Regular expression notation is a compact way of specifying a pattern for subject strings that can be searched and from which specific parts can be extracted by StringRegExp() or replaced when using StringRegExpReplace().

More precisely, the regular expression engine tries to match a pattern (a kind of programmatic format) with a subject string, both from left to right. Should a mismatch occur, the engines tries to backtrack (return to successively previous states) as much as needed, expecting that the rest of the pattern will match as well.

Backtracking is a fundamental feature of regular expression engines and one that every novice programmer already understand and uses daily. It is like leaving a specific mark on every fork in the road and going back to the last untried path when the choosen path turns out to be a dead end: you backtrack as needed until you find the right point (match success) or explore every path without reaching your goal (match failure). Searching a given filename with optional wildcards inside a directory tree is no different.

AutoIt uses the PCRE engine. PCRE means "Perl-Compatible Regular Expressions" and is the most comprehensive open-source engine available. This implementation includes Unicode Category Properties (UCP) support, which allows fine-grain processing of most human languages.
However to maintain compatibility with previous versions and keep matching speed at its best, **the UCP support is not enabled by default**. You can enable it by prepending the string **(*UCP)** at the very start of your pattern. When enabled, the UCP setting changes the extend of a number of regular expression elements, as documented below where applicable.

This page is only a summary for the most used pattern elements. For full in-depth discussion of regular expressions as implemented in AutoIt, refer to the **complete description of PCRE patterns**.
Unless you are already familiar with regular expressions you will probably need to read several parts of this summary more than once to grasp how they work and inter-relate.

Caution: bad regular expressions can produce a quasi-infinite loop hogging the CPU, and can even cause a crash.

`

Global settings

These settings are only recognized at the start of the pattern and affect it globally.

Newline conventions

Newline sequences affect where the **^** and **$** anchors match and what **\N** and **.** do not match. By default the newline sequence is @CRLF as an unbreakable sequence or lone @CR or @LF.
The default can be changed by prepending one of the following sequence at the start of a pattern.

(*CR)	Carriage return (@CR).
(*LF)	Line feed (@LF).
(*CRLF)	Carriage return immediately followed by linefeed (@CRLF).
(*ANYCRLF)	Any of @CRLF, @CR or @LF. This is the default newline convention.
(*ANY)	Any Unicode newline sequence: @CRLF, @LF, VT, FF, @CR or \x85.

What \R matches

(*BSR_ANYCRLF)	By default **\R** matches @CRLF, @CR or @LF only.
(*BSR_UNICODE)	Changes **\R** to match any Unicode newline sequence: @CRLF, @LF, VT, FF, @CR or \x85.

Options

PCRE patterns may contain options, which are enclosed in **(?)** sequences. Options can be grouped together: "(?imx)". Options following an hyphen are negated: "(?im-sx)". Options appearing outside a group affect the remaining of the pattern from that point onwards. Options appearing inside a group affect that group only. Options lose their special meaning inside a character class, where they are treated literally.

(?i)	Caseless: matching becomes case-insensitive from that point on. By default, matching is case-sensitive. When UCP is enabled casing applies to the entire Unicode plane 0, else applies by default to ASCII letters A-Z and a-z only.
(?m)	Multiline: **^** and **$** match at newline sequences within data. By default, multiline is off.

(?s)	Single-line or *DotAll*: **.** matches anything including a newline sequence. By default, DotAll is off hence **.** does not match a newline sequence.
(?U)	Ungreedy: quantifiers become lazy (non-greedy) from that point on. By default, matching is greedy - see below for further explanation.
(?x)	eXtended: whitespaces outside character classes are ignored and **#** starts a comment up to the next solid newline in pattern. Meaningless whitespaces between components make regular expressions much more readable. By default, whitespaces match themselves and **#** is a literal character.

