8 encoded bytes and support interpolation.

```
IO.puts "Begin String \n============"
str = "string"
str |> is_binary # true
```

While internally represented as a sequence of bytes, the String module splits the codepoints into strings.

```
str |> String.codepoints
```

The bytes can be accessed by appending a null byte to the string

```
str <> <<0>>
```

Strings can be evaluated using `?` before a character in the command line or in a string, then evaluating the string

```
?a # 97
Code.eval_string("?b") # 98
Code.eval_string("?ł") # 322
```

Char Lists

Char lists are simply lists of characters. Elixir will attempt to convert number values to characters if a string could be formed from the values. Char lists represent characters as single quotes and still allow for interpolation.

```
IO.inspect "Begin Char List \n============="
[115, 116, 114, 105, 110, 103]
ch = "hi"
'string #{ch}'
```

Again, since 0 cannot be rendered as a character, adding it to a char list will return the char list

```
'string #{ch}'++[0]
```

Output:

Begin String
============
"string"
true
["s", "t", "r", "i", "n", "g"]
<<115, 116, 114, 105, 110, 103, 0>>
97
98
322
Begin Char List
===============
'string'
'string hi'
[115, 116, 114, 105, 110, 103, 32, 104, 105, 0]

Emacs Lisp (/wiki/Category:Emacs_Lisp)

Strings

The only string literal is a double-quote

```
"This is a string."
```

Backslash gives various special characters similar to C, such as `\n` for newline and `\"` for a literal double-quote. `\\` is a literal backslash. See "Syntax for Strings" in the elisp manual.

Characters

A character is an integer in current Emacs. (In the past character was a separate type.) `?` is the read syntax.

```
?z    => 122
?\n   => 10
```

See "Basic Char Syntax" in the elisp manual.

Erlang (/wiki/Category:Erlang)

Erlang strings are lists containing integer values within the range of the ASCII or (depending on version and settings) Unicode characters.

```
"This is a string".
[$T,$h,$i,$s,$ ,$a,$ ,$s,$t,$r,$i,$n,$g,$,,$ ,$t,$o,$o].
```

Characters are represented either as literals (above) or integer values.

```
97 == $a. % => true
```

With the string syntax, characters can be escaped with \.

```
"\"The quick brown fox jumps over the lazy dog.\"".
```

Factor (/wiki/Category:Factor)

A basic character:

```
CHAR: a
```

Characters are Unicode code points (integers in the range `[0-2,097,152]`).

`CHAR:` is a parsing word that takes a literal character, escape code, or Unicode code point name and adds a Unicode code point to the parse tree.

```
CHAR: x                      ! 120
CHAR: \u000032               ! 50
CHAR: \u{exclamation-mark}   ! 33
CHAR: exclamation-mark       ! 33
CHAR: ugaritic-letter-samka  ! 66450
```

Strings are represented as fixed-size mutable sequences of Unicode code points.

A basic string:

```
"Hello, world!"
```

We can take a look under the hood:

```
"Hello, world!" { } like ! { 72 101 108 108 111 44 32 119 111 114 108 100 33 }
```

Both `CHAR:` and strings support the following escape codes:

Escape code	Meaning
\\	\
\s	a space
\t	a tab
\n	a newline
\r	a carriage return
\b	a backspace (ASCII 8)
\v	a vertical tab (ASCII 11)
\f	a form feed (ASCII 12)
\0	a null byte (ASCII 0)
\e	escape (ASCII 27)

\"	"
\xxx	The Unicode code point with hexadecimal number xxx
\uxxxxxx	The Unicode code point with hexadecimal number xxxxxx
\u{name}	The Unicode code point named name

Some examples of strings with escape codes:

```
"Line one\nLine two" print
```

Output:

```
Line one
Line two
```

Putting quotation marks into a string:

```
"\"Hello,\" she said." print
```

Output:

```
"Hello," she said.
```

Strings can span multiple lines. Newlines are inserted where they occur in the literal.

```
"2\u{superscript-two} = 4
2\u{superscript-three} = 8
2\u{superscript-four} = 16" print
```

Output:

```
2² = 4
2³ = 8
2⁴ = 16
```

The `multiline` vocabulary provides support for verbatim strings and here-strings.

A verbatim string:

```
USE: multiline
[[ escape codes \t are literal \\ in here
but newlines \u{plus-minus-sign} are still
inserted " for each line the string \" spans.]] print
```

Output:

```
escape codes \t are literal \\ in here
but newlines \u{plus-minus-sign} are still
inserted " for each line the string \" spans.
```

Note that the space after `[[` is necessary for the Factor parser to recognize it as a word. `"` is one of very few special cases where this is not necessary. The space will not be counted as part of the string.

A here-string:

```
USE: multiline
HEREDOC: END
Everything between the line above
         and the final line (a user-defined token)
      is parsed into a string where whitespace
  is         significant.
END
print
```

Output:

```
Everything between the line above
         and the final line (a user-defined token)
      is parsed into a string where whitespace
  is         significant.
```

`STRING:` is similar to `HEREDOC:` except instead of immediately placing the string on the data stack, it defines a word that places the string on the data stack when called.

```
USE: multiline
STRING: random-stuff
ABC
123
    "x y z
;
random-stuff print
```

Output:

```
ABC
123
    "x y z
```

Finally, the `interpolate` vocabulary provides support for interpolating lexical variables, dynamic variables, and data stack values into strings.

```
USING: interpolate locals namespaces ;

"Sally" "name" set
"bicycle"
"home"

[let

"crying" :> a

[I ${name} crashed her ${1}. Her ${1} broke.
${name} ran ${} ${a}.
I]

]
```

Output:

```
Sally crashed her bicycle. Her bicycle broke.
Sally ran home crying.
```

`${}` consumes values from the stack. With a number n inside, you can reference (and re-reference!) the data stack value n places from the top of the data stack.

Forth (/wiki/Category:Forth)

In the interpreter:

```
char c   emit
s" string"   type
```

In the compiler:

```
: main
  [char] c   emit
  s" string"   type ;
```

Strings may contain any printable character except a double quote, and may not span multiple lines. Strings are done via the word S" which parses ahead for a terminal quote. The space directly after S" is thus not included in the string.

Works with: GNU Forth (/wiki/GNU_Forth)
GNU Forth has a prefix syntax for character literals, and another string literal word S\" which allows escaped characters, similar to C (/wiki/C).

```
'c emit
s\" hello\nthere!"
```

Fortran (/wiki/Category:Fortran)

First Fortran (1958) did not offer any means to manipulate text except via the H (for Hollerith) code in FORMAT statements of the form nH where n was an integer that counted the *exact* numbers of characters following the H, any characters, that constituted the text literal. Miscounts would cause a syntax error, if you were lucky. This woul be used for output to annotate the numbers, but consider the following:

```
      DIMENSION ATWT(12)
      PRINT 1
    1 FORMAT (12HElement Name,F9.4)
      DO 10 I = 1,12
        READ  1,ATWT(I)
   10   PRINT 1,ATWT(I)
      END
```

Evidently, the syntax highlighter here does not recognise the Hollerith style usage. Nor do some compilers, even if in its original home within FORMAT statements.

The first PRINT statement writes out a heading, here with lower case letters as an anachronism. Then the loop reads a deck of cards containing the name of an element and its atomic weight into an array ATWT, but the special feature is that the first twelve characters of each card replace the text in the FORMAT statement, and thus the following PRINT statement shows the name of the element followed by its atomic weight as just read.

Fortran IV introduced a text literal, specified within apostrophes, with two apostrophes in a row indicating an apostrophe within the text. Later, either an apostrophe or a double quote could be used to start a text string (and the same one must be used to end it) so that if one or the other were desired within a text literal, the other could be used as its delimiters. If both were desired, then there would be no escape from doubling for one. Because spaces are significant within text literals, a long text literal continued on the next line would have the contents of column seven of the continuation line immediately following the contents of column 72 of the continued line - except that (for some compilers reading disc files) if such lines did not extend to column 72 (because trailing spaces were trimmed from the records) rather less text would be defined. So, even though this is in fixed-format (or card image) style, again misinterpreted by the syntax highlighter,

```
      BLAH = "
     1Stuff"
```

might be the equivalent of only `BLAH = "Stuff"` instead of defining a text literal with many leading spaces. F90 formalised an opportunity for free-format source files; many compilers had also allowed usage beyond column 72.

Within the text literal, any character whatever may be supplied as text grist, according to the encodement recognised by the card reader as this was a fixed-format file - cards have an actual physical size. This applied also to source text held in disc files, as they were either fixed-size records or, for variable-length records, records had a length specification and the record content was not involved. Variable-length records were good for omitting the storage of the trailing spaces on each line, except that the sequence numbers were at the end of the line! In this case they might be omitted (unlike a card deck, a disc file's records are not going to be dropped) or there may be special provision for them to be at the start of each line with the source text's column one staring in column nine of the record. But, for the likes of paper tape, the question "How long is a record?" has no natural answer, and record endings were marked by a special symbol. Such a symbol (or symbol sequence) could not appear within a record such as within a text literal and be taken as a part of the text. This style has been followed by the ASCII world, with variously CR, CRLF, LFCR and CR sequences being used to mark end-of-record. Such characters cannot be placed within a text literal, but the CHAR(n) function makes them available in character expressions. Some compilers however corrupt the "literal" nature of text *literals* by allowing escape sequences to do so, usually in the style popularised by C, thus \n, and consequently, \\ should a single be desired.

Some examples, supposing that TEXT is a CHARACTER variable.

```
      TEXT = 'That''s right!'            !Only apostrophes as delimiters. Doubling required.
      TEXT = "That's right!"             !Chose quotes, so that apostrophes may be used freely.
      TEXT = "He said ""That's right!""" !Give in, and use quotes for a "quoted string" source style.
      TEXT = 'He said "That''s right!"'  !Though one may dabble in inconsistency.
      TEXT = 23HHe said "That's right!"  !Some later compilers allowed Hollerith to escape from FORMAT.
```

A similar syntax enables the specification of hexadecimal, octal or binary sequences, as in `X = Z"01FE"` for hexadecimal (the "H" code already being used for "Hollerith" even if the H-usage is not supported by the compiler) but this is for numerical values, not text strings. While one could mess about with EQUIVALENCE statements, numbers fill up from the right while text strings fill from the left and there would be "endian" issues as well, so it is probably not worth the bother. Just use the CHAR function in an expression, as in

```
      TEXT = "That's"//CHAR(10)//"right!" !For an ASCII linefeed (or newline) character.
```

Which may or may not be acted upon by the output device. A lineprinter probably would ignore a linefeed character but a teletype would not - it would roll the printing carriage one line up without returning to the column one position, thus the usage LFCR (or CRLF) to add the carriage return action. Some systems regard the LF as also implying a CR and for these the notation \n for "newline" is mnemonic even though there is no "newline" character code in ASCII - though there is in EBCDIC. Display screens do not handle glyph construction via overprinting though teletypes (and lineprinters) do. Similarly, a display screen may or may not start a new screen with a formfeed character and a lineprinter won't start a new page - at least if attached to a mainframe computer.

FreeBASIC (/wiki/Category:FreeBASIC)

```
Print "Hello, World."
Print Chr(34); "Hello, World." & Chr(34)

Print "Tom said, ""The fox ran away."""
Print "Tom said," + "'The fox ran away.'"
```

friendly interactive shell (/wiki/Category:Friendly_interactive_shell)

```
echo Quotes are optional in most cases.
echo
echo 'But they are when using either of these characters (or whitespace):'
echo '# $ % ^ & * ( ) { } ; \' " \\ < > ?'
echo
echo Single quotes only interpolate \\ and \' sequences.
echo '\In \other \cases, \backslashes \are \preserved \literally.'
echo
set something variable
echo "Double quotes interpolates \\, \" and \$ sequences and $something accesses."
```

FurryScript (/mw/index.php?title=Category:FurryScript&action=edit&redlink=1)

A name literal starts with ` and is one word long; it functions like a string.

A normal string literal uses angle brackets (< and >) around it. You can have additional < > pairs inside (which can nest to any level) in order to represent subroutine calls (they are not called where the string literal appears; they are called only once it is processed).

