- GETTING TO GRIPS WITH -

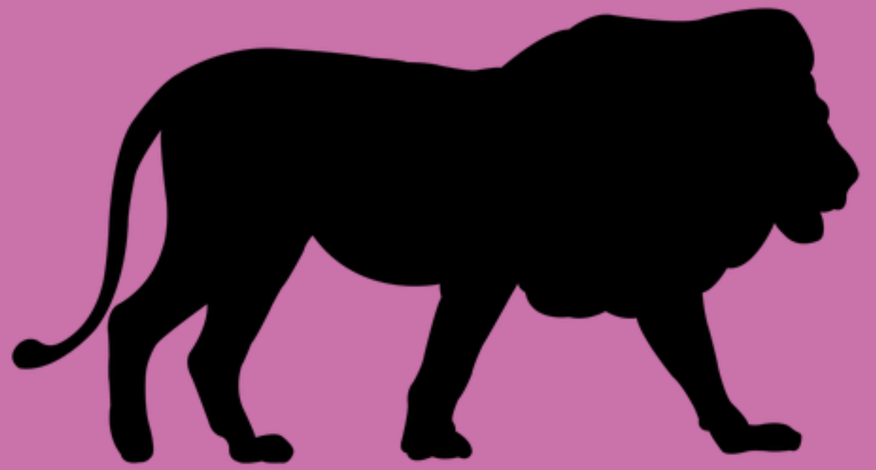
Regular Expressions

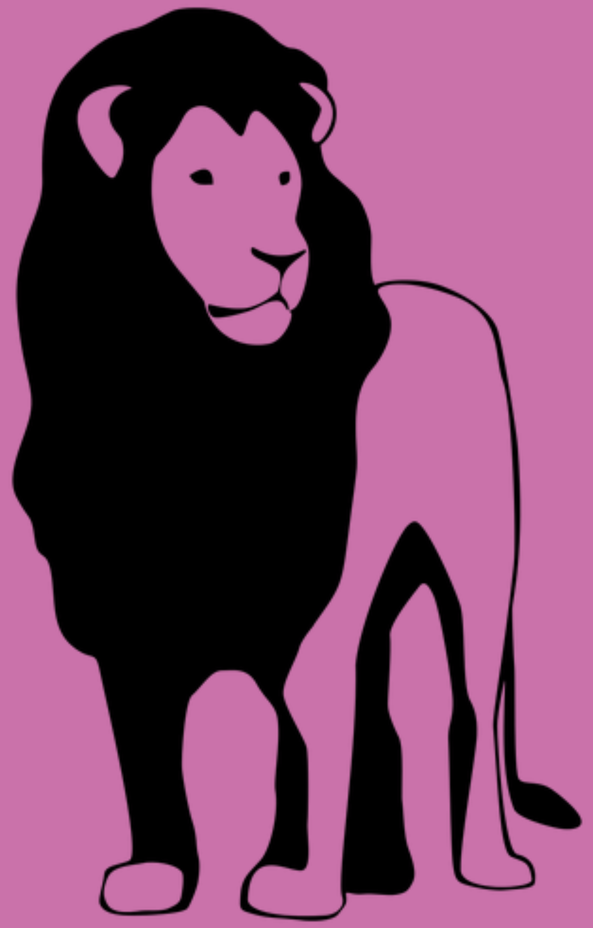
- DREW MCLELLAN -

- CONFOO 2015 -

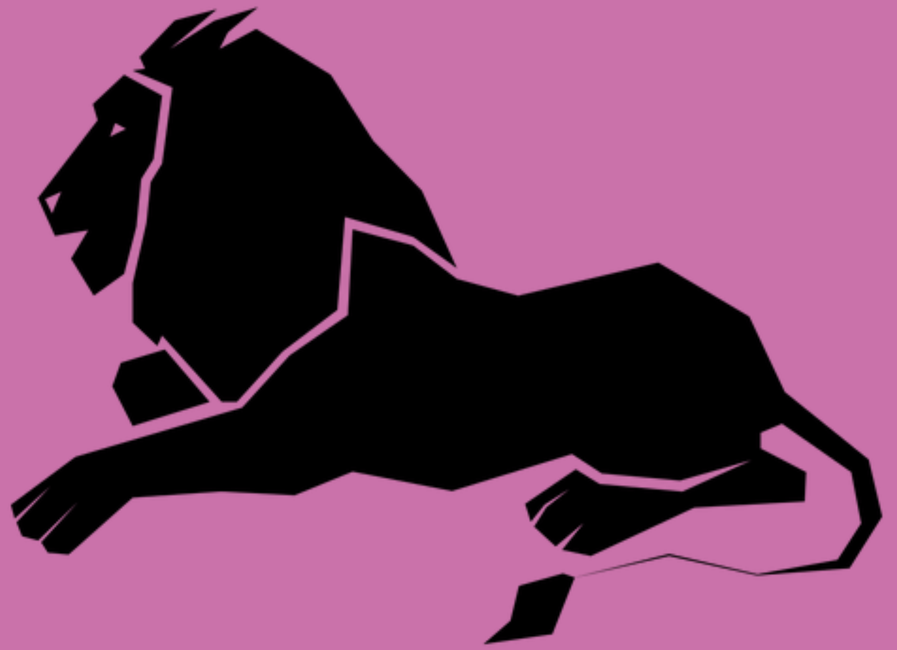
For the fearful.

Hello!





Created by Christy Presler from the Noun Project



Created by Yi Chen from the Noun Project

Humans are great at matching patterns.

RegExp are great at matching patterns.

RegExp

Humans

Donec in euismod mi. Ut a ullamcorper eros, id ultricies odio. In ullamcorper lobortis finibus. Nunc molestie, ex id ultrices lobortis, ante elit **Finding** mauris consequat lacus, at scelerisque leo nisl vitae leo. cursus lacus eu erat euismod tincidunt. Etiam ultrices elementum nulla, eu ornare elit eleifend a. Mauris lacinia velit non maximus ultrices. Praesent in condimentum metus. Curabitur hendrerit eget **text** id egestas. Nam et sodales dui. Suspendisse potenti. Mauris sed suscipit dui. Suspendisse ultricies felis non lacus maximus rutrum. Duis vel ante et neque ornare sagittis eu a nisi. Curabitur ultrices aliquet magna ut venenatis. Duis nec rhoncus **that**, sed pulvinar dui. Nunc pellentesque tortor sem, convallis eleifend nibh pharetra eu. Nulla congue, nisi vitae consectetur sollicitudin, felis nisl malesuada tortor, ut semper sem tellus ut dui. Donec eget augue quis justo vestibulum sodales sit amet eget tortor. Donec viverra risus turpis, sit