

Andrei Zmievski
andrei@php.net • @a

ConFoo
2010

Andrei's Regex Clinic



USER FRIENDLY by J.D. "Illiad" Frazer



COPYRIGHT © 2007 J.D. "Illiad" Frazer HTTP://WWW.USERFRIENDLY.ORG/

I'VE BEEN EXPERIMENTING A LOT TO PUSH THE BOUNDARIES OF MY KNOWLEDGE.

I'D SAY YOU'VE COME A LONG WAY. APPROACHING MASTERY, EVEN.



YOU REALLY THINK SO?

WELL, FOR ONE THING I CAN'T TELL IF THAT'S A REGEX OR LINE NOISE.

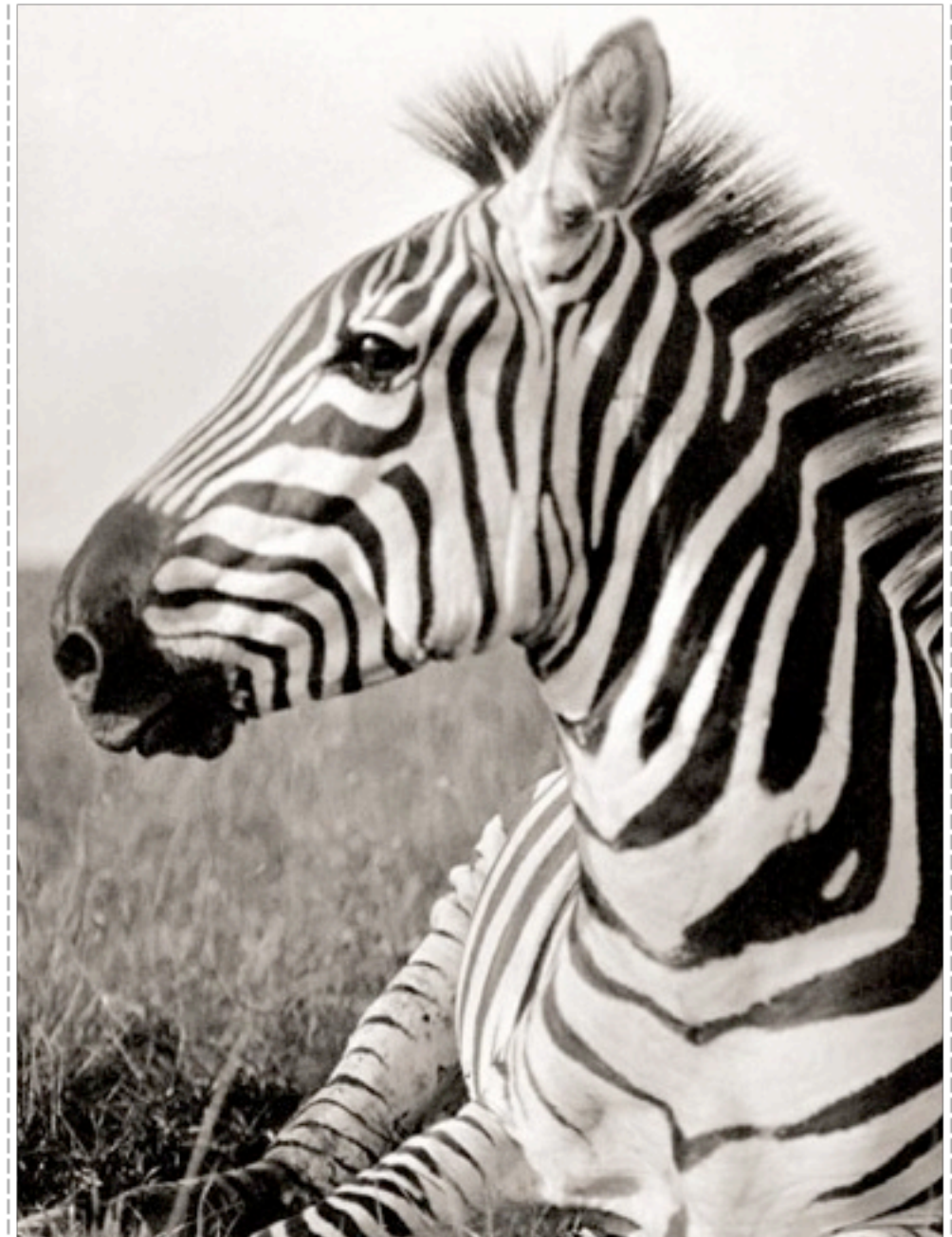


userfriendly: line noise



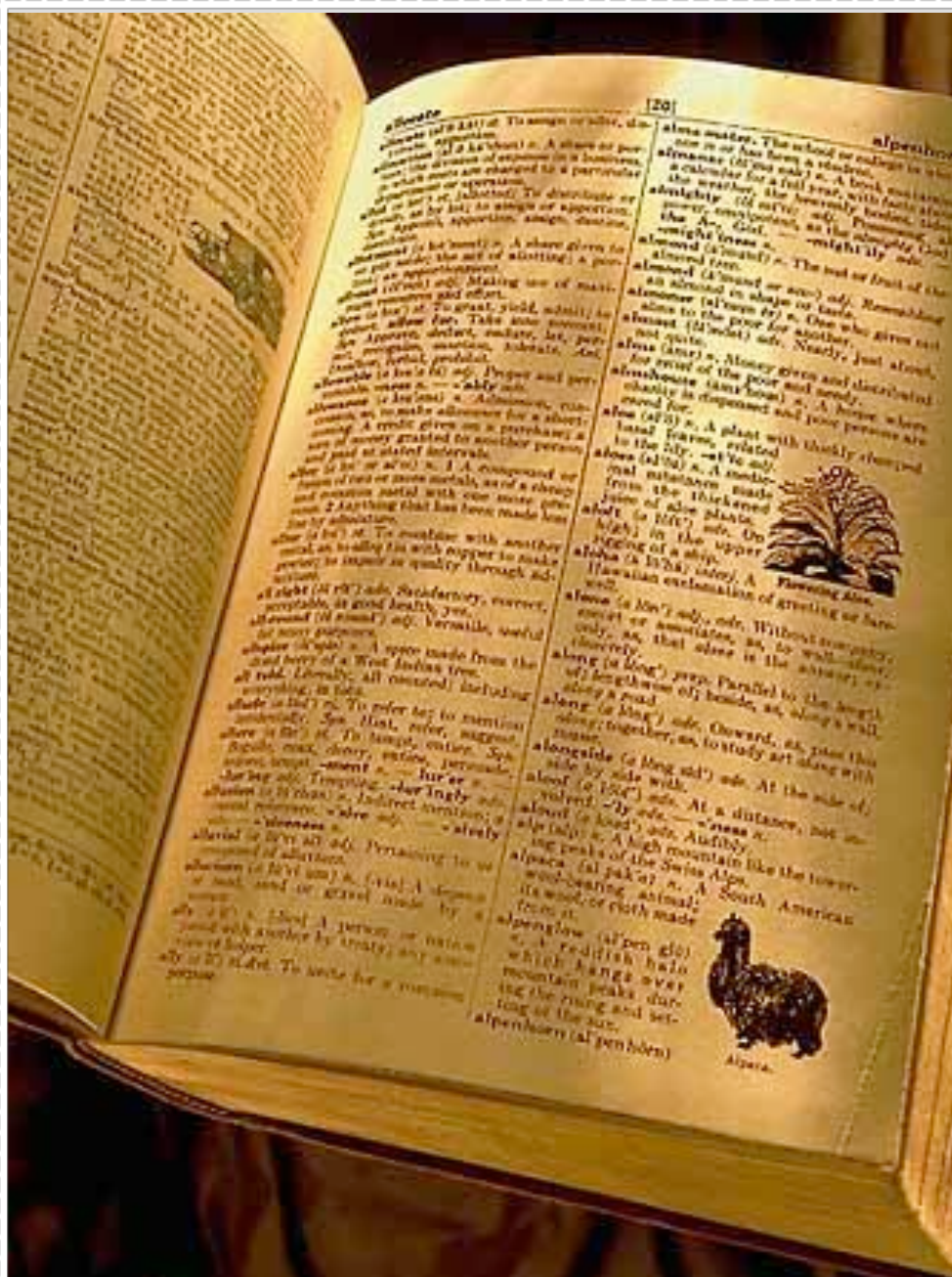
what are they good for?