A story text uses {|| and ||} around it, and can contain any text, with no escapes supported.

All three kinds are string literals.

GAP (/wiki/Category:GAP)

```
IsChar('a');
# true
IsString("abc");
# true
IsString('a');
# false
IsChar("a");
# false
```

gecho (/wiki/Category:Gecho)

```
'a outascii
```

Just one character.

```
'yo...dawg. print
```

A string.

Go (/wiki/Category:Go)

See the language specification sections on rune literals (https://golang.org/ref/spec#Rune_literals) and string literals (https://golang.org/ref/spec#String_literals).

In Go, character literals are called "rune literals" and can be any single valid Unicode code point. They are written as an integer value or as text within single quotes.

```
ch := 'z'
ch = 122          // or 0x7a or 0172 or any other integer literal
ch = '\x7a'       // \x{2*hex}
ch = '\u007a'     // \u{4*hex}
ch = '\U0000007a' // \U{8*hex}
ch = '\172'       // \{3*octal}
```

A rune literal results in an untyped integer. When used in a typed constant or stored in a variable, usually the type is either `byte` or `rune` to distinguish character values from integer values. These are aliases for `uint8` and `int32` respectively, but like other integer types in Go, they are distinct and require an explicate cast.

```
ch := 'z'          // ch is type rune (an int32 type)
var r rune = 'z'   // r is type rune
var b byte = 'z'   // b is type byte (an uint8 type)
b2 := byte('z')    // equivalent to b
const z = 'z'      // z is untyped, it may be freely assigned or used in any integer expression
b = z
r = z
ch2 := z           // equivalent to ch (type rune)
var i int = z
const c byte = 'z' // c is a typed constant
b = c
r = rune(c)
i = int(c)
b3 := c            // equivalent to b
```

Strings literals are are either interpreted or raw.

Interpreted string literals are contained in double quotes. They may not contain newlines but may contain backslash escapes.

```
str := "z"
str = "\u007a"
str = "two\nlines"
```

This means that 'z' and "z" are different. The former is a character while the latter is a string.

Unicode may be included in the string literals. They will be encoded in UTF-8.

```
str := "日本語"
```

Raw string literals are contained within back quotes. They may contain any character except a back quote. Backslashes have no special meaning.

```
`\n` == "\\n"
```

Raw string literals, unlike regular string literals, may also span multiple lines. The newline is included in the string (but not any `'\r'` characters):

```
`abc
def` == "abc\ndef", // never "abc\r\ndef" even if the source file contains CR+LF line endings
```

Go raw string literals serve the purpose of here-strings in other languages. There is no variable expansion in either kind of string literal (the Go text/template package provides something like variable expansion).

Groovy (/wiki/Category:Groovy)

In Groovy (/wiki/Groovy), unlike in Java (/wiki/Java), a String literal is delimited with *single quotes* (apostrophes(')).

```
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) string = 'Able was I'
```

There is a *double quote* (quotation mark(")) delimited syntax in Groovy, but it represents an expression construct called a *GString* (I know, I know). Inside of a GString, sub-expression substitution of the form ${*subexpression*} may take place. Thus the following results:

```
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) gString = "${string} ere I saw Elba"

println (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20println) gString

//Outputs:
//Able was I ere I saw Elba
```

UNIX Shell (/wiki/UNIX_Shell) command line users should recognize these forms of syntax as *strong* ('-delimited) and *weak* ("-delimited) quoting. And like UNIX Shell (/wiki/UNIX_Shell) weak quoting syntax, the evaluated subexpression part of the GString syntax loses its special meaning when preceded by a backslash (\):

```
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) gString2 = "1 + 1 = ${1 + 1}"
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) gString2 == '1 + 1 = 2'

def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) gString3 = "1 + 1 = \${1 + 1}"
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) gString3 == '1 + 1 = ${1 + 1}'
```

Groovy also supports multi-line String literals and multi-line GString expressions.

```
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) multiLineString = '''
A man
A plan
A canal
'''

def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) multiLineGString = """
${multiLineString.trim()}:
Panama!
"""

println (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20println) multiLineGString

//Outputs:
//
//A man
//A plan
//A canal:
//Panama!
//
```

UNIX Shell (/wiki/UNIX_Shell) programmers should recognize these forms of syntax as similar in function to the strong and weak forms of *Here Document* syntax.

Both String literals and GString expressions support a number of *special characters* (usually non-printable characters) which consist of a single character preceded by a backslash (hence *X*), or a 4-hexadecimal-digit UNICODE encoding preceded by a backslash and a lowercase "u" (hence *uXXXX*).

One of these special characters is the backslash itself, denoted with in a String or GString as \\. This actually interferes with regular expression syntax in which literal backslashes play various important regular-expression-specific roles. Thus it can become quite onerous to write regular expressions using String or GString quoting syntax, since every regex backslash would have to be written as \\.

However, Groovy (/wiki/Groovy) has a special GString syntax that uses slash (/) as a GString delimiter rather that quote (''). In this special syntax, most backslash usages tha would require a double backslash in a regular String or GString require only a single backslash (\). This does not create a "regular expression object" (there is not such a thin in Groovy (/wiki/Groovy)); however, it does evaluate to form a "regular expression ready" String, as demonstrated in the following:

```
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) regexString = /(\[[Tt]itle\]|\[[Ss]ubject\])${10 * 5}/

assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) regexString == '(\\[[Tt]itle\\]|\\[[Ss]ubject\\])50'
```

Javascript (/wiki/Javascript) users (and others) will recognize the roots of this "regex-ready" syntax as a feature in their own language.

Since apostrophe is used to delimit String literals, that delimiter syntax is not available, as it is in Java (/wiki/Java), to denote single character literals (type char or Character) However, single character string literals can be converted to character literals by casting. Shown in the examples below are casting using the *as* operator, Java-style parenthetical casting, and forced coercion in the intialization of a variable of type char or Character.

```
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) 'a' instanceof (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20instanceof) String (https://www.google.de/search?as_q=String&num=100&hl=en&as_occt=url&as_sitesearch=java.sun.com%2Fj2se%2F1%2E5%2E0%2Fdocs%2Fapi%2F)
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) ('a' as (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20as) char (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20char)) instanceof (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20instanceof) Character (https://www.google.de/search?as_q=Character&num=100&hl=en&as_occt=url&as_sitesearch=java.sun.com%2Fj2se%2F1%2E5%2E0%2Fdocs%2Fapi%2F)
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) ((char (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20char))'a') instanceof (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20instanceof) Character (https://www.google.de/search?as_q=Character&num=100&hl=en&as_occt=url&as_sitesearch=java.sun.com%2Fj2se%2F1%2E5%2E0%2Fdocs%2Fapi%2F)

char (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20char) x = 'a'
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) x instanceof (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20instanceof) Character (https://www.google.de/search?as_q=Character&num=100&hl=en&as_occt=url&as_sitesearch=java.sun.com%2Fj2se%2F1%2E5%2E0%2Fdocs%2Fapi%2F)
Character (https://www.google.de/search?as_q=Character&num=100&hl=en&as_occt=url&as_sitesearch=java.sun.com%2Fj2se%2F1%2E5%2E0%2Fdocs%2Fapi%2F) y = 'b'
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) y instanceof (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20instanceof) Character (https://www.google.de/search?as_q=Character&num=100&hl=en&as_occt=url&as_sitesearch=java.sun.com%2Fj2se%2F1%2E5%2E0%2Fdocs%2Fapi%2F) && (x+1 == y)
```

As in Java (/wiki/Java), backslash is also used to mask a string delimiter. Thus the following two assignments represent strings containing a single quote and a single apostrophe respectively

```
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) quote = "\""
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) apostrophe = '\''
```

Of course, if you are not using GString subexpression evaluation, you can just use apostrophe delimiters to contain a quote, or quote delimiters to contain an apostrophe.

```
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) quote2 = '"'
def (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20def) apostrophe2 = "'"
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) quote == quote2
assert (https://www.google.de/search?q=site%3Agroovy.codehaus.org/%20assert) apostrophe == apostrophe2
```

Haskell (/wiki/Category:Haskell)

language support

Characters use single quotes, strings use double quotes. Both allow Unicode. Escape sequences start with a backslash. There are no verbatim strings, no here-strings, and no expansion of variables in strings.

Strings may be split across lines, even indented, using the 'gap' syntax:

```
"abcdef" == "abc\
            \def"

"abc\ndef" == "abc\n\
              \def"
```

You can also use \& which expands into nothing (but can be useful to interrupt another escape sequence).

The Haskell 98 Report section Character and String Literals (https://www.haskell.org/onlinereport/lexemes.html#sect2.6) has more information.

using raw-strings-qq package

using raw-strings-qq (https://hackage.haskell.org/package/raw-strings-qq-1.0.2/docs/Text-RawString-QQ.html) package:

```
{-# LANGUAGE QuasiQuotes #-}
import Text.RawString.QQ

"abc\ndef" == [r|abc
def|]
```

HicEst (/wiki/Category:HicEst)

HicEst makes no distinction between single characters and strings. One can use single quotes, or double quotes, or most non-standard characters.

```
CHARACTER c1='A', c2="B", c3=&C&
CHARACTER str1='single quotes', str2="double quotes", str3*100

str3 = % delimit "Nested 'strings' " if needed %
```

A null character CHAR(0) is printed as " ", displayed as "." in dialogs, but ends the string in Windows controls such as StatusBar or ClipBoard

```
str3 = 'a string' // CHAR(0) // "may contain" // $CRLF // ~ any character ~
```

Named literal constants in HicEst:

```
$TAB == CHAR(9)              ! evaluates to 1 (true)
$LF  == CHAR(10)
$CR  == CHAR(13)
$CRLF == CHAR(13) // CHAR(10) ! concatenation
```

Icon (/wiki/Category:Icon) and Unicon (/wiki/Category:Unicon)

Below is a little program to demonstrate string literals.

```
procedure main()

   # strings are variable length are not NUL terminated
   # at this time there is no support for unicode or multi-byte charactersets

   c1 := 'aaab'                                        # not a string - cset
   s1 := "aaab"                                        # string
   s2 := "\"aaab\b\d\e\f\n\l\n\r\t\v\'\"\\\000\x00\^c"   # with escapes and imbedded zero

   # no native variable substitution, a printf library function is available in the IPL

   every x := c1|s1|s2 do                              # show them
      write(" size=",*x,", type=", type(x),", value=", image(x))
end
```

Output:

```
  size=2, type=cset, value='ab'
  size=4, type=string, value="aaab"
  size=21, type=string, value="\"aaab\b\d\e\f\n\n\n\r\t\v'\"\\\x00\x00\x03"
```

IDL (/wiki/Category:IDL)

The single and double quotes are fairly interchangeable allowing one to use whichever isn't to be quoted (though single-quotes seem more well-behaved around integers in strings). Thus the following are both valid character-constant assignments:

```
a = " that's a string "
b = ' a "string" is this '
```

In a pinch, a character constant doesn't absolutely have to be terminated, rendering the following valid:

```
a = " that's a string
```

Duplicating either of them quotes them. Thus the following contains three single quotes and no double-quotes:

```
a = ' that''s a string
print,a
;==>  that's a string
```

Things in quotes are not expanded. To get to the content of a variable, leave it unquoted:

```
b = 'hello'
a = b+' world
print,a
;==>  hello world
```

Single-quoted strings of valid hex or octal digits will be expanded if followed by "x" or "o":

```
print,'777'x
;==>   1911
print,'777'o
;==>    511
print,'777'
;==>    777
```

so will be unterminated double-quoted strings if they represent valid octal numbers:

```
print,"777
;==>    511
print,"877
;==>    877
```

Note that this renders the following false (common trip-up for IDL newbies):

```
a = "0"
;==> Syntax error.
```

...because the number zero indicates that an octal number follows, but the second double-quote is not a valid octal digit.

