Wolfram *Mathematica*® Tutorial Collection

ADVANCED STRING PATTERNS

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Content authored by:
Oyvind Tafjord

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Introduction

The general symbolic string patterns in *Mathematica* allow you to perform powerful string manipulation efficiently. What follows discusses the details of string patterns, including usage and implementation notes. The emphasis is on issues not mentioned elsewhere in the help system.

At the heart of *Mathematica* is a powerful language for describing patterns in general expressions. This language is used in function definitions, substitutions, and searches, with constructs like x_, a | b, x .., and so on.

```
In[1]:= MatchQ[{a, b, c, d}, {___, x_, x_, ___}]
Out[1]= False

In[2]:= MatchQ[{a, b, c, c, d}, {___, x_, x_, ___}]
Out[2]= True

In[3]:= Cases[{a, 3, 4, b, c, 8}, _Integer]
Out[3]= {3, 4, 8}
```

A *Mathematica* string pattern uses the same constructs to describe patterns in a text string. You can think of a string as a sequence of characters and apply the principles of general *Mathematica* patterns. In addition there are several useful string-specific pattern constructs.

```
In[4]:= StringMatchQ["abcd", ___ ~~ x_ ~~ x_ ~~ ___]
Out[4]= False

In[5]:= StringMatchQ["abccd", ___ ~~ x_ ~~ x_ ~~ ___]
Out[5]= True

In[6]:= StringCases["a34bc8", DigitCharacter]
Out[6]= {3, 4, 8}
```

Regular expressions can be used as an alternative way to specify string patterns. These tend to be more compact, but less readable.

```
In[7]:= StringMatchQ["abcd", RegularExpression[".*(.)\\1.*"]]
Out[7]= False

In[8]:= StringMatchQ["abccd", RegularExpression[".*(.)\\1.*"]]
Out[8]= True
```

```
In[9]:= StringCases["a34bc8", RegularExpression["\\d"]]
Out[9]= {3, 4, 8}
```

Here is a list of several functions that recognize string patterns.

`StringMatchQ["s", patt]`	test whether *s* matches *patt*
`StringFreeQ["s", patt]`	test whether *s* is free of substrings matching *patt*
`StringCases["s", patt]`	give a list of the substrings of *s* that match *patt*
`StringCases["s", lhs->rhs]`	replace each case of *lhs* by *rhs*
`StringPosition["s", patt]`	give a list of the positions of substrings that match *patt*
`StringCount["s", patt]`	count how many substrings match *patt*
`StringReplace["s", lhs->rhs]`	replace every substring that matches *lhs*
`StringReplaceList["s", lhs->rhs]`	give a list of all ways of replacing *lhs*
`StringSplit["s", patt]`	split *s* at every substring that matches *patt*
`StringSplit["s", lhs->rhs]`	split at *lhs*, inserting *rhs* in its place

Functions that support string patterns.

General String Patterns

A general string pattern is formed from pattern objects similar to the general pattern objects in *Mathematica*. To join several string pattern objects, use the `StringExpression` operator `~~`.

```
In[10]:= FullForm["a" ~~ _]
Out[10]//FullForm= StringExpression["a", Blank[]]
```

`StringExpression` is closely related to `StringJoin`, except nonstrings are allowed and lists are not flattened. For pure strings, they are equivalent.

```
In[11]:= "aa" ~~ "bbb" ~~ "c"
Out[11]= aabbbc
```

The list of objects that can appear in a string pattern closely matches the list for ordinary *Mathematica* patterns. In terms of string patterns, a string is considered a sequence of characters, that is, "abc" can be thought of as something like `String[a, b, c]`, to which the ordinary pattern constructs apply.

The following objects can appear in a symbolic string pattern.

"*string*"	a literal string of characters
`_`	any single character
`__`	any substring of one or more characters
`___`	any substring of zero or more characters
x`_`, *x*`__`, *x*`___`	substrings given the name *x*
x:*pattern*	pattern given the name *x*
pattern..	pattern repeated one or more times
pattern...	pattern repeated zero or more times
{$patt_1$, $patt_2$, …} or $patt_1 \mid patt_2 \mid \ldots$	a pattern matching at least one of the $patt_i$
patt/;*cond*	a pattern for which *cond* evaluates to `True`
pattern?*test*	a pattern for which *test* yields `True` for each character
`Whitespace`	a sequence of whitespace characters
`NumberString`	the characters of a number
`DatePattern`[*spec*]	the characters of a date
charobj	an object representing a character class (see below)
`RegularExpression`["*regexp*"]	substring matching a regular expression
`StringExpression`[…]	an arbitrary string expression

The following represent classes of characters.

{c_1, c_2, …}	any of the "c_i"
`Characters`["$c_1 c_2$…"]	any of the "c_i"
`CharacterRange`["c_1","c_2"]	any character in the range "c_1" to "c_2"
`HexadecimalCharacter`	hexadecimal digit 0–9, a–f, A–F
`DigitCharacter`	digit 0–9
`LetterCharacter`	letter
`WhitespaceCharacter`	space, newline, tab or other whitespace character
`WordCharacter`	letter or digit
`Except`[*p*]	any character except ones matching *p*

The following represent positions in strings.

`StartOfString`	start of the whole string
`EndOfString`	end of the whole string
`StartOfLine`	start of a line
`EndOfLine`	end of a line
`WordBoundary`	boundary between word characters and others
`Except[WordBoundary]`	anywhere except a word boundary

The following determine which match will be used if there are several possibilities.

`Shortest[`*p*`]`	the shortest consistent match for *p*
`Longest[`*p*`]`	the longest consistent match for *p* (default)

Some nontrivial issues regarding these objects follow.

The _, __, and ___ wildcards match any characters including newlines. To match any character except newline (analogous to the "." in regular expressions), use `Except["\n"]`, `Except["\n"]..`, and `Except["\n"]...`.

```
In[12]:= StringCases["line1\nline2\n", __]
Out[12]= {line1
         line2
         }
```

```
In[13]:= StringCases["line1\nline2\n", Except["\n"]..]
Out[13]= {line1, line2}
```

```
In[14]:= StringCases["line1\nline2\n", RegularExpression[".+"]]
Out[14]= {line1, line2}
```

A list of patterns, such as `{"a", "b", "c"}`, is equivalent to a list of alternatives, such as `"a" | "b" | "c"`. This is convenient in that functions like `Characters` and `CharacterRange` can be used to specify classes of characters.

```
In[15]:= StringReplace["the cat in the hat", x : Characters["aeiou"] :> x <> x]
Out[15]= thee caat iin thee haat
```

When Condition (/;) is used, the patterns involved are treated as strings as far as the rest of *Mathematica* is concerned, so you need to use ToExpression in some cases.

```
In[16]:= StringCases["a13 a18 a41 a42",
          "a" ~~ x : DigitCharacter .. ~~ WordBoundary /; PrimeQ[ToExpression[x]] :> x]
Out[16]= {13, 41}
```

Similar to ordinary *Mathematica* patterns, the function in PatternTest (?) is applied to each individual character.

```
In[17]:= StringCases["125378132", __?(ToExpression[#] < 5 &)]
Out[17]= {12, 3, 132}
```

The Whitespace construct is equivalent to WhitespaceCharacter ...

```
In[18]:= StringReplace["13   \t 17 \n22   19", Whitespace -> ","]
Out[18]= 13,17,22,19
```

You can insert a RegularExpression object into a general string pattern.

```
In[19]:= StringCases["a13b12c17a32", "a" ~~ x : RegularExpression["\\d+"] :> x]
Out[19]= {13, 32}
```

This inserts a lookbehind constraint (see "Regular Expressions") to ensure that you only pick words preceded by "the ".

```
In[20]:= StringCases["the cat in the hat",
          RegularExpression["(?<=the )"] ~~ WordCharacter ..]
Out[20]= {cat, hat}
```

StringExpression objects can be nested.

```
In[21]:= StringCases["ba3a1a78a2b7ba9", "b" ~~ ("a" ~~ DigitCharacter) ..]
Out[21]= {ba3a1a7, ba9}
```

The Except construct for string patterns takes a single argument that should represent a single character or a class of single characters.