amet congue dolor vel **matches**. Pellentesque sollicitudin purus a ligula tristique, et posuere justo faucibus. Pellentesque vehicula id nisl sit amet mollis. Integer tempor eros id varius aliquam. Phasellus vel est ullamcorper, dignissim nulla et, iaculis ex. Maecenas a dictum orci, eu sagittis felis. Vestibulum scelerisque diam elit, vitae placerat ipsum congue nec. Nulla blandit magna vel velit feugiat, eget maximus tortor feugiat. In vel metus ex. Ut molestie enim vel dolor elementum, at **patterns** turpis volutpat. Sed pulvinar dignissim eros et interdum. Quisque scelerisque diam et facilisis consequat. Etiam gravida sodales ornare. Donec tristique sem vitae ipsum gravida, in finibus sem vulputate. Sed in ex at dolor euismod commodo sed nec augue. Maecenas sed dictum turpis, nec bibendum neque. Pellentesque dapibus mi vitae elit porttitor elementum. Vestibulum porttitor porta nunc, et laoreet eros finibus ac. Suspendisse potenti. Nunc a gravida nisi. Morbi et massa magna. Cras ligula erat, congue sit amet dignissim a, porttitor vel felis.

Regular Expressions

Server rewrite rules.

Form validation.

Text editor search & replace.

Application code.

Flavours

POSIX basic & extended.

Perl and Perl-compatible (PCRE).

Most common implementations are Perl-like
(PHP, JavaScript and HTML5, mod_rewrite, nginx)

In this exciting episode

Basic syntax.

Matching.

Repeating.