Byte-arrays that are converted into strings are converted to the ascii-characters represented by the bytes. E.g.

```
crlf = string([13b,10b])
```

Inform 7 (/wiki/Category:Inform_7)

String literals are enclosed in double quotes. These may include raw line breaks, or expressions to be substituted enclosed in square brackets.

```
Home is a room. The description is "This is where you live...

...with your [number of animals in Home] pet[s]."
```

Single quotes in a string are translated to double quotes when they occur outside of a word: the string literal

```
"'That's nice,' said the captain."
```

will print as

```
"That's nice," said the captain.
```

Raw linebreak must be double -- single linebreaks will be collapsed unless explicitly marked with `[line break]`. In addition, leading whitespace is stripped from each line. This

```
"
\
 \
  \
   \"
```

will print as:

```
\\\\
```

while this:

```
"
[line break]\
[line break] \
[line break]  \
[line break]   \"
```

will insert line breaks and preserve the following whitespace, printing as:

```
\
 \
  \
   \
```

There are no character literals: phrases that manipulate characters pass them as single-character strings.

J (/wiki/Category:J)

Like C, J treats strings as lists of characters. Character literals are enclosed in single quotes, and there is no interpolation. Therefore, the only "escape" character neccessary is the single-quote itself, and within a character literal is represented by a pair of adjacent single quotes (much like in C, where within a character literal, a slash is represented by a pair of adjacent slashes).

Examples:

```
'x'                 NB.  Scalar character
'string'            NB.  List of characters, i.e. a string
'can''t get simpler'  NB.  Embedded single-quote
```

Like VB, J can include newlines and other special characters in literals with concatentation. Also like VB, J comes with certain constants predefined for some characters:

```
'Here is line 1',LF,'and line two'

'On a mac, you need',CR,'a carriage return'

'And on windows, ',CRLF,'you need both'

TAB,TAB,TAB,'Everyone loves tabs!'
```

These constants are simply names assigned to selections from the ASCII alphabet. That is, the standard library executes lines like this:

```
CR   =:  13 { a.
LF   =:  10 { a.
CRLF =:  CR,LF     NB.  Or just  10 13 { a.
TAB  =:  9 { a.
```

Since these constants are nothing special, it can be seen that any variable can be similarly included in a literal:

```
NAME =:  'John Q. Public'
'Hello, ',NAME,' you may have already won $1,000,000'
```

For multiline literals, you may define an explicit noun, which is terminated by a lone `)`

```
template =: noun define
Hello, NAME.

My name is SHYSTER, and I'm here to tell
you that you my have already won $AMOUNT!!

To collect your winnings, please send $PAYMENT
to ADDRESS.
)
```

Simple substitution is most easily effected by using loading a standard script:

```
load 'strings'

name    =:  'John Q. Public'
shyster =:  'Ed McMahon'
amount  =:  1e6
payment =:  2 * amount
address =:  'Publisher''s Clearing House'

targets =:  ;:   'NAME SHYSTER AMOUNT PAYMENT ADDRESS'
sources =:  ":&.> name;shyster;amount;payment;address

message =: template rplc targets,.sources
```

While C-like interpolation can be effected with another:

```
   load 'printf'

   'This should look %d%% familiar \nto programmers of %s.' sprintf 99;'C'
This should look 99% familiar
to programmers of C.
```

Java (/wiki/Category:Java)

```
  char a = 'a';  // prints as: a
  String (https://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) b = "abc";  // prints as:
 abc
  char doubleQuote = '"';  // prints as: "
  char singleQuote = '\'';  // prints as: '
  String (https://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) singleQuotes = "'";  //
prints as: ''
  String (https://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) doubleQuotes = "\"\"";  /
/ prints as: ""
```

Null characters ('\0') are printed as spaces in Java. They will not terminate a String as they would in C or C++. So, the String "this \0is \0a \0test" will print like this:

```
this  is  a  test
```

JavaScript (/wiki/Category:JavaScript)

A JavaScript string is a sequence of zero or more characters enclosed in either 'single quotes' or "double quotes". Neither form prevents escape sequences: `"\n"` and `'\n'` are both strings of length 1. There is no variable interpolation.

Unicode characters can be entered as literals or as 4 character hexadecimal escapes. The following expressions are equivalent:

```
(function () {
    return "αβγδ 中间来点中文 🐫 אבגד";
})();


(function() {
    return "\u03b1\u03b2\u03b3\u03b4 \u4e2d\u95f4\u6765\u70b9\u4e2d\u6587 \ud83d\udc2b \u05d0\u05d1\u05d2\u05d3";
})();
```

Note that in the case of the Emoji character above, where more than 4 hexadecimal characters are needed, ES5 requires us to separately write a pair of surrogate halves, ar
the **String.length** of such characters is 2.

ES6 introduces Unicode code point escapes such as

```
'\u{2F804}'
```

allowing direct escaping of code points up to 0x10FFFF.

ES6 also introduces template literals, which are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them. Template literals are enclosed by the backtick (`` ` ``) (grave accent) character instead of double or single quotes.

```
const multiLine = `string text line 1
string text line 2`
const expression = `expressions are also supported, using \$\{\}: ${multiLine}`

console.log(expression)
```

Output:

```
expressions are also supported, using ${}: string text line 1
string text line 2
```

jq (/wiki/Category:Jq)

jq supports all JSON types, including JSON strings; jq also supports "string interpolation".

The rules for constructing JSON string literals are explained elsewhere (notably at json.org), so here we'll focus on "string interpolation" -- a technique for creating JSON strings programmatically using string literals, much like ruby's "#{...}", for example. The twist is that the string literal for specifying string interpolation is (by design) not itself valid JSON string.

Suppose that:

- s is (or is a reference to) a JSON entity (e.g. a string or a number), and
- we wish to create a JSON string that is some combination of JSON strings and the string value of s, for example: "The value of s is " + (s|tostring).

jq allows the shorthand: "The value of s is \(s)", and in general, arbitrarily many such interpolations may be made.

JSON (/wiki/Category:JSON)

A JSON string literal is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. Unicode code points can be represented as a hexadecimal escape sequence, like `"\u002F"` which is the same as `"/"`. Multi-line strings are not supported.

Julia (/wiki/Category:Julia)

Concatenation:

```
greet = "Hello"
whom = "world"
greet * ", " * whom * "."
```

Interpolation:

```
"$greet, $whom."
```

Both will output:

```
Hello, world.
```

Triple-quoted strings

```
str = """Hello,
        world.
         """

print(str)
```

Will output:

```
Hello,
world.
```

Kotlin (/wiki/Category:Kotlin)

Kotlin supports two kinds of string literals (UTF-16 encoded):

- escaped string literals, enclosed in double-quotes, which can contain 'escaped characters'.
- raw string literals, enclosed in triple double-quotes, which ignore escaping but can contain new lines.

The language also supports character literals - a single UTF-16 character (including an escaped character) enclosed in single quotes.

Here are some examples of these :

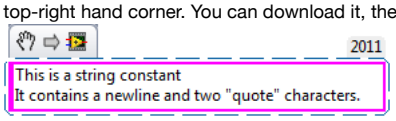
```
// version 1.0.6

fun main(args: Array<String>) {
    val (https://scala-lang.org) cl = 'a'          // character literal - can contain escaped character
    val (https://scala-lang.org) esl = "abc\ndef"  // escaped string literal - can contain escaped character(s)
    val (https://scala-lang.org) rsl = """
              This is a raw string literal
              which does not treat escaped characters
              (\t, \b, \n, \r, \', \", \\, \$ and \u)
              specially and can contain new lines.

              "Quotes" or doubled ""quotes"" can
              be included without problem but not
              tripled quotes.
              """
    val (https://scala-lang.org) msl = """
              |Leading whitespace can be removed from a raw
              |string literal by including
              |a margin prefix ('|' is the default)
              |in combination with the trimMargin function.
              """.trimMargin()
    println(cl)
    println(esl)
    println(rsl)
    println(msl)
}
```

Output:

```
a
abc
def

              This is a raw string literal
              which does not treat escaped characters
              (\t, \b, \n, \r, \', \", \\, \$ and \u)
              specially and can contain new lines.

              "Quotes" or doubled ""quotes"" can
              be included without problem but not
              tripled quotes.

Leading whitepace can be removed from a raw
string literal by including
a margin ('|' is the default)
in combination with the trimMargin function.
```

LabVIEW (/wiki/Category:LabVIEW)

LabVIEW is a graphical language so it uses graphical string delimiters. No escaping is needed.
This image is a VI Snippet (http://zone.ni.com/devzone/cda/tut/p/id/9330), an executable image of LabVIEW (/wiki/LabVIEW) code. The LabVIEW version is shown on the top-right hand corner. You can download it, then drag-and-drop it onto the LabVIEW block diagram from a file browser, and it will appear as runnable, editable code.



(/wiki/File:LabVIEW_Literals_String.png)

Lasso (/wiki/Category:Lasso)

All strings in Lasso are Unicode strings. This means that a string can contain any of the characters available in Unicode. Lasso supports two kinds of string literals: quoted and ticked. Quoted strings can contain escape sequences, while ticked strings cannot. Both quoted and ticked string literals can contain line breaks and they both return same type of string object. [1] (http://lassoguide.com/language/literals.html)

Quoted Strings

```
'I\'m a 2" string\n'
"I'm a 2\" string\n"
```

Ticked Strings

In the below example here \n would not be a line feed, it represents a backslash and n.

```
`I'm also a 2" string\n`
```

LaTeX (/wiki/Category:LaTeX)

Since LaTeX is a markup language rather than a programming language, quotes are displayed rather than interpreted. However, quotes do deserve special mention in LaTeX Opening (left) quotes are denoted with backquotes and closing (right) quotes are denoted with quotes. Single quotes use a single symbol and double quotes use double symbols. For example, to typeset 'a' is for "apple" in LaTeX, one would type

```
\documentclass (http://www.golatex.de/wiki/index.php?title=%5Cdocumentclass){minimal}
\begin{document}
`a' is for ``apple"
\end{document}
```

One common mistake is to use the same symbol for opening and closing quotes, which results in the one of the quotes being backward in the output. Another common mistake is to use a double quote symbol in the input file rather than two single quotes in order to produce a double quote in the output.

Liberty BASIC (/wiki/Category:Liberty_BASIC)

```
'Liberty BASIC does not support escape characters within literal strings.
print "Quotation mark:"
print chr$(34)
print

'Print literal string
print "Hello, World."
'Print literal string displaying quotation marks.
print chr$(34) + "Hello, World." + chr$(34)
```

Lingo (/wiki/Category:Lingo)

- Lingo only supports single quotes for string literals. Single quotes inside string literals have to be replaced by "&QUOTE&":

```
str = "Hello "&QUOTE&"world!"&QUOTE
put str
-- "Hello "world!""
```

- Lingo does not support heredoc syntax, but only multiline string literals by using the line continuation character "\":

```
str = "This is the first line.\
This is the second line.\
This is the third line."
```

- Lingo does not support automatic variable expansion in strings. But the function value() can be used to expand template strings in the current context:

```
template = QUOTE&"Milliseconds since last reboot: "&QUOTE&"&_system.milliseconds"

-- expand template in current context
str = value(template)
put str
-- "Milliseconds since last reboot: 20077664"
```

Lisaac (/wiki/Category:Lisaac)

Characters:

```
c1 := 'a';
c2 := '\n'; // newline
c3 := '\''; // quote
c4 := '\101o'; // octal
c5 := '\10\'; // decimal
c6 := '\0Ah\'; // hexadecimal
c7 := '\10010110b\'; // binary
```

Strings:

```
s1 := "this is a\nsample"; // newline
s2 := "\""; // double quote
s3 := "abc\
      \xyz"; // "abcxyz", cut the gap
```

LiveCode (/wiki/Category:LiveCode)

LiveCode has only one string representation using quotes. Characters are accessed through chunk expressions, specifically char. Some special characters are built-in

constants such as quote, space, comma, cr, return. There is no support for escaping characters or multiline literals.

```
put "Literal string"  -- Literal string
put char 1 of "Literal string"  -- L
put char 1 to 7 of "Literal string"  -- Literal
put word 1 of "Literal string"  -- Literal
put quote & "string" & quote -- "string"
```

Logo (/wiki/Category:Logo)

Logo does not have a string or character type that is separate from its symbol type ("word"). A literal word is specified by prefixing a double-quote character. Reserved and delimiting characters, ()[];~+-*/\=<>| and newline, may be used if preceded by a backslash. Alternatively, the string may be wrapped in vertical bars, in which case only backslash and vertical bar need be escaped.

```
print "Hello\,\ world
print "|Hello, world|
```

Lua (/wiki/Category:Lua)

Strings can be enclosed using singlequotes or doublequotes. Having two different types of quotation symbols enables either of the symbols to be embedded within a string enclosed with the other symbol.

```
singlequotestring = 'can contain "double quotes"'
doublequotestring = "can contain 'single quotes'"
longstring = [[can contain
              newlines]]
longstring2 = [==[ can contain [[ other ]=] longstring " and ' string [===[ qualifiers]==]
```

Note that interpolation of variables names within a string does not take place. However, interpolation of literal characters escape sequences does occur, irrespective of whether singlequote or doublequote enclosures are being used.

