Learn regular expressions in about 55 minutes

By Sam Hughes

Regular expressions ("regexes") are **supercharged Find/Replace string operations**. Regular expressions are used when editing text in a text editor, to:

- check whether the text contains a certain *pattern*
- *find* those pattern matches, if there are any
- pull *information* (i.e. substrings) out of the text
- make *modifications* to the text.

As well as text editors, almost every high-level programming language includes support for regular expressions. In this context "the text" is just a string variable, but the operations available are the same. Some programming languages (Perl, JavaScript) even provide dedicated syntax for regular expression operations.

But what are they?

A regular expression is just a *string*. There's no length limit, but typically the string is quite short. Some examples are:

- `I had a \S+ day today`
- `[A-Za-z0-9\-_]{3,16}`
- `\d\d\d\d-\d\d-\d\d`
- `v(\d+)(\.\d+)*`
- `TotalMessages="(.*?)"`
- `<[^<>]>`

The string is actually an extremely tiny *computer program*, and regular expression syntax is a small, terse, *domain-specific programming language*. Bearing this in mind, it shouldn't surprise you to learn that:

- Every regular expression can be broken down into a sequence of *instructions*. "Find this, then find that, then find one of these..."
- A regular expression has *input* (the text) and *output* (pattern matches, and sometimes the modified text).
- There are syntax errors - not every string is a legal regular expression!
- The syntax has some quirks and, arguably, horrors.
- A regular expression can sometimes be *compiled* to run faster.

There are also significant variations in implementation. For this document I'm going to stay focused on the core syntax which is shared by almost every regular expression implementation.

Exercise

Get a text editor which supports regular expressions. I prefer Notepad++.

Download a long prose story such as Project Gutenberg's version of H. G. Wells' *The Time Machine* and open it.

Download a dictionary such as this, unzip it and open it.

That's all for right now. There will be more exercises shortly.

Note: Regular expressions are completely incompatible with file globbing syntax, such as `*.xml`.

Basic regular expression syntax

Literals

Regular expressions contain a mixture of *literal* characters, which just represent themselves, and special characters called *metacharacters*, which do special things.

Here are those examples again. I'll underline the metacharacters.

- `I had a \S+ day today`
- `[A-Za-z0-9\-_]{3,16}`
- `\d\d\d\d-\d\d-\d\d`
- `v(\d+)(\.\d+)*`
- `TotalMessages="(.*?)"`
- `<[^<>]*>`

Most characters, including all alphanumeric characters, occur as literals. This means that they find themselves. For example, the regular expression

```
cat
```

means "find a `c`, followed by an `a`, followed by a `t`".

So far so good. This is exactly like

- a conventional Find dialogue
- Java's `String.indexOf()` function
- PHP's `strpos()` function
- etc.

Note: Unless otherwise specified, regular expressions are case-sensitive. However, almost all implementations provide a flag to enable case-insensitivity.

Another note: It is important to know whether "the text" is a sequence of *bytes* or a sequence of *Unicode characters*. In plain ASCII, the two are the same, and you can get a long way without needing to know the difference...

The dot

Our first metacharacter is the full stop, `.`. A `.` finds any single character. The regular expression

```
c.t
```

means "find a `c`, followed by any character, followed by a `t`".

In a piece of text, this will find `cat`, `cot`, `czt` and even the literal string `c.t` (`c`, full stop, `t`), but not `ct`, or `coot`.

Whitespace is significant in regular expressions. The regular expression

```
c t
```

means "find a `c`, followed by a space, followed by a `t`".

Any metacharacter can be escaped using a backslash, `\`. This turns it back into a literal. So the regular expression

```
c\.t
```

means "find a `c`, followed by a full stop, followed by a `t`".

The backslash is a metacharacter, which means that it too can be escaped using a backslash. So the regular expression

```
c\\t
```

means "find a `c`, followed by a backslash, followed by a `t`".

