Regex Modifiers—Turning them On

This is a reference page—don't feel you have to read it as it is rather terse. Other sections link to it when needed.

Regex modifiers turn up at every corner on this site. Rather than repeatedly explain what they do and the multiple ways to turn them on in every regex flavor, I decided to gather the four common ones (`i`, `s`, `m` and `x`) in one place. The final section briefly surveys other modifiers, which are usually language-specific.

Jumping Points
For easy navigation, here are some jumping points to various sections of the page:

(direct link)

Case Insensitivity: `i`

By default, all major regex engines match in case-sensitive mode. If you want patterns such as `Name: [a-z]+` to match in case-insensitive fashion, we need to turn that feature on.
Yes, but…

What does *case-insensitive* really mean?
As long as you stick to the 26 letters of the English alphabet, the definition of upper-case and lower-case is straightforward. When you branch out into typographical niceties or other languages and scripts, things are not always so simple. Here are some questions you may run into.

✻ Will *ﬂ* (one-character, i.e. the *fl* ligature) match *FL*?
✻ Will *à* (one character) match *À*?
✻ Will *à* (two characters, i.e. the letter *a* and the grave accent) match *À*? All

engines seem to handle that correctly.
✻ Will *ß* match *ss*? No engine seems to do so.
✻ Will *i* match *İ* (Turkish capital *i*) as well as *I*?

These questions are just the tip of the iceberg. Even if I knew all the answers, it would be impossible to include them all in this section: if you use a particular script, you'll need to research how your specific engine handles case-insensitive matching in that script.

More than one way
For several engines, note that there are two ways of turning on case-insensitive matching: as an inline modifier `(?i)` or as an option in the regex method or function.

Inline Modifier `(?i)`
In .NET, PCRE (C, PHP, R…), Perl, Python, Java and Ruby (but not JavaScript), you can use the inline modifier `(?i)`, for instance in `(?i)cat`. See the section on inline modifiers for juicy details about three additional features (unavailable in Python): turning it on in mid-string, turning it off with `(?-i)`, or applying it only to the content of a non-capture group with `(?i:foo)`

.NET
Apart from the `(?i)` inline modifier, .NET languages have the `IgnoreCase` option. For instance, in C# you can use:

```
var catRegex = new Regex("cat", RegexOptions.IgnoreCase);
```

Perl
Apart from the `(?i)` inline modifier, Perl lets you add the `i` flag after your pattern's closing delimiter. For instance, you can use:

```
if ($the_subject =~ m/cat/i) { … }
```

PCRE (C, PHP, R…)
Note that in PCRE, to use case-insensitive matching with non-English letters that aren't part of your locale, you'll have to turn on Unicode mode—for instance with the `(*UTF8)` special start-of-pattern modifier.

Apart from the `(?i)` inline modifier, PCRE lets you set the `PCRE_CASELESS` mode when calling the `pcre_compile()` (or similar) function:

```
cat_regex = pcre_compile( "cat", PCRE_CASELESS,
                          &error, &erroroffset, NULL );
```

In PHP, the `PCRE_CASELESS` option is passed via the `i` flag, which you can add in your regex string after the closing delimiter. For instance, you can use:

```
$cat_regex = '~cat~i';
```

In R, the `PCRE_CASELESS` option is passed via the `ignore.case=TRUE` option. For instance, you can use:

```
grep("cat", subject, perl=TRUE, value=TRUE, ignore.case=TRUE);
```

Python
Apart from the `(?i)` inline modifier, Python has the `IGNORECASE` option. For instance, you can use:

```
cat_regex = re.compile("cat", re.IGNORECASE)
```

Java
Apart from the `(?i)` inline modifier, Java has the `CASE_INSENSITIVE` option. For instance, you can use:

```
Pattern catRegex = Pattern.compile( "cat",
                                    Pattern.CASE_INSENSITIVE |
                                    Pattern.UNICODE_CASE );
```

The `UNICODE_CASE` option added here ensures that the case-insensitivity feature is Unicode-aware. If you're only working with ASCII, you don't have to use it.

JavaScript
In JavaScript, your only option is to add the `i` flag after your pattern's closing delimiter. For instance, you can use:

```
var catRegex = /cat/i;
```

Ruby
Apart from the `(?i)` inline modifier, Ruby lets you add the `i` flag after your pattern's closing delimiter. For instance, you can use:

```
cat_regex = /cat/i
```

(direct link)

DOTALL (Dot Matches Line Breaks): `s` (with exceptions)

By default, the dot `.` doesn't match line break characters such as line feeds and carriage returns. If you want patterns such as `BEGIN .*? END` to match across lines, we need to turn that feature on.