- ❖ Literal string searches are fast but inflexible
- ❖ With regular expressions you can:
 - ❖ Find out whether a certain pattern occurs in the text
 - ❖ Locate strings matching a pattern and remove them or replace them with something else
 - ❖ Extract the strings matching the pattern





terminology



Regex

a pattern describing a set of strings

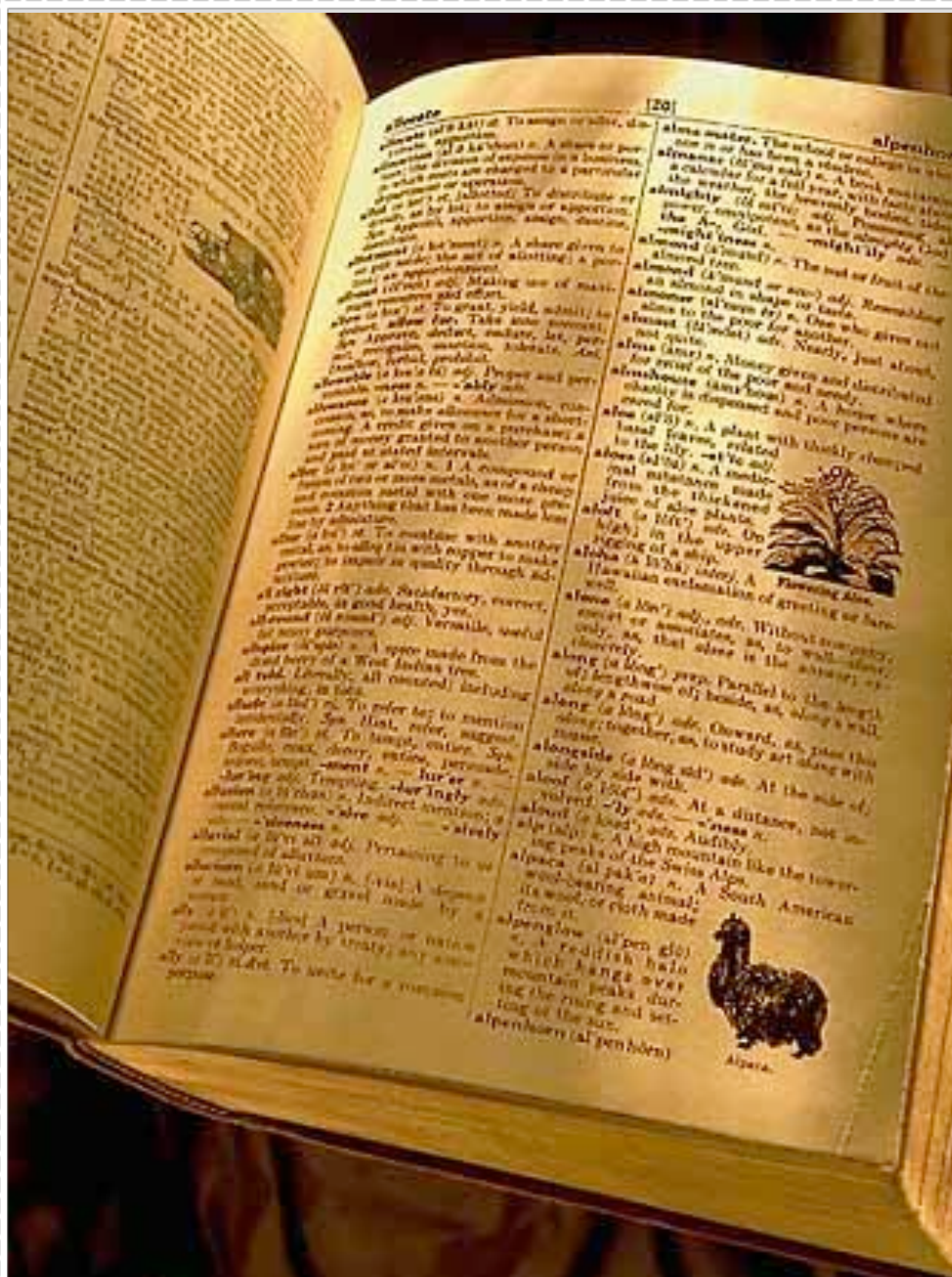
a b c d e f



apple

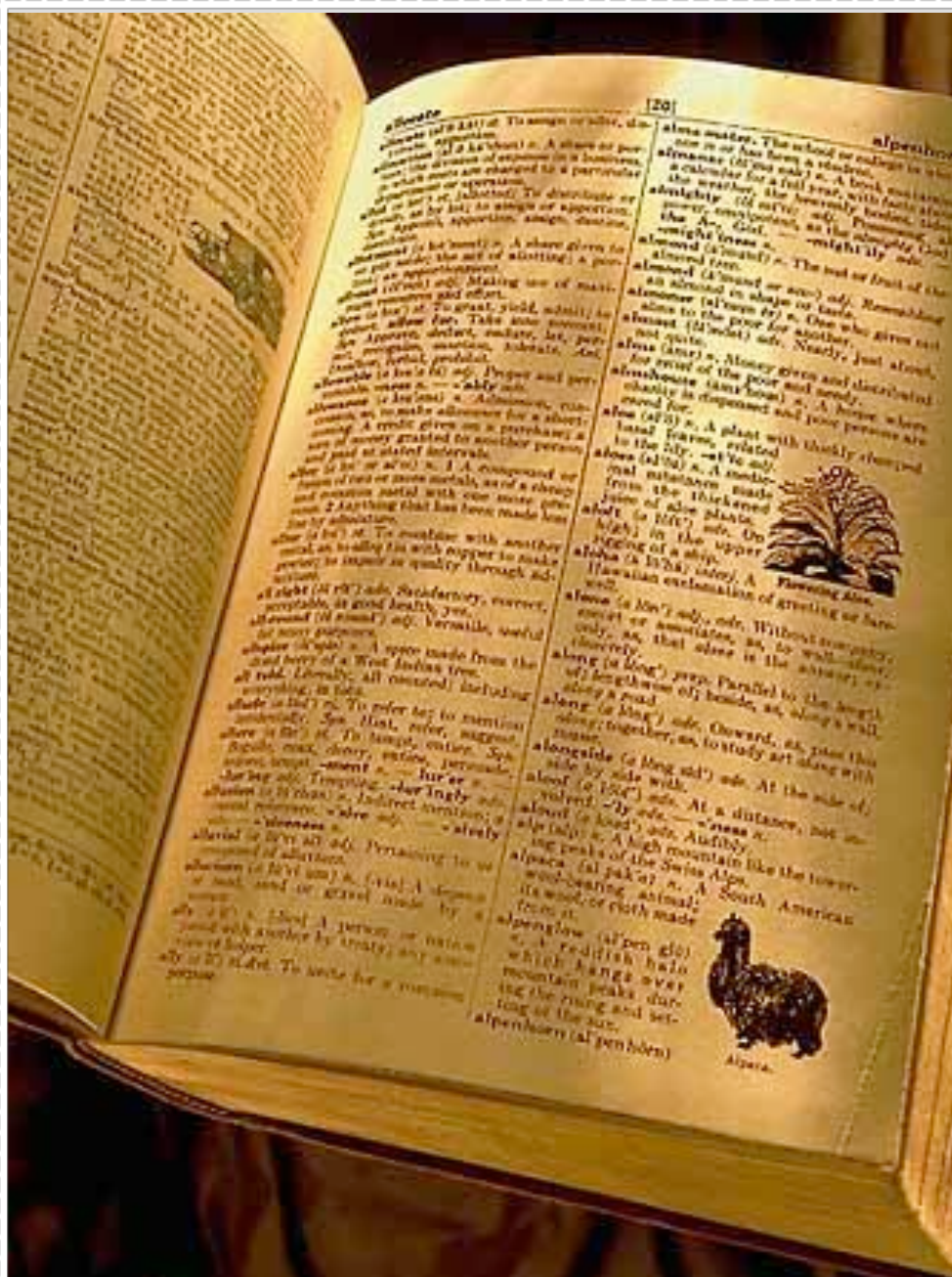
Subject String

text that the regex is applied to





terminology



apple

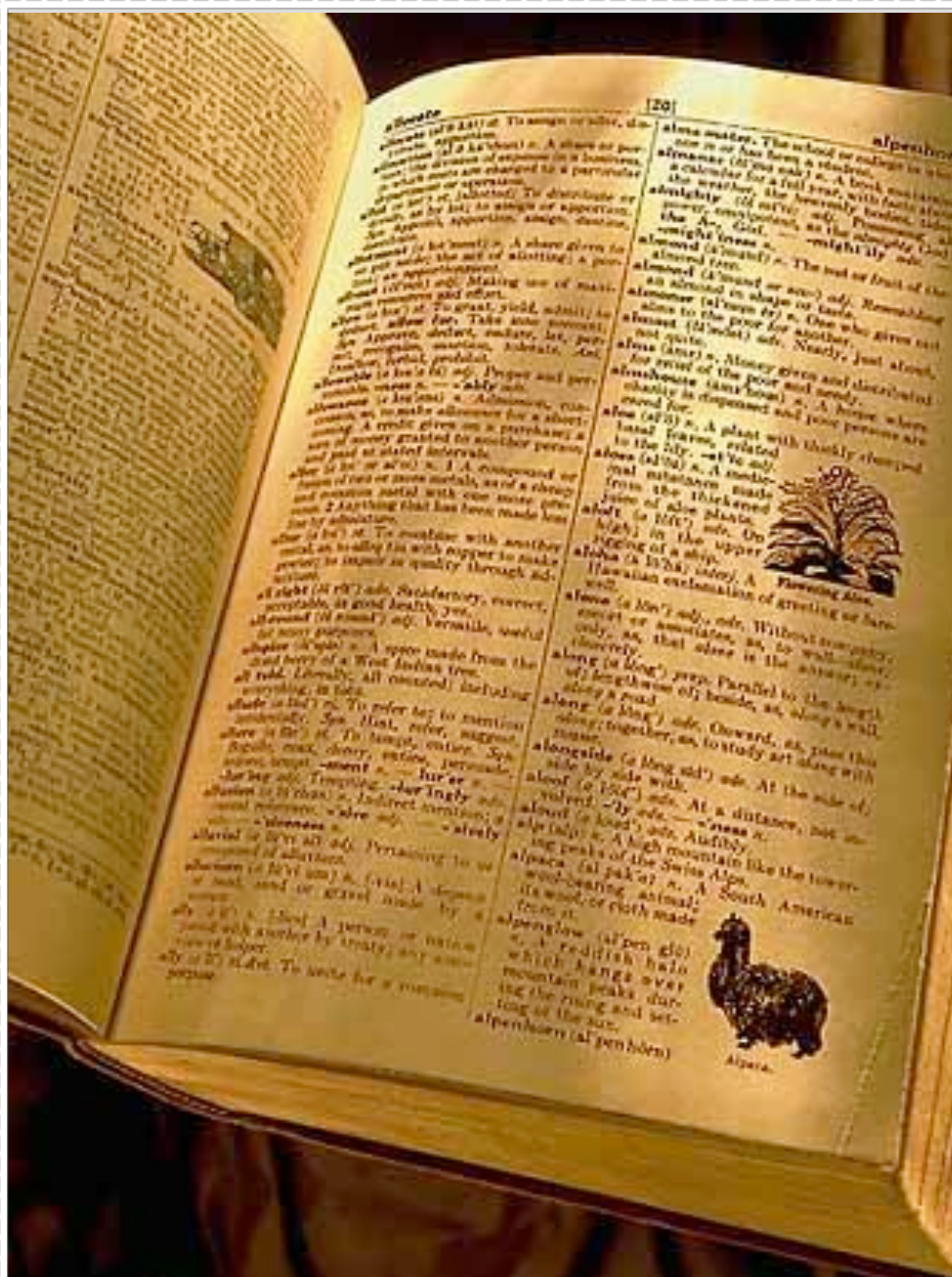
Match

a portion of the string that is successfully
described by the regex





terminology



Engine

A program or a library that obtains matches given a regex and a string

PCRE



how an NFA engine works

- ❖ The engine bumps along the string trying to match the regex
- ❖ Sometimes it goes back and tries again



how an NFA engine works

- ❖ Two basic things to understand about the engine

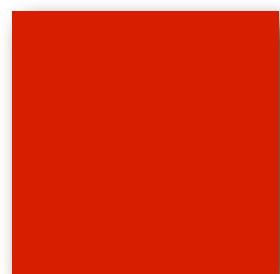
- ❖ It will always return the earliest (leftmost) match it finds

The topic of the day is isotopes.

- ❖ Given a choice it always favors match over a non-match



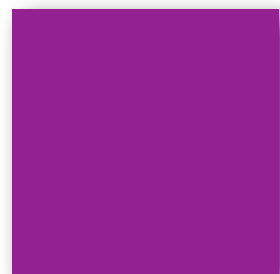
color legend



regular expression



subject string



match







characters

- ❖ Special set is a well-defined subset of ASCII
- ❖ Ordinary set consist of all characters not designated special
- ❖ Special characters are also called metacharacters

a 0 4
x K

^ * . !
?



matching literals

123

- ❖ The most basic regex consists of a single ordinary character
- ❖ It matches the first occurrence of that character in the string