Characters, metacharacters and escaping (or quoting)

Characters

Regular expressions patterns consist of literal Unicode text parts which match themselves, intermixed with regular expression specifiers or options. Specifiers and options use a few metacharacters which have a special meaning by themselves or introduce special pattern elements described in the tables below.
In literal parts, alphanumeric characters always stand for themselves: the pattern "literal part with 中国文字" matches exactly the string "literal part with 中国文字" ("中国文字" means "chinese text".)
Some non-alphanumeric characters called metacharacters have special behavior, discussed thereafter.

Representing some characters literally

The special sequences below are used to represent certain characters literally.

\a	Represents "alarm", the BEL character (Chr(7)).
\c*X*	Represents "control-*X*", where *X* is any 7-bit ASCII character. For example, "\cM" represents ctrl-M, same as \x0D or \r (Chr(13)).
\e	Represents the "escape" control character (Chr(27)). *Not to be confused with the escaping of a character!*
\f	Represents "formfeed" (Chr(12)).
\n	Represents "linefeed" (@LF, Chr(10)).
\r	Represents "carriage return" (@CR, Chr(13)).

\t	Represents "tab" (@TAB, Chr(9)).
*****ddd*	Represents character with octal code *ddd*, OR backreference to capturing group number *ddd* in decimal. For example, **([a-z])\1** would match a doubled letter. **Best avoided** as it can be ambiguous! Favor the hex representations below.
\x*hh*	Represents Unicode character with hex codepoint *hh*: "\x7E" represents a tilde, "~".
\x{*hhhh***}**	Represents Unicode character with hex codepoint *hhhh*: "\x{20AC}" represents the Euro symbol, "€" (ChrW(0x20AC)).
*****x*	where *x* is non-alphanumeric, stands for a literal *x*. Used to represent metacharacters literally: "\.\[" represents a dot followed by a left square bracket, ".[".
\Q ... **\E**	Verbatim sequence: metacharacters lose their special meaning between **\Q** and **\E**: "\Q(.)\E" matches "(.)" and is equivalent to, but more readable than, "\(\.\)".

Metacharacters

PCRE metacharacters are **\ . ^ $ | [({ * + ? #** which have one or more special meaning, depending on context.
To insert a literal metacharacter, preceede it by a backslash (this is called **escaping (or quoting) a character**): "\$" means the dollar character.
Metacharacters will be discussed in separate sections where their behavior or meaning belong.

Character types

.	Matches any single character except, by default, a newline sequence. Matches newlines as well when option **(?s)** is active.
\d	Matches any decimal digit (any Unicode decimal digit in any language when UCP is enabled).
\D	Matches any non-digit.
\h	Matches any horizontal whitespace character (see table below).

\H	Matches any character that is not a horizontal whitespace character.
\N	Matches any character except a newline sequence regardless of option **(?s)**.
\p{*ppp***}**	Only when UCP is enabled: matches any Unicode character having the property *ppp*. E.g. "\b\p{Cyrillic}+" matches any cyrillic word; "\p{Sc}" matches any currency symbol. See reference documentation for details.
\P{*ppp***}**	Only when UCP is enabled: matches any Unicode character not having the property *ppp*.
\R	Matches any Unicode newline sequence by default, or the currently active (*BSR_...) setting. By default **\R** matches "(?>\r\n|\n|\r)" where "(?>...)" is an atomic group, making the sequence "\r\n" (@CRLF) unbreakable.
\s	Matches any whitespace character (see table below).
\S	Matches any non-whitespace character.
\v	Matches any vertical whitespace character (see table below).
\V	Matches any character that is not a vertical whitespace character.
\w	Matches any "word" character: any digit, any letter or underscore "_" (any Unicode digit, any Unicode letter in any language or underscore "_" when UCP is enabled).
\W	Matches any non-word character.
\X	Only when UCP is enabled: matches any Unicode extended grapheme cluster - an unbreakable sequence of codepoints which represent a single character for the user. As a consequence **\X** may match more than one character in the subject string, contrary to all other sequences in this table.