This deletes all nonvowel characters from the string.

```
In[22]:= StringReplace["the cat in the hat", Except[Characters["aeiou"]] -> ""]
Out[22]= eaiea
```

When trying to match patterns of variable length (such as __ and `patt ..`), the longest possible match is tried first by default. To force the matcher to try the shortest match first, you can wrap the relevant part of the pattern in `Shortest[ ]`.

```
In[23]:= StringCases["(ab) (cde)", "(" ~~ __ ~~ ")"]

Out[23]= {(ab) (cde)}
```

```
In[24]:= StringCases["(ab) (cde)", Shortest["(" ~~ __ ~~ ")"]]

Out[24]= {(ab), (cde)}
```

If for some reason you need a longest match within the short match, you can use `Longest`.

```
In[25]:= StringCases["(ab132cd)137(ef576gh)",
          Shortest["(" ~~ ___ ~~ x : DigitCharacter .. ~~ ___ ~~ ")"] :> x]

Out[25]= {1, 5}
```

```
In[26]:= StringCases["(ab132cd)(ef576gh)",
          Shortest["(" ~~ ___ ~~ Longest[x : DigitCharacter ..] ~~ ___ ~~ ")"] :> x]

Out[26]= {132, 576}
```

You could alternatively rewrite this pattern without use of `Longest`.

```
In[27]:= StringCases["(ab132cd)(ef576gh)",
          "(" ~~ Shortest[___] ~~ x : DigitCharacter .. ~~ Shortest[___] ~~ ")" :> x]

Out[27]= {132, 576}
```

Regular Expressions

The regular expression syntax follows the underlying Perl Compatible Regular Expressions (PCRE) library, which is close to the syntax of Perl. (See [1] for further information and documentation.) A regular expression in *Mathematica* is denoted by the head `RegularExpression`.

The following basic elements can be used in regular expression strings.

c	the literal character c
`.`	any character except newline
$[c_1c_2\ldots]$	any of the characters c_i
$[c_1\text{–}c_2]$	any character in the range c_1–c_2
$[\hat{}\,c_1c_2\ldots]$	any character except the c_i
$p*$	p repeated zero or more times
$p+$	p repeated one or more times
$p?$	zero or one occurrence of p
$p\{m,n\}$	p repeated between m and n times
$p*?$, $p+?$, $p??$	the shortest consistent strings that match
$p*+$, $p++$, $p?+$	possessive match
$(p_1p_2\ldots)$	strings matching the sequence p_1, p_2, …
$p_1 \mid p_2$	strings matching p_1 or p_2

The following represent classes of characters.

`\\d`	digit 0–9
`\\D`	nondigit
`\\s`	space, newline, tab or other whitespace character
`\\S`	non-whitespace character
`\\w`	word character (letter, digit or _)
`\\W`	nonword character
[[:*class*:]]	characters in a named class
[^[:*class*:]]	characters not in a named class

The following named classes can be used: *alnum*, *alpha*, *ascii*, *blank*, *cntrl*, *digit*, *graph*, *lower*, *print*, *punct*, *space*, *upper*, *word*, and *xdigit*.

The following represent positions in strings.

`^`	the beginning of the string (or line)
`$`	the end of the string (or line)
`\\A`	the beginning of the string
`\\z`	the end of the string
`\\Z`	the end of the string (allowing for a single newline character first)
`\\b`	word boundary
`\\B`	anywhere except a word boundary

The following set options for all regular expression elements that follow them.

`(?i)`	treat uppercase and lowercase as equivalent (ignore case)
`(?m)`	make `^` and `$` match start and end of lines (multiline mode)
`(?s)`	allow `.` to match newline
`(?x)`	disregard all whitespace and treat everything between "#" and "\n" as comments
`(?-\#c)`	unset options

The following are lookahead/lookbehind constructs.

`(?=`*p*`)`	the following text must match *p*
`(?!`*p*`)`	the following text cannot match *p*
`(?<=`*p*`)`	the preceding text must match *p*
`(?<!`*p*`)`	the preceding text cannot match *p*

Discussion of a few issues regarding regular expressions follows.

This looks for runs of word characters of length between 2 and 4.

```
In[28]:= StringCases["a bb ccc dddd eeeee", RegularExpression["\\b\\w{2,4}\\b"]]
Out[28]= {bb, ccc, dddd}
```

With the possessive "+" quantifier, as many characters as possible are grabbed by the matcher, and no characters are given up, even if the rest of the patterns require it.

```
In[29]:= StringCases["a2 b6", RegularExpression["\\w+\\d"]]
Out[29]= {a2, b6}
```

```
In[30]:= StringCases["a2 b6", RegularExpression["\\w++\\d"]]
Out[30]= {}
```

```
In[31]:= StringCases["a2 b6", RegularExpression["\\D++\\d"]]
Out[31]= {a2,  b6}
```

`[[:`*xdigit*`:]]` corresponds to characters in a hexadecimal number.

```
In[32]:= StringCases["ff, 13, 1a3, xyz, 3b", RegularExpression["[[:xdigit:]]+"]]
Out[32]= {ff, 13, 1a3, 3b}
```

The complete list of characters that need to be escaped in a regular expression consists of `.`, `\`, `?`, `(`, `)`, `{`, `}`, `[`, `]`, `^`, `$`, `*`, `+`, and `|`. For instance, to write a literal period, use `"\\."` and to write a literal backslash, use `"\\\\"`.

Inside a character class `"[...]"`, the complete list of escaped characters is `^`, `-`, `\`, `[`, and `]`.

By default, ^ and $ match the beginning and end of the string, respectively. In multiline mode, these match the beginning/end of lines instead.

```
In[33]:= StringCases["line1\nline2", RegularExpression["^.*"]]
Out[33]= {line1}
```

```
In[34]:= StringCases["line1\nline2", RegularExpression["(?m)^.*"]]
Out[34]= {line1, line2}
```

In multiline mode, \\ A and \\ Z can be used to denote the beginning and end of the string.

```
In[35]:= StringCases["line1\nline2", RegularExpression["(?m)\\A.*"]]
Out[35]= {line1}
```

The `(?x)` modifier allows you to add whitespace and comments to a regular expression for readability.

```
In[36]:= StringCases["12.45   bc58.11",
          RegularExpression["\<(?x)
         \\d+  \\.  #remember to escape the period
         \\d+\>"]]
Out[36]= {12.45, 58.11}
```

Named subpatterns are achieved by surrounding them with parentheses (*subpatt*); they then become numbered subpatterns. The number of a given subpattern counts the opening parenthesis, starting from the start of the pattern. You can refer to these subpatterns using `\\` *n* for the *n*th pattern later in the pattern, or by "$*n*" in the right-hand side of a rule. "$0" refers to all of the matched pattern.

```
In[37]:= StringCases["a1b6a3b3a3c3a8b8", RegularExpression["(a(\\d))b\\2"]]
Out[37]= {a3b3, a8b8}
```

```
In[38]:= StringCases["a1b6a3b3a3c3a8b8",
          RegularExpression["(a(\\d))b\\2"] → {"$0", "$1", "Number:$2"}]
Out[38]= {{a3b3, a3, Number:3}, {a8b8, a8, Number:8}}
```

If you need a literal $ in this context (when the head of the left-hand side is `RegularExpression`), you can escape it by using backslashes (for example, "\\$2").

```
In[39]:= StringCases["a1b6a3b3a3c3a8b8",
          RegularExpression["(a(\\d))b\\2"] → {"$0", "$1", "Number:$2", "Literal:\\$2"}]
Out[39]= {{a3b3, a3, Number:3, Literal:$2}, {a8b8, a8, Number:8, Literal:$2}}
```

If you happen to need a single literal backslash followed by a literal $ under these circumstances, you need to be a bit tricky and split into two strings temporarily.

```
In[40]:= StringCases["a1b6a3b3a3c3a8b8", RegularExpression["(a(\\d))b\\2"] :>
           {"$0", "$1", "Number:$2", "Literal:\\" <> "\\$2"}]
Out[40]= {{a3b3, a3, Number:3, Literal:\$2}, {a8b8, a8, Number:8, Literal:\$2}}
```

If you need to group a part of the pattern, but you do not want to count the group as a numbered subpattern, you can use the `(?:patt)` construct.

```
In[41]:= StringCases["a11b16c22b77", RegularExpression["(?:a|b)(\\d)\\1"]]
Out[41]= {a11, b77}
```

Lookahead and lookbehind patterns are used to ensure a pattern is matched without actually including that text as part of the match.