This mode is sometimes called *single-line* (hence the *s*) because as far as the dot is concerned, it turns the whole string into one big line—`.*` will match from the first character to the last, no matter how many line breaks stand in between.

The mode is also called *DOTALL* in PCRE, Python and Java (to be more precise, the PCRE documentation uses *PCRE_DOTALL*). To me, the name *DOTALL* is a sensible way to call this mode. The third option *dot-matches-line-breaks* is descriptive but a bit of a mouthful.

For several engines, note that there are two ways of turning it on: as an inline modifier or as an option in the regex method or function.

JavaScript
JavaScript does not support single-line mode. To match any character in JavaScript, including line breaks, use a construct such as `[\D\d]`. This character class matches one character that is either a non-digit `\D` or a digit `\d`. Therefore it matches any character.

Another JavaScript solution is to use the XRegExp regex library. If you've got infinite time on your hands, you can also try porting PCRE to JavaScript using Emscripten, as Firas seems to have done on regex 101.

Inline Modifier `(?s)`
In .NET, PCRE (C, PHP, R…), Perl, Python and Java (but not Ruby), you can use the inline modifier `(?s)`, for instance in `(?s)BEGIN .*? END`. See the section on inline modifiers for juicy details about three additional features (unavailable in Python): turning it on in mid-string, turning it off with `(?-s)`, or applying it only to the content of a non-capture group with `(?s:foo)`

.NET
Apart from the `(?s)` inline modifier, .NET languages have

the `Singleline` option. For instance, in C# you can use:

```
var blockRegex = new Regex( "BEGIN .*? END",
                            RegexOptions.IgnoreCase |
                            RegexOptions.Singleline );
```

Perl
Apart from the `(?s)` inline modifier, Perl lets you add the `s` flag after your pattern's closing delimiter. For instance, you can use:

```
if ($the_subject =~ m/BEGIN .*? END/s) { … }
```

PCRE (C, PHP, R…)
Apart from the `(?s)` inline modifier, PCRE lets you set the `PCRE_DOTALL` mode when calling the `pcre_compile()` (or similar) function:

```
block_regex = pcre_compile( "BEGIN .*? END", PCRE_DOTALL,
                            &error, &erroroffset, NULL );
```

In PHP, the `PCRE_DOTALL` option is passed via the `s` flag, which you can add in your regex string after the closing delimiter. For instance, you can use:

```
$block_regex = '~BEGIN .*? END~s';
```

Python
Apart from the `(?s)` inline modifier, Python has the `DOTALL` option. For instance, you can use:

```
block_regex = re.compile("BEGIN .*? END", re.IGNORECASE | re.DOTALL)
```

Java
Apart from the `(?s)` inline modifier, Java has the `DOTALL` option. For instance, you can use:

```
Pattern blockRegex = Pattern.compile( "BEGIN .*? END",
                                      Pattern.CASE_INSENSITIVE |
                                      Pattern.DOTALL );
```

Ruby: `(?m)` modifier and `m` flag

In Ruby, you can use the inline modifier `(?m)`, for instance in `(?m)BEGIN .*? END`. This is an odd Ruby quirk as other engines use `(?m)` for the "`^` and `$` match on every line" mode.

See the section on inline modifiers for juicy details about three additional features: turning it on in mid-string, turning it off with `(?-m)`, or applying it only to the content of a non-capture group with `(?m:foo)`

Ruby also lets you to add the `m` flag at the end of your regex string. For instance, you can use:

```
block_regex = /BEGIN .*? END/m
```

(direct link)
Origins of `DOTALL`
The single-line mode is also often called *`DOTALL`* (which stands for "dot matches all") because of the `PCRE_DOTALL` option in PCRE, the `re.DOTALL` option in Python and the `Pattern.DOTALL` option in Java.