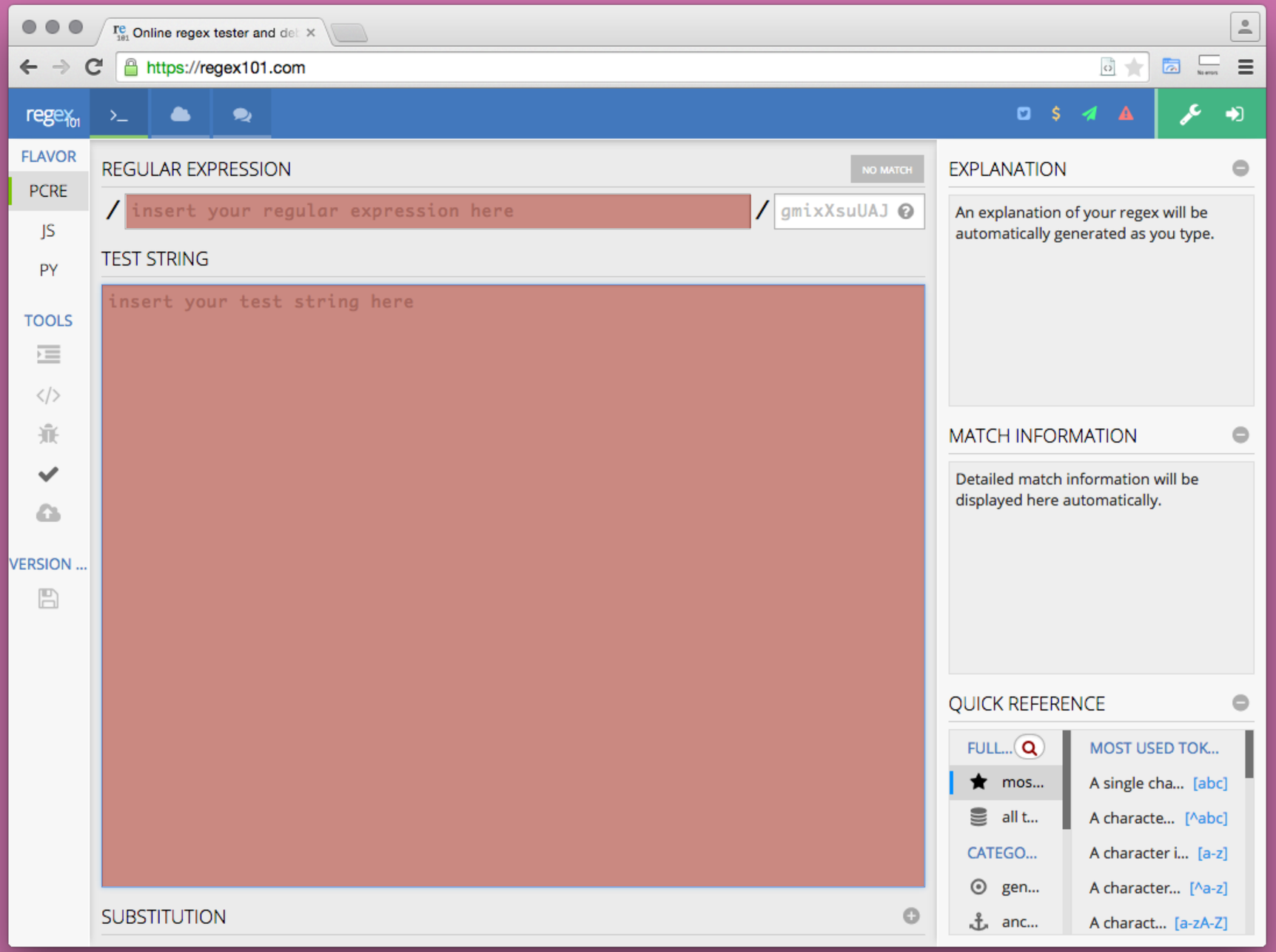
Grouping.