M2000 Interpreter (/wiki/Category:M2000_Interpreter)

```
Print "Hello {World}"
Print {Hello "World"}
Report {Multiline String
      2nd line
      }
Print """Hello There"""={"Hello There"}
Print Quote$("Hello There")={"Hello There"}
```

M4 (/wiki/Category:M4)

The quoting characters are ` and ', but can be changed by the `changequote` macro:

```
`this is quoted string'
```

```
changequote(`[',`]')dnl
[this is a quoted string]
```

Maple (/wiki/Category:Maple)

There is no separate character type in Maple; a character is just a string of length equal to 1.

```
> "foobar";
                                "foobar"

> "foo\nbar"; # string with a newline
"foo
    bar"

> "c"; # a character
                                  "c"
```

Note that adjacent strings in the input (separated only by white-space) are concatenated automatically by the parser.

```
> "foo"   "bar";
                                "foobar"
```

Since variable names are not distinguished lexically from other text (such as by using a "$" prefix, as in some shells), Maple does not do any kind of variable expansion insid strings.

Mathematica (/wiki/Category:Mathematica)/Wolfram Language (/wiki/Category:Wolfram_Language)

```
There is no character type in Mathematica, only string type.
"c";          // String (result: "c")
"\n";         // String (result: newline character)
```

MATLAB (/wiki/Category:MATLAB)

Strings start and end with single quotes, the escape sequence for a single quote with in a string, is the use of two consequtive single quotes

```
    s1 = 'abcd'   % simple string
    s2 = 'ab''cd'   % string containing a single quote
```

Output:

```
>>     s1 = 'abcd'   % simple string
s1 = abcd
>>     s2 = 'ab''cd'   % string containing a single quote
s2 = ab'cd
```

Maxima (/wiki/Category:Maxima)

```
/* A string */
"The quick brown fox jumps over the lazy dog";

/* A character - just a one character string */
"a"
```

Metafont (/wiki/Category:Metafont)

In Metafont there's no difference between a single character string and a single character. Moreover, the double quotes (which delimites a string) cannot be inserted directly into a string; for this reason, the basic Metafont macro set defines

```
string ditto; ditto = char 34;
```

i.e. a string which is the single character having ASCII code 34 ("). Macro or variables expansion inside a string block is inhibited.

```
message "You've said: " & ditto & "Good bye!" & ditto & ".";
```

ML/I (/wiki/Category:ML/I)

ML/I treats all input and programs as character streams. Strings do not have to be quoted; they are taken 'as is'. If one wishes to ensure that a string is taken literally (i.e. no evaluated), it is enclosed in *literal brackets*. There are no predefined literal brackets; the programmer can define anything suitable, usually by setting up a *matched text skip*, using the MCSKIP operation macro. By convention, the pair <> is used for literal brackets, unless this clashes in the case of a particular processing task.

Input

```
MCSKIP "WITH" NL
"" Literals/String
MCINS %.
MCSKIP MT,<>
"" Demonstration of literal string
MCDEF Bob AS Alice
"" The following two lines both mention Bob. The first line is
"" evaluated, but the second is surrounded by literal brackets and is not
"" evaluated.
This is the first mention of Bob
<and here we mention Bob again>
```

Output

```
This is the first mention of Alice
and here we mention Bob again
```

Modula-3 (/wiki/Category:Modula-3)

Characters in Modula-3 use single quotes.

```
VAR char: CHAR := 'a';
```

Strings in Modula-3 use double quotes.

```
VAR str: TEXT := "foo";
```

`TEXT` is the string type in Modula-3. Characters can be stored in an array and then converted to type TEXT using the function `Text.FromChars` in the `Text` module.

Strings (of type `TEXT`) can be converted into an array of characters using the function `Text.SetChars`.

```
VAR str: TEXT := "Foo";
VAR chrarray: ARRAY [1..3] OF CHAR;

Text.SetChars(chrarray, str);
(* chrarray now has the value ['F', 'o', 'o'] *)
```

MUMPS (/wiki/Category:MUMPS)

All strings are delimited by the double quotes character. But you can escape the double quotes to add a double quotes character to a string.

```
USER>SET S1="ABC"

USER>SET S2="""DEF"""

USER>SET S3="""GHI"

USER>W S1
ABC
USER>W S2
"DEF"
USER>W S3
"GHI
```

Nemerle (/wiki/Category:Nemerle)

Character literals are enclosed in single quotes. Regular strings are enclosed in double quotes, and use \ to delimit special characters, whitespace similar to C. A @ preceding the double quotes indicates a literal string. A $ preceding the double quote indicates string interpolation, identifiers prefixed with $ inside the string literal will be replaced with their value. Nemerle also has a recursive string literal, enclosed within <# #>, that is the same as a literal string, except that it allows nesting of strings.

```
'a'                                          // character literal
'\n'                                         // also a character literal
"foo\nbar"                                   // string literal
@"x\n"                                       // same as "x\\n"
@"x
 y"                                          // same as "x\n y"
@"""Hi!"""                                   // "" replaces \" to escape a literal quote mark
<#This string type can contain any symbols including "
and new lines. It does not support escape codes
like "\n".#>                                 // same as "This string type can contain any symbols including \"\nand new lines. "
                                             //       + "It does not\nsupport escape codes\nlike \"\\n\"."
<#Test <# Inner #> end#>                     // same as "Test <# Inner #> end" (i.e. this string type support recursion.
```

Nim (/wiki/Category:Nim)

```
var c = 'c'
var s = "foobar"
var l = """foobar
and even
more test here"""

var f = r"C:\texts\text.txt" # Raw string
```

OASYS Assembler (/wiki/Category:OASYS_Assembler)

There are two kinds, strings with quotation marks and strings with braces. Both kinds are treated exactly like numeric tokens for all purposes.

Strings with quotation marks can contain repeated quotation marks to represent a quotation mark, a tilde to represent a line break, or a line break to represent a space (in which case any leading spaces on the following line are ignored).

Strings with braces start with { and end with the next } (they don't nest), and all characters (except a right-brace) are treated as-is.

There are no character literals.

Objeck (/wiki/Category:Objeck)

Objeck string support is similar to Java except that string elements are 1-byte in length. In addition, string literals may terminated using a NULL character or the string's length calculation.

Objective-C (/wiki/Category:Objective-C)

The same as C, with the addition of the new string literal

```
@"Hello, world!"
```

which represents a pointer to a statically allocated string object, of type `NSString *`, similar to string literals in Java. You can use this literal like other object pointers, e.g. call methods on it `[@"Hello, world!" uppercaseString]`.

OCaml (/wiki/Category:OCaml)

Characters are contained in single quotes:

```
# 'a';;
- : char (http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#TYPEchar) = 'a'
```

Strings are contained in double quotes:

```
# "Hello world";;
- : string (http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#TYPEstring) = "Hello world"
```

Strings may be split across lines and concatenated using the following syntax: (the newline and any blanks at the beginning of the second line is ignored)

```
# "abc\
    def";;
- : string (http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#TYPEstring) = "abcdef"
```

If the above syntax is not used then any newlines and whitespace are included in the string:

```
# "abc
   def";;
- : string (http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#TYPEstring) = "abc\n def"
```

Another syntax to include verbatim text:

```
# {id|
    Hello World!
  |id} ;;
- : string (http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#TYPEstring) = "\n  Hello World!\n"
```

Octave (/wiki/Category:Octave)

Strings can be defined in Octave with single or double quotes. In order to maintain compatible with Matlab, it is recommended to use single quotes for defining strings.

```
    s1 = 'abcd'   % simple string
    s2 = 'ab''cd'   % string containing a single quote using an escaped single quote
    s3 = 'ab"cd'   % simple string containing a double quote
    s4 = "ab'cd"   % string containing a single quote
    s5 = "ab""cd"   % string containing a double quote using an escaped double quote
```

Output:

```
octave:5>     s1 = 'abcd'   % simple string
s1 = abcd
octave:6>     s2 = 'ab''cd'   % string containing a single quote using an escaped single quote
s2 = ab'cd
octave:7>     s3 = 'ab"cd'   % simple string containing a double quote
s3 = ab"cd
octave:8>     s4 = "ab'cd"   % string containing a single quote
s4 = ab'cd
octave:9>     s5 = "ab""cd"   % string containing a double quote using an escaped double quote
s5 = ab"cd
```

Oforth (/wiki/Category:Oforth)

Oforth uses single quotes for characters and double quotes for strings.

There is no character type : characters are integers representing unicode value of the character.

```
'a'
'\''
"abcd"
"ab\ncd"
"ab\" and \" cd"
```

Oz (/wiki/Category:Oz)

```
declare
Digit0 = &0       %% the character '0'
NewLine = &\n     %% a character with special representation
NewLine = &\012   %% characters can also be specified with octals

%% Strings are lists of characters, but can also be written in double quotes:
[&H &e &l &l &o] = "Hello"

AnAtom = 'Hello'         %% single quotes are used for atoms
Atom2 = hello = 'hello'  %% for atoms starting with a lower case letter, they are optional

%% To build strings out of other values, so-called virtual strings are used:
MyName = "Peter"
MyAge = 8
{System.showInfo MyName # " is " # MyAge # " years old."}
```

PARI/GP (/wiki/Category:PARI/GP)

There are just three escapes:

```
\e escape
\n newline
\t tab
```

Any other escaped character simply represents itself; `\\` and `\"` are the most useful. There are no characters or character strings as such, but Vectorsmall("string") is ver similar to a character array.

Version 2.4.3 added the functions `printf` and `Strprintf` which allow interpolation (typically with %Ps).

Pascal (/wiki/Category:Pascal)

See Delphi (/wiki/Literals/String#Delphi)

Perl (/wiki/Category:Perl)

Perl makes no distinction between single characters and strings. One can use single or double quotes, but they are different. Double-quotes allows you to interpolate variables and escape sequences, while single-quotes do not.

```
'c';                    # character
'hello';                # these two strings are the same
"hello";
'Hi $name. How are you?'; # result: "Hi $name. How are you?"
"Hi $name. How are you?"; # result: "Hi Bob. How are you?"
'\n';                   # 2-character string with a backslash and "n"
"\n";                   # newline character
`ls`;                   # runs a command in the shell and returns the output as a string
q/hello/;               # same as 'hello', but allows custom delimiters, eg: q(hi) and q!hi!
qq/hello/;              # same as "hello", but allows custom delimiters, eg: qq{$hi} and qq#hi#
qw/one two three/;      # same as ('one', 'two', 'three'); constructs a list of the words
qx/ls/;                 # quoted execution, same as `ls`
qr/regex/;              # creates a regular expression
<<END;                  # Here-Document
Hi, whatever goes here gets put into the string,
including newlines and $variables,
until the label we put above
END
<<'END';                # Here-Document like single-quoted
Same as above, but no interpolation of $variables.
END
```

Phix (/wiki/Category:Phix)

Library: Phix/basics (/wiki/Category:Phix/basics)
single character literals (incidentally entirely equivalient to their ascii value) require single quotes, eg

```
constant UPPERCASEJ = 'J'   -- equivalent to 74
```

string literals use double quotes, eg

```
constant hw = "Hello World!",
         mt = ""    -- empty string
```

Note that 'z' and "z" are quite different. In Phix there is a strong difference between a character and a string.