I've heard it claimed several times that "DOTALL is a Python thing" but this seemed to come from people who hadn't heard about the equivalent options in PCRE and Java. Still this made me wonder: where did `DOTALL`appear first? Looking at the PCRE Change Log and old Python documentation, it seems that it appeared in PCRE with version 0.96 (October 1997), in Python with version 1.5 (February 1998), then in Java 1.4 (February 2002). The gap between the PCRE and Python introductions wasn't conclusive—the word might have been in circulation in earlier beta versions, or even in other tools—so I asked Philip Hazel (the father of PCRE) about it. He replied:

> “ I believe I invented it — I certainly had not seen it elsewhere when I was trying to think of a name for the PCRE option that corresponds to Perl's `/s` option. ("S" there stands for "single-line" (...) so I wanted a better name.) ”

So there. Those who like a bit of history might enjoy this tasty nugget.

(direct link)
Multiline (`^` and `$` Match on Every Line): `m` (except Ruby)

By default, in all major engines except Ruby, the anchors `^` and `$` only match (respectively) at the beginning and the end of the string.

In Ruby, they match at the beginning and end of each line, and there is no way to turn that feature off. This is actually a reasonable way of doing things, with which Ruby partially redeems itself for using `m` for DOTALL mode when other engines use `s`.

In other engines, if you want patterns such as `^Define` and `>>>$` to match (respectively) at the beginning and the end of each line, we need to turn that feature on.

This feature is usually called *multi-line* (hence the *m*) because the anchors `^` and `$` operate on multiple lines.

For several engines, note that there are two ways of turning it on: as an inline modifier `(?m)` or as an option in the regex method or function.

Ruby
In Ruby, the anchors `^` and `$` always match on all lines. There is no way to turn this option off. This is actually quite a nice way to do things, since, as in most flavors, there are separate anchors for the beginning and end of strings: `\A`, `\Z` and `\z`.

On the other hand, one can regret Ruby's choice to use the `m` flag and modifier in a non-standard way (see DOTALL).

Inline Modifier `(?m)`
In .NET, PCRE (C, PHP, R…), Perl, Python, Java and Ruby (but not JavaScript), you can use the inline modifier `(?m)`, for instance in `(?m)^cat`. See the section on inline modifiers for juicy details about three additional features (unavailable in Python): turning it on in mid-string, turning it off with `(?-m)`, or applying it only to the content of a non-capture group with `(?m:foo)`

.NET
Apart from the `(?m)` inline modifier, .NET languages have the `Multiline` option. For instance, in C# you can use:

```
var catRegex = new Regex("^cat", RegexOptions.IgnoreCase |
                                 RegexOptions.Multiline);
```

Perl
Apart from the `(?m)` inline modifier, Perl lets you add the `m` flag after your pattern's closing delimiter. For instance, you can use:

```
if ($the_subject =~ m/^cat/m) { … }
```

PCRE (C, PHP, R…)
Apart from the `(?m)` inline modifier, PCRE lets you set the `PCRE_MULTILINE` mode when calling the `pcre_compile()` (or similar) function:

```
cat_regex = pcre_compile( "^cat",
                          PCRE_CASELESS | PCRE_MULTILINE,
                          &error, &erroroffset, NULL );
```

In PHP, the `PCRE_MULTILINE` option is passed via the `m` flag, which you can add in your regex string after the closing delimiter. For instance, you can use:

```
$cat_regex = '~^cat~m';
```

Python
Apart from the `(?m)` inline modifier, Python has the `MULTILINE` option. For instance, you can use:

```
cat_regex = re.compile("^cat", re.IGNORECASE | re.MULTILINE)
```

Java
Apart from the `(?m)` inline modifier, Java has the `MULTILINE` option. For instance, you can use:

```
Pattern catRegex = Pattern.compile( "^cat",
                                    Pattern.CASE_INSENSITIVE |
                                    Pattern.MULTILINE );
```

JavaScript
In JavaScript, your only option is to add the `m` flag after your pattern's closing delimiter. For instance, you can use:

```
var catRegex = /^cat/m;
```

(direct link)

Free-Spacing: `x` (except JavaScript)

By default, any space in a regex string specifies a character to be matched. In languages where you can write regex strings on multiple lines, the line breaks also specify literal characters to be matched. Because you cannot insert spaces to separate groups that carry different meanings (as you do between phrases and pragraphs when you write in English), a regex can become hard to read, as for instance the Meaning of Life regex from the regex humor page:

```
^(?=(?!(.)\1)([^\DO:105-93+30])(?-1)(?<!\d(?<=(?![5-90-3])\d))).[^\WHY?]$
```