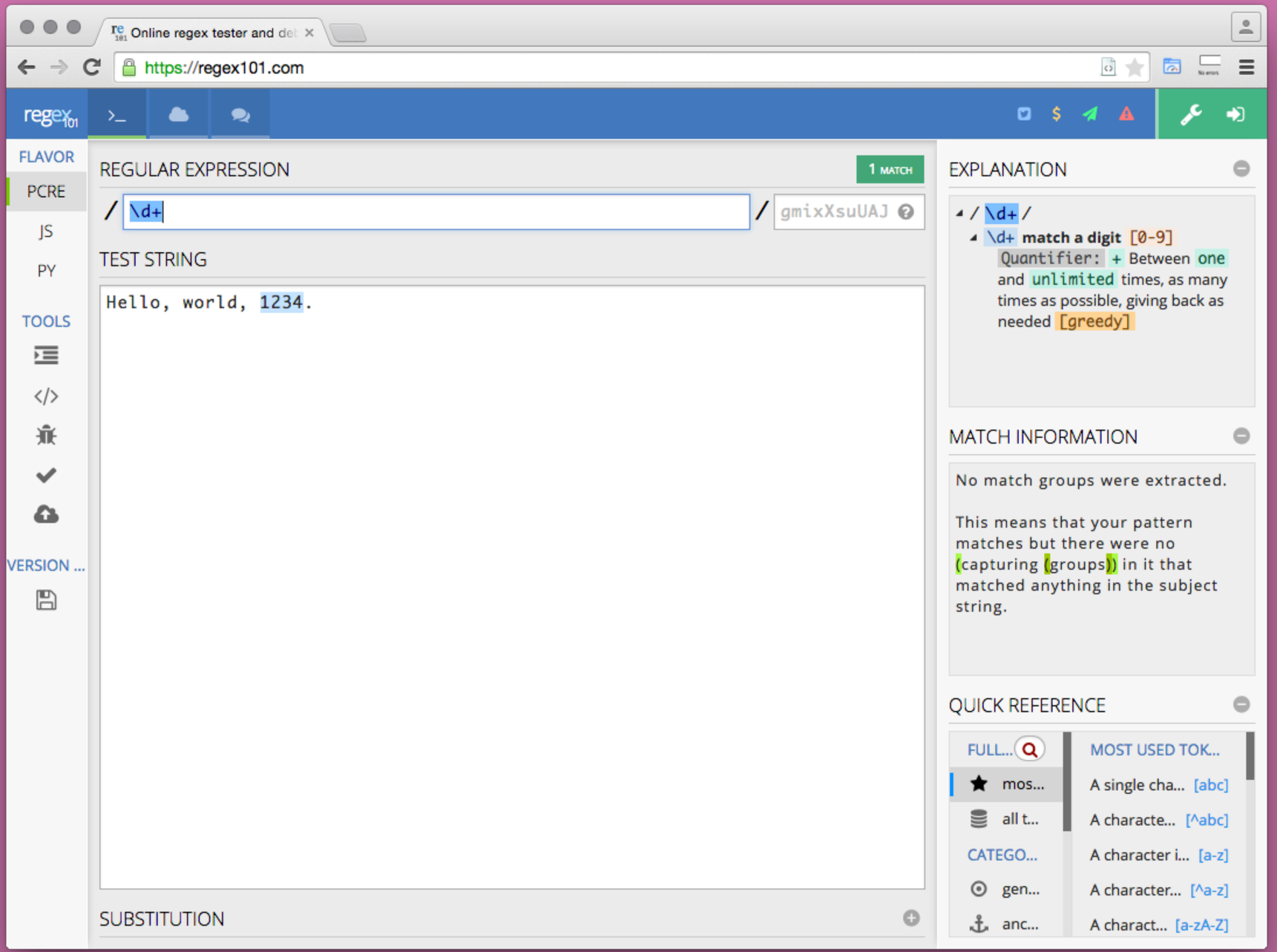
Replacing.

But first...

A regular expression *tester* is a great way to try things out.

There's an excellent online tester at:
regex101.com


Online regex tester and de
https://regex101.com
regex101
FLAVOR
PCRE
JS
PY
TOOLS
VERSION ...
REGULAR EXPRESSION
NO MATCH
insert your regular expression here
gmixXsuUAJ
TEST STRING
insert your test string here
SUBSTITUTION
EXPLANATION
An explanation of your regex will be automatically generated as you type.
MATCH INFORMATION
Detailed match information will be displayed here automatically.
QUICK REFERENCE
FULL...
MOST USED TOK...
mos...
A single cha... [abc]
all t...
A characte... [^abc]
CATEGO...
A character i... [a-z]
gen...
A character... [^a-z]
anc...
A charact... [a-zA-Z]


Online regex tester and de
https://regex101.com
regex101
FLAVOR
PCRE
JS
PY
TOOLS
VERSION ...
REGULAR EXPRESSION
1 MATCH
/ \d+ / gmixXsuUAJ
TEST STRING
Hello, world, 1234.
SUBSTITUTION
EXPLANATION
/ \d+ /
\d+ match a digit [0-9]
Quantifier: + Between one and unlimited times, as many times as possible, giving back as needed [greedy]
MATCH INFORMATION
No match groups were extracted.
This means that your pattern matches but there were no (capturing (groups)) in it that matched anything in the subject string.
QUICK REFERENCE
FULL...
MOST USED TOK...
mos...
A single cha... [abc]
all t...
A characte... [^abc]
CATEGO...
A character i... [a-z]
gen...
A character... [^a-z]
anc...
A charact... [a-zA-Z]

RegExp Basics

Basics

Delimiters are usually slashes by default.