All strings are ansi or utf8, depending on the encoding of the source file, eg

```
s = "日本語"
```

Utf8 strings are byte-subscripted rather than character-subscripted, so s[3] is not necessarily the third character.
Phix strings have a length field in the (internal) header, /and/ a terminating null, so they can be used directly when interfacing to C-style languages.
Phix strings can also be used to hold "raw binary", ie instead of a sequence of characters, a sequence of any bytes in the range 0 to 255.
Strings are fully mutable: you can append, prepend, replace, substitute, and crop characters and slices (/substrings) any way you like, eg

```
string s = "food"
    s[2..3] = 'e'       -- s is now "feed" (replace all)
    s[2..2] = "east"    -- s is now "feasted" (replace substring)
    s[2..5] = ""        -- s is now "fed"
```

Special characters may be entered (between quotes) using a back-slash:

```
       Code     Value   Meaning
        \n       #10     newline
        \r       #13     carriage return
        \b       #08     backspace
        \t       #09     tab
        \\       #5C     backslash
        \"       #22     double quote (the \ is optional in ', mandatory in ")
        \'       #27     single quote (the \ is optional in ", mandatory in ')
        \0       #00     null
        \#HH     #HH     any hexadecimal byte
        \xHH     #HH     any hexadecimal byte
        \uH4      -      any 16-bit unicode point, eg "\u1234", max #FFFF
        \UH8      -      any 32-bit unicode point, eg "\U00105678", max #10FFFF
```

There are no other automatic substitutions or interpolation, other than through explict function calls such as [s]printf().

Strings can also be entered by using triple quotes or backticks intead of double quotes to include linebreaks and avoid any backslash interpretation. If the literal begins with a newline, it is discarded and any immediately following leading underscores specify a (maximum) trimming that should be applied to all subsequent lines. Examples:

```
ts = ""`
this
string\thing`""

ts = ""`
_____this
     string\thing`""

ts = `this
string\thing`

ts = "this\nstring\\thing"
```

which are all equivalent.

On a practical note, as long as you have at least 2GB of physical memory, you should experience no problems whatsoever constructing a string with 400 million characters, and you could more than triple that by allocating things up front, however deliberately hogging the biggest block of memory the system will allow is generally considered ba programming practice, and may lead to disk thrashing.

Hex string literals are also supported (mainly for compatibility with OpenEuphoria, x/u/U for 1/2/4 byte codes), eg:

```
?x"68 65 6c 6c 6f";     -- displays "hello"
```

PHP (/wiki/Category:PHP)

PHP makes no distinction between single characters and strings. One can use single or double quotes, but they are different. Double-quotes allows you to interpolate variables and escape sequences, while single-quotes do not.

```
'c';                    # character
'hello';                # these two strings are the same
"hello";
'Hi $name. How are you?'; # result: "Hi $name. How are you?"
"Hi $name. How are you?"; # result: "Hi Bob. How are you?"
'\n';                   # 2-character string with a backslash and "n"
"\n";                   # newline character
`ls`;                   # runs a command in the shell and returns the output as a string
<<END (http://www.php.net/end)                    # Here-Document
Hi, whatever goes here gets put into the string,
including newlines and $variables,
until the label we put above
END (http://www.php.net/end);
<<'END'                 # Here-Document like single-quoted
Same as above, but no interpolation of $variables.
END (http://www.php.net/end);
```

PicoLisp (/wiki/Category:PicoLisp)

PicoLisp doesn't have a string data type. Instead, symbols are used. Certain uninterned symbols, called "transient symbols" (http://software-lab.de/doc/ref.html#transient), however, look and behave like strings on other languages.

Syntactically, transient symbols (called "strings" in the following) are surrounded by double quotes.

```
: "ab\"cd"
-> "ab\"cd"
```

Double quotes in strings are escaped with a backslash.

ASCII control characters can be written using the hat ('^') character:

```
: "ab^Icd^Jef"  # Tab, linefeed
```

There is no special character type or representation. Individual characters are handled as single-character strings:

```
: (chop "abc")
-> ("a" "b" "c")

: (pack (reverse @))
-> "cba"
```

A limited handling of here-strings is available with the 'here (http://software-lab.de/doc/refH.html#here)' function.

Pike (/wiki/Category:Pike)

```
'c';          // Character code (ASCII) (result: 99)
"c";          // String (result: "c")
"\n";         // String (result: newline character)
"hi " + world // String (result: "hi " and the contents of the variable world)
#"multiple line
string using the
preprocessor" // single literal string with newlines in it
```

PL/I (/wiki/Category:PL/I)

```
'H'                 /* a single character as a literal.            */
'this is a string'
''                  /* an empty string literal.                    */
'John''s cat'       /* a literal containing an embedded apostrophe. */
                    /* stored are <<John's cat>>                   */
'101100'b           /* a bit string, stored as one bit per digit.   */
```

Plain TEX (/wiki/Category:PlainTeX)

```
`a' is for ``apple"
\end
```

The same as LaTeX case (/wiki/Quotes#LaTeX), even though one should say the opposite.

The `` and '' in TeX (plainTeX, LaTeX and many more) are just examples of ligatures.

Pop11 (/wiki/Category:Pop11)

In Pop11 charaters literals are written in inverted quotes (backticks)

`a` ;;; charater a

String are written in quotes

```
'a'    ;;; string consisting of single character
```

Backslash is used to insert special charaters into strings:

```
'\'\n' ;;; string consisting of quote and newline
```

PowerShell (/wiki/Category:PowerShell)

PowerShell makes no distinction between characters and strings. Single quoted strings do not interpolate variable contents but double quoted strings do. Also, escape sequences are quoted literally as separate characters within single quotes.

PowerShell here-strings begin with @' (or @") followed immediately by a line break and end with a line break followed by '@ (or "@). Escape sequences and variables are interpolated in @" quotes but not in @' quotes.

Prolog (/wiki/Category:Prolog)

Standard Prolog has no string types. It has atoms which can be formed in two ways, one of which is wrapping arbitrary text in single quotation marks:

```
'This is an "atom" and not a string.'
```

Such atoms can be (and are) treated as immutable strings in Prolog in many cases. Another string-like form wraps text in double quotation marks:

```
"This 'string' will fool you if you're in a standard Prolog environment."
```

While this appears as a string to non-Prolog users, it is in reality a linked list of integers with each node containing the integer value of the character (or for Unicode-capable systems, code point) at that location. For example:

```
?- [97, 98, 99] = "abc".
true (http://pauillac.inria.fr/~deransar/prolog/bips.html).
```

Individual character constants are special forms of integer (syntax sugar) using a 0' prefix:

```
?- 97 = 0'a.
true.
```

Works with: SWI Prolog (/wiki/SWI_Prolog) version 7.0
SWI-Prolog, beginning with version 7, introduced a new native string type. Unless options are specifically set by the user, character sequences wrapped in double quotes ar now a string data type. The older list-based version uses back quotes instead:

```
?- [97, 98, 99] = "abc".
false.
?- [97, 98, 99] = `abc`.
true (http://pauillac.inria.fr/~deransar/prolog/bips.html).
```

Also starting with SWI-Prolog version 7, quasiquotation became possible. While not exactly a string type directly, they can be (ab)used to give multi-line strings. More importantly, however, they permit special string handling to be embedded into Prolog code, in effect permitting entire other languages inside of Prolog to be used natively as per this example:

```
test_qq_odbc :-
        myodbc_connect_db(Conn),
        odbc_query(Conn, {|odbc||
select
 P.image,D.description,D.meta_keywords,C.image,G.description
from
 product P, product_description D, category C, category_description G, product_to_category J
where
 P.product_id=D.product_id and
 P.product_id=J.product_id and C.category_id=J.category_id and
 C.category_id=G.category_id
        |}, Row),
        writeln(Row).
```

In this example, the test_qq_odbc/0 predicate connects to an ODBC database and performs a query. The query is wrapped into a multi-line quasiquotation (beginning with { and ending with |}) that checks the syntax and security of the query, so not only is the query a multi-line string, it is a **checked** multiline string in this case.

PureBasic (/wiki/Category:PureBasic)

PureBasic supports char in ASCII and UNICODE as well as both dynamic and fixed length strings.

```
; Characters (*.c), can be ASCII or UNICODE depending on compiler setting
Define.c  AChar='A'
; defines as *.a it will be ASCII and *.u is always UNICODE
Define.a  A='b'
Define.u  U='水'

; Strings is defined as **.s or ending with '$'
Define.s  AStrion     ="String #1"
Define    BStrion.s   ="String #2"
Define    CString$    ="String #3"
; Fixed length stings can be defined if needed
Define XString.s{100} ="I am 100 char long!"

; '"' can be included via CHR() or its predefined constant
Define AStringQuotes$=Chr(34)+"Buu"+Chr(34)+" said the ghost!"
Define BStringQuotes$=#DOUBLEQUOTE$+"Buu"+#DOUBLEQUOTE$+" said yet a ghost!"
```

To dynamically detect the current sizes of a character, e.g. ASCI or UNICODE mode, StringByteLength() can be used.

```
Select StringByteLength("X")
  Case 1
    Print("ASCII-mode;  Soo, Hello world!")
  Case 2
    Print("UNICODE-mode; Soo, 您好世界!")
EndSelect
```

Python (/wiki/Category:Python)

Python makes no distinction between single characters and strings. One can use single or double quotes.

```
'c' == "c" # character
'text' == "text"
' " '
" ' "
'\x20' == ' '
u'unicode string'
u'\u05d0' # unicode literal
```

As shown in the last examples, Unicode strings are single or double quoted with a "u" or "U" prepended thereto.

Verbatim (a.k.a. "raw") strings are contained within either single or double quotes, but have an "r" or "R" prepended to indicate that backslash characters should NOT be treated as "escape sequences." This is useful when defining regular expressions as it avoids the need to use sequences like \\\\ (a sequence of four backslashes) in order to get one literal backslash into a regular expression string.

```
r'\x20' == '\\x20'
```

The Unicode and raw string modifiers can be combined to prefix a raw Unicode string. This **must** be done as "ur" or "UR" (not with the letters reversed as it: "ru").

Here-strings are denoted with triple quotes.

```
''' single triple quote '''
""" double triple quote """
```

The "u" and "r" prefixes can also be used with triple quoted strings.

Triple quoted strings can contain any mixture of double and single quotes as well as embedded newlines, etc. They are terminated by unescaped triple quotes of the same type that initiated the expression. They are generally used for "doc strings" and other multi-line string expressions --- and are useful for "commenting out" blocks of code.

Quackery (/wiki/Category:Quackery)

A character literal is denoted by the word `char`. The character is the first non-whitespace character following `char`.

A string literal is denoted by the word `$`. The string is delimited by the first non-whitespace character following `$`.

Character and string literals illustrated in the Quackery shell (REPL):

```
/O> char X emit
... char Y emit
... char Z emit cr
... $ "This is a 'string'." echo$ cr
... $ 'This is a "string" too.' echo$ cr
... $ ~This is one with "quotes" and 'apostrophes'.~ echo$ cr
... $ \Any non-whitespace character can be the delimiter.\ echo$ cr
...
XYZ
This is a 'string'.
This is a "string" too.
This is one with "quotes" and 'apostrophes'.
Any non-whitespace character can be the delimiter.
```

R (/wiki/Category:R)

R makes no distinction between characters and strings, and uses single and double quotes interchangeably, though double quotes are considered to be preferred. Verbatim strings are not supported. See ?Quotes (http://stat.ethz.ch/R-manual/R-patched/library/base/html/Quotes.html) for more information.

```
str1 <- "the quick brown fox, etc."
str2 <- 'the quick brown fox, etc.'
identical(str1, str2)   #returns TRUE
```

R also supports testing string literals with **==**, e.g.,

```
modestring <- 'row,col'
mode.vec <- unlist(strsplit(modestring, ','))
mode.vec[1] # "row"
mode.vec[2] # "col"
if (mode.vec[2] == 'col') { cat('Col!\n') } # Col! (with no quotes)
if (mode.vec[1] == "row") { cat('Row!\n') } # Row!
```

R also uses backticks, for creating non-standard variable names (amongst other things).

```
`a b` <- 4
`a b`   # 4
a b     # Error: unexpected symbol in "a b"
```

R will print different styles of single and double quote using sQuote and dQuote

```
options(useFancyQuotes=FALSE)
cat("plain quotes: ", dQuote("double"), "and", sQuote("single"), "\n")
```

returns

```
plain quotes:  "double" and 'single'
```

```
options(useFancyQuotes=TRUE)
cat("fancy quotes: ", dQuote("double"), "and", sQuote("single"), "\n")
```

returns

```
fancy quotes:  “double” and ‘single’
```

```
options(useFancyQuotes="TeX")
cat("fancy quotes: ", dQuote("double"), "and", sQuote("single"), "\n")
```

returns

```
 TeX quotes:  ``double'' and `single'
```

Racket (/wiki/Category:Racket)

Characters are specified as hash-backslash-character, sometime using a name for the character.

```
  #\a
  #\space
  #\return
```

Strings are double-quoted, and have most of the usual C-style escapes. To include a double-quote in strings, escape it with a backslash, and the same goes for doubly-escaped backslashes (leading to the usual regexp fun).