- ❖ Characters can be added together to form longer regexes



extended characters

- ❖ To match an extended character, use `\xhh` notation where `hh` are hexadecimal digits
- ❖ To match Unicode characters (in UTF-8 mode) mode use `\x{hhh..}` notation





metacharacters

To use one of these literally, escape it, that is prepend it with a backslash

\ \$

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



metacharacters

To escape a sequence of characters, put them between **\Q** and **\E**

Price is **\Q\$12.36\E**

will match

Price is \$12.36

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



metacharacters

So will the backslashed version

Price is \\$12.36

will match

Price is \$12.36

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



character classes

[]

- ❖ Consist of a set of characters placed inside square brackets
- ❖ Matches one and only one of the characters specified inside the class



character classes

[]

⬢ matches an English vowel (lowercase)

[aeiou]

⬢ matches **surf** or **turf**

[st]urf



negated classes

[^]

- ❖ Placing a caret as the first character after the opening bracket negates the class
- ❖ Will match any character not in the class, *including newlines*
- ❖ `[^<>]` would match a character that is not left or right bracket

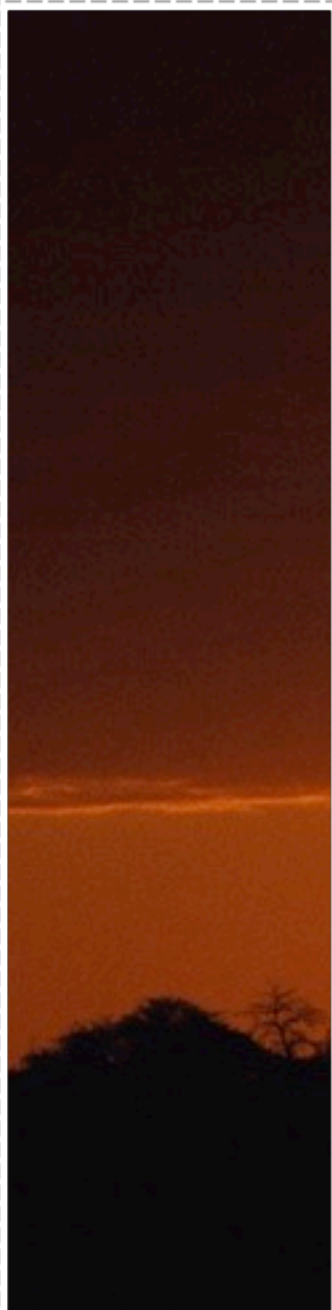


character ranges

[-]

- ❖ Placing a dash (-) between two characters creates a range from the first one to the second one
- ❖ Useful for abbreviating a list of characters

[a-z]

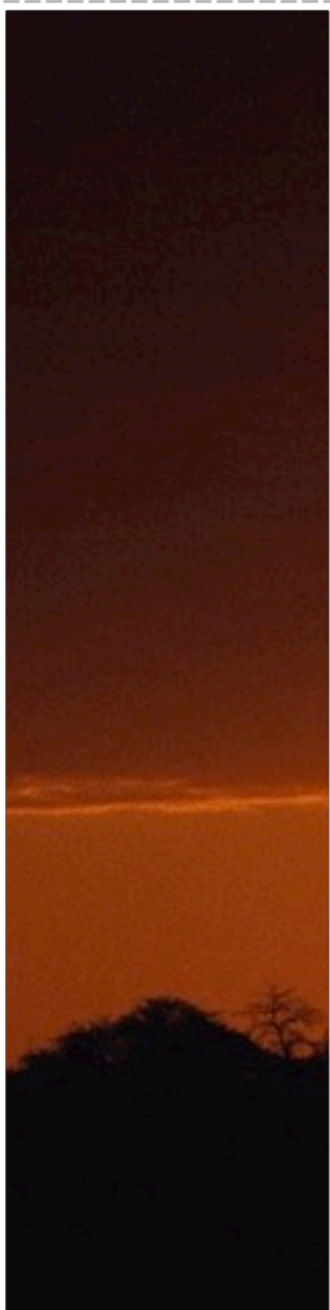


character ranges

[-]

❖ Ranges can be reversed

[z-a]

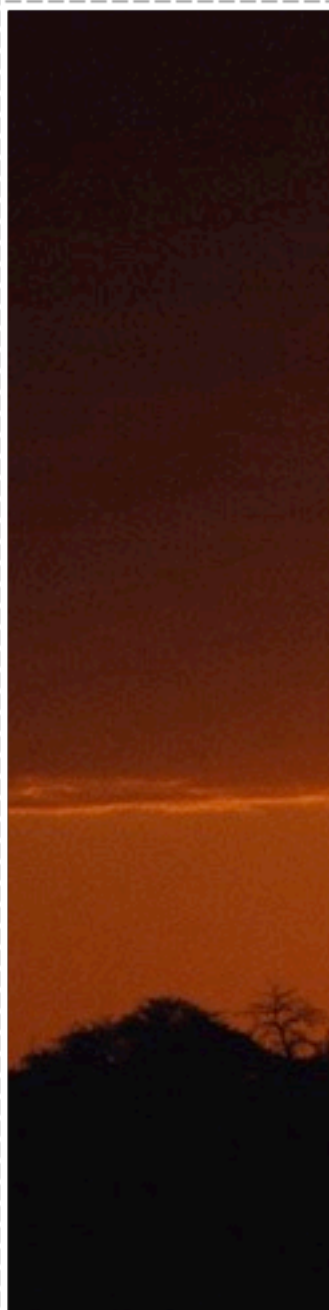


character ranges

[-]