Some engines allow you to use other delimiters.

Modifiers include things like case sensitivity.

```
/regex goes here/

/regex goes here/modifiers

/[A-Z]\w[A-Z]/i
```

Basics

Delimiters and other special characters need to be escaped with backslashes.

```
/this\/that/
```

Basics

Anything proceeded by a backslash has a special meaning.

There are also a number of meta-characters with special meaning.

Most other things are literal.

```
/\w\s\d/

+ . * ? ^ | / () {} []

/ferret/
```

Matching

Words

Matches an alphanumeric character, including underscore.

```
\w (lowercase W)

/\w/

Hello, world, 1234.
```

Global modifier

The ‘g’ global modifier returns all matches.

Doesn’t stop at the first match.

Words

Matches an alphanumeric character, including underscore.

```
\w (lowercase W)

/\w/g

Hello, world, 1234.
```

Digits

Matches single digits 0-9.

```
\d
```

```
/\d/
```

```
Hello, world, 1234.
```

```
/\d/g
```

```
Hello, world, 1234.
```

Spaces

Matches single whitespace character.

Includes spaces, tabs, new lines.

```
\s

/\s/

Hello, world, 1234.

/\s/g

Hello, world, 1234.
```

Character classes

These are all ***shorthand character classes***.

Character classes match one character, but offer a set of acceptable possibilities for the match.

The tokens we've looked at a shorthand for more complex character classes.

Words

Character classes match one character only.

They can use ranges like A-Z.

They are denoted by [square brackets].

```
\w

[A-Za-z0-9_]
```

Digits

Character classes match one character only.

They can use ranges like A-Z.

They are denoted by [square brackets].

```
\d

[0-9]
```

Spaces

Character classes match one character only.

They can use ranges like A-Z.

They are denoted by [square brackets].

```
\s

[\r\n\t\f ]
```

!!!

\r	Carriage return
\n	New line
\t	Tab
\f	Form feed

Custom classes

```
[ol3]

/[ol3]/g

Hello, world, 1234.


[a-z0-9-]

/[a-z0-9-]/g

/2009/nice-title
```

Negative classes

Use a caret to indicate the class should match **none** of the given characters.

```
[^ol3]

/[^ol3]/g

Hello, world, 1234.


[^a-z0-9-]

/[^a-z0-9-]/g

/2009/nice-title
```

Dot

A dot (period) matches any character other than a line break.

It’s often over-used. Try to use something more specific if possible.

Dot

Matches any character other than a line break.

```
/./g
```

```
Hello, world, 1234.
```

Developer joke time.

```
!false
```

So where does this get us?

Matching

So that’s something, right?

```
Hello world (1980-02-21).
```

```
/\d\d\d\d-\d\d-\d\d/
```

```
Hello world (1980-02-21).
```

Repetition

Repetition

Matching single characters gets old fast.

There are four main operators or ‘quantifiers’ for specifying repetition.

Repetition

`?`	Match zero or once.
`+`	Match once or more.
`*`	Match zero or more.
`{x}`	Match *x* times.
`{x,y}`	Match between *x* and *y* times.

Repetition

```
/\d\d\d\d-\d\d-\d\d/
```

```
/\d{4}-\d{2}-\d{2}/
```

```
/[a-z0-9-]+/g
```

```
/2009/nice-title
```

Greediness

Repetition quantifiers are ‘greedy’ by default. They’ll try to match as many times as possible, within their scope.

Sometimes that’s not quite what we want, and we can change this behaviour to make them ‘lazy’.

Greediness

Repetition quantifiers try to match as many times as they’re allowed to.

```
/<.+>/
```

```
This <em>is</em> some HTML.
```

EXPECTED:

```
This <em>is</em> some HTML.
```

ACTUAL:

```
This <em>is</em> some HTML.
```

Greediness

Quantifiers can be made ‘lazy’ with a question mark.

```
/<.+?>/

This <em>is</em> some HTML.
```

Anchors

Anchors

Anchors don’t match characters, but the position within the string.

There are three main anchors in common use.