Racket source code is read as UTF-8 text so strings can include Unicode characters -- but the internal representation is UCS-4. This includes "\NNN" for octals and "\xHH" for hex and "\uHHHH" for higher characters. See the docs (https://docs.racket-lang.org/reference/reader.html#%28part._parse-string%29) for a complete specification.

Racket also has here strings, and a more sophisticated facility for text that includes interpolation-like features, which is described in the Here Document (https://rosettacode.org/wiki/Here_document#Racket) entry

Raku (/wiki/Category:Raku)

(formerly Perl 6) Unlike most languages that hardwire their quoting mechanisms, the quote mechanism in Raku is extensible, and all normal-looking quotes actually derive from a parent quoting language called Q via grammatical mixins, applied via standard Raku adverbial syntax. The available quote mixins, straight from current spec S02, are

```
Short       Long           Meaning
=====       ====           =======
:x          :exec          Execute as command and return results
:w          :words         Split result on words (no quote protection)
:ww         :quotewords    Split result on words (with quote protection)
:v          :val           Evaluate word or words for value literals
:q          :single        Interpolate \\, \q and \' (or whatever)
:qq         :double        Interpolate with :s, :a, :h, :f, :c, :b
:s          :scalar        Interpolate $ vars
:a          :array         Interpolate @ vars
:h          :hash          Interpolate % vars
:f          :function      Interpolate & calls
:c          :closure       Interpolate {...} expressions
:b          :backslash     Interpolate \n, \t, etc. (implies :q at least)
:to         :heredoc       Parse result as heredoc terminator
            :regex         Parse as regex
            :subst         Parse as substitution
            :trans         Parse as transliteration
            :code          Quasiquoting
:p          :path          Return a Path object (see S16 for more options
```

In any case, an initial `Q`, `q`, or `qq` may omit the initial colon to form traditional Perl quotes such as `qw//`. And Q can be used by itself to introduce a quote that has no escape at all except for the closing delimiter:

```
my $raw = Q'$@\@#)&!#';
```

Note that the single quotes there imply no single quoting semantics as they would in Perl 5. They're just the quotes the programmer happened to choose, since they were most like the raw quoting. Single quotes imply `:q` only when used as normal single quotes are, as discussed below. As in Perl 5, you can use any non-alphanumeric, non-whitespace characters for delimiters with the general forms of quoting, including matching bracket characters, including any Unicode brackets.

Using the definitions above, we can derive the various standard "sugar" quotes from Q, including:

```
Normal  Means
======  =====
q/.../  Q :q /.../
qq/.../ Q :qq /.../
'...'   Q :q /.../
"..."   Q :qq /.../
<...>   Q :q :w :v /.../
«...»   Q :qq :ww :v /.../
/.../   Q :regex /.../
quasi {...} Q :code {...}
```

The `:qq`-derived languages all give normal Perlish interpolation, but individual interpolations may be chosen or suppressed with extra adverbs.

Unlike in Perl 5, we don't use backticks as shorthand for what is now expressed as `qqx//` in Raku. (Backticks are now reserved for user-defined syntax.) Heredocs now hav no special << syntax, but fall out of the `:to` adverb:

```
say qq:to/END/;
    Your ad here.
    END
```

Indentation equivalent to the ending tag is automatically removed.

Backslash sequences recognized by `:b` (and hence `:qq`) include:

```
"\a"        # BELL
"\b"        # BACKSPACE
"\t"        # TAB
"\n"        # LINE FEED
"\f"        # FORM FEED
"\r"        # CARRIAGE RETURN
"\e"        # ESCAPE
"\x263a"    # ☺
"\o40"      # SPACE
"\0"        # NULL

"\cC"       # CTRL-C
"\c8"       # BACKSPACE
"\c[13,10]" # CRLF
"\c[LATIN CAPITAL LETTER A, COMBINING RING ABOVE]"
```

Leading `0` specifically does not mean octal in Perl 6; you must use `\o` instead.

Retro (/wiki/Category:Retro)

Strings begin with a single quote and end on space. Underscores are replaced with spaces.

ASCII characters are prefixed by a single dollar sign.

```
$c
'hello,_world!
'This_is_'a_string'
```

REXX (/wiki/Category:REXX)

There are two types of quotes used for REXX literals:

- " (sometimes called a double quote or quote)
- ' (sometimes called a single quote or apostrophe)

There is no difference between them as far as specifying a REXX literal.
You can double them (code two of them adjacent) to specify a quote within the string.

```
char1 = "A"
char2 = 'A'
str = "this is a string"
another = 'this is also a string'
escape1 = "that's it!"
escape2 = 'that''s it!'
```

Variable expansion is not possible within REXX literals.
Simply concatenate the string with the variable:

```
amount = 100
result = "You got" amount "points."
say result
```

Output:

```
You got 100 points.
```

It's also possible to express characters in hexadecimal notation in a string:

```
lf = '0A'x
cr = '0D'x

mmm = '01 02 03 34 ee'x
ppp = 'dead beaf 11112222 33334444 55556666 77778888 00009999 c0ffee'X

lang = '52455858'x     /*which is "REXX" on ASCII-computers.*/
```

Binary strings are also possible:

```
jjj = '01011011'B
jjj = '01011011'b
jjj = "0101 1011"b
jjj = '0101 1011 1111'b
longjjj = '11110000 10100001 10110010 11100011 11100100'B
```

Ring (/wiki/Category:Ring)

```
see 'This is a "quoted string"'
```

Ruby (/wiki/Category:Ruby)

Quotes that do not interpolate:

```
'single quotes with \'embedded quote\' and \\backslash'
%q(not interpolating with (nested) parentheses
and newline)
```

Quotes that interpolate:

```
a = 42
"double quotes with \"embedded quote\"\nnewline and variable interpolation: #{a} % 10 = #{a % 10}"
%Q(same as above)
%|same as above|
```

Heredocs

```
print <<HERE
With an unquoted delimiter, this interpolates:
a = #{a}
HERE
print <<-INDENTED
   This delimiter can have whitespace before it
   INDENTED
print <<'NON_INTERPOLATING'
This will not interpolate: #{a}
NON_INTERPOLATING
```

S-lang (/wiki/Category:S-lang)

S-Lang supports character literals with single-quote apostrophe. These are normally limited to byte sized 0 thru 255 (ASCII, extended into 8 bits). Wide (unicode) character literals can be introduced with apostrophe `'\x{Hex}'`, for example `'\x{1D7BC}'` for math symbol sans-serif bold italic small sigma. Character literals are treated as Integer_Type data.

Double-quotes are used for strings, but these may not include literal newlines or NUL bytes without using backslash escapes. S-Lang 2.2 introduced verbatim strings using backtick. These may include newline literals, but other platform concerns can make NUL byte literals in the source code a tricky business. Both double-quote and back-quote allow a suffix:

- R for no backslash escapes (default for back-quoted strings)
- Q to force backslash escaping (default for double-quoted strings)
- B to produce a binary string, BString_Type
- $ to request dollar prefix variable substitutions within the given string.

```
% String literals
variable c, ch, s, b, r, v;

c = 'A';
ch = '\x{1d7bc}';

printf (https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Double quotes\n");
s = "this is a single line string with\t\tbackslash substitutions";
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)(s, stdout);
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)("\n", stdout);

s = "this is a single line string without\t\tbackslash substitutions"R;
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)(s, stdout);
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)("\n", stdout);

s = "string with backslash escaped newline \
takes up two lines in source, but no newline is in the string";
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)(s, stdout);
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)("\n", stdout);

printf (https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("\nBack quotes\n");
r = `this is a multi line string with
  backslash substitutions and \t(tabs)\t`Q;
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)(r, stdout);
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)("\n", stdout);

r = `this is a multi line string without
  backslash substitutions and \t(tabs)\t`;
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)(r, stdout);
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)("\n", stdout);

printf (https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("\nvariable substitution\n");
v = "variable substitution with $$c as $c"$;
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)(v, stdout);
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)("\n", stdout);

v = "no variable substitutions, $$c as $c";
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)(v, stdout);
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)("\n", stdout);

printf (https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("\nBString_Type\n");
b = "this is a binary string, NUL \0 \0 bytes allowed"B;
print(b);
% display of b will be stopped at the NUL byte using stdio streams
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)(b, stdout);
() = fputs (https://www.opengroup.org/onlinepubs/009695399/functions/fputs.html)("\n", stdout);
printf (https://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("strlen(b) is %d, bstrlen(b) is %d\n", strlen (https://w
ww.opengroup.org/onlinepubs/009695399/functions/strlen.html)(b), bstrlen(b));
```

Output:

```
prompt$ slsh strlit.sl
Double quotes
this is a single line string with               backslash substitutions
this is a single line string without\t\tbackslash substitutions
string with backslash escaped newline takes up two lines in source, but no newline is in the string

Back quotes
this is a multi line string with
  backslash substitutions and   (tabs)
this is a multi line string without
  backslash substitutions and \t(tabs)\t

variable substitution
variable substitution with $c as 65
no variable substitutions, $$c as $c

BString_Type
"this is a binary string, NUL \000 \000 bytes allowed"
this is a binary string, NUL
strlen(b) is 29, bstrlen(b) is 46
```

With substitutions, if a literal dollar sign or backquote is required, the character needs to be doubled. "$$c" is the literal "$c" when using dollar suffixed strings.