- ❖ Ranges can be reversed
- ❖ A class can have more than one range and combine ranges with normal lists

[a-z0-9:]



\w word character [A-Za-z0-9_]

\d decimal digit [0-9]

\s whitespace [\n\r\t\f]

\W not a word character [^A-Za-z0-9_]

\D not a decimal digit [^0-9]

\S not whitespace [^ \n\r\t\f]

shortcuts for ranges

[-]



classes and metacharacters

]

\

^

_

❖ Inside a character class, most metacharacters lose their meaning

❖ Exceptions are:



classes and metacharacters

]

\

^

_

❖ Inside a character class, most metacharacters lose their meaning

❖ Exceptions are:

❖ closing bracket

classes and metacharacters

]

\

^

_

❖ Inside a character class, most metacharacters lose their meaning

❖ Exceptions are:

❖ closing bracket

❖ backslash



classes and metacharacters

]

\

^

_

❖ Inside a character class, most metacharacters lose their meaning

❖ Exceptions are:

❖ closing bracket

❖ backslash

❖ caret

classes and metacharacters

]

\

^

-

❖ Inside a character class, most metacharacters lose their meaning

❖ Exceptions are:

❖ closing bracket

❖ backslash

❖ caret

❖ dash

classes and metacharacters

`[ab\]`

`[ab^]`

`[a-z-]`

To use them literally, either escape them with a backslash or put them where they do not have special meaning





dot metacharacter



- ❖ By default matches any single character



dot metacharacter

❖ By default matches any single character

❖ Except a newline

~~\n~~



dot metacharacter

Is equivalent to

[^\n]



dot metacharacter

❖ Use dot carefully - it might match something you did not intend

❖ **12.45** will match literal **12.45**

❖ But it will also match these:

12345

12945

12a45

12-45

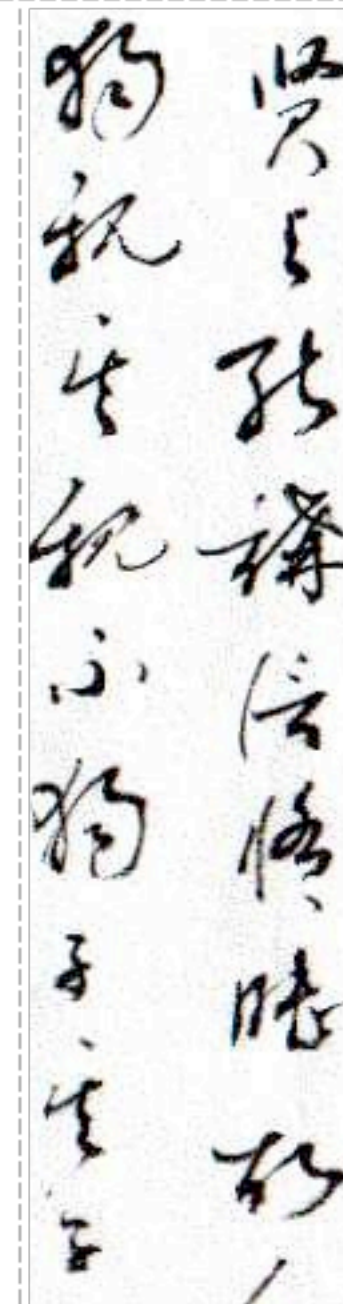
78812 45839



quantifiers



We are almost never sure about the contents of the text.



quantifiers



Quantifiers help us deal with this uncertainty

?

*

+

{ }



quantifiers



They specify how many times a regex component must repeat in order for the match to be successful

?

*

+

{ }



repeatable components

a

literal character

\w \d \s
\W \D \S

range shortcuts

■

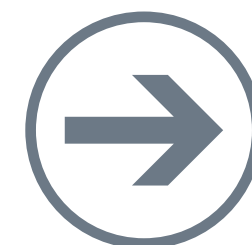
dot metacharacter

[]

character class

subpattern

backreference



zero-or-one



- ❖ Indicates that the preceding component is optional
- ❖ Regex **welcome!?** will match either **welcome** or **welcome!**
- ❖ Regex **super\s?strong** means that **super** and **strong** may have an optional whitespace character between them
- ❖ Regex **hello[!]??** Will match **hello**, **hello!**, or **hello?**



- ❖ Indicates that the preceding component has to appear once or more
- ❖ Regex **a+h** will match **ah**, **aah**, **aaah**, etc
- ❖ Regex **-\d+** will match negative integers, such as **-33**
- ❖ Regex **[^"]+** means to match a sequence (more than one) of characters until the next quote

one-or-more



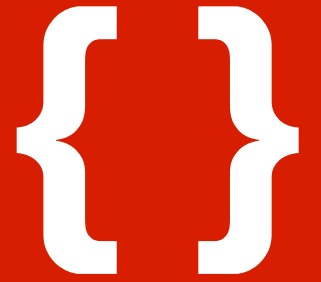


zero-or-more

- ❖ Indicates that the preceding component can match zero or more times
- ❖ Regex `\d+\.\d*` will match **2.**, **3.1**, **0.001**
- ❖ Regex `<[a-z][a-z0-9]*>` will match an opening HTML tag with no attributes, such as **** or **<h2>**, but not **<>** or **</i>**



general repetition



- ❖ Specifies the minimum and the maximum number of times a component has to match
- ❖ Regex `ha{1,3}` matches `ha`, `haa`, `haaa`
- ❖ Regex `\d{8}` matches exactly 8 digits
- ❖ If second number is omitted, no upper range is set
- ❖ Regex `go{2,}al` matches `goal`, `gooaal`, `gooooaal`, etc



general repetition

{ }

$\{0, 1\}$ = ?

$\{1, \}$ = +

$\{0, \}$ = *





greediness

 matching as much as possible, up to a limit



greediness

PHP 5?

PHP 5 is better than Perl 6

\d{2,4}

10/26/2004



greediness

- ❖ Quantifiers try to grab as much as possible by default
- ❖ Applying `<.+>` to `<i>greediness</i>` matches the whole string rather than just `<i>`



greediness

- ❖ If the entire match fails because they consumed too much, then they are forced to give up as much as needed to make the rest of regex succeed



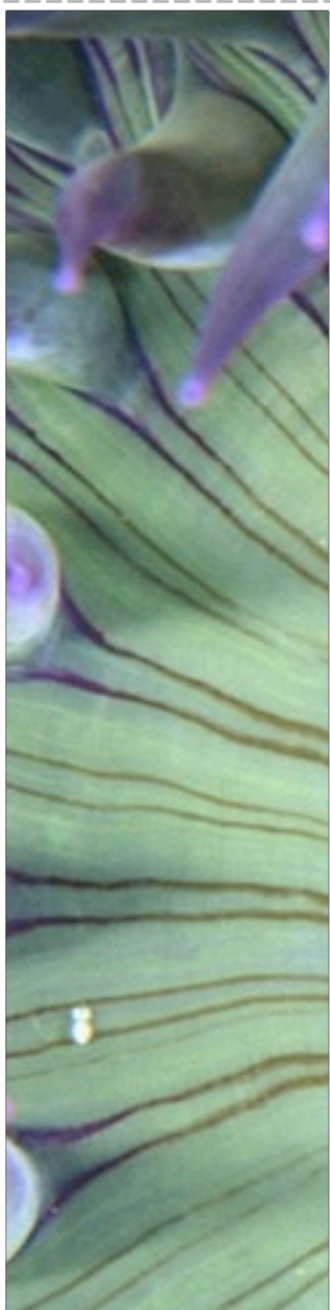
greediness

- ❖ To find words ending in **ness**, you will probably use **\w+ness**
- ❖ On the first run **\w+** takes the whole word
- ❖ But since **ness** still has to match, it gives up the last 4 characters and the match succeeds



overcoming greediness

- ❖ The simplest solution is to make the repetition operators non-greedy, or **lazy**
- ❖ Lazy quantifiers grab as little as possible
- ❖ If the overall match fails, they grab a little more and the match is tried again



overcoming greediness

*?

+?

{ , }?

??

- ❖ To make a greedy quantifier lazy, append ?
- ❖ Note that this use of the question mark is different from its use as a regular quantifier



overcoming greediness

*?

+?

{ , }?

??