This picks out words following the string "the ".

```
In[42]:= StringCases["the cat in the hat", RegularExpression["(?<=the )\\w+"]]
Out[42]= {cat, hat}
```

This tries to pick out all even numbers in the string, but it will find matches that include partial numbers.

```
In[43]:= StringCases["a23b42c63d80, 123",
          x : RegularExpression["\\d+"] /; Mod[ToExpression[x], 2] == 0]
Out[43]= {2, 42, 6, 80, 12}
```

Using lookbehind/lookahead, you can ensure that the characters before/after the match are not digits (note that the lookbehind test is superfluous in this particular case).

```
In[44]:= StringCases["a23b42c63d80, 123",
          x : RegularExpression["(?<!\\d)\\d+(?!\\d)"] /; Mod[ToExpression[x], 2] == 0]
Out[44]= {42, 80}
```

RegularExpression versus StringExpression

There is a close correspondence between the various pattern objects that can be used in general symbolic string patterns and in regular expressions. Here is a list of examples of patterns written as regular expressions and as symbolic string patterns.

regular expression	*general string pattern*	*explanation*
"abc"	"abc"	the literal string "abc"
"."	Except["\n"]	any character except newline
"(?s)."	_	any character
"(?s).+"	__	one or more characters (greedy)
"(?s).+?"	Shortest[__]	one or more characters (nongreedy)
"(?s).*"	___	zero or more characters
".*"	Except["\n"]...	zero or more characters (except newlines)
"a?b"	"a"|""~~"b"	zero or one "a" followed by a "b" (that is, "b" or "ab")

`"[abef]"`	`Characters["abef"]`	any of the characters "a", "b", "e", or "f"
`"[abef]+"`	`Characters["abef"]..`	one or more of the characters "a", "b", "e", or "f"
`"[a-f]"`	`CharacterRange["a","f"]`	any character in the range between "a" and "f"
`"[^abef]"`	`Except[Characters["abef"]]`	any character except the characters "a", "b", "e", or "f"
`"ab|efg"`	`"ab"|"efg"`	match the strings "ab" or "efg"
`"(ab|ef)gh"` or `"(?:ab|ef)gh"`	`("ab"|"ef")~~"gh"`	"ab" or "ef" followed by "gh" (that is, "abgh" or "efgh")
`"\\s"`	`WhitespaceCharacter`	any whitespace character
`"\\s+"`	`Whitespace`	one or more characters of whitespace
`"(a|b)\\1"`	`x:"a"|"b"~~x_`	this will match either "aa" or "bb"
`"\\d"`	`DigitCharacter`	any digit character
`"\\D"`	`Except[DigitCharacter]`	any nondigit character
`"\\d+"`	`DigitCharacter..`	one or more digit characters
`"\\w"`	`WordCharacter|"_"`	any digit, letter, or "_" character
`"[[:alpha:]]"`	`LetterCharacter`	any letter character
`"[^[:alpha:]]"`	`Except[LetterCharacter]`	any nonletter character
`"^abf"` or `"\\Aabc"`	`StartOfString~~"abf"`	the string "abf" at the start of the string
`"(?m)^abf"`	`StartOfLine~~"abf"`	the string "abf" at the start of a line
`"wxz$"` or `"wxz\\z"`	`"wxz"~~EndOfString`	the string "wxz" at the end of the string
`"wxz\\Z"`	`"wxz"~~"\n"|""~~EndOfString`	the string "wxz" at the end of the string or before newline at the end of the string

Pattern objects that can be used in general string patterns, but not in regular expressions, include conditions (`/;`) and pattern tests (`?`) that can access general *Mathematica* code during the match.

Some special constructs in regular expressions are not directly available in general string patterns. These include lookahead/lookbehinds and repeats of a given length. They can be embedded into a larger general string pattern by inserting a `RegularExpression` object.

String Manipulation Functions

The following discusses some particulars and subtleties in the various string manipulation functions (see the reference pages for more information on these functions).

StringMatchQ

`StringMatchQ` is used to check whether a whole string matches a certain pattern.

```
In[45]:= StringMatchQ["test", "t" ~~ __ ~~ "t"]
Out[45]= True
```

```
In[46]:= StringMatchQ["tester", "t" ~~ __ ~~ "t"]
Out[46]= False
```

`StringMatchQ` is special in that it also allows the metacharacters `*` and `@` to be entered as wildcards (for backward compatibility reasons). `*` is equivalent to `Shortest[___]` (`RegularExpression["(?s).*?"]`) and `@` is equivalent to `Except[CharacterRange["A", "Z"]]` (`RegularExpression["[^A-Z]"]`).

The following three patterns are therefore equivalent.

```
In[47]:= StringMatchQ["test", _ ~~ "e*"]
Out[47]= True
```

```
In[48]:= StringMatchQ["test", _ ~~ "e" ~~ Shortest[___]]
Out[48]= True
```

```
In[49]:= StringMatchQ["test", RegularExpression["(?s).e.*?"]]
Out[49]= True
```

Note that technically the appearance of `Shortest` does not make a difference here, since we are only looking for a possible match.

If you need to access parts of the string matched by subpatterns in the pattern, use `StringCases` instead.

`StringMatchQ` has a `SpellingCorrection` option for finding matches allowing for a small number of discrepancies. This only works for patterns consisting of a single literal string.

```
In[50]:= StringMatchQ["alpha", "alpa", SpellingCorrection → True]
Out[50]= True
```

StringFreeQ

`StringFreeQ` is used to check whether a string contains a substring matching the pattern. You cannot extract the matching substring; to do this you would use `StringCases`.

```
In[51]:= StringFreeQ["abcde", "b" ~~ __ ~~ "d"]
Out[51]= False
```

```
In[52]:= StringFreeQ["abcde", RegularExpression["b.*d"]]
Out[52]= False
```

StringCases

`StringCases` is a general purpose function for finding occurrences of patterns in a string, picking out subpatterns, and processing the results.

Find substrings matching a pattern.

```
In[53]:= StringCases["a1b2a26d15a42", "a" ~~ _]
Out[53]= {a1, a2, a4}
```

Pick apart the matching substring.

```
In[54]:= StringCases["a1b2a26d15a42", "a" ~~ x : DigitCharacter .. → x]
Out[54]= {1, 26, 42}
```

```
In[55]:= StringCases["a1b2a26d15a42", RegularExpression["a(\\d+)"] → "$1"]
Out[55]= {1, 26, 42}
```

Restrict the number of matches.

```
In[56]:= StringCases["a b c d e", LetterCharacter, 3]
Out[56]= {a, b, c}
```

You can use a list of rules.

```
In[57]:= StringCases["a13bF5b1Aa33", {"a" ~~ x : DigitCharacter .. → f1[x],
            "b" ~~ x : (DigitCharacter | CharacterRange["A", "F"]) .. → hex[x]}]
Out[57]= {f1[13], hex[F5], hex[1A], f1[33]}
```

You can also give a list of strings as the first argument for efficient processing of many strings (see "Tips and Tricks for Efficient Matching" for a discussion).

```
In[58]:= StringCases[{"cat", "in", "the", "hat"}, __ ~~ "t" ~~ EndOfString]
Out[58]= {{cat}, {}, {}, {hat}}

In[59]:= Flatten[%]
Out[59]= {cat, hat}
```

The Overlaps Option

The `Overlaps` option for `StringCases`, `StringPosition`, and `StringCount` deals with how the matcher proceeds after finding a match. It has three possible settings: `False`, `True`, or `All`. The default is `False` for `StringCases` and `StringCount`, while it is `True` for `StringPosition`.