Sets of "whitespaces characters"

Horizontal whitespace characters matched by \h
\h is equivalent to "[\x09 \xA0]" by default (or "[\x09 \xA0\x{1680}\x{180E}\x{2000}-\x{200A}\x{202F}\x{205F}\x{3000}]" when UCP is enabled.)
This set is: Horizontal tab (HT), Space, Non-break space (adding: Ogham space mark, Mongolian vowel separator, En quad, Em quad, En space, Em space, Three-per-em space, Four-per-em space, Six-per-em space, Figure space, Punctuation space, Thin space, Hair space, Narrow no-break space, Medium mathematical space, Ideographic

space when UCP is enabled.)

Vertical whitespace characters matched by \v
\v is equivalent to "[\x0A-\x0D]" by default (or "[\x0A-\x0D\x{0085}\x{2028}\x{2029}]" when UCP is enabled.)
This set is: Linefeed (LF), Vertical tab (VT), Form feed (FF), Carriage return (CR)
(adding: Next line (NEL), Line separator, Paragraph separator when UCP is enabled.)

Whitespace characters matched by \s
\s is equivalent to "[\h\x0A\x0C\x0D]".
This set is: all characters in **\h** plus Linefeed (LF), Form feed (FF), Carriage return (CR).

Whitespace characters matched by [[:space:]]
[[:space:]] is equivalent to "\s".
This set is: all characters in **\s** plus Vertical tab (VT).

Character classes and POSIX classes

Character classes

A character classes defines a set of allowed (resp. disallowed) characters, which the next character in subject is expected to match (resp. not to match).
Inside a character classes, most metacharacters lose their meaning (like $. or *) or mean something else (like ^).

[...]

Matches any character in the explicit set: "[aeiou]" matches any lowercase vowel. A contiguous (in Unicode codepoint increasing order) set can be defined by putting an hyphen between the starting and ending characters: "[a-z]" matches any lowercase ASCII letter. To include a hyphen (-) in a set, put it as the first or last character of the set or escape it (\-).

Notice that the pattern "[A-z]" is not the same as "[A-Za-z]": the former is equivalent to "[A-Z\[\\\]^_`a-z]".

To include a closing bracket in a set, use it as the first character of the set or escape it: "[][]" and "[\[\]]" will both match either "[" or "]".

Note that in a character class, only **\d**, **\D**, **\h**, **\H**, **\p{}**, **\P{}**, **\s**, **\Q**...**\E**, **\S**, **\v**, **\V**, **\w**, **\W**, and **\x** sequences retain their special meaning, while **\b** means the backspace character (Chr(8)).

[^ ...]	Matches any character **not** in the set: "[^0-9]" matches any non-digit. To include a caret (^) in a set, put it after the beginning of the set or escape it (\^).

POSIX classes

These are named sets specifications to be used themselves within a character class: "[z[:digit:]w-y]" is the same as "[w-z0-9]". To negate a POSIX character class, put a caret (^) after the first colon: "[[:^digit:]]".

When UCP is enabled, several POSIX classes extend to some Unicode character subset, else they are by default restricted to 7-bit ASCII.

[:alnum:]	ASCII letters and digits (same as [^\W_] or [A-Za-z0-9]). When UCP is enabled: Unicode letters and digits (same as [^\W_] or \p{Xan}).
[:alpha:]	ASCII letters (same as [^\W\d_] or [A-Za-z]). When UCP is enabled: Unicode letters (same as [^\W\d_] or \p{L}).
[:ascii:]	ASCII characters (same as [\x00-\x7F]).
[:blank:]	Space or Tab (@TAB) (same as \h or [\x09\x20]). When UCP is enabled: Unicode horizontal whitespaces (same as \h).
[:cntrl:]	ASCII control characters (same as Chr(0) ... Chr(31) and Chr(127)).
[:digit:]	ASCII decimal digits (same as \d or [0-9]). When UCP is enabled: Unicode decimal digits (same as \d or \p{Nd}).
[:graph:]	ASCII printing characters, excluding space (same as Chr(33) ... Chr(126)).
[:lower:]	ASCII lowercase letters (same as [a-z]). When UCP is enabled: Unicode lowercase letters (same as \p{Ll}).
[:print:]	ASCII printing characters, including space (same as Chr(32) ... Chr(126)).
[:punct:]	ASCII punctuation characters, [:print:] excluding [:alnum:] and space, (33-47, 58-64, 91-96, 123-126).
[:space:]	ASCII white space (same as [\h\x0A-\x0D]). **[:space:]** is not quite the same as **\s**: it includes VT, Chr(11)).