Caution! In some implementations, `.` finds any character *except a line break*. The meaning of "line break" can vary from implementation to implementation, too. Check your documentation. For this document, however, I'll assume that `.` finds any character.

In either case, there is usually a flag to adjust this behaviour, which is usually called `DOTALL` or similar.

Exercise

In the dictionary, using regular expressions and only what you know so far, find the word(s) with two `z`s that are as far apart as possible.

Answer

Exercise

In *The Time Machine*, use regular expressions to find a sentence ending in a preposition.

Answer

Character classes

A *character class* is a collection of characters in square brackets. This means, "find any one of these characters".

- The regular expression `c[aeiou]t` means, "find a `c` followed by a vowel followed by a `t`". In a piece of text, this will find `cat`, `cet`, `cit`, `cot` and `cut`.
- The regular expression `[0123456789]` means "find a digit".
- The regular expression `[a]` means the same as `a`: "find an `a`".
- The regular expression `[ ]` means "find a space".

Some examples of escaping:

- `\[a\]` means "find a left square bracket followed by an `a` followed by a right square bracket".
- `[\[\]ab]` means "find a left square bracket or a right square bracket or an `a` or a `b`".
- `[\\\[\]]` means "find a backslash or a left square bracket or a right square bracket". (Urgh!)

Order and duplication of characters are not important in character classes. `[dabaaabcc]` is the same as `[abcd]`.

An important note

The "rules" inside a character class are different from the rules outside of a character class. Some characters act as metacharacters inside a character class, but as literals outside of a character class. Some characters do the opposite. Some characters act as metacharacters in *both* situations, but do different things in each situation!

In particular, `.` means "find any character", but `[.]` means "find a full stop". Not the same thing!

Exercise

In the dictionary, using only what you have learned so far, use regular expressions to find the word(s) with the most consecutive vowels and the word(s) with the most consecutive consonants.

Answers

Character class ranges

Within a character class, you can express a *range* of letters or digits, using a hyphen:

- `[b-f]` is the same as `[bcdef]` and means "find a `b` or a `c` or a `d` or an `e` or an `f`".
- `[A-Z]` is the same as `[ABCDEFGHIJKLMNOPQRSTUVWXYZ]` and means "find an upper-case letter".
- `[1-9]` is the same as `[123456789]` and means "find a non-zero digit".

Outside of a character class, the hyphen has no special meaning. The regular expression `a-z` means "find an `a` followed by a hyphen followed by a `z`".

Ranges and isolated characters may coexist in the same character class:

- `[0-9.,]` means "find a digit or a full stop or a comma".
- `[0-9a-fA-F]` means "find a hexadecimal digit".
- `[a-zA-Z0-9\-]` means "find an alphanumeric character or a hyphen".

Although you can *try* using non-alphanumeric characters as endpoints in a range (e.g. `abc[!-/]def`), this is not legal syntax in every implementation. Even where it is legal, it is not obvious to a reader exactly which characters are included in such a range. Use caution (by which I mean, don't do this).

Equally, range endpoints should be chosen from the same range. Even if a regular expression like `[A-z]` is legal in your implementation of choice, it may not do what you think. (Hint: there are characters between `Z` and `a`...)

Caution. Ranges are ranges of characters, not ranges of numbers. The regular expression `[1-31]` means "find a `1` or a `2` or a `3`", *not* "find an integer from `1` to `31`".

Exercise

Using what you have learned so far, write a regular expression to find a date in YYYY-MM-DD format.

Answer

Character class negation

You may *negate* a character class by putting a caret at the very beginning.

- `[^a]` means "find any character other than an `a`".
- `[^a-zA-Z0-9]` means "find a non-alphanumeric character".
- `[\^abc]` means "find a caret or an `a` or a `b` or a `c`".
- `[^\^]` means "find any character other than a caret". (Ugh!)

Exercise

In the dictionary, use regular expressions to find counterexamples to the rule "`i` before `e` except after `c`".