Luckily, many engines support a *free-spacing* mode that allows you to aerate your regex. For instance, you can add spaces between the tokens. In PHP, you could write this—note the `x` flag after the final delimiter `~`:

```
$word_with_digit_and_cap_regex = '~ ^ (?=\D*\d) \w*[A-Z]\w* $ ~x';
```

But why stay on one line? You can spread your regex over as many lines as you like—indenting and adding comments—which are introduced by a `#`. For instance, in C# you can do something like this:

```
var wordWithDigitAndCapRegex = new Regex(
@"(?x)      # Free-spacing mode

  ^         # Assert that position = beginning of string

  #########   Lookahead   ##########
  (?=       # Start lookahead
     \D*       # Match any non-digits
     \d        # Match one digit
  )         # End lookahead

  ######## Matching Section ########
  \w*       # Match any word chars
  [A-Z]     # Match one upper-case letter
  \w*       # Match any word chars

  $         # Assert that position = end of string
");
```

This mode is called *free-spacing mode*. You may also see it called *whitespace mode*, *comment mode* or *verbose mode*.

It may be overkill in a simple regex like the one above (although anyone who has to maintain your code will thank you for it). But if you're building a serious regex pattern like the one in the trick to match numbers in plain English… Unless you're a masochist, you have no choice.

Note that inside a character class, the space character and the # (which otherwise introduces comments) are still honored—except in Java, where they both need to be escaped if you mean to match these characters.

For several engines, there are two ways of turning the free-spacing mode on: as an inline modifier or as an option in the regex method or function.

Free-spacing mode is wonderful, but there are a couple of minor hazards you should be aware of, as they may leave you scratching your head wondering why a pattern is not working as you expect.

Trip Hazard #1: The Meaning of Space
First, you can no longer use `Number: \d+` to match a string such as *Number: 24*. The reason is that the space in `: \d` no longer matches a space. We're in free-spacing mode, remember? That's the whole point.

To match a space character, you need to specify it. The two main ways to do so are to place it inside a character class, or to escape it with a backslash. Either of those would work: `Number:[ ]\d+` or `Number:\ \d+`

Of course `Number:\s\d+` would also match, but remember that `\s` matches much more than a space character. For instance, it could match a tab or a line break. This may not be what you want.

Trip Hazard #2: Late Start
Second, you may get overconfident in the power of free-spacing and try something like this in order to let the regex stand on its own:

```
var wordWithDigitAndCapRegex = new Regex(@"
       (?x)      # Free-spacing mode
       ^         # Beginning of string
       etc       # Match the literal chars e,t,c
");
```

The problem with this is that although it may look as though the free-spacing modifier `(?x)` is the first thing in your regex, it is not.

After the opening double-quote `"`, we have a line break and a number of spaces. The engine tries to match those, because at that stage we are not yet in free-spacing mode. That mode is turned on only when we encounter `(?x)`. This regex will never match the string *etc and more*, because by the time we encounter the beginning of string anchor `^`, we're supposed to already have matched a line break and space characters!

This is why if you look at the first example, you will see that the free-spacing modifier `(?x)` is the very first thing after the opening quote character.

Whitespace is not just trimmed out of the pattern
Even though whitespace is ignored, the position of a whitespace still separates the previous token from the next. For instance,

✻ `(A)\1 2` is not the same as `(A)\12`. The former matches *AA2*, the latter matches *A\n* in .NET, PCRE, Perl and Ruby (12 is the octal code for the linefeed character)
✻ `\p{Nd}` is valid, but `\p{N d}` is not—except in Perl and Ruby

JavaScript
JavaScript does not support free-spacing mode. In JavaScript, to match any character including line breaks, use a construct such as `[\D\d]`. This character class matches one character that is either a non-digit `\D` or a digit `\d`. Therefore it matches any character.

Another JavaScript solution is to use the XRegExp regex library. If you've got infinite time on your hands, you can also try porting PCRE to JavaScript using Emscripten, as Firas seems to have done on regex 101.

Inline Modifier `(?s)`
In .NET, PCRE (C, PHP, R…), Perl, Python, Java and Ruby (but not JavaScript), you can use the inline modifier `(?x)`, for instance, this is an aerated regex to match repeated words:

```
(?x)  (\w+)  [ \r\n]+  \1\b
```

Also see the section on inline modifiers.