Anchors

^	The beginning of the string.
$	The end of the string.
\b	A word boundary.

Anchors

Anchors find matches based on position.

```
/^Hello/g
```

```
Hello, Hello
```

```
/Hello$/g
```

```
Hello, Hello
```

Anchors

Word boundaries are useful for avoiding accidental sub-matches.

```
/cat/g
```

```
cat concatenation
```

```
/\bcat\b/g
```

```
cat concatenation
```

Developer joke time.

```
['hip', 'hip']
```

Grouping

Grouping

Parts of a pattern can be grouped together with (parenthesis).

This enables repetition to be applied on the group, and enables us to control how the result is ‘captured’.

Grouping

Round brackets enable us to create groups that can then be repeated.

```
abc123-def456-ghi789
```

```
/[a-z]{3}[0-9]{3}-?/
```

```
/([a-z]{3}[0-9]{3}-?)+/
```

```
[
    'abc123-',
    'def456-',
    'ghi789'
]
```

Grouping

Groups are captured by default.

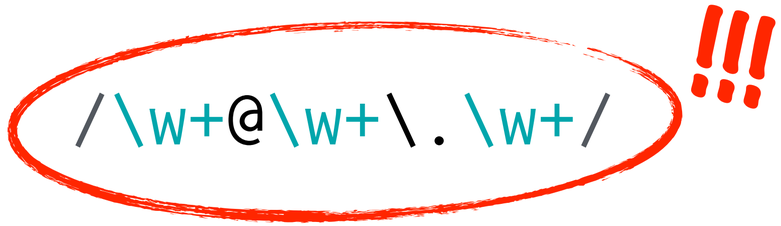
If you don’t need the group to be captured, make it non-capturing.

```
/([a-z]{3}[0-9]{3}-?)+/
```

```
/(?:[a-z]{3}[0-9]{3}-?)+/
```

Grouping

Capturing groups is very useful!




```
drew@allinthehead.com

/(\w+)@(\w+\.\w+)/

[
    'drew',
    'allinthehead.com'
]
```

Grouping

Some engines offer named groups.

```
/(?<user>\w+)@(?<domain>\w+\.\w+)/

[
    user: ‘drew’,
    domain: ‘allinthehead.com’
]
```

Replacing

Replacing

If you've used capturing groups in your pattern, you can re-insert any of those matched values back into your replacement.

This is done with 'back references'.

Back references use the index number of the captured group.

Replacing with back references

PHP uses the **preg** (Perl Regular Expression) functions to perform matches and replacements.

```
<?php

$str = 'drew@allinthehead.com';

$pattern = '/(\w+)@(\w+\.\w+)/';

$replacement = '$1 is now fred@$2';

$result = preg_replace($pattern,
$replacement, $str);

echo $result;

> drew is now fred@allinthehead.com
```

Replacing with back references

JavaScript uses the **replace()** method of a string object.

```
var str = 'drew@allinthehead.com';

var pattern = /(\w+)@(\w+\.\w+)/;

var replacement = '$1 is now fred@$2';

var result = str.replace(pattern,
replacement);

console.log(result);

> drew is now fred@allinthehead.com
```

Putting it to use

HTML5 input validation

HTML5 adds the pattern attribute on form fields.

They’re parsed using the browser’s JavaScript engine.

```
<input name="sku" type="text"
pattern="[A-Z]{3}[0-9]{8-10}">
```

Apache mod rewrite

URL rewriting in Apache uses PCRE.

```
RewriteEngine On

RewriteRule
^news/([1-2]{1}[0-9]{3})/([a-z0-9-]+)/?
/news.php?year=$1&slug=$2
```

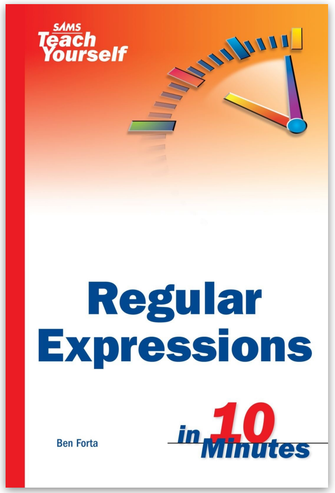
Your application code

Don’t copy this example - it’s simplified and insecure.

```
<?php

$str = 'Look at this https://
www.youtube.com/watch?v=loab4A_SqoQ and
this https://www.youtube.com/watch?
v=I-19GRsBW-Y';

$pattern = '/(\w+:\/\/[^\s"]+)/';

$replacement = '<a href="$1">$1</a>';

echo preg_replace($pattern,
$replacement, $str);

> Look at this <a href="https://
www.youtube.com/watch?
v=loab4A_SqoQ">https://www.youtube.com/
watch?v=loab4A_SqoQ</a> and this <a
href="https://www.youtube.com/watch?
v=I-19GRsBW-Y">https://www.youtube.com/
watch?v=I-19GRsBW-Y</a>
```