Answer

Freebie character classes

The regular expression `\d` means the same as `[0-9]`: "find a **d**igit". (To find a backslash followed by a `d`, use the regular expression `\\d`.)

`\w` means the same as `[0-9A-Za-z_]`: "find a **w**ord character".

`\s` means "find a **s**pace character (space, tab, carriage return or line feed)".

Furthermore,

- `\D` means `[^0-9]`: "find a non-digit".
- `\W` means `[^0-9A-Za-z_]`: "find a non-word character".
- `\S` means "find a non-space character".

These are extremely common and you must learn them.

As you may have noticed, the full stop, `.`, is essentially **a character class containing every possible character**.

Many implementations provide numerous additional character classes, or flags which extend these existing classes to cover characters beyond plain ASCII. Hint: Unicode contains more "digit characters" than just `0` to `9`, and the same is true of "word" and "space" characters. Check your documentation.

Exercise

Simplify the regular expression `[0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]`.

Answer

Multipliers

You can use braces to put a *multiplier* after a literal or a character class.

- The regular expression `a{1}` is the same as `a` and means "find an `a`".
- `a{3}` means "find an `a` followed by an `a` followed by an `a`".
- `a{0}` means "find the empty string". By itself, this appears to be useless. If you use this regular expression on any piece of text, you will immediately get a match, right at the point where you started searching. This remains true even if your text is the empty string!
- `a\{2\}` means "find an `a` followed by a left brace followed by a `2` followed by a right brace".
- Braces have no special meaning inside character classes. `[{}]` means "find a left brace or a right brace".

Caution. Multipliers have no memory. The regular expression `[abc]{2}` means "find `a` or `b` or `c`, followed by `a` or `b` or `c`". This is the same as "find `aa` or `ab` or `ac` or `ba` or `bb` or `bc` or `ca` or `cb` or `cc`". It does *not* mean "find `aa` or `bb` or `cc`"!

Exercises

Simplify these regular expressions:

- `z.......z`
- `\d\d\d\d-\d\d-\d\d`
- `[aeiou][aeiou][aeiou][aeiou][aeiou][aeiou]`
- `[bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz][bcdfghjklmnpqrstvwxyz]`

Answers

Multiplier ranges

Multipliers may have *ranges*:

- `x{4,4}` is the same as `x{4}`.
- `colou{0,1}r` means "find `colour` or `color`".
- `a{3,5}` means "find `aaaaa` or `aaaa` or `aaa`".

Note that the longer possibility is most preferred, because multipliers are *greedy*. If your input text is `I had an aaaaawful day` then this regular expression will find the `aaaaa` in `aaaaawful`. It won't just stop after three `a`s.

The multiplier is greedy, but it won't disregard a perfectly good match. If your input text is `I had an aaawful daaaaay`, then this regular expression will find the `aaa` in `aaawful` on its first match. Only if you then say "find me another match" will it continue searching and find the `aaaaa` in `daaaaay`.

Multiplier ranges may be *open-ended*:

- `a{1,}` means "find one or more `a`s in a row". Your multiplier will still be greedy, though. After finding the first `a`, it will try to find as many more `a`s as possible.
- `.{0,}` means "find anything". No matter what your input text is - even the empty string - this regular expression will successfully match the entire text and return it to you.

Exercises

Write a regular expression to find a double-quoted string. The string may have any number of characters.

Modify your regular expression so that the double-quoted string that you find has no extra double-quotes between it. Again, use only what you've already learned.

Answers

Freebie multipliers

`?` means the same as `{0,1}`. For example, `colou?r` means "find `colour` or `color`".

`*` means the same as `{0,}`. For example, `.*` means "find anything", exactly as above.

`+` means the same as `{1,}`. For example, `\w+` means "find a word". Here a "word" is a sequence of 1 or more "word characters", such as `_var` or `AccountName1`.