.NET
Apart from the `(?x)` inline modifier, .NET languages have the `IgnorePatternWhitespace` option. For instance, in C# you can use:

```
var repeatedWordRegex = new Regex(@"
                           (\w+) [ \r\n]+ \1\b",
                           RegexOptions.IgnorePatternWhitespace
                        );
```

Perl
Apart from the `(?x)` inline modifier, Perl lets you add the `x` flag after your pattern's closing delimiter. For instance, you can use:

```
if ($the_subject =~ m/(\w+) [ \r\n]+ \1\b/x) { … }
```

PCRE (C, PHP, R…)
Apart from the `(?x)` inline modifier, PCRE lets you set the `PCRE_EXTENDED` mode when calling the `pcre_compile()` (or similar) function:

```
repeated_word_regex = pcre_compile( "(\w+) [ \r\n]+ \1\b",
                                    PCRE_EXTENDED,
                                    &error, &erroroffset, NULL );
```

In PHP, the `PCRE_EXTENDED` option is passed via the `x` flag, which you can add in your regex string after the closing delimiter. For instance, you can use:

```
$repeated_word_regex = '~(\w+) [ \r\n]+ \1\b~x';
```

Python
Apart from the `(?x)` inline modifier, Python has the `VERBOSE` option. For instance, you can use:

```
repeated_word_regex = re.compile(r"(\w+) [ \r\n]+ \1\b", re.VERBOSE)
```

Java
Unlike in other engines, inside a Java character class hashes introduce comments and spaces are ignored, so you need to escape them if you want to use these characters in a class, e.g. `[\#\ ]+`

Apart from the `(?x)` inline modifier, Java has the `COMMENTS` option. For instance, you can use:

```
Pattern repeatedWordRegex = Pattern.compile(
                                    "(\\w+) [ \\r\\n]+ \\1\\b",
                                    Pattern.COMMENTS );
```

Ruby
Apart from the `(?x)` inline modifier, Ruby lets you add the `x` flag at the end of your regex string. For instance, you can use:

```
repeated_word_regex = /(\w+) [ \r\n]+ \1\b/x
```

(direct link)
Other Modifiers

Some engines support modifiers and flags in addition to `i`, `s`, `m` and `x`. I plan to cover those in the pages dedicated to those flavors.

For instance,

✻ .NET has the `(?n)` modifier (also accessible via the `ExplicitCapture` option). This turns all *(parentheses)* into non-capture groups. To capture, you must use named groups.

✻ Java has the `(?d)` modifier (also accessible via the `UNIX_LINES` option). When this is on, the line feed character `\n` is the only one that affects the dot `.` (which doesn't match line breaks unless DOTALL is on) and the anchors `^` and `$` (which match line beginnings and endings in multiline mode.)

✻ Perl has several other flags. See the documentation's modifier section.

✻ PCRE has the `(?J)` modifier (also available in code via the `PCRE_DUPNAMES` option). When set, different capture groups are allowed to use the same name—though they will be assigned different numbers.

(direct link)
✻ PCRE has the `(?U)` modifier (also available in code via the `PCRE_UNGREEDY` option). When set, quantifiers are ungreedy by default. Appending a `?` makes them greedy.

(direct link)
✻ PCRE has the `(?X)` modifier (also available in code via the `PCRE_EXTRA` option). Historically, this mode has been used to enable new features in development. At the moment, it triggers errors if tokens such as `\B` are used in a character class (where normally it matches the capital letter *B*, unlike outside a character class, where it is a not-a-word-boundary assertion).

✻ PCRE has a number of **special** modifiers that can be set at the start of the pattern (these are shown below). In addition, many options can be sent to the `pcre_compile()` family of functions, if you have access to them. For details on those, get *pcre_compile.html* from the doc folder by downloading PCRE.

(direct link)

PCRE's Special Start-of-Pattern Modifiers

PCRE has a number of "special modifiers" you can set at the start of a pattern. Instead of the standard `(?z)` syntax for inline modifiers, the special modifier syntax looks like `(*MYMODIFIER)`. These modifiers are particularly useful in contexts where PCRE is integrated within a tool or a language—as they replace a number of options you would send to `pcre_compile()`.

UTF Strings

Assuming PCRE is compiled with the relevant options, you can instruct the engine to treat the subject string as various kinds of UTF strings.

✻ `(*UTF)` is a generic way to treat the subject as a UTF string—detecting whether it should be treated as UTF-8, UTF-16 or UTF-32.
✻ `(*UTF8)`, `(*UTF16)` and `(*UTF32)` treat the string as one of three specific UTF encodings.