With `Overlaps -> False`, the matcher continues the match testing at the character following the last matched substring.

```
In[60]:= StringCases["(a(b)c(d)", Shortest["(" ~~ __ ~~ ")"]]
Out[60]= {(a(b), (d)}
```

With `Overlaps -> True`, the matcher continues at the character following the first character of the last matched substring (when a single pattern is involved).

```
In[61]:= StringCases["(a(b)c(d)", Shortest["(" ~~ __ ~~ ")"], Overlaps → True]
Out[61]= {(a(b), (b), (d)}
```

With `Overlaps -> All`, the matcher keeps starting at the same position until no more new matches are found.

```
In[62]:= StringCases["(a(b)c(d)", Shortest["(" ~~ __ ~~ ")"], Overlaps → All]
Out[62]= {(a(b), (a(b)c(d), (b), (b)c(d), (d)}
```

If multiple patterns are given in a list, `Overlaps -> True` will cause the matcher to start at the same position once for each of the patterns before proceeding to the next character.

```
In[63]:= StringCases["(a(b)c(d)",
          {Shortest["(" ~~ __ ~~ ")"], Shortest["(" ~~ __ ~~ "("]}, Overlaps → True]
Out[63]= {(a(b), (a(, (b), (b)c(, (d)}
```

```
In[64]:= StringCases["(a(b)c(d)",
          {Shortest["(" ~~ __ ~~ ")"], Shortest["(" ~~ __ ~~ "("]}, Overlaps → False]
Out[64]= {(a(b), (d)}
```

Note that with `Overlaps -> True`, there can thus be a difference between specifying a list of patterns and using the alternatives operator (`|`).

```
In[65]:= StringCases["ab", {_, __}, Overlaps → True]
Out[65]= {a, ab, b, b}
```

```
In[66]:= StringCases["ab", _ | __, Overlaps → True]
Out[66]= {a, b}
```

StringPosition

`StringPosition` works much like `StringCases`, except the positions of the matching substrings are returned.

```
In[67]:= StringPosition["a1b2a26d15a42", "a" ~~ _]
Out[67]= {{1, 2}, {5, 6}, {11, 12}}
```

```
In[68]:= StringTake["a1b2a26d15a42", #] & /@ %
Out[68]= {a1, a2, a4}
```

The `Overlaps` option is `True` by default (see "The Overlaps Option" for more details on this option).

```
In[69]:= StringPosition["(a(b)c(d)", Shortest["(" ~~ __ ~~ ")"]]
Out[69]= {{1, 5}, {3, 5}, {7, 9}}
```

Note that even empty strings can be matches.

```
In[70]:=  StringPosition["abc", ___]
Out[70]=  {{1, 3}, {2, 3}, {3, 3}, {4, 3}}
```

StringCount

`StringCount` returns the number of matching substrings (which are found by `StringPosition` or `StringCases`). It is useful for cases with many matches where memory for storing all the substrings might be an issue.

```
In[71]:=  StringCount["abaababba", "a" ~~ ___ ~~ "b", Overlaps → All]
Out[71]=  12

In[72]:=  StringCases["abaababba", "a" ~~ ___ ~~ "b", Overlaps → All] // Length
Out[72]=  12
```

Note that `Overlaps -> False` is the default for `StringCount`.

StringReplace

`StringReplace` is used for substituting substrings matching the given patterns.

```
In[73]:=  StringReplace["abcde", {"a" → "A", "cd" → "XX"}]
Out[73]=  AbXXe
```

Named patterns can be used as strings on the right-hand side of the replacement rules. Note the use of `RuleDelayed` (:→) to avoid premature evaluation.

```
In[74]:=  StringReplace["this is a test", x : WordCharacter .. :→ StringReverse[x]]
Out[74]=  siht si a tset
```

When using regular expressions, it is convenient to remember that `"$0"` on the right-hand side refers to the whole matched substring.

```
In[75]:=  StringReplace["this is a test", RegularExpression["\\w+"] :→ StringReverse["$0"]]
Out[75]=  siht si a tset
```

You can limit the number of replacements made by specifying a third argument.

```
In[76]:= StringReplace["this is a test", x : WordCharacter .. :> StringReverse[x], 1]
Out[76]= siht is a test
```

Note that the replacement does not have to be a string. If the result is not a string, a `StringExpression` is returned.

```
In[77]:= StringReplace["some <b>bold</b> and <i>italics</i>.",
          Shortest["<" ~~ x___ ~~ ">"] :> Tag[x]]
Out[77]= some  ~~ Tag[b] ~~ bold ~~ Tag[/b] ~~  and  ~~ Tag[i] ~~ italics ~~ Tag[/i] ~~ .
```

```
In[78]:= InputForm[%]
Out[78]//InputForm= StringExpression["some ", Tag["b"], "bold", Tag["/b"], " and ", Tag["i"], "italics", Tag[
```

There is limited support for using the old `MetaCharacters` option in conjunction with general string patterns, but this option is deprecated and its use should be avoided.

StringReplaceList

`StringReplaceList` returns a list of strings where a *single* string replacement has been made in all possible ways.

```
In[79]:= StringReplaceList["abaac", "a" ~~ x_ :> ToUpperCase[x]]
Out[79]= {Baac, abAc, abaC}
```

If a list of strings is given as input, the output is a nested list of results.

```
In[80]:= StringReplaceList[{"abaac", "baaba"}, "a" ~~ x_ :> ToUpperCase[x]]
Out[80]= {{Baac, abAc, abaC}, {bAba, baBa}}
```

StringSplit

`StringSplit` is useful for splitting a string into many strings at delimiters matching a pattern. By default, the splits happen at runs of whitespace.

```
In[81]:= StringSplit["this is a test"]
Out[81]= {this, is, a, test}
```

For instance, to split a normal sentence into words, you need to also include punctuation in the delimiter.

```
In[82]:= StringSplit["A sentence: with commas, semicolons; etc...!?",
          Characters[":,;.!? "] ..]

Out[82]= {A, sentence, with, commas, semicolons, etc}
```

By default, empty strings at the beginning and the end of the result are removed.

```
In[83]:= StringSplit[":a:b:c:", ":"]

Out[83]= {a, b, c}
```

These can be included by specifying `All` as a third argument.

```
In[84]:= StringSplit[":a:b:c:", ":", All]

Out[84]= {, a, b, c, }
```

The third argument can also be a number giving the maximum number of strings to split into.

```
In[85]:= StringSplit["this is a test", Whitespace, 2]

Out[85]= {this, is a test}
```

This splits a string into individual lines.

```
In[86]:= StringSplit["line1\nthis is line 2\nline3", "\n"]

Out[86]= {line1, this is line 2, line3}
```

You can also split at patterns that match positions, such as `StartOfLine`. This keeps the newline characters in the result.

```
In[87]:= StringSplit["line1\nthis is line 2\nline3", StartOfLine]

Out[87]= {line1
         , this is line 2
         , line3}
```

You can keep the delimiters, or parts of the delimiters, in the output by using a rule as the second argument.

```
In[88]:= StringSplit["this is a test", " " → " "]

Out[88]= {this,  , is,  , a,  , test}
```

```
In[89]:= StringSplit["this is a test", " " → ":"]

Out[89]= {this, :, is, :, a, :, test}
```

```
In[90]:= StringSplit["the <tag1>first</tag1> and the <tag2>second</tag2>",
          Shortest["<" ~~ __ ~~ ">"]]

Out[90]= {the , first,  and the , second}
```

```
In[91]:= StringSplit["the <tag1>first</tag1> and the <tag2>second</tag2>",
          Shortest["<" ~~ x__ ~~ ">"] :> Tag[x]]

Out[91]= {the , Tag[tag1], first, Tag[/tag1],  and the , Tag[tag2], second, Tag[/tag2]}
```

You can give a list of patterns and rules as well; the delimiters matching the patterns will be left out of the result.

```
In[92]:= StringSplit["the <tag1>first</tag1> and the <tag2>second</tag2>",
           {Whitespace, Shortest["<" ~~ x__ ~~ ">"] :> Tag[x]}] // InputForm

Out[92]//InputForm= {"the", "", Tag["tag1"], "first", Tag["/tag1"], "", "and", "the", "", Tag["tag2"], "secon
```

For Perl Users

Overview

With the addition of general string patterns, *Mathematica* can be a powerful alternative to languages like Perl and Python for many general, everyday programming tasks. For people familiar with Perl syntax, and the way Perl does string manipulation, the following rough guide shows how to get similar functionality in *Mathematica*.

Here is an overview of the *Mathematica* functions involved in constructing Perl-like functions.