These are extremely common and you must learn them. Also:

- `\?\*\+` means "find a question mark followed by an asterisk followed by a plus sign".
- `[?*+]` means "find a question mark or an asterisk or a plus sign".

Exercises

Simplify these regular expressions:

- `".{0,}"` and `"[^"]{0,}"`
- `x?x?x?`
- `y*y*`
- `z+z+z+z+`

Answers

Exercise

Write an expression to find two words separated by non-word characters. What about three words? What about six?

Answers

Non-greed

The regular expression `".*"` means "find a double quote, followed by as many characters as possible, followed by a double quote". Notice how the inner characters, caught by `.*`, could easily be more double quotes. This is not usually very useful.

Multipliers can be made *non-greedy* by appending a question mark. This reverses the order of preference:

- `\d{4,5}?` means "find `\d\d\d\d` or `\d\d\d\d\d`". This has exactly the same behaviour as `\d{4}`.
- `colou??r` is `colou{0,1}?r` which means "find `color` or `colour`". This has the same behaviour as `colou?r`.
- `".*?"` means "find a double quote, followed by as *few* characters as possible, followed by a double quote". This, unlike the two examples above, is actually useful.

Alternation

You can match one of several *choices* using pipes:

- `cat|dog` means "find `cat` or `dog`".
- `red|blue|` and `red||blue` and `|red|blue` all mean "find `red` or `blue` or the empty string".
- `a|b|c` is the same as `[abc]`.
- `cat|dog|\|` means "find `cat` or `dog` or a pipe".
- `[cat|dog]` means "find `a` or `c` or `d` or `g` or `o` or `t` or a pipe".

Exercises

Simplify these regular expressions as much as you can:

- `s|t|u|v|w`
- `aa|ab|ba|bb`
- `[abc]|[^abc]`
- `[^ab]|[^bc]`
- `[ab][ab][ab]?[ab]?`

Answers

Exercise

Write a regular expression to match an integer between 1 and 31 inclusive. Remember, `[1-31]` is not the right answer.

Answer

Grouping

You may *group* expressions using parentheses:

- To find a day of the week, use `(Mon|Tues|Wednes|Thurs|Fri|Satur|Sun)day`.
- `(\w*)ility` is the same as `\w*ility`. Both mean "find a word ending in `ility`". For why the first form might be useful, see later...
- `\(\)` means "find a left parenthesis followed by a right parenthesis".
- `[()]` means "find a left parenthesis or a right parenthesis".

Exercise

In *The Time Machine*, use regular expressions to find an expression wrapped in parentheses. Then, modify your answer so that the match has no parentheses within it.

Answer

Groups may contain the empty string:

- `(red|blue|)` means "find `red` or `blue` or the empty string".
- `abc()def` means the same as `abcdef`.

You may use multipliers on groups:

- `(red|blue)?` means the same as `(red|blue|)`.
- `\w+(\s+\w+)*` means "find one or more words separated by whitespace".

Exercise

Simplify `\w+\W+\w+\W+\w+` and `\w+\W+\w+\W+\w+\W+\w+\W+\w+\W+\w+`.

Answers

Word boundaries

A *word boundary* is the place between a word character and a non-word character. Remember, a word character is `\w` which is `[0-9A-Za-z_]`, and a non-word character is `\W` which is `[^0-9A-Za-z_]`.

The beginning and end of the text always count as word boundaries too.

In the input text `it's a cat`, there are eight word boundaries. If we added a trailing space after `cat`, there would be nine word boundaries.

- The regular expression `\b` means "find a word boundary".
- `\b\w\w\w\b` means "find a three-letter word".
- `a\ba` means "find `a`, followed by a word boundary, followed by `a`". This regular expression will never successfully find a match, no matter what the input text.

Word boundaries are not characters. They have zero width. The following regular expressions are identical in behaviour:

- `(\bcat)\b`
- `(\bcat\b)`
- `\b(cat)\b`
- `\b(cat\b)`

Exercise

Find the longest word(s) in the dictionary.