Applying **<.+?>**

to **<i>greediness</i>**

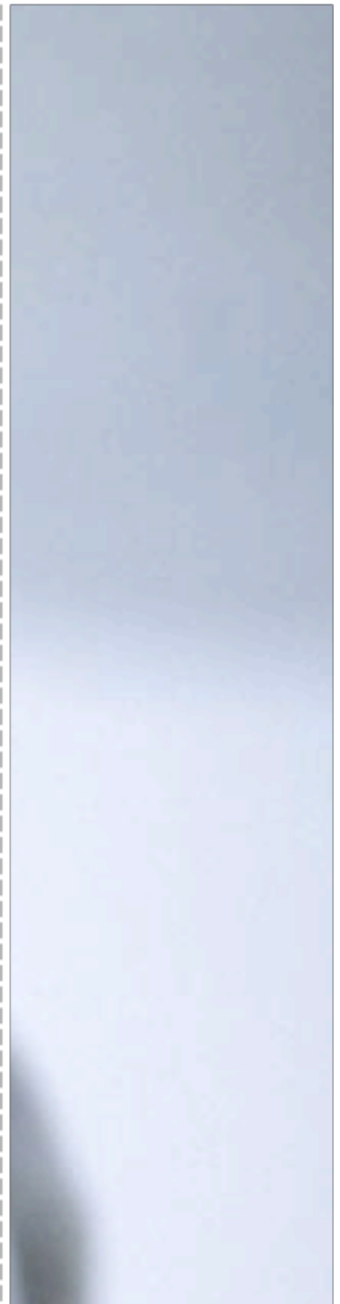
gets us **<i>**



overcoming greediness



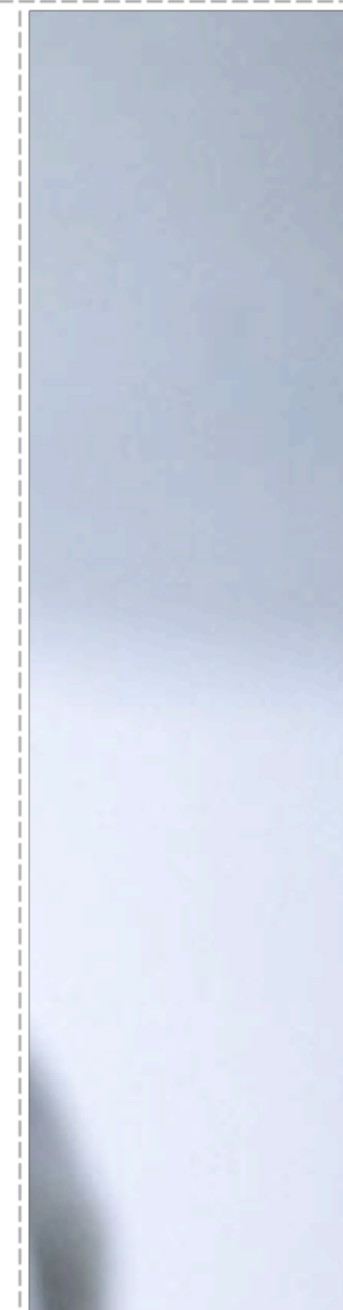
- ❖ Another option is to use negated character classes
- ❖ More efficient and clearer than lazy repetition



overcoming greediness



- ❖ `<.+?>` can be turned into `<[^>]+>`
- ❖ Note that the second version will match tags spanning multiple lines
- ❖ Single-line version:
`<[^>\r\n]+>`



assertions and anchors

- ❖ An assertion is a regex operator that
 - ❖ expresses a statement about the current matching point
 - ❖ consumes no characters



assertions and anchors

- ❖ The most common type of an assertion is an anchor
- ❖ Anchor matches a certain position in the subject string



caret



❖ C caret, or circumflex, is an anchor that matches at the beginning of the subject string

❖ **^F** basically means that the subject string has to start with an **F**

^F



Fandango



dollar sign



❖ Dollar sign is an anchor that matches at the end of the subject string or right before the string-ending newline

❖ `\d$` means that the subject string has to end with a digit

❖ The string may be `top 10` or `top 10\n`, but either one will match

`\d$`



`top 10`



multiline matching

- ❖ Often subject strings consist of multiple lines
- ❖ If the multiline option is set:
 - ❖ Caret (^) also matches immediately after any newlines
 - ❖ Dollar sign (\$) also matches immediately before any newlines

^t.+



one
two
three



absolute start/end

❖ Sometimes you really want to match the absolute start or end of the subject string when in the multiline mode

❖ These assertions are always valid:

❖ `\A` matches only at the very beginning

❖ `\Z` matches only at the very end

❖ `\Z` matches like `$` used in single-line mode

`\At. +`



three
tasty
truffles

word boundaries

\b \B

\bto\b



right **|to|** vote

- ❖ A word boundary is a position in the string with a word character (\w) on one side and a non-word character (or string boundary) on the other
- ❖ **\b** matches when the current position is a word boundary
- ❖ **\B** matches when the current position is not a word boundary



word boundaries

\b \B

\B2\b



doc2html

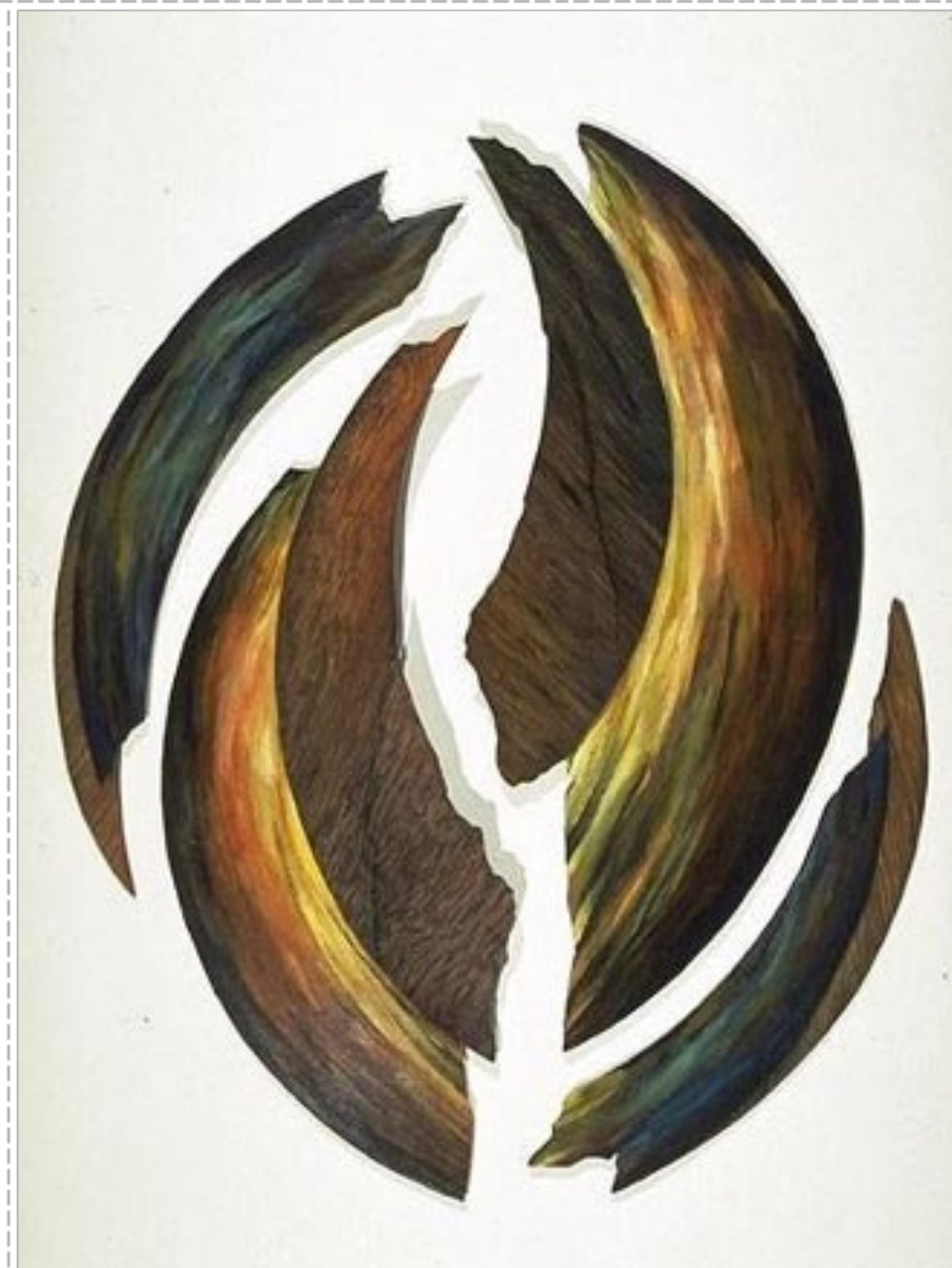
- ❖ A word boundary is a position in the string with a word character (\w) on one side and a non-word character (or string boundary) on the other
- ❖ **\b** matches when the current position is a word boundary
- ❖ **\B** matches when the current position is not a word boundary



subpatterns

()

- ❖ Parentheses can be used group a part of the regex together, creating a subpattern
- ❖ You can apply regex operators to a subpattern as a whole



grouping

()

- ❖ Regex `is(land)?` matches both `is` and `island`
- ❖ Regex `(\d\d,)*\d\d` will match a comma-separated list of double-digit numbers



capturing subpatterns

()

- ❖ All subpatterns by default are capturing
- ❖ A capturing subpattern stores the corresponding matched portion of the subject string in memory for later use



capturing subpatterns

()

❖ Subpatterns are numbered by counting their opening parentheses from left to right