Perl construct	*Mathematica function*	*explanation*
`m/.../`	`StringFreeQ` or `StringCases`	match a string with a regular expression, possibly extracting subpatterns
`s/.../.../`	`StringReplace`	replace substrings matching a regular expression
`split (...)`	`StringSplit`	split a string at delimiters matching a regular expression
`tr/.../.../`	`StringReplace`	replace characters by other characters
`/i`	`IgnoreCase->True` or `"(?i)"`	case-insensitive modifier
`/s`	`"(?s)"`	force `"."` to match all characters (including newlines)
`/x`	`"(?x)"`	ignore whitespace and allow extended comments in regular expression
`/m`	`"(?m)"`	multiline mode ("^" and "$" match start/end of lines)

Following are some common Perl constructs in more detail.

m /.../

The match operator `m / regex /` tests whether a string contains a substring matching the `regex`. For simple matches of this sort in *Mathematica*, use `StringFreeQ`.

Here is a Perl snippet for testing whether a string contains a "b" somewhere after an "a".

```
$string = "sdakdb";
if ($string =~ m/a.*b/){
  print "Match!";
}
```

Here is a *Mathematica* version of the same test.

```
In[93]:= string = "sdakdb";
         If[! StringFreeQ[string, RegularExpression["a.*b"]], Print["Match!"]]

            Match!
```

If parts of the matched string need to be accessed later, using $1, $2, … in Perl, the best *Mathematica* function to use is normally `StringCases`.

Here is Perl code for extracting an error message.

```
$res = "ERROR = paper jam";
if ($res =~ m/ERROR = (.*)/){
  print "Hey, you should check the $1!";
}
```

Here is a *Mathematica* version.

```
In[95]:= res = "ERROR = paper jam";
         With[{test = StringCases[res, RegularExpression["ERROR = (.*)"] → "$1"]},
          If[test =!= {}, Print["Hey, you should check the ", test[[1]], "!"]]]

           Hey, you should check the paper jam!
```

Here is Perl code for extracting several subpatterns at once.

```
$date = "88/6/13";
($year, $month, $day) = $date =~ m/^(\d+)/(\d+)/(\d+)$/;
```

In *Mathematica*, this is done with `StringCases`.

```
In[97]:= date = "88/6/13";
         {year, month, day} = StringCases[date,
            RegularExpression["^(\\d+)/(\\d+)/(\\d+)$"] -> {"$1", "$2", "$3"}][[1]]
Out[98]= {88, 6, 13}
```

This is similar to assigning all the matches to an array using the /g modifier.

```
$text = "128.32.13.117";
@nums = $text =~ m/\d+/g;
```

The same thing is easily done with `StringCases` in *Mathematica*.

```
In[99]:= text = "128.32.13.117";
         nums = StringCases[text, RegularExpression["\\d+"]]
Out[100]= {128, 32, 13, 117}
```

s/.../.../

The obvious *Mathematica* version of the Perl s/.../.../ substitution operator is `StringReplace`.

```
$text = "abcagh";
$text =~ s/a./XX/;
```

The default Perl behavior is to do a single replacement.

```
In[101]:= text = "abcagh";
          StringReplace[text, RegularExpression["a."] -> "XX", 1]
Out[102]= XXcagh
```

The /g modifier in Perl does global replacement of all matches.

```
$text =~ s/a./XX/g
```

```
In[103]:= StringReplace[text, RegularExpression["a."] -> "XX"]
Out[103]= XXcXXh
```

Using the evaluation /e modifier, Perl can use subpatterns as part of the replacement. This is easily done in *Mathematica*.

```
$text = "13 27 3";
$text =~ s/(\d+)/$1$1/eg
```

```
In[104]:= text = "13 27 3";
          StringReplace[text, RegularExpression["(\\d+)"] :> "$1$1"]
Out[105]= 1313 2727 33
```

split(...)

The Perl `split` command is similar to `StringSplit` in *Mathematica*.

```
$text = "ab:cd:efg";
split(/:/, $text)
```

```
In[106]:= text = "ab:cd:efg";
          StringSplit[text, ":"]
Out[107]= {ab, cd, efg}
```

You can specify the number of blocks to split into in both Perl and *Mathematica*.

```
split(/:/, $text,2)
```

```
In[108]:= StringSplit[text, ":", 2]
Out[108]= {ab, cd:efg}
```

A `split` with capturing parentheses in the pattern, for which the captured substrings are included in the result, can be done in *Mathematica* using rules in the second argument of `StringSplit`. Compared to Perl, in *Mathematica* it is easy to then apply a function to these substrings.

```
$text = "test with <tag1>tags</tag1> and <b>more</b>";
split(/<([^>]*)>/, $text)
```

```
In[109]:= text = "test with <tag1>tags</tag1> and <b>more</b>";
          StringSplit[text, RegularExpression["<([^>]*)>"] → "$1"] // InputForm
Out[110]//InputForm= {"test with ", "tag1", "tags", "/tag1", " and ", "b", "more", "/b"}
```

```
In[111]:= text = "test with <tag1>tags</tag1> and <b>more</b>";
          StringSplit[text, RegularExpression["<([^>]*)>"] :→ Tag["$1"]] // InputForm
Out[112]//InputForm= {"test with ", Tag["tag1"], "tags", Tag["/tag1"], " and ", Tag["b"], "more", Tag["/b"]}
```

tr/.../.../

The Perl `tr` command can be simulated using *Mathematica* `StringReplace` together with the appropriate list of rules.

Here is the simplest form where the characters "a", "b", and "c" are replaced by "X", "Y", and "Z", respectively.

```
$text = "abcdef";
$text =~ tr/abc/XYZ/
```

This generates the appropriate rules in *Mathematica* using `Thread`.

```
In[113]:= text = "abcdef";
          StringReplace[text, Thread[Rule[Characters["abc"], Characters["XYZ"]]]]
Out[114]= XYZdef
```

Here is an example where the replacement list is shorter than the character list, so "d", "e", and "f" are all replaced by "Z".

```
$text = "abcdefghi";
$text =~ tr/abcdef/WXYZ/
```

```
In[115]:= text = "abcdefghi";
          StringReplace[text, Append[
            Thread[Rule[Characters["abc"], Characters["WXY"]]], Characters["def"] → "Z"]]
Out[116]= WXYZZZghi
```

Character ranges in Perl are emulated using `CharacterRange` in *Mathematica*.

```
$text = "this and that";
$text =~ tr/a-z/x/
```

```
In[117]:= text = "this and that";
          StringReplace[text, CharacterRange["a", "z"] → "x"]
Out[118]= xxxx xxx xxxx
```

With the /d modifier, the surplus characters are instead deleted.

```
$text = "abcdefghi";
$text =~ tr/abcdef/WXYZ/d
```

```
In[119]:= text = "abcdefghi";
          StringReplace[text, Append[
            Thread[Rule[Characters["abcd"], Characters["WXYZ"]]], Characters["ef"] → ""]]
Out[120]= WXYZghi
```

With the /c modifier, the complement of the character list is used.

```
$text =~ tr/aeh/ /c
```

```
In[121]:= StringReplace[text, Except[Characters["aeh"]] → " "]
Out[121]= a   e  h
```

```
In[122]:= StringReplace[text, RegularExpression["[^aeh]"] → " "]
Out[122]= a   e  h
```

The /s modifier squeezes down to one any run of characters translating into the same character.

```
$text = "abbcccddddeeeeeeffeeded";
$text =~ tr/abcde/ABCD/s
```

You get the same effect in *Mathematica* using `Repeated (..)`.

```
In[123]:= text = "abbcccddddeeeeeeffeeded";
          StringReplace[text,
           Append[Thread[Rule[Repeated /@ Characters["abc"], Characters["ABC"]]],
            Characters["de"].. → "D"]]
Out[124]= ABCDffD
```

Some Examples

Some brief examples of practical uses of string patterns are presented in this section.