Answer

Line boundaries

Every piece of text breaks down into one or more *lines*, separated by *line breaks*, like this:

- Line
- Line break
- Line
- Line break
- ...
- Line break
- Line

Note how the text always ends with a line, never with a line break. However, any of the lines may contain zero characters, including the final line.

A *start-of-line* is the place between a line break and the first character of the next line. As with word boundaries, the beginning of the text also counts as a start-of-line.

An *end-of-line* is the place between the last character of a line and the line break. As with word boundaries, the end of the text also counts as an end-of-line.

So our new breakdown is:

- Start-of-line, line, end-of-line
- Line break
- Start-of-line, line, end-of-line
- Line break
- ...
- Line break
- Start-of-line, line, end-of-line

Based on this:

- The regular expression `^` means "find a start-of-line".
- The regular expression `$` means "find an end-of-line".
- `^$` means "find an empty line".
- `^.*$` will find your entire text, because a line break is a character and `.` will find it. To find a single line, use a non-greedy multiplier, `^.*?$`.
- `\^\$` means "find a caret followed by a dollar sign".
- `[$]` means "find a dollar sign". **However, `[^]` is not a valid regular expression.** Remember that the caret has a *different* special meaning inside square brackets! To put a caret in a character class, use `[\^]`.

Like word boundaries, line boundaries are not characters. They have zero width. The following regular expressions are identical in behaviour:

- `(^cat)$`
- `(^cat$)`
- `^(cat)$`
- `^(cat$)`

Exercise

Use regular expressions to find the longest line in *The Time Machine*.

Answer

Text boundaries

Many implementations provide a flag which changes the meaning of `^` and `$` from "start-of-line" and "end-of-line" respectively to "start-of-text" and "end-of-text" respectively.

Other implementations provide the separate metacharacters `\A` and `\z` for this purpose.

Capturing and replacing

This is where regular expressions start to get extremely powerful.

Capture groups

You already know that parentheses are used to denote groups. They are also used to capture substrings. If a regular expression is a very small computer program, the *capture groups* are (part of) its output.

The regular expression `(\w*)ility` means "find a word ending in `ility`". Capture group 1 is the part matched by `\w*`. For example, if our text contains the word `accessibility`, capture group 1 is `accessib`. If our text just contains `ility` all by itself, capture group 1 is the empty string.

You can have multiple capture groups, and they can even nest. Capture groups are numbered from left to right. Just count the left-parentheses.

Suppose our regular expression is `(\w+) had a ((\w+) \w+)`. If our input text is `I had a nice day`, then

- Capture group 1 is `I`.
- Capture group 2 is `nice day`.
- Capture group 3 is `nice`.

In some implementations, you will also have access to capture group 0, which is the entire match: `I had a nice day`.

In some implementations, if there are no capture groups, capture group 1 is automatically populated with the value of capture group 0.

Yes, all of this does mean that parentheses are somewhat overloaded. Some implementations provide a separate syntax to declare a "non-capturing group", but the syntax isn't standardised and so won't be covered here.

The number of capture groups returned from a successful match is *always* equal to the number of capture groups in the original regular expression. Remember this, as it can help you with some confusing cases.

The regular expression `((cat)|dog)` means "find `cat` or `dog`". There are **always two capture groups**. If our input text is `dog`, then capture group 1 is `dog`, and capture group 2 is the empty string, because that choice was not used.

The regular expression `a(\w)*` means "find a word beginning with `a`". There is **always one capture group**:

- If the input text is `a`, capture group 1 is the empty string.
- If the input text is `ad`, capture group 1 is `d`.
- If the input text is `apricot`, capture group 1 is `t`. However, capture group 0 would be the entire word, `apricot`.