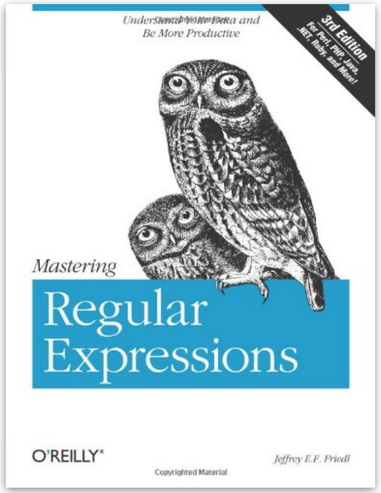
Further reading

Further reading



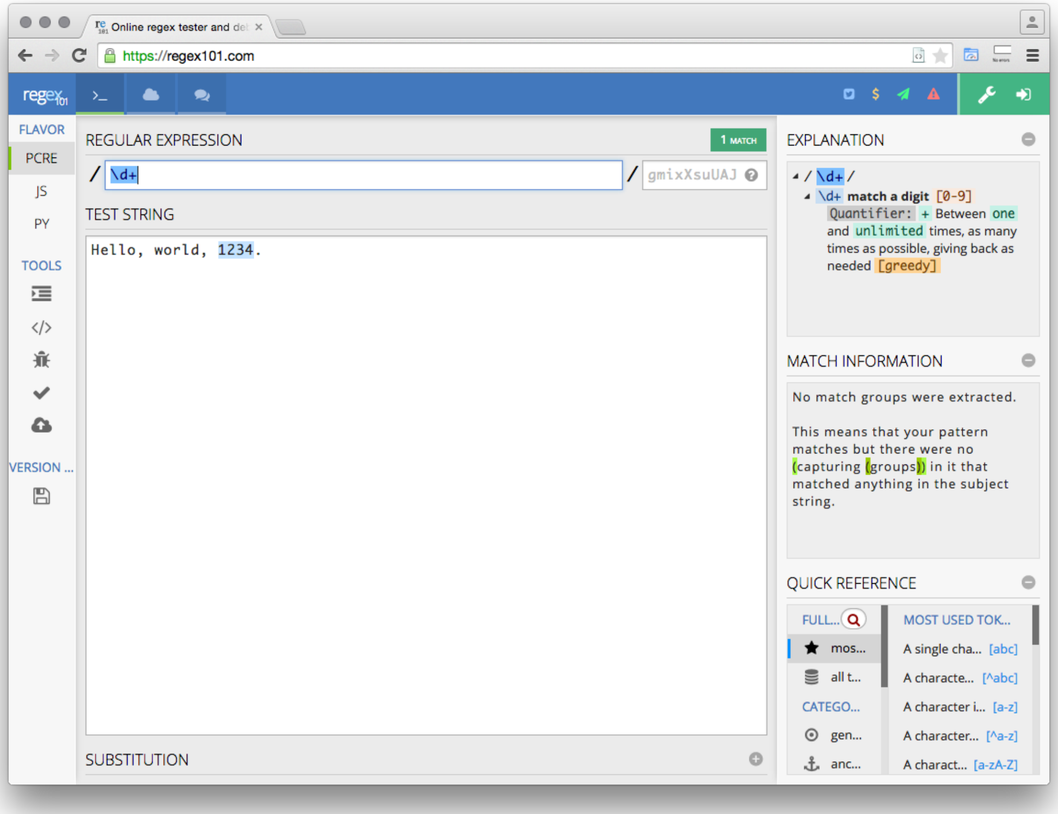
Teach Yourself Regular Expressions in 10 minutes, by Ben Forta.

(Not actually in 10 minutes.)



Mastering Regular Expressions, by Jeffrey E. F. Friedl.

Further learning



regex101.com

Thanks!

@drewm

speakerdeck.com/drewm/learn-to-love-regular-expressions