Unicode Properties for `\d` and `\w`

By default, `\d` only matches ASCII digits, whereas `\w` only matches ASCII digits, letters and underscores.

The `(*UCP)` modifier (which stands for *Unicode Character Properties*) allows these tokens to match Unicode digits and word characters.

For instance, `(*UCP)\d+ :: \w+` matches *1٣٢١८৮8 :: Aれま래도ᚠᚱᚠ* (See demo).

In combination with `(*UCP)`, you may also need to use one of the `(*UTF)` modifiers. To see how this works, consider the output of this program with a standard Xampp PHP:

```
$string = '1٣٢١८৮8 :: Aれま래도ᚠᚱᚠ';
$utfregex[0] = "~\d+ :: \w+~";
$utfregex[1] = "~(*UCP)\d+ :: \w+~";
$utfregex[2] = "~(*UTF)(*UCP)\d+ :: \w+~";
$utfregex[3] = "~(*UTF)\d+ :: \w+~";
$utfregex[4] = "~\d+ :: \w+~u";

foreach (range(0, 4) as $i) {
          echo "$i: ".preg_match($utfregex[$i],$string)."<br />";
          }

//   Output:
```

```
//   0: 0
//   1: 0
//   2: 1 => (*UTF)(*UCP)
//   3: 0
//   4: 1 => The u flag produces the same result as (*UTF)(*UCP)
```

Line Break Modifiers
By default, when PCRE is compiled, you tell it what to consider to be a line break when encountering a `.` (as the dot it doesn't match line breaks unless in dotall mode), as well the `^` and `$` anchors' behavior in multiline mode. You can override this default with the following modifiers:

✻ `(*CR)` Only a carriage return is considered to be a line break
✻ `(*LF)` Only a line feed is considered to be a line break (as on Unix)
✻ `(*CRLF)` Only a carriage return followed by a line feed is considered to be a line break (as on Windows)
✻ `(*ANYCRLF)` Any of the above three is considered to be a line break
✻ `(*ANY)` Any Unicode newline sequence is considered to be a line break

For instance, `(*CR)\w+.\w+` matches *Line1\nLine2* because the dot is able to match the *\n*, which is not considered to be a line break. See demo.

Controling `\R`
By default, the `\R` metacharacter matches any Unicode newline sequence. When UTF-8 mode is off, these newline sequences are the `\r\n`pair, as well as the carriage return, line feed, vertical tab, form feed or next line characters. In UTF-8 mode, the token also matches the line separator and the paragraph separator character.

Two start-of-pattern modifiers let you change the behavior of `\R`:

✻ With `(*BSR_ANYCRLF)`, `\R` only matches the `\r\n` sequence, `\r` or `\n`. This can also be set when PCRE is compiled or requested via the `PCRE_BSR_ANYCRLF` option
✻ With `(*BSR_UNICODE)`, `\R` matches any Unicode newline sequence (overriding the `PCRE_BSR_ANYCRLF` option if set). This can also be set when PCRE is compiled or requested via the `PCRE_BSR_UNICODE` option

Controling Runaway Patterns
To limit the number of times PCRE calls the `match()` function, use the `(*LIMIT_MATCH=x)` modifier, setting x to the desired number.

To limit recursion, use `(*LIMIT_RECURSION=d)`, setting d to the deepest recursion level allowed.

Turning Off Optimizations

(direct link)

By default, PCRE studies the pattern and automatically makes a quantified token atomic when the following token is incompatible—for instance turning `A+X` into `A++X`. The `(*NO_AUTO_POSSESS)` modifier disables this optimization. Use this when you want to use pcretest to benchmark two patterns and make yourself feel good about all the cycles auto-possessification is saving you.

By default, PCRE performs several optimizations to find out faster whether a match will fail. The `(*NO_START_OPT)` modifier disables these optimizations.

Disabling Empty Matches

In PCRE2, `(*NOTEMPTY)` tells the engine not to return empty matches. Likewise, `(*NOTEMPTY_ATSTART)` tells the engine not to return empty matches found at the start of the subject.

Disabling Automatic Anchoring Optimization

In PCRE2, `PCRE2_NO_DOTSTAR_ANCHOR` tells the engine not to automatically anchor patterns that start with `.*`

You can read more about this flag on the PCRE2 API page (search for *PCRE2_NO_DOTSTAR_ANCHOR*).