❖ Regex `(\d\d-(\w+)-\d{4})` has two subpatterns

`(\d\d-(\w+)-\d{4})`



12-May-2004



capturing subpatterns

()

- ❖ Subpatterns are numbered by counting their opening parentheses from left to right
- ❖ Regex `(\d\d-(\w+)-\d{4})` has two subpatterns
- ❖ When run against **12-May-2004** the second subpattern will capture **May**

`(\d\d-(\w+)-\d{4})`



12-May-2004



non-capturing subpatterns

- ❖ The capturing aspect of subpatterns is not always necessary
- ❖ It requires more memory and more processing time



non-capturing subpatterns

- ❖ Using **?:** after the opening parenthesis makes a subpattern be a purely grouping one
- ❖ Regex **box(?:ers)?** will match **boxers** but will not capture anything
- ❖ The **(?:)** subpatterns are not included in the subpattern numbering



named subpatterns

- ❖ It can be hard to keep track of subpattern numbers in a complicated regex
- ❖ Using **?P<name>** after the opening parenthesis creates a *named* subpattern
- ❖ Named subpatterns are still assigned numbers
- ❖ Pattern **(?P<number>\d+)** will match and capture **99** into subpattern named **number** when run against **99 bottles**



- ❖ Alternation operator allows testing several sub-expressions at a given point
- ❖ The branches are tried in order, from left to right, until one succeeds
- ❖ Empty alternatives are permitted
- ❖ Regex **sailing|cruising** will match either **sailing** or **cruising**

alternation

|



- ❖ Since alternation has the lowest precedence, grouping is often necessary
- ❖ **sixth|seventh sense** will match the word **sixth** or the phrase **seventh sense**
- ❖ **(sixth|seventh) sense** will match **sixth sense** or **seventh sense**

alternation

|



- ❖ Remember that the regex engine is eager
- ❖ It will return a match as soon as it finds one
- ❖ **camel|came|camera** will only match **came** when run against **camera**
- ❖ Put more likely pattern as the first alternative

alternation

|



- ❖ Applying **?** to assertions is not permitted but..
- ❖ The branches may contain assertions, such as anchors for example
- ❖ **(^|my|your) friend** will match **friend** at the beginning of the string and after **my** or **your**

alternation

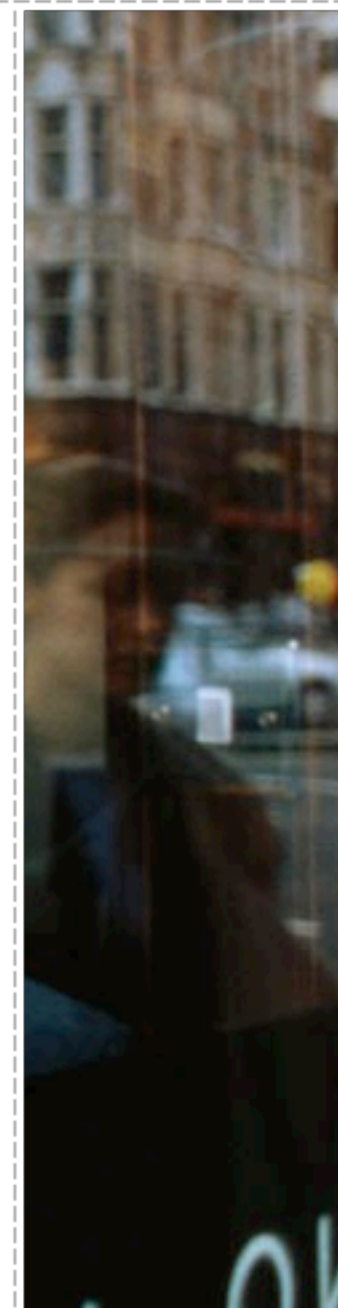
|



backtracking



- ❖ Also known as “if at first you don’t succeed, try, try again”
- ❖ When faced with several options it could try to achieve a match, the engine picks one and remembers the others



backtracking



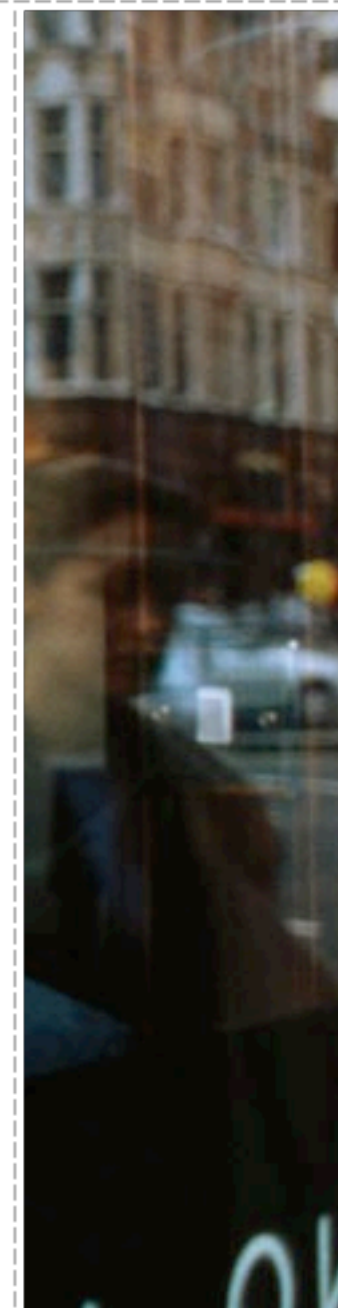
❖ If the picked option does not lead to an overall successful match, the engine backtracks to the decision point and tries another option



backtracking



- ❖ This continues until an overall match succeeds or all the options are exhausted
- ❖ The decision points include quantifiers and alternation

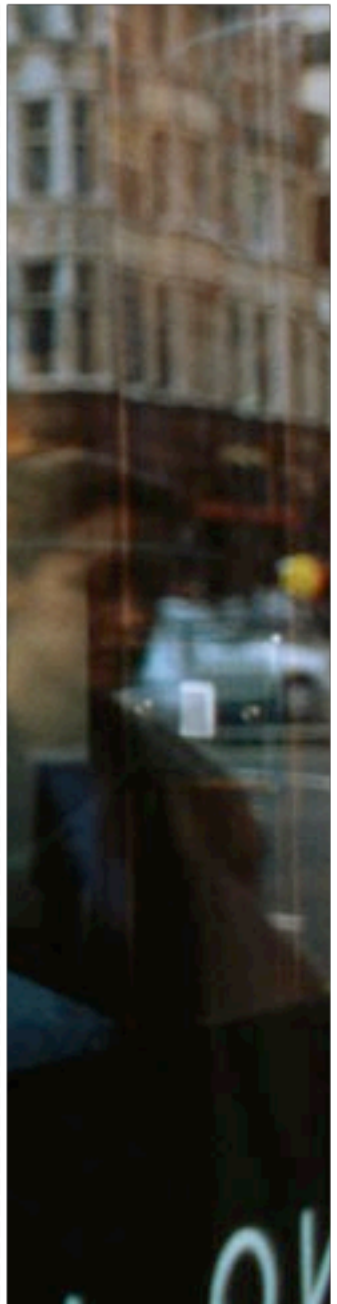


backtracking



Two important rules to remember

- ❖ With greedy quantifiers the engine always attempts the match, and with lazy ones it delays the match
- ❖ If there were several decision points, the engine always goes back to the most recent one