Replacement

Once you've used a regular expression to find a string, you can specify another string to replace it with. The second string is the *replacement expression*. At first, this is exactly like

- a conventional Replace dialogue
- Java's `String.replace()` function
- PHP's `str_replace()` function
- etc.

Exercise

Replace all the vowels in *The Time Machine* with the letter `r`. Be sure to use the correct case!

Answer

However, you can refer to capture groups in your replacement expression. This is the only special thing you can do in replacement expressions, and it's incredibly powerful because it means you don't have to completely destroy the thing you just found.

Let's say you're trying to replace American-style dates (MM/DD/YY) with ISO 8601 dates (YYYY-MM-DD).

- Start with the regular expression `(\d\d)/(\d\d)/(\d\d)`. Note that this has three capture groups: the month, the day and the two-digit year.

- Capture groups are referred to using a backslash and then the capture group number. So, your replacement expression is `20\3-\1-\2`.

- If our input text contains `03/04/05` (representing March 4, 2005), then

  - Capture group 1 is `03`.
  - Capture group 2 is `04`.
  - Capture group 3 is `05`.
  - The replacement string is `2005-03-04`.

You can refer to capture groups more than once in the replacement expression.

- To double up vowels, use the regular expression `([aeiou])` and the replacement expression `\1\1`.

Backslashes must be escaped in the replacement expression. For example, let's say you have some text which you want to use in a string literal in a computer program. That means you need to put a backslash in front of every double quote or backslash in the original text.

- Your regular expression would be `([\\"])`. Capture group 1 is the double quote or backslash.
- Your replacement expression would be `\\\1`; a literal backslash followed by the captured double quote or backslash.

Instead of a backslash, some implementations use the dollar sign `$` to indicate capture groups.

Exercise

Write a regular expression and replacement expression which could take timestamps such as `23h59` and turn them into `23:59`.

Answer

Back-references

You can also refer to a captured group later in the same regular expression. This is called a *back-reference*.

For example, recall that the regular expression `[abc]{2}` means "find `aa` or `ab` or `ac` or `ba` or `bb` or `bc` or `ca` or `cb` or `cc`". But the regular expression `([abc])\1` means "find `aa` or `bb` or `cc`".

Exercise

In the dictionary, find the longest word which consists of the same string repeated twice (e.g. `papa`, `coco`).

Answer

Programming with regular expressions

Some notes specific to this task:

Excessive backslash syndrome

In some programming languages, such as Java, there is no special support for strings containing regular expressions. Strings have their own escaping rules, which are added on top of the escaping rules for regular expressions, commonly resulting in backslash overload. For example (still Java):

- To find a digit, the regular expression `\d` becomes `String re = "\\d";` in source code.
- To find a double-quoted string, `"[^"]*"` becomes `String re = "\"[^\"]*\"";`.
- To find a backslash or a left square bracket or a right square bracket, the regular expression `[\\\[\]]` becomes `String re = "[\\\\\\[\\]]";`.
- `String re = "\\s";` and `String re = "[ \t\r\n]";` are equivalent. Note the different "levels" of escaping.

In other programming languages, regular expressions are marked out by a special delimiter, typically the forward slash `/`. Here's some JavaScript:

- To find a digit, `\d` simply becomes `var regExp = /\d/;`.
- To find a backslash or a left square bracket or a right square bracket, `var regExp = /[\\\[\]]/;`.
- `var regExp = /\s/;` and `var regExp = /[ \t\r\n]/;` are equivalent.
- Of course, this means forward slashes must be escaped instead of double quotes. To find the first part of a URL: `var regExp = /https?:\/\//;`.

I hope you see why I've been trying to inoculate you against backslashes up to this point.

Dynamic regular expressions

Be cautious when constructing a regular expression string dynamically. If the string that you use is not fixed, then it could contain unexpected metacharacters. This could make for a syntax error. Worse, it could result in a syntactically correct regular expression, but one with unexpected behaviour.