Highlight Patterns

This defines a 1000-base random DNA string.

```
In[131]:= SeedRandom[1234];
          dna = StringJoin[Table[{"a", "c", "g", "t"}[[RandomInteger[{1, 4}]]], {1000}]]

Out[131]= acaaccgccgcgaattctcacaaacgtcgagtgtgatatagaaaatcccagatcacactatagggtggaaaccaggtgatagttgcctctgcca
             tgcatatgcgattaaatgttcgttgaatatgagtaaagaatctaagcgtagtttttatagtaaagaccccgcgcctctgcgcgtgatagtg
             ttaccgacgcatctcgatgttgtacatgtagcactgtacgtaatcattatacgatttccataacgtaagctgggtaacagacctaacgtag
             ggttcatctacgcgcttatcctccgacctaggattgcgtctagaaaactgaacaagtaaaccgtactcctttatccgccgacagtccagaa
             cagtctgacttccagctacttaatggtttcccagatttcctgcggaatacctcgaccgtgtggccattgctccaccaccgcaattcgcctc
             ttctgcacaggtccacgcacgttttccctgagcataaaaacccagcaatacgaaaggttctctacacatcagcagcttcccgagtgacctg
             attggggctgcgctataacgtcggtcgcgtttccatcaggacgcatgcagcgacgcctgcagcagcagtccccttcacagcgtacagggct
             ctggtaagggcagccagtttcgctaacggtcctgttgcttacatgcgcatacaattatgccaaacggacacgtgctatccagacgaggtgt
             cgtaaaggggatttctaagtgaccagaattactgtcagacgaccttaagatagtcaggctttcagcggtagataggcgggatgaatcgaaa
             gcaatgacaaggcccggtcgccagagagacaggcttagtattcagtaagcagtagcgcgacatacccgaaactccgcgcgggtatagagta
             catctactaggtgtgtatctgcagcacattagggctattcagaccgttaattccggcctgaggccatgccgacagaacaaattgcct
```

This highlights parts of the DNA that match a certain pattern.

```
In[132]:= StringReplace[dna,
           x : ("ag" ~~ _ ~~ _ ~~ "t" ~~ _ ~~ "ca") ⧴ "\!\(\*StyleBox[\"" <> x <>
             "\",FontColor->RGBColor[1,0,0],FontSize->18,FontWeight->\"Bold\"]\)"]

Out[132]= acaaccgccgcgaattctcacaaacgtcgagtgtgatatagaaaatcccagatcacactatagggtggaaaccaggtgatagttgcctctgcca
             tgcatatgcgattaaatgttcgttgaatatgagtaaagaatctaagcgtagtttttatagtaaagaccccgcgcctctgcgcgtgatagtg
             ttaccgacgcatctcgatgttgtacatgtagcactgtacgtaatcattatacgatttccataacgtaagctgggtaacagacctaacg
             tagggttcat
             ctacgcgcttatcctccgacctaggattgcgtctagaaaactgaacaagtaaaccgtactcctttatccgccgacagtccagaacagtctg
             acttccagctacttaatggtttcccagatttcctgcggaatacctcgaccgtgtggccattgctccaccaccgcaattcgcctcttctgca
             caggtccacgcacgttttccctgagcataaaaacccagcaatacgaaaggttctctacacatcagcagcttcccgagtgacctgattgggg
             ctgcgctataacgtcggtcgcgtttccatcaggacgcatgcagcgacgcctgcagcagcagtccccttcacagcgtacagggctctgg
             taagggcagccagtttcgctaacggtcctgttgcttacatgcgcatacaattatgccaaacggacacgtgctatccagacgaggtgtcgta
             aaggggatttctaagtgaccagaattactgtcagacgaccttaagatagtcaggctttcagcggtagataggcgggatgaatcgaaagcaa
             tgacaaggcccggtcgccagagagacaggcttagtattcagtaagcagtagcgcgacatacccgaaactccgcgcgggtat
            agagtaca
            tctactaggtgtgtatctgcagcacattagggctattcagaccgttaattccggcctgaggccatgccgacagaacaaattgcct
```

Here is the same result using a regular expression.

```
In[133]:= StringReplace[dna, RegularExpression["ag..t.ca"] ⧴
            "\!\(\*StyleBox[\"$0\",FontColor->RGBColor[1,0,0],FontSize->18,FontWeight->\"
              Bold\"]\)"]

Out[133]= acaaccgccgcgaattctcacaaacgtcgagtgtgatatagaaaatcccagatcacactatagggtggaaaccaggtgatagttgcctctgcca⋱
            tgcatatgcgattaaatgttcgttgaatatgagtaaagaatctaagcgtagtttttatagtaaagaccccgcgcctctgcgcgtgatagtg⋱
            ttaccgacgcatctcgatgttgtacatgtagcactgtacgtaatcattatacgatttccataacgtaagctgggtaacagacctaacg⋱
            tagggttcat⋱
            ctacgcgcttatcctccgacctaggattgcgtctagaaaactgaacaagtaaaccgtactcctttatccgccgacagtccagaacagtctg⋱
            acttccagctacttaatggtttcccagatttcctgcggaatacctcgaccgtgtggccattgctccaccaccgcaattcgcctcttctgca⋱
            caggtccacgcacgttttccctgagcataaaaacccagcaatacgaaaggttctctacacatcagcagcttcccgagtgacctgattgggg⋱
            ctgcgctataacgtcggtcgcgtttccatcaggacgcatgcagcgacgcctgcagcagcagtccccttcacagcgtacagggctctgg⋱
            taagggcagccagtttcgctaacggtcctgttgcttacatgcgcatacaattatgccaaacggacacgtgctatccagacgaggtgtcgta⋱
            aaggggatttctaagtgaccagaattactgtcagacgaccttaagatagtcaggctttcagcggtagataggcgggatgaatcgaaagcaa⋱
            tgacaaggcccggtcgccagagagacaggcttagtattcagtaagcagtagcgcgacatacccgaaactccgcgcgggtat
          agagtaca
          tctactaggtgtgtatctgcagcacattagggctattcagaccgttaattccggcctgaggccatgccgacagaacaaattgcct
```

HTML Parsing

String patterns are useful for taking raw HTML and extracting information from it.

Here is the source from www.google.com.

```
In[134]:= text = "\<<html><head><meta http-equiv='content-type'
              content='text/html;charset=UTF-8'><title>Google</title><style><!--body,td
              ,a,p,.h{font-family:arial,sans-serif;}
          .h{font-size:20px;}
          .q{color:#0000cc;}
          //-->
              </style>
              <script>
              <!--function sf(){document.f.q.focus();}
          //-->
              </script>
              </head><body bgcolor=#ffffff text=#000000 link=#0000cc
              vlink=#551a8b alink=#ff0000 onLoad=sf()><center><table border=0
              cellspacing=0 cellpadding=0><tr><td><img src='/images/logo.gif'
              width=276 height=110 alt='Google'></td></tr></table><br>
              <form action='/search' name=f><script><!--function
              qs(el) {if (window.RegExp&&window.encodeURIComponent)
              {var qe=encodeURIComponent(document.f.q.value);if
              (el.href.indexOf('q=')≠-1) {el.href=el.href.replace(new
              RegExp('q=[^&$]*'),'q='+qe);} else {el.href+='&q='+qe;}}return 1;}
          //-->
              </script><table border=0 cellspacing=0
              cellpadding=4><tr><td nowrap class=q><font size=-1><b><font
              color=#000000>Web</font></b>&nbsp;&nbsp;&nbsp;&nbsp;<a
              id=1a class=q href='/imghp?hl=en&tab=wi' onClick='return
              qs(this);'>Images</a>&nbsp;&nbsp;&nbsp;&nbsp;<a id=2a
              class=q href='/grphp?hl=en&tab=wg' onClick='return
              qs(this);'>Groups</a>&nbsp;&nbsp;&nbsp;&nbsp;<a id=4a
              class=q href='/nwshp?hl=en&tab=wn' onClick='return
              qs(this);'>News</a>&nbsp;&nbsp;&nbsp;&nbsp;<a id=5a
              class=q href='/froogle?hl=en&tab=wf' onClick='return
              qs(this);'>Froogle</a>&nbsp;&nbsp;&nbsp;&nbsp;<b><a
              href='/options/index.html'
              class=q>more&nbsp;&raquo;</a></b></font></td></tr></table>  <table
              cellspacing=0 cellpadding=0><tr><td width=25%>&nbsp;</td><td
              align=center><input type=hidden name=hl value=en><span
              id=hf></span><input type=hidden name=ie value='UTF-8'><input
              maxLength=256 size=55 name=q value=''><br><input
              type=submit value='Google Search' name=btnG><input
              type=submit value='I'm Feeling Lucky' name=btnI></td><td
              valign=top nowrap width=25%><font size=-2>&nbsp;&nbsp;<a
              href=/advanced_search?hl=en>Advanced&nbsp;Search</a><br>&nbsp;&nbsp;<a
              href=/preferences?hl=en>Preferences</a><br>&nbsp;&nbsp;<a
              href=/language_tools?hl=en>Language
              Tools</a></font></td></tr></table></form><br><br><font
              size=-1><a href='/ads/'>Advertising&nbsp;Programs</a>- <a
              href='/services/'>Business&nbsp;Solutions</a>- <a href=/about.html>About
              Google</a><span id=hp style='behavior:url(#default#homepage)'></span>
              <script>
              //<!--if (!hp.isHomePage('http://www.google.com/'))
```

```
    {document.write('<p><a href=\'/mgyhp.html\'
    onClick=\'style.behavior='url(#default#homepage)';setHomePage('http://www
    .google.com/');\'>Make Google Your Homepage!</a>');}
//-->
    </script></font><p><font size=-2>&copy;2004 Google-Searching
    4,285,199,774 web pages</font></p></center></body></html>\>";
```

```
In[135]:= StringLength[text]

Out[135]= 2639
```