[:upper:]	ASCII uppercase letters (same as [A-Z]). When UCP is enabled: Unicode uppercase letters (same as \p{Lu}).
[:word:]	ASCII "Word" characters (same as \w or [[:alnum:]_]). When UCP is enabled: Unicode "word" characters (same as \w or [[:alnum:]_] or \p{Xwd}).
[:xdigit:]	Hexadecimal digits (same as [0-9A-Fa-f]).

Groups

Groups are used to delimit subpatterns and are the building blocks of powerful expressions. Groups can be either capturing or not and may be nested irrespective of their nature, except comments groups. A regular expression can contain up to 65535 capturing groups.

Option letters (discussed above) can be conveniently inserted between the "?" and the ":" of non-capturing groups: "(?-i:[aeiou]{5})" matches 5 lowercase vowels. In this case options are local to the group.

(... **)**	Capturing group. The elements in the group are treated in order and can be repeated as a block. E.g. "(ab)+c" will match "abc" or "ababc", but not "abac". Capturing groups remember the text they matched for use in backreferences and they populate the optionally returned array. They are numbered starting from 1 in the order of appearance of their opening parenthesis. Capturing groups may also be treated as subroutines elsewhere in the pattern, possibly recursively.
(?<*name***>** ... **)**	Named capturing group. Can be later referenced by name as well as by number. Avoid using the name "DEFINE" (see "conditional patterns").
(?: ... **)**	Non-capturing group. Does not record the matching characters in the array and cannot be re-used as backreference.
(?| ... **)**	Non-capturing group with reset. Resets capturing group numbers in each top-level alternative it contains: "(?|(Mon)|(Tue)s|(Wed)nes|(Thu)rs|(Fri)|(Sat)ur|(Sun))day" matches a weekday name and captures its abbreviation in group number 1.

(?> ... **)**	Atomic non-capturing group: locks and never backtracks into (gives back from) what has been matched (see also Quantifiers and greediness below). Atomic groups, like possessive quantifiers, are always greedy.
(?# ... **)**	Comment group: always ignored (but may not contain a closing parenthesis, hence comment groups are not nestable).

Quantifiers and greediness

Quantifiers (or repetition specifiers) specify how many of the preceding character, class, reference or group are expected to match. Optional greediness qualifiers denote how aggressively the repetition will behave. For instance "\d{3,5}" will match at least 3 and no more than 5 decimal digits.

By default, patterns are "greedy", which means that quantifiers *** + ? {**...**}** will match the longest string which doesn't cause the rest of the pattern to fail. Greediness can be inverted for the entire pattern by giving option (?U) at the head of the pattern, or locally by placing a question mark following a quantifier.

Non-greedy (lazy) repetitions will match the smallest string that still allows the rest of the pattern to match. E.g. given the subject "aaab", the pattern "(a*)([ab]+)" will capture "aaa" then "b", but "(?U)(a*)([ab]+)" will capture "" then "a": indeed, capturing an empty string is good enough to satisfy the lazy "(a*)" and capturing "a" matches the lazy "([ab]+)" subpattern.

Possessive quantifiers are atomic and greedy. In fact they are a short notation for simple atomic groups. "\d++" is a shorthand notation for "(?>\d+)" and its behavior is "match a complete sequence of one or more digits, but never give back any". As a consequence "\d++**(\d**)" can never match since the last digit (in bold) is already matched and locked by "\d++". This is in contrast with simple greediness, where "\d+(\d)" will first match a complete sequence of digits with "\d+", but then backtrack the last one to allow "**(\d)**" to capture it.