backtracking example

\d+00

12300

start





backtracking example

\d+00

12300

add 1





backtracking example

\d+00

12300

add 2





backtracking example

\d+00

12300

add 3





backtracking example

\d+00

12300

add 0





backtracking example

\d+00

12300

add 0





backtracking example

`\d+00`

12300

string exhausted
still need to match **00**





backtracking example

\d+00

12300

give up 0





backtracking example

\d+00

12300

give up 0





backtracking example

\d+00

12300

add 00





backtracking example

\d+00

12300

success





backtracking example

\d+ff

123dd

start





backtracking example

\d+ff

123dd

add 1





backtracking example

\d+ff

123dd

add 2





backtracking example

\d+ff

123dd

add 3





backtracking example

\d+_f

123dd

cannot match **f** here





backtracking example

\d+_f

123dd

give up **3**
still cannot match **f**



backtracking example

\d+_f

123dd

give up **2**
still cannot match **f**



backtracking example

\d+_ff

123dd

cannot give up more
because of +





backtracking example

\d+ff

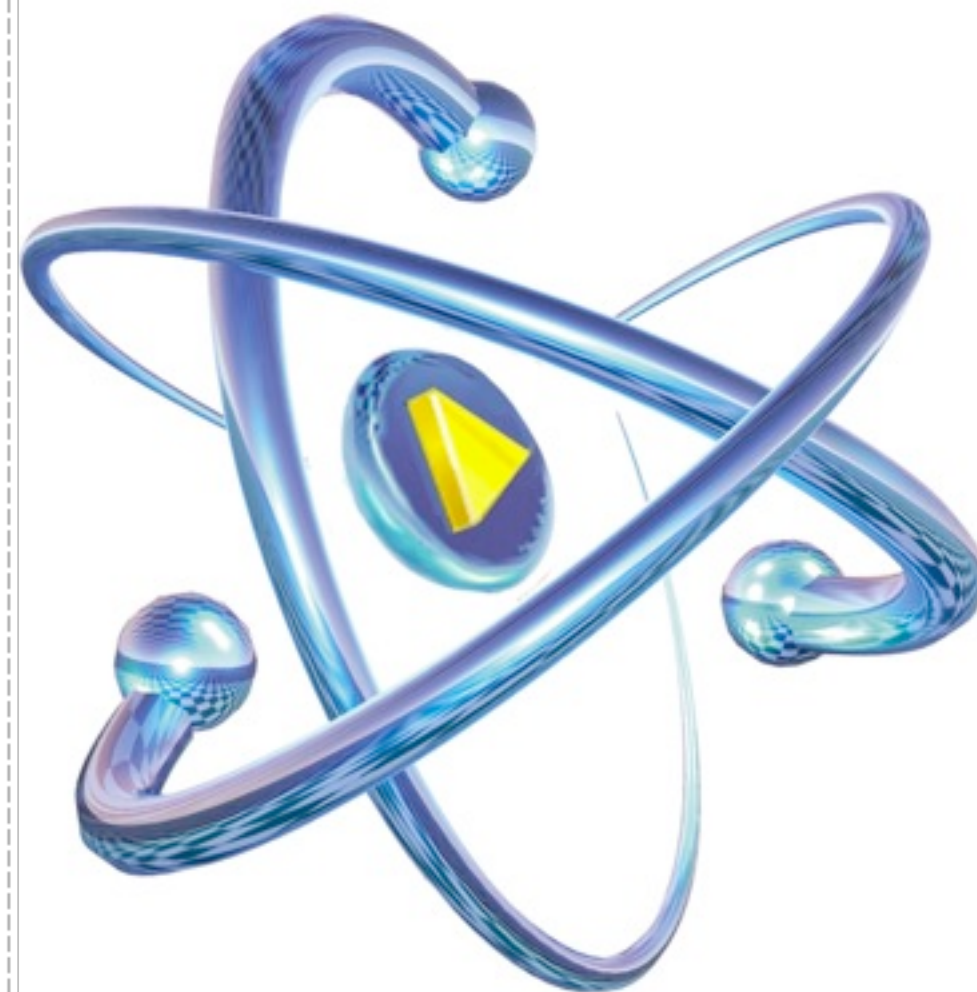
123dd

failure



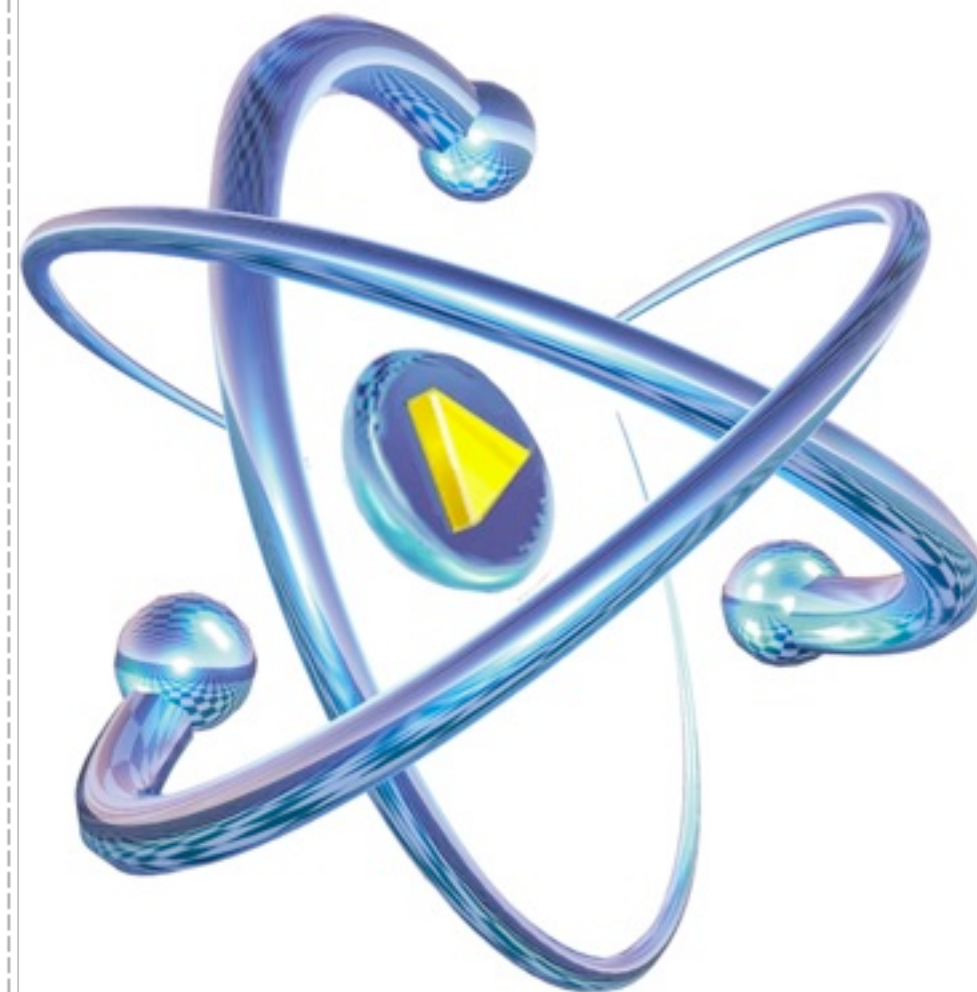
atomic grouping

- ❖ Disabling backtracking can be useful
- ❖ The main goal is to speed up failed matches, especially with nested quantifiers



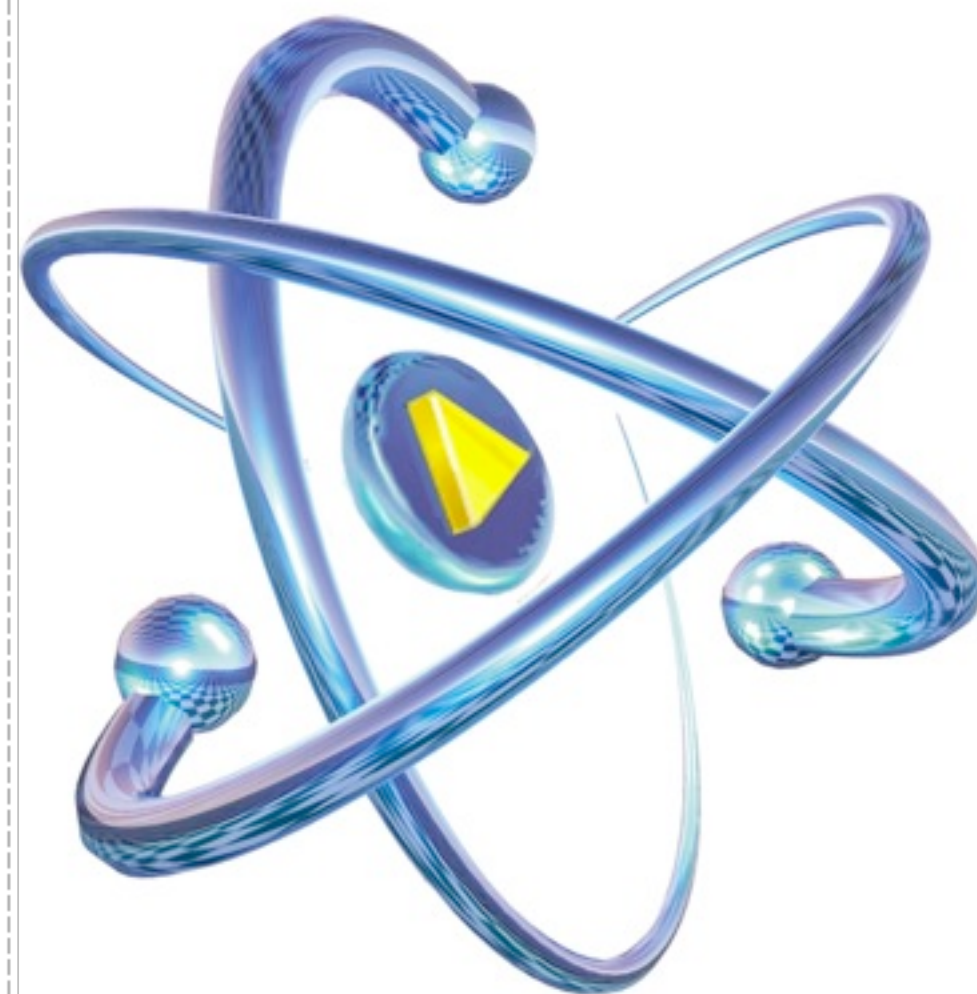
atomic grouping

- ❖ **(?>regex)** will treat **regex** as a single atomic token, no backtracking will occur inside it
- ❖ All the saved states are forgotten