This extracts all the direct hyperlinks in the source.

```
In[136]:= StringCases[text, Shortest["<a" ~~
              __ ~~ "href=" ~~ ref__ ~~ (WhitespaceCharacter | ">") ~~ ___ ~~ ">"] :> ref]

Out[136]= {'/imghp?hl=en&tab=wi', '/grphp?hl=en&tab=wg', '/nwshp?hl=en&tab=wn',
           '/froogle?hl=en&tab=wf', '/options/index.html', /advanced_search?hl=en, /preferences?hl=en,
           /language_tools?hl=en, '/ads/', '/services/', /about.html, \'/mgyhp.html\'}
```

This deletes everything inside tags < … >.

```
In[137]:= StringReplace[text, Shortest["<" ~~ ___ ~~ ">"] → ""]

Out[137]= Google





            Web&nbsp;&nbsp;&nbsp;&nbsp;Images&nbsp;&nbsp;&nbsp;&nbsp;Groups&nbsp;&nbsp;&nbsp;&nbsp;News&
            nbsp;&nbsp;&nbsp;&nbsp;Froogle&nbsp;&nbsp;&nbsp;&nbsp;more&nbsp;&raquo;
            &nbsp;&nbsp;&nbsp;Advanced&nbsp;Search&nbsp;&nbsp;Preferences&nbsp;&nbsp;Language
            ToolsAdvertising&nbsp;Programs- Business&nbsp;Solutions- About Google

            //Make Google Your Homepage!');}
          //-->
            &copy;2004 Google-Searching 4,285,199,774 web pages
```

Find Money

Here is some text to scan for strings that look like dollar amounts.

```
In[138]:= text = "This $100 sentence can be bought for $85.00, at 15% discount"

Out[138]= This $100 sentence can be bought for $85.00, at 15% discount
```

This is one way to do the search using symbolic string patterns.

```
In[139]:= StringCases[text, "$" ~~ DigitCharacter .. ~~ (("." ~~ DigitCharacter ..) | "")]

Out[139]= {$100, $85.00}
```

Here is the same search using regular expressions (note that you must remember to escape the dollar sign).

```
In[140]:= StringCases[text, RegularExpression["\\$\\d+(\.\\d+)?"]]

Out[140]= {$100, $85.00}
```

There is also a built-in pattern object, `NumberString`, for this particular situation.

```
In[141]:= StringCases[text, "$" ~~ NumberString]

Out[141]= {$100, $85.00}
```

Find Text in Files

Here is a very simple grep-like function for finding lines in a text file containing text matching a given pattern.

```
In[142]:= Grep[file_, patt_] := With[{data = Import[file, "Lines"]},
            Pick[Transpose[{Range[Length[data]], data}], StringFreeQ[data, patt], False]]
```

This creates a sample text file.

```
In[143]:= Export["test.txt", {"this is a line",
            "a line with 2 numbers 5", "third line and more", "line 4"}, "Lines"]

Out[143]= test.txt
```

This returns the line numbers and lines in `"text.txt"` containing any digit characters.

```
In[144]:= Grep["test.txt", DigitCharacter] // TableForm

Out[144]//TableForm=
         2  a line with 2 numbers 5
         4  line 4
```

This finds lines containing `"a"` as a standalone word.

```
In[145]:= Grep["test.txt", RegularExpression["\\ba\\b"]] // TableForm

Out[145]//TableForm=
         1  this is a line
         2  a line with 2 numbers 5
```

Tips and Tricks for Efficient Matching

This section addresses some issues involving efficiency in string pattern matching.

StringExpression versus RegularExpression

Since a string pattern written in *Mathematica* syntax is immediately translated to a regular expression and then compiled and cached, there is very little overhead in using the *Mathematica* syntax as opposed to the regular expression syntax directly. An exception to this happens when many different patterns are used a few times; in that case the overhead might be noticeable.

Conditions and PatternTests

If a pattern contains `Condition` (`/;`) or `PatternTest` (?) statements, the general *Mathematica* evaluator must be invoked during the match, thus slowing it down. If a pattern can be written without such constructs, it will typically be faster.

```
In[146]:= SeedRandom[1234];
          test = StringJoin[Table[FromCharacterCode[RandomInteger[{48, 80}]], {200}]];

In[147]:= StringCases[test, DigitCharacter ..] // Length // Timing

Out[147]= {0. Second, 45}

In[148]:= StringCases[test, __?DigitQ] // Length // Timing

Out[148]= {0.03 Second, 45}
```

Avoid Nested Quantifiers

Because of the nondeterministic finite automaton (NFA) algorithm used in the match, patterns involving nested quantifiers (such as __ and `patt ..` or the regular expression equivalents) can become arbitrarily slow. Such patterns can usually be "unrolled" into more efficient versions (see Friedl [2] for additional information).

Avoid Many Calls to a Function

If you are searching through a long list of strings for certain matches, it is more efficient to feed the whole list to a string function at once, rather than using something like `Select` and `StringMatchQ` (see the earlier dictionary example for an illustration). Here is another example that generates a list of 2000 strings with 10 characters each and searches for the strings that start with an "a" and contain "ggg" as a substring.

```
In[149]:= SeedRandom[1234]; test = Table[StringJoin[
             {"a", "c", "g", "t"}[[#]] & /@ Table[RandomInteger[{1, 4}], {10}]], {2000}];
```

```
In[150]:= Take[test, 3]
Out[150]= {acaaccgccg, cgaattctca, caaacgtcga}
```

Here is the slower version, using `Select` and `StringMatchQ`.

```
In[151]:= Select[test, StringMatchQ[#, "a" ~~ ___ ~~ "ggg" ~~ ___] &] // Timing
Out[151]= {0.01 Second, {acgtagggtt, attagggcta, atagggctct, aagggccgtc, agtgttaggg, aggggtggca, aggggcggag,
            agcgggactc, acagagggtg, atgggacatc, agggataaga, accacgggct, aaaagggcat, agtaagggac, agggtagtta,
            agctacgggc, ataagccggg, atagggagaa, acttgatggg, acagtgaggg, agggcaggga, agggttctag,
            agaggggaac, atgcagggat, atcgtagggc, aggggaagct, agtggggctg, aaacaaggga, aagtgggatg,
            aagagggaat, agggacggag, attcgggagc, aataactggg, agggcgccca, agaggggatt, agggacgaag,
            aagggatatt, agggcaggtg, agaacgggta, aattgggtct, agcgggtagg, actcgggccc, agggcctcct,
            aagggagggg, aagggcatgt, aagttgaggg, aaaacggggt, agagggcgta, aagtctaggg, agggagcgtc}}
```