There are two reasons for using an atomic group or a possessive quantifier: either for matching a sequence of characters that may also appear individually (e.g. "\r\n" in the definition of **\R**), or for forcing a quick failure in certain situations involving unbounded repetitions, where the engine would normally spend a very long time trying a huge number of grouping combinations before failing.

?	0 or 1, greedy.
?+	0 or 1, possessive.

??	0 or 1, lazy.
*****	0 or more, greedy.
***+**	0 or more, possessive.
***?**	0 or more, lazy.
+	1 or more, greedy.
++	1 or more, possessive.
+?	1 or more, lazy.
{*x***}**	exactly *x*.
{*x***,***y***}**	at least *x* and no more than *y*, greedy.
{*x***,***y***}+**	at least *x* and no more than *y*, possessive.
{*x***,***y***}?**	at least *x* and no more than *y*, lazy.
{*x***,}**	*x* or more, greedy.
{*x***,}+**	*x* or more, possessive.
{*x***,}?**	*x* or more, lazy.

Alternation

X|*Y*	Matches either subpattern *X* or *Y*: "ac|dc|ground" matches "ac" or "dc" or "ground".

Backreferences and references to subroutines

Backreferences permit reuse of the content of a previously captured group.

n	References a previous capturing group by its absolute number. WARNING: if no group number *n* exists, it evaluates as the character with value *n* provided *n* is a valid octal value, else errors out. Due to this ambiguity, this form is not recommended. Favor the next forms for a safe semantic.

\g*n*	References a previous capturing group by its absolute number.
\g{*n***}**	References a previous capturing group by its absolute number. Similar to above but clearly delimits where *n* ends: useful when the following character(s) is(are) digits.
\g-*n*	References a previous capturing group by its relative number.
\k<*name***>**	References a previous capturing group by its name.

References to subroutines

Capturing groups are subpatterns that can be invoked (possibly recursively) exactly like subroutines in a programming language. The subpattern is simply re-run at the current matching point. See reference documentation for details and examples.

(?R) or **(?0)** Recurses into the entire regular expression.

(?*n***)**	Calls subpattern by absolute number.
(?+*n***)**	Calls subpattern by relative number.
(?-*n***)**	Calls subpattern by relative number.
(?&*name***)**	Calls subpattern by name.

Anchors and assertions

Anchors and assertions are tests that do not change the matching position and therefore do not consume nor capture anything.

Anchors test the position of the current matching point.

^	Outside a character class, the caret matches at the start of the subject text, and also just after a non-final newline sequence if option **(?m)** is active. By default the newline sequence is @CRLF. Inside a character class, a leading **^** complements the class (excludes the characters listed there).
$	Outside a character class, the dollar matches at the end of the subject text, and also just before a newline sequence if option **(?m)** is active. Inside a character class, **$** means itself, a dollar sign.

\A	Matches only at the absolute beginning of subject string, irrespective of the multiline option **(?m)**. Will never match if offset is not 1.
\G	Matches when the current position is the first matching position in subject.
\z	Matches only at end of subject string, irrespective of the multiline option **(?m)**.
\Z	Matches only at end of subject string, or before a newline sequence at the end, irrespective of the multiline option **(?m)**.

Assertions test the character(s) preceding (look-behind), at (word boundary) or following (look-ahead) the current matching point.

\b	Matches at a "word" boundary, i.e. between characters not both **\w** or **\W**. See **\w** for the meaning of "word". Inside a character class, **\b** means "backspace" (Chr(8)).
\B	Matches when not at a word boundary.
(?=*X***)**	Positive look-ahead: matches when the subpattern *X* matches starting at the current position.
(?!*X***)**	Negative look-ahead: matches when the subpattern *X* does not match starting at the current position.
(?<=*X***)**	Positive look-behind: matches when the subpattern *X* matches characters preceding the current position. Pattern *X* must match a fixed-length string, i.e. may not use any undefinite quantifier *** + or ?**.
(?<!*X***)**	Negative look-behind: matches when the subpattern *X* does not match characters preceding the current position. Pattern *X* must match a fixed-length string, i.e. may not use any undefinite quantifier *** + or ?**.