Backslash escapes include:

- `\"` -- double quote
- `\'` -- single quote
- `\\` -- backslash
- `\0` -- NUL byte

- `\a` -- bell character (ASCII 7)
- `\t` -- tab character (ASCII 9)
- `\n` -- newline character (ASCII 10)
- `\e` -- escape character (ASCII 27)
- `\xhh` -- byte expressed in HEXADECIMAL notation
- `\ooo` -- byte expressed in OCTAL notation
- `\dnnn` -- byte expressed in DECIMAL
- `\u{h..h}` -- the Unicode character U+h..h
- `\x{h..h}` -- the Unicode character U+h..h [modal]

Scala (/wiki/Category:Scala)

Character literals use single quotes marks:

```
val (https://scala-lang.org) c = 'c'
```

However, symbols are denoted with a single quote, so care must be taken not to confuse the two:

```
val (https://scala-lang.org) sym = 'symbol
```

Strings can use either double quotes, or three successive double quotes. The first allows special characters, the second doesn't:

```
scala> "newline and slash: \n and \\"
res5: java.lang.String =
newline and slash:
 and \

scala> """newline and slash: \n and \\"""
res6: java.lang.String = newline and slash: \n and \\
```

However, Unicode characters are expanded wherever they happen, even inside comments. So, for instance:

```
scala> val (https://scala-lang.org) uniquote = \u0022normal string"
uniquote: java.lang.String = normal string

scala> val insidequote = """an inside \u0022 quote"""
insidequote: java.lang.String = an inside " quote
```

Finally, on version 2.7, the triple-double-quoted string ends at the third consecutive quote, on version 2.8 it ends on the last quote of a series of at least three double-quotes

Scala 2.7

```
scala> val (https://scala-lang.org) error = """can't finish with a quote: """"
<console>:1: error: unterminated string
       val error = """can't finish with (https://scala-lang.org) a quote: """"
                                                ^
```

Scala 2.8

```
scala> val (https://scala-lang.org) success = """but it can on 2.8: """"
success: java.lang.String = but it can on 2.8: "
```

Scheme (/wiki/Category:Scheme)

Characters are specified using the "#\" syntax:

```
#\a
#\A
#\?
#\space
#\newline
```

Strings are contained in double quotes:

```
"Hello world"
```

Literal symbols, lists, pairs, etc. can be quoted using the quote syntax:

```
'apple
'(1 2 3) ; same as (list 1 2 3)
'()      ; empty list
'(a . b) ; same as (cons 'a 'b)
```

Seed7 (/wiki/Category:Seed7)

The type char (http://seed7.sourceforge.net/manual/types.htm#char) describes Unicode characters encoded with UTF-32. A character literal (http://seed7.sourceforge.net/manual/tokens.htm#Character_literals) is written as UTF-8 encoded Unicode character enclosed in single quotes.

```
var char: ch is 'z';
```

The type string (http://seed7.sourceforge.net/manual/types.htm#string) describes sequences of Unicode characters. The characters in the string use the UTF-32 encoding. A string literal (http://seed7.sourceforge.net/manual/tokens.htm#String_literals) is a sequence of UTF-8 encoded Unicode characters surrounded by double quotes.

```
var string: stri is "hello";
```

This means that 'z' and "z" are different. The former is a character while the latter is a string. Seed7 strings are not null terminated (they do not end with \0). They can contain any sequence of UNICODE (UTF-32) characters (including a \0). Empty strings are also allowed. In order to represent non-printable characters and certain printable characters the following escape sequences may be used.

audible alert	BEL	\a
backspace	BS	\b
escape	ESC	\e
formfeed	FF	\f
newline	NL (LF)	\n
carriage return	CR	\r
horizontal tab	HT	\t
vertical tab	VT	\v
backslash	(\)	\\
apostrophe	(')	\'
double quote	(")	\"
control-A		\A
...		
control-Z		\Z

A backslash followed by an integer literal and a semicolon is interpreted as character with the specified ordinal number. Note that the integer literal is interpreted decimal unless it is written as based integer (http://seed7.sourceforge.net/manual/types.htm#based_integer).

```
"Euro sign: \8364;"
```

There is also a possibility to break a string into several lines.

```
var string: example is "this is a string\
                       \ which continues in the next line\n\
                       \and contains a line break";
```

There is no built-in mechanism for expanding variables within strings.

Sidef (/wiki/Category:Sidef)

Quotes that do not interpolate:

```
'single quotes with \'embedded quote\' and \\backslash';
,unicode single quoted’;
%q(not interpolating with (nested) parentheses
and newline);
```

Quotes that interpolate:

```
var a = 42;
"double \Uquotes\E with \"embedded quote\"\nnewline and variable interpolation: #{a} % 10 = #{a % 10}";
„same as above”;
%Q(same as above);
```

Heredocs:

```
print <<EOT
Implicit double-quoted (interpolates):
a = #{a}
EOT

print <<"EOD"
Explicit double-quoted with interpolation:
a = #{a}
EOD

print <<'NON_INTERPOLATING'
This will not interpolate: #{a}
NON_INTERPOLATING
```

Slate (/wiki/Category:Slate)

Characters are specified using the $ syntax:

```
$a
$D
$8
$,
$\s
$\n
```

Strings are contained in single quotes, with backslash for escaping:

```
'Hello\'s the word.'
```

SQL (/wiki/Category:SQL)

String literals in SQL use single-quotation. There are no escapes, but you can double a `'` mark to make a single `'` in the text.

```
SELECT 'The boy said ''hello''.';
```

Standard ML (/wiki/Category:Standard_ML)

Characters are contained in the `#""` syntax:

```
- #"a";
val it = #"a" : char
```

Strings are contained in double quotes:

```
- "Hello world";
val it = "Hello world" : string
```

Strings may be split across lines and concatenated by having two backslashes around the newline and whitespace:

```
- "abc\
    \def";
val it = "abcdef" : string
```

Swift (/wiki/Category:Swift)

```
let you = "You"
let str1 = "\(you) can insert variables into strings."
let str2 = "Swift also supports unicode in strings ıʃƒəß´™¡à"
let str3 = "Swift also supports control characters \n\tLike this"
let str4 = "'" // '
let str5 = "\"" // "
println(str3)
```

Output:

```
Swift also supports control characters
        Like this
```

Swift 4 introduced multi-line string literals called long strings. Long strings are strings delimited by `"""triple quotes"""` that can contain newlines and individual `"`

characters without the need to escape them.

```
let author = "Author"
let xml == """
    <?xml version="1.0"?>
    <catalog>
        <book id="bk101" empty="">
            <author>\(author)</author>
            <title>XML Developer's Guide</title>
            <genre>Computer</genre>
            <price>44.95</price>
            <publish_date>2000-10-01</publish_date>
            <description>An in-depth look at creating applications with XML.</description>
        </book>
    </catalog>
    """

println(xml)
```

Output:

```
<?xml version="1.0"?>
<catalog>
    <book id="bk101" empty="">
        <author>Author</author>
        <title>XML Developer's Guide</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publish_date>2000-10-01</publish_date>
        <description>An in-depth look at creating applications with XML.</description>
    </book>
</catalog>
```

To allow free formatting of the literal an indentation stripping operation is applied whereby any whitespace characters in front of the closing delimiter are removed from each of the lines in the literal. As part of this process any initial linefeed is also removed. This allows the developer to paste literal content directly into the string without modification.

Tcl (/wiki/Category:Tcl)

Tcl makes no distinction between single characters and strings.

Double quotes allow command and variable interpolation:

```
set str "This is Tcl $::tcl_version\tIt is [clock format [clock seconds]]"
puts $str ;# ==> This is Tcl 8.5        It is Mon Apr 06 16:49:46 EDT 2009
```

Braces prevent interpolation

```
set str {This is Tcl $::tcl_version\tIt is [clock format [clock seconds]]}
puts $str ;# ==> This is Tcl $::tcl_version\tIt is [clock format [clock seconds]]
```

TI-89 BASIC (/wiki/Category:TI-89_BASIC)

Double quotes enclose strings, e.g. `"Hello Rosetta Code"`. There are no escape characters. Quotes in strings are doubled: `"This > "" < is one double-quote."`

TOML (/mw/index.php?title=Category:TOML&action=edit&redlink=1)

There are four ways to express strings: basic, multi-line basic, literal, and multi-line literal. All strings must contain only valid UTF-8 characters.

Basic strings are surrounded by quotation marks. Any Unicode character may be used except those that must be escaped: quotation mark, backslash, and the control characters other than tab (U+0000 to U+0008, U+000A to U+001F, U+007F).

```
str = "I'm a string. \"You can quote me\". Name\tJos\u00E9\nLocation\tSF."
```

Multi-line basic strings are surrounded by three quotation marks on each side and allow newlines. A newline immediately following the opening delimiter will be trimmed. A other whitespace and newline characters remain intact.

```
str1 = """
Roses are red
Violets are blue"""
```

When the last non-whitespace character on a line is a `\` ("line ending backslash"), it will be trimmed along with all whitespace (including newlines) up to the next non-whitespace character or closing delimiter. All of the escape sequences that are valid for basic strings are also valid for multi-line basic strings.

```
# The following strings are byte-for-byte equivalent:
str1 = "The quick brown fox jumps over the lazy dog."

str2 = """
The quick brown \


  fox jumps over \
    the lazy dog."""

str3 = """\
       The quick brown \
       fox jumps over \
       the lazy dog.\
       """
```

Literal strings are surrounded by single quotes and do not support escaping. This means that there is no way to write a single quote in a literal string. Like basic strings, the must appear on a single line.

```
# What you see is what you get.
winpath  = 'C:\Users\nodejs\templates'
winpath2 = '\\ServerX\admin$\system32\'
quoted   = 'Tom "Dubs" Preston-Werner'
regex    = '<\i\c*\s*>'
```

Multi-line literal strings are surrounded by three single quotes on each side and allow newlines. Like literal strings, there is no escaping whatsoever. A newline immediately following the opening delimiter will be trimmed. All other content between the delimiters is interpreted as-is without modification. One or two single quotes are allowed anywhere within a multi-line literal string, but sequences of three or more single quotes are not permitted.

```
regex2 = '''I [dw]on't need \d{2} apples'''
lines  = '''
The first newline is
trimmed in raw strings.
   All other whitespace
   is preserved.
'''
```

TUSCRIPT (/wiki/Category:TUSCRIPT)

```
$$ MODE TUSCRIPT,{}
s1=*
DATA "string"
s2=*
DATA + "double" quotes
s3=*
DATA + 'single' quotes
s4=*
DATA + "double" + 'single' quotes
show=JOIN(s1," ",s2,s3,s4)
show=JOIN(show)
PRINT show
```

Output:

```
"string" + "double" quotes + 'single' quotes + "double" + 'single' quotes
```

UNIX Shell (/wiki/Category:UNIX_Shell)

Works with: Bourne Shell (/wiki/Bourne_Shell)
Works with: bash (/wiki/Bash)
The Unix shell supports several types of quotation marks:

- singlequotes - for literal string quotation
- doublequotes - for interpolated string quotation
- backticks - Used to capture the output from an external program

Quotation marks within a literal String

It is possible to place singlequote characters within a string enclosed with doublequotes and to put doublequote characters in a string enclosed within singlequotes:

```
echo "The boy said 'hello'."
echo 'The girl said "hello" too.'
```

We can also use an escapesequence to put doublequote characters in an interpolated string:

```
print "The man said \"hello\".";
```

Here documents

The shell supports the use of here documents for the passing of quoted text as input into a command. Here documents cannot be used to represent literal strings as an expression for variable assignment.

```
cat << END
1, High Street,
SMALLTOWN,
West Midlands.
WM4 5HD.
END
```

Ursala (/wiki/Category:Ursala)

Single characters are denoted with a back quote.

```
a = `x
```

Unprintable character constants can be expressed like this.

```
cr = 13%cOi&
```

Strings are enclosed in single forward quotes.

```
b = 'a string'
```

A single quote in a string is escaped by another single quote.

```
c = 'Hobson''s choice'
```

Multi-line strings are enclosed in dash-brackets.

```
d =

-[this is a list
of strings]-
```

Dash-bracket enclosed text can have arbitrary nested unquoted expressions, provided they evaluate to lists of character strings.

```
e = -[the front matter -[ d ]- the rest of it]-

f = -[text -[ d ]- more -[ e ]- text ]-
```

This notation can also be used for defining functions.

```
g "x" = -[ Dear -[ "x" ]- bla bla ]-
```

The double quotes aren't for character strings but dummy variables.

V (/wiki/Category:V)

A simple quoted string is of the form 'string' e.g

```
'hello world' puts
```

Vim Script (/wiki/Category:Vim_Script)

A string constant delimited by double quotes " may contain escape sequences like \n, \t, \123 (byte value in octal), \xab (byte value in hexadecimal), \\ (backslash), \" or \u12ff (character code in hexadecimal according to the current encoding).

If a string constant is delimited by single quotes ' all characters are taken as they are. In order to use a single quote inside a literal string it must be escaped with another single quote, i.e. two single quotes stand for one.

Strings must always end at the current line and characters are just strings of length one.

Visual Basic (/wiki/Category:Visual_Basic)

Works with: Visual Basic (/wiki/Visual_Basic) version 5
Works with: Visual Basic (/wiki/Visual_Basic) version 6
Works with: VBA (/wiki/VBA) version Access 97
Works with: VBA (/wiki/VBA) version 6.5
Works with: VBA (/wiki/VBA) version 7.1

```
   Debug.Print "Tom said, ""The fox ran away."""
   Debug.Print "Tom said, 'The fox ran away.'"
```

Output:

```
Tom said, "The fox ran away."
Tom said, 'The fox ran away.'
```

Visual Basic .NET (/wiki/Category:Visual_Basic_.NET)

Visual Basic only supports single-line strings. The only escape sequence supported is the double double-quote (""), which is translated into a single double-quote.

```
Dim s = "Tom said, ""The fox ran away."""
Result: Tom said, "The fox ran away."
```

WEB (/mw/index.php?title=Category:WEB&action=edit&redlink=1)

WEB supports single-quoted strings exactly like Pascal strings (duplicate a ' to represent a literal ').