If you instead feed the whole list to `StringMatchQ` at once, it will be much faster. Then `Pick` can be used to extract the wanted elements.

```
In[152]:= Pick[test, StringMatchQ[test, "a" ~~ ___ ~~ "ggg" ~~ ___]] // Timing
Out[152]= {0. Second, {acgtagggtt, attagggcta, atagggctct, aagggccgtc, agtgttaggg, aggggtggca, aggggcggag,
            agcgggactc, acagagggtg, atgggacatc, agggataaga, accacgggct, aaaagggcat, agtaagggac, agggtagtta,
            agctacgggc, ataagccggg, atagggagaa, acttgatggg, acagtgaggg, agggcaggga, agggttctag,
            agaggggaac, atgcagggat, atcgtagggc, aggggaagct, agtggggctg, aaacaaggga, aagtgggatg,
            aagagggaat, agggacggag, attcgggagc, aataactggg, agggcgccca, agaggggatt, agggacgaag,
            aagggatatt, agggcaggtg, agaacgggta, aattgggtct, agcgggtagg, actcgggccc, agggcctcct,
            aagggagggg, aagggcatgt, aagttgaggg, aaaacggggt, agagggcgta, aagtctaggg, agggagcgtc}}
```

Alternatively, you could use `StringCases`, which is also fast. Note that you need to anchor the pattern using `StartOfString` to ensure that the `"a"` is at the start (the `EndOfString` is superfluous in this particular case).

```
In[153]:= Flatten[StringCases[test,
             StartOfString ~~ "a" ~~ ___ ~~ "ggg" ~~ ___ ~~ EndOfString]] // Timing
Out[153]= {0. Second, {acgtagggtt, attagggcta, atagggctct, aagggccgtc, agtgttaggg, aggggtggca, aggggcggag,
            agcgggactc, acagagggtg, atgggacatc, agggataaga, accacgggct, aaaagggcat, agtaagggac, agggtagtta,
            agctacgggc, ataagccggg, atagggagaa, acttgatggg, acagtgaggg, agggcaggga, agggttctag,
            agaggggaac, atgcagggat, atcgtagggc, aggggaagct, agtggggctg, aaacaaggga, aagtgggatg,
            aagagggaat, agggacggag, attcgggagc, aataactggg, agggcgccca, agaggggatt, agggacgaag,
            aagggatatt, agggcaggtg, agaacgggta, aattgggtct, agcgggtagg, actcgggccc, agggcctcct,
            aagggagggg, aagggcatgt, aagttgaggg, aaaacggggt, agagggcgta, aagtctaggg, agggagcgtc}}
```

Rewrite General Expression Searches as String Searches

Because the string-matching algorithm is different than the algorithm *Mathematica* uses for general expression matching (string matching can assume a finite alphabet and a flat structure, for instance), there are cases where it is advantageous to translate a normal expression-matching problem to a string-matching problem. A typical case is matching a long list of symbols against a pattern involving several occurrences of __ and ___.

As an example, assume you want to find primes (after prime number 1000000, say) that have at least four identical digits. Using ordinary pattern matching, it could be accomplished like this.

```
In[154]:= Select[Array[Prime, 1000, 1 000 000],
            MatchQ[IntegerDigits[#], {___, x_, ___, x_, ___, x_, ___, x_, ___}] &] // Timing

Out[154]= {0.16 Second, {15 488 881, 15 491 117, 15 491 171, 15 491 711, 15 493 333, 15 493 999,
            15 496 111, 15 499 111, 15 499 199, 15 499 399, 15 499 499, 15 499 919, 15 499 997, 15 500 557,
            15 501 119, 15 501 121, 15 501 151, 15 501 553, 15 501 559, 15 501 911, 15 502 111}}
```

By converting the list of integers to a string, you can use string matching instead.

```
In[155]:= Select[Array[Prime, 1000, 1 000 000],
            StringMatchQ[FromCharacterCode[48 + IntegerDigits[#]],
              StringExpression[___, x_, ___, x_, ___, x_, ___, x_, ___]] &] // Timing

Out[155]= {0.05 Second, {15 488 881, 15 491 117, 15 491 171, 15 491 711, 15 493 333, 15 493 999,
            15 496 111, 15 499 111, 15 499 199, 15 499 399, 15 499 499, 15 499 919, 15 499 997, 15 500 557,
            15 501 119, 15 501 121, 15 501 151, 15 501 553, 15 501 559, 15 501 911, 15 502 111}}
```

By using the previous tips of using `Pick` or `StringCases`, you can speed it up even more.

```
In[156]:= With[{list = Array[Prime, 1000, 1 000 000]},
            Pick[list, StringMatchQ[FromCharacterCode[48 + IntegerDigits[#]] & /@ list,
              StringExpression[___, x_, ___, x_, ___, x_, ___, x_, ___]]]] // Timing

Out[156]= {0.04 Second, {15 488 881, 15 491 117, 15 491 171, 15 491 711, 15 493 333, 15 493 999,
            15 496 111, 15 499 111, 15 499 199, 15 499 399, 15 499 499, 15 499 919, 15 499 997, 15 500 557,
            15 501 119, 15 501 121, 15 501 151, 15 501 553, 15 501 559, 15 501 911, 15 502 111}}

In[157]:= Flatten[StringCases[FromCharacterCode[48 + IntegerDigits[#]] & /@
              Array[Prime, 1000, 1 000 000], StringExpression[StartOfString,
              ___, x_, ___, x_, ___, x_, ___, x_, ___, EndOfString]]] // Timing

Out[157]= {0.04 Second, {15488881, 15491117, 15491171, 15491711, 15493333,
            15493999, 15496111, 15499111, 15499199, 15499399, 15499499, 15499919, 15499997,
            15500557, 15501119, 15501121, 15501151, 15501553, 15501559, 15501911, 15502111}}
```

For long sequences, the difference can be significant.

```
In[158]:= test = Range[100]; test[[{50, 75}]] = 5;

In[159]:= Position[test, 5]

Out[159]= {{5}, {50}, {75}}

In[160]:= MatchQ[test, {___, x_, ___, x_, ___, x_, ___}] // Timing

Out[160]= {0.121 Second, True}

In[161]:= teststr = FromCharacterCode[test];

In[162]:= StringPosition[teststr, FromCharacterCode[5]]

Out[162]= {{5, 5}, {50, 50}, {75, 75}}
```

```
In[163]:= StringMatchQ[teststr, StringExpression[___, x_, ___, x_, ___, x_, ___]] // Timing
Out[163]= {0. Second, True}
```

Implementation Details

String pattern matching in *Mathematica* is built on top of the PCRE (Perl Compatible Regular Expressions) library by Philip Hazel [1].

In some cases the pre-5.1 *Mathematica* algorithms are used (for example, when the pattern is just a single, literal string).

Any symbolic string pattern is first translated to a regular expression. You can see this translation by using the internal `StringPattern`PatternConvert` function.

```
In[164]:= StringPattern`PatternConvert["a" | "" ~~ DigitCharacter ..] // InputForm
Out[164]//InputForm= {"(?ms)a?\\d+", {}, {}, {}, Hold[None]}
```

The first element returned is the regular expression, while the rest of the elements have to do with conditions, replacement rules, and named patterns.

The regular expression is then compiled by PCRE, and the compiled version is cached for future use when the same pattern appears again. The translation from symbolic string pattern to regular expression only happens once.

Mathematica conditions in the pattern are handled by external call-outs from the PCRE library to the *Mathematica* evaluator, so this will slow down the matching.

Explicit `RegularExpression` objects embedded into a general string pattern will be spliced into the final regular expression (surrounded by noncapturing parentheses "`(?:...)`"), so the counting of named patterns can become skewed compared to what you might expect.

Because PCRE currently does not support preset character classes with characters beyond character code 255, the word and letter character classes (such as `WordCharacter` and `LetterCharacter`) only include character codes in the Unicode range 0–255. Thus `LetterCharacter` and `_?LetterQ` do not give equivalent results beyond character code 255.

Because of a similar PCRE restriction, case-insensitive matching (for example, with `IgnoreCase -> True`) will only apply to letters in the Unicode range 0–127 (that is, the normal English letters "`a`"–"`z`" and "`A`"–"`Z`").

References

[1] Hazel, P. "PCRE—Perl Compatible Regular Expressions." 2008. www.pcre.org

[2] Friedl, J. E. F. *Mastering Regular Expressions*. (2nd ed.) O'Reilly & Associates, 2002.