Buggy Java code:

```
String sep = System.getProperty("file.separator");
String[] directories = filePath.split(sep);
```

The bug: `String.split()` expects `sep` to be a regular expression. But on Windows, `sep` is a string consisting of a single backslash, `"\\"`. This is not a syntactically correct regular expression! The result: a `PatternSyntaxException`.

Any good programming language provides a mechanism to escape all the metacharacters in a string. In Java, you would do this:

```
String sep = System.getProperty("file.separator");
String[] directories = filePath.split(Pattern.quote(sep));
```

Regular expressions in loops

Compiling a regular expression string into a working "program" is a relatively expensive operation. You may find performance improves if you can avoid doing this inside a loop.

Miscellaneous advice

Input validation

Regular expressions can be used to validate user input. But excessively strict validation can make users' lives very difficult. Examples follow:

Payment card numbers

On a website, I entered my card number as `1234 5678 8765 4321`. The site rejected it. It was validating the field using `\d{16}`.

The regular expression should allow for spaces. And hyphens.

In fact, why not just strip out all non-digit characters first and *then* perform the validation? To do this, use the regular expression `\D`, and an empty string for the replacement expression.

Exercise

Write a regular expression which could validate my card number without stripping out non-digit characters first.

Answer

Names

Do not use regular expressions to validate people's names. In fact, do not validate names at all if you can possibly help it.

Falsehoods programmers believe about names:

- Names do not contain spaces.
- Names do not contain punctuation.
- Names use only ASCII characters.
- Names are restricted to *any* particular character set.
- Names are always at least *M* characters long.
- Names are never more than *N* characters long.
- People always have exactly one forename.

- People always have exactly one middle name.
- People always have exactly one surname.
- ...

Email addresses

Do not use regular expressions to validate email addresses.

Firstly, this is extremely difficult to do correctly. Email addresses do indeed conform to a regular expression, but the expression is apocalyptically long and complicated. Anything shorter is likely to yield false negatives. (Did you know? Email addresses can contain comments!)

Secondly, even if the supplied email address conforms to the regular expression, this doesn't prove it exists. **The only way to validate an email address is to send an email to it.**

Markup

Do not use regular expressions to parse HTML or XML for serious applications. Parsing HTML/XML is

1. Impossible using simple regular expressions
2. Incredibly difficult even in general
3. A solved problem.

Find an existing parsing library which can do this for you.

And that's 55 minutes

In summary:

- Literals: `a` `b` `c` `d` `1` `2` `3` `4` etc.
- Character classes: `.` `[abc]` `[a-z]` `\d` `\w` `\s`
  - `.` means "any character"
  - `\d` means "a digit"
  - `\w` means "a word character", `[0-9A-Za-z_]`
  - `\s` means "a space, tab, carriage return or line feed character"
  - Negated character classes: `[^abc]` `\D` `\W` `\S`
- Multipliers: `{4}` `{3,16}` `{1,}` `?` `*` `+`
  - `?` means "zero or one"
  - `*` means "zero or more"
  - `+` means "one or more"
  - Multipliers are greedy unless you put a `?` afterwards

- Alternation and grouping: `(Septem|Octo|Novem|Decem)ber`
- Word, line and text boundaries: `\b` `^` `$` `\A` `\z`
- To refer back to a capture groups: `\1` `\2` `\3` etc. (works in both replacement expressions and matching expressions)
- List of metacharacters: `.` `\` `[` `]` `{` `}` `?` `*` `+` `|` `(` `)` `^` `$`
- List of metacharacters when inside a character class: `[` `]` `\` `-` `^`
- You can always escape a metacharacter using a backslash: `\`

Thanks for reading

Regular expressions are ubiquitous and incredibly useful. Everybody who spends any amount of time editing text or writing computer programs should know how to use them.

So far today we have only scratched the surface...

Exercise

Go and read the documentation for your regular expression implementation of choice. I guarantee that there are more features there than we've covered here.

Back to Things Of Interest