Resetting the match

There are situations where it is necessary to "forget" that something has matched so far, in order to match more pertinent data later in the subject string.

\K	Resets start of match at the current point in subject string. Note that groups already captured are left alone and still populate the returned array; it is therefore always possible to backreference to them later on. Action of **\K** is similar but not identical to a look-behind, in that **\K** can work on alternations of varying lengths.

Conditional patterns

These constructs are similar to ***If...EndIf*** and ***If...Else...EndIf*** blocks.

(?*(condition)yes-pattern***)**	Allows conditional execution of pattern.
(?*(condition)yes-pattern* **|** *no-pattern***)**	Chooses between distinct patterns depending on the result of *(condition)*.

where *(condition)* can be any of the following form:

(*n***)**	Tests whether the capturing group with absolute number *n* matched.
(+*n***)**	Tests whether the capturing group with relative number +*n* matched.
(-*n***)**	Tests whether the capturing group with relative number -*n* matched.
(<*name***>)**	Tests whether the capturing group with name *name* matched.
(R)	Tests whether any kind of recursion occurred.
(R*n***)**	Tests whether the most recent recursion was for capturing group with absolute number *n*.
(R&*name***)**	Tests whether the most recent recursion was for capturing group with name *name*.
(DEFINE)	Used without *no-pattern*, permits definition of a subroutine useable from elsewhere. "(?x) (?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]?\d))" defines a subroutine named "byte" which matches any component of an IPv4 address. Then an actual address can be matched by "\b (?&byte) (\.(?&byte)){3} \b".

(*assertion***)**	Here *assertion* is one of positive or negative, look-ahead or look-behind assertion.

Miscellaneous advanced features

These options, escapes and constructs are simply mentioned here; see reference documentation for detail on why, when and how to use them, if at all.

Uncommon settings and options

(?J)	Enables duplicate group or subroutine names (not discussed further here).
(?X)	Causes some out-of-context sequences to raise an error, instead of being benign.
(*J)	Enables Javascript compatibility (not discussed further here).
(*LIMIT_MATCH=*n***)**	Limits number of matches to *n*.
(*LIMIT_RECURSION=*n***)**	Limits recursion to *n* levels.
(*NO_START_OPT)	Disables several optimizations (not discussed further here).

Backtracking control

(*ACCEPT)	Forces an immediate match success in the current subroutine or top-level pattern.
(*FAIL) or **(*F)**	Forces an immediate match failure.
(*MARK:*name***)** or **(*:***name***)**	*(See reference documentation.)*
(*COMMIT)	*(See reference documentation.)*
(*PRUNE)	*(See reference documentation.)*
(*PRUNE:*name***)**	*(See reference documentation.)*
(*SKIP)	*(See reference documentation.)*

(*SKIP:*name***)** *(See reference documentation.)*

(*THEN) *(See reference documentation.)*

(*THEN:*name***)** *(See reference documentation.)*

General comments about AutoIt regular expressions

1. When UCP is active, case sense matching applies to the full Unicode plane 0. There are also a small number of many-to-one mappings in Unicode, like the Greek lowercase letter sigma; these are supported by PCRE with UCP enabled.

2. Alternate forms of several escapes exist for compatibility with Perl, Ruby, Python, JavaScript, .NET and other engines. Do not use constructs not listed here: some will simply not work, some will supply wrong results, some will cause severe issues or merely crash.

3. The default newline convention is the unbreakable sequence @CRLF or a separate @CR or @LF. Similarly \R matches the same set. Know your data! If you know that your subjects use separate @LF or @CR to mean something else than a newline, you may have to change the newline convention and/or the matching of \R (see "Settings").

See also the Regular Expression tutorial, in which you can run a script to test your regular expression(s).