Double-quoted strings are "pool strings"; they are replaced by a numeric literal by the preprocessor, and placed into a string pool file (duplicate a " to represent a literal ").

Pool strings consisting of exactly one character represent numbers 0-255 according to their ASCII character code.

Wren (/wiki/Category:Wren)

In Wren a string is an immutable array of bytes. They are usually interpreted as UTF-8 but don't have to be and invalid UTF-8 sequences are permitted. A string can also include the zero byte (\0) which is not interpreted as a string terminator as would be the case in C.

All strings are instances of the built-in String class and there is no separate Character class as such. Characters are simply strings consisting of a single byte or Unicode code point (1 to 4 bytes).

Ordinary string literals must be surrounded in double quotes and support the following escape characters:

Character	Meaning
\0	The NUL byte: 0
\"	A double quote character
\\	A backslash
\%	A percent sign (see below)
\a	Alarm beep
\b	Backspace
\e	ESC character
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\xhh	A single byte with hex value '0xhh'
\uhhhh	A Unicode code point within the basic multilingual plane
\Uhhhhhhhh	Any Unicode code point including emojis

Ordinary string literals also allow interpolation. If you have a percent sign (%) followed by a parenthesized expression, the expression is evaluated and can be arbitrarily complex. Consequently, if you need to include a normal % character in a string literal, you have to use the escaped form \%.

From v0.4.0 Wren also supports *raw* string literals. These are any text surrounded by triple double quotes, """, and are interpreted verbatim i.e. any control codes and/or interpolations are not processed as such. They can include single or double double quotes without problem.

```
var s = "abc123"
var t = "abc\t123\%"
var u = "\U0001F64A\U0001F680"
var v = "%("abc" * 3)"
var w = """a"bc""def\n%(v)"""

System.print([s, t, u, v, w])
```

Output:

```
[abc123, abc    123%, 🙊🚀, abcabcabc, a"bc""def\n%(v)]
```

Xojo (/wiki/Category:Xojo)

Xojo only supports single-line strings. The only escape sequence supported is the double double-quote (""), which is translated into a single double-quote.

```
Dim s As String = "Tom said, ""The fox ran away."""
Result: Tom said, "The fox ran away."
```

XPL0 (/wiki/Category:XPL0)

```
The literal value of a character is specified by preceding it with a
caret. For example, this assigns the ASCII value of A (hex 41) to Ch:

Ch:= ^A;

Strings are one-dimensional byte arrays.
For example, this assigns the address of the string enclosed in quote marks to Str:

Str:= "Hello, world!";

The ASCII code for each character is stored in sequential bytes.
By default strings are terminated with the most significant bit set
on the last character.
The exclamation point would be stored as hex A1.
Strings can also be terminated with a zero byte (like in the C language).
If the command:

string 0;

occurs anywhere before a string is set up then it will have an extra zero
byte at the end.

A quote mark can be included in a string by preceding it with a caret.
Carets are also included this way.
For example:

"^"^^^" is a ^"caret^""

results in:

"^" is a "caret"

Carets can also be used to specify control characters.
For example, this is escape E (hex 1B C5):

"^[E"

Strings can any length and can span lines, for example:

"Hello,
world!"

A carriage return (hex 0D) and line feed (hex 0A) are in the middle.

Strings are output to various devices (such as the console screen or
printer) with the Text intrinsic.
```

XSLT (/wiki/Category:XSLT)

XSLT is based on XML, and so can use either " or ' to delimit strings. Since XML attribute values are defined using double-quotes, one must use single-quotes for string literals within attributes.

```
<xsl:if test="starts-with(@name, 'Mr.')">Mister</xsl:if>
```

Double and single quote characters may also be escaped with XML entities: " and ' respectively.

zkl (/wiki/Category:Zkl)

Interpreted string: "hoho". Raw string: 0'|hoho| where | is user choosen. \b, \f, \n, \r, \t, \e escapes are supported. Two adjacent strings are treated as one: "foo" 0'~bar~ --> "foobar". No variable expansion.

here-strings:

```
text:=
0'|foo|
"bar\n";
```

Output:

```
"foobar\n"
```

```
n:=7; text:=String(
   "foo = ",3,"\n"
   "bar=",n,"\n"
);
```

Output:

```
text = "foo = 3\nbar=7"
```

Categories (/wiki/Special:Categories): Programming Tasks (/wiki/Category:Programming_Tasks) | Basic language learning (/wiki/Category:Basic_language_learning) | String manipulation (/wiki/Category:String_manipulation) | Syntax elements (/wiki/Category:Syntax_elements) | 11l (/wiki/Category:11l) | Ada (/wiki/Category:Ada) | Aime (/wiki/Category:Aime) | ALGOL 68 (/wiki/Category:ALGOL_68) | ALGOL W (/wiki/Category:ALGOL_W) | ARM Assembly (/wiki/Category:ARM_Assembly) | Arturo (/wiki/Category:Arturo) | AutoHotkey (/wiki/Category:AutoHotkey) | AWK (/wiki/Category:AWK) | Axe (/wiki/Category:Axe) | BASIC (/wiki/Category:BASIC) | Applesoft BASIC (/wiki/Category:Applesoft_BASIC) | IS-BASIC (/wiki/Category:IS-BASIC) | BASIC256 (/wiki/Category:BASIC256) | ZX Spectrum Basic (/wiki/Category:ZX_Spectrum_Basic) | BBC BASIC (/wiki/Category:BBC_BASIC) | Bc (/wiki/Category:Bc) | Befunge (/wiki/Category:Befunge) | Bracmat (/wiki/Category:Bracmat) | C (/wiki/Category:C) | C sharp (/wiki/Category:C_sharp) | C++ (/wiki/Category:C%2B%2B) | Clojure (/wiki/Category:Clojure) | COBOL (/wiki/Category:COBOL) | Common Lisp (/wiki/Category:Common_Lisp) | D (/wiki/Category:D) | Delphi (/wiki/Category:Delphi) | DWScript (/wiki/Category:DWScript) | Dyalect (/wiki/Category:Dyalect) | Déjà Vu (/wiki/Category:D%C3%A9j%C3%A0_Vu) | E (/wiki/Category:E) | Ela (/wiki/Category:Ela) | Elena (/wiki/Category:Elena) | Elixir (/wiki/Category:Elixir) | Emacs Lisp (/wiki/Category:Emacs_Lisp) | Erlang (/wiki/Category:Erlang) | Factor (/wiki/Category:Factor) | Forth (/wiki/Category:Forth) | Fortran (/wiki/Category:Fortran) | FreeBASIC (/wiki/Category:FreeBASIC) | Friendly interactive shell (/wiki/Category:Friendly_interactive_shell) | FurryScript (/mw/index.php?title=Category:FurryScript&action=edit&redlink=1) | GAP (/wiki/Category:GAP) | Gecho (/wiki/Category:Gecho) | Go (/wiki/Category:Go) | Groovy (/wiki/Category:Groovy) | Haskell (/wiki/Category:Haskell) | HicEst (/wiki/Category:HicEst) | Icon (/wiki/Category:Icon) | Unicon (/wiki/Category:Unicon) | IDL (/wiki/Category:IDL) | Inform 7 (/wiki/Category:Inform_7) | J (/wiki/Category:J) | Java (/wiki/Category:Java) | JavaScript (/wiki/Category:JavaScript) | Jq (/wiki/Category:Jq) | JSON (/wiki/Category:JSON) | Julia (/wiki/Category:Julia) | Kotlin (/wiki/Category:Kotlin) | LabVIEW (/wiki/Category:LabVIEW) | Lasso (/wiki/Category:Lasso) | LaTeX (/wiki/Category:LaTeX) | Liberty BASIC (/wiki/Category:Liberty_BASIC) | Lingo (/wiki/Category:Lingo) | Lisaac (/wiki/Category:Lisaac) | LiveCode (/wiki/Category:LiveCode) | Logo (/wiki/Category:Logo) | Lua (/wiki/Category:Lua) | M2000 Interpreter (/wiki/Category:M2000_Interpreter) | M4 (/wiki/Category:M4) | Maple (/wiki/Category:Maple) | Mathematica (/wiki/Category:Mathematica) | Wolfram Language (/wiki/Category:Wolfram_Language) | MATLAB (/wiki/Category:MATLAB) | Maxima (/wiki/Category:Maxima) | Metafont (/wiki/Category:Metafont) | ML/I (/wiki/Category:ML/I) | Modula-3 (/wiki/Category:Modula-3) | MUMPS (/wiki/Category:MUMPS) | Nemerle (/wiki/Category:Nemerle) | Nim (/wiki/Category:Nim) | OASYS Assembler (/wiki/Category:OASYS_Assembler) | Objeck (/wiki/Category:Objeck) | Objective-C (/wiki/Category:Objective-C) | OCaml (/wiki/Category:OCaml) | Octave (/wiki/Category:Octave) | Oforth (/wiki/Category:Oforth) | Oz (/wiki/Category:Oz) | PARI/GP (/wiki/Category:PARI/GP) | Pascal (/wiki/Category:Pascal) | Perl (/wiki/Category:Perl) | Phix (/wiki/Category:Phix) | Phix/basics (/wiki/Category:Phix/basics) | PHP (/wiki/Category:PHP) | PicoLisp (/wiki/Category:PicoLisp) | Pike (/wiki/Category:Pike) | PL/I (/wiki/Category:PL/I) | PlainTeX (/wiki/Category:PlainTeX) | Pop11 (/wiki/Category:Pop11) | PowerShell (/wiki/Category:PowerShell) | Prolog (/wiki/Category:Prolog) | PureBasic (/wiki/Category:PureBasic) | Python (/wiki/Category:Python) | Quackery (/wiki/Category:Quackery) | R (/wiki/Category:R) | Racket (/wiki/Category:Racket) | Raku (/wiki/Category:Raku) | Retro (/wiki/Category:Retro) | REXX (/wiki/Category:REXX) | Ring (/wiki/Category:Ring) | Ruby (/wiki/Category:Ruby) | S-lang (/wiki/Category:S-lang) | Scala (/wiki/Category:Scala) | Scheme (/wiki/Category:Scheme) | Seed7 (/wiki/Category:Seed7) | Sidef (/wiki/Category:Sidef) | Slate (/wiki/Category:Slate) | SQL (/wiki/Category:SQL) | Standard ML (/wiki/Category:Standard_ML) | Swift (/wiki/Category:Swift) | Tcl (/wiki/Category:Tcl) | TI-89 BASIC (/wiki/Category:TI-89_BASIC) | TOML (/mw/index.php?title=Category:TOML&action=edit&redlink=1) | TUSCRIPT (/wiki/Category:TUSCRIPT) | UNIX Shell (/wiki/Category:UNIX_Shell) | Ursala (/wiki/Category:Ursala) | V (/wiki/Category:V) | Vim Script (/wiki/Category:Vim_Script) | Visual Basic (/wiki/Category:Visual_Basic) | Visual Basic .NET (/wiki/Category:Visual_Basic_.NET) | WEB (/mw/index.php?title=Category:WEB&action=edit&redlink=1) | Wren (/wiki/Category:Wren) | Xojo (/wiki/Category:Xojo) | XPL0 (/wiki/Category:XPL0) | XSLT (/wiki/Category:XSLT) | Zkl (/wiki/Category:Zkl)

This page was last modified on 27 June 2021, at 16:52.
Content is available under GNU Free Documentation License 1.2 (https://www.gnu.org/licenses/fdl-1.2.html) unless otherwise noted.

(https://www.gnu.org/licenses/fdl-1.2.html)
(//www.mediawiki.org/)

(https://www.semantic-mediawiki.org/wiki/Semantic_MediaWiki)