atomic grouping

- ❖ `(?>\d+)ff` will lock up all available digits and fail right away if the next two characters are not **ff**
- ❖ Atomic groups are not capturing



possessive quantifiers



- ❖ Atomic groups can be arbitrarily complex and nested
- ❖ Possessive quantifiers are simpler and apply to a single repeated item



possessive quantifiers



- ❖ To make a quantifier possessive append a single **+**
- ❖ **\d++ff** is equivalent to **(?>\d+)ff**



possessive quantifiers



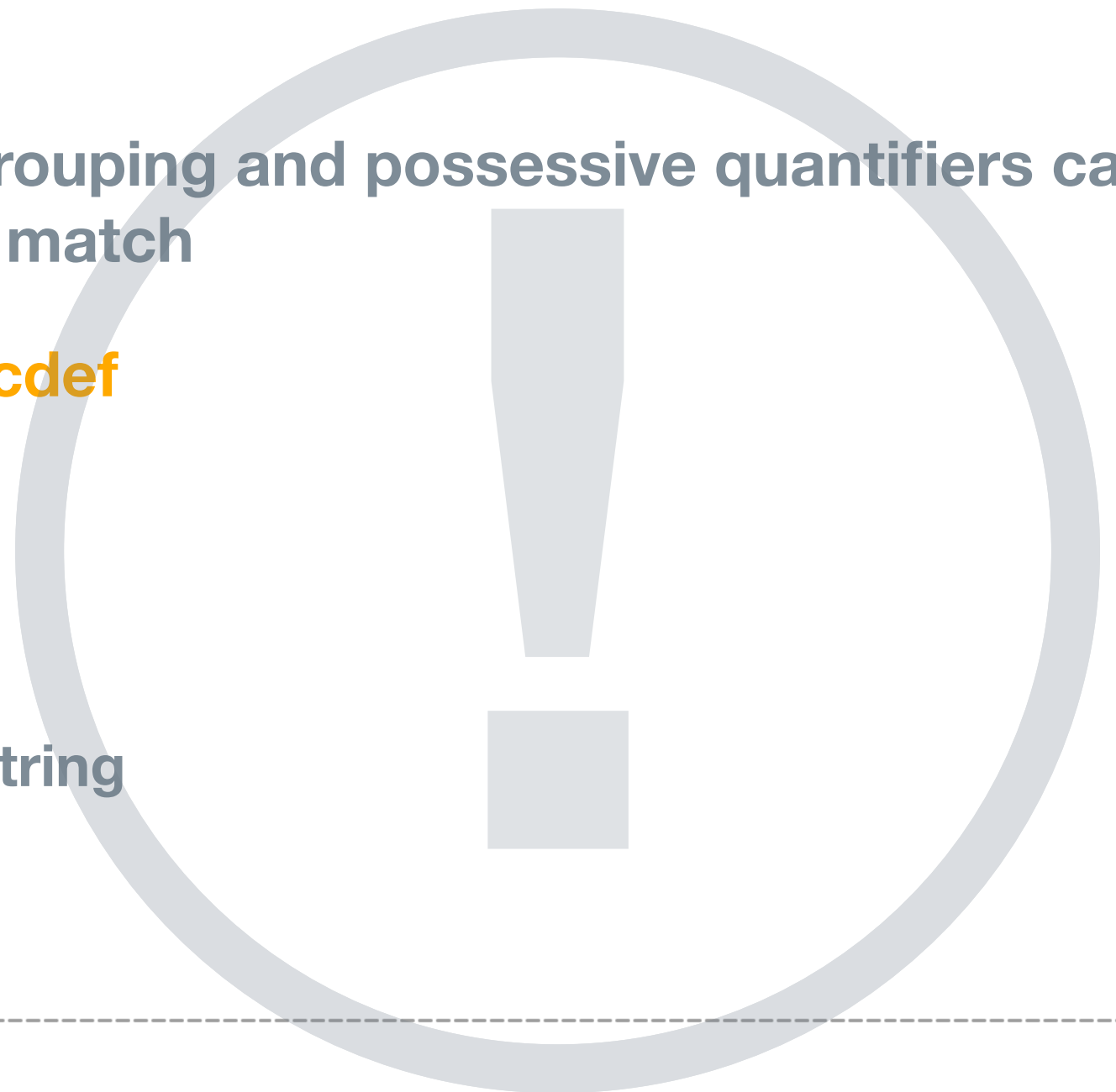
❖ Other ones are $*+$, $?+$, and $\{m,n\}+$

❖ Possessive quantifiers are always greedy



do not over-optimize

- ❖ Keep in mind that atomic grouping and possessive quantifiers can change the outcome of the match
- ❖ When run against string **abcdef**
 - ❖ **\w+d** will match **abcd**
 - ❖ **\w++d** will not match at all
 - ❖ **\w+** will match the whole string



backreferences

\n

- ❖ A backreference is an alias to a capturing subpattern
- ❖ It matches whatever the referent capturing subpattern has matched



backreferences

\n

- ❖ `(re|le)\w+\1` matches words that start with **re** or **le** and end with the same thing
- ❖ For example, retire and legible, but not revocable or lecture
- ❖ Reference to a named subpattern can be made with **(?P=name)**



lookaround

- ❖ Assertions that test whether the characters before or after the current point match the given regex
- ❖ Consume no characters
- ❖ Do not capture anything
- ❖ Includes lookahead and lookbehind



positive lookahead

(?=)



- ❖ Tests whether the characters after the current point match the given regex
- ❖ `(\w+)(?=:.*)` matches **surfing: a sport** but colon ends up in the second subpattern



negative lookahead

(?!)

- ❖ Tests whether the characters after the current point do not match the given regex
- ❖ **fish(?!ing)** matches **fish** not followed by ing
- ❖ Will match **fisherman** and **fished**



negative lookahead

(?!)

- ❖ Difficult to do with character classes
- ❖ `fish[^i][^n][^g]` might work but will consume more than needed and fail on subjects shorter than 7 letters
- ❖ Character classes are no help at all with something like `fish(?!hook|ing)`



positive lookbehind

(?<=)



- ❖ Tests whether the characters immediately preceding the current point match the given regex
- ❖ The regex must be of fixed size, but branches are allowed
- ❖ **(?<=foo)bar** matches **bar** only if preceded by **foo**, e.g. **my foobar**



negative lookbehind

(?<!)

- ❖ Tests whether the characters immediately preceding the current point do not match the given regex
- ❖ Once again, regex must be of fixed size
- ❖ **(?<!foo)bar** matches **bar** only if not preceded by **foo**, e.g. **in the bar** but not **my foobar**



conditionals

- ❖ Conditionals let you apply a regex selectively or to choose between two regexes depending on a previous match

`(?(condition)yes-regex)`

`(?(condition)yes-regex|no-regex)`

- ❖ There are 3 kinds of conditions

- ❖ Subpattern match

- ❖ Lookaround assertion

- ❖ Recursive call (not discussed here)



subpattern conditions **(?(n))**

- ❖ This condition is satisfied if the capturing subpattern number **n** has previously matched
- ❖ **(“)? \b\w+\b (?(1))”** matches words optionally enclosed by quotes
- ❖ There is a difference between **(“)?** and **(“?)** in this case: the second one will always capture



assertion conditions

- ❖ This type of condition relies on lookahead assertions to choose one path or the other

href=(? (?=[“”]) ([“”])\S+\1 | \S+)

- ❖ Matches **href=**, then

- ❖ If the next character is single or double quote match a sequence of non-whitespace inside the matching quotes

- ❖ Otherwise just match it without quotes



inline options

(?i)

The matching can be modified by options you put in the regular expression

enables case-insensitive mode

(?m)

enables multiline matching for **^** and **\$**

(?s)

makes dot metacharacter match newline also

(?x)

ignores literal whitespace

(?U)

makes quantifiers ungreedy (lazy) by default



inline options

(?i)

(?m)

(?s)

(?x)

(?U)

❖ Options can be combined and unset

(?im-sx)

❖ At top level, apply to the whole pattern

❖ Localized inside subpatterns

(a(?i)b)c





comments

?#

Here's a regex I wrote when working on Smarty templating engine

```
^\$\w+(?>(\[(\d+|\$\w+|\w+(\.\w+)?)\])|((\.|->)\$\w+))*(>|@?\w+(:(>"[^"\\]*(?:\\\.|"[^"]*"|'[^']*')*\'|[\^|+))*)*$
```



comments

?#

Let me blow that up for you

```
^\$\\w+(?>(\[ (\d+ | \$\\w+ | \\w+ (\\.\\w+)? ) \] ) |
( (\\.|->) \\$?\\w+ ) ) * (?>\\ | @?\\w+ ( : (?>" [ ^"\\\\\\\\ ] *
( ? : \\\\\. [ ^"\\\\\\\\ ] * ) * " | \\ ' [ ^\\'\\\\\\\\ ] *
( ? \\\\\. [ ^\\'\\\\\\\\ ] * ) * \\ ' | [ ^| ]+ ) ) * ) * $
```

Would you like some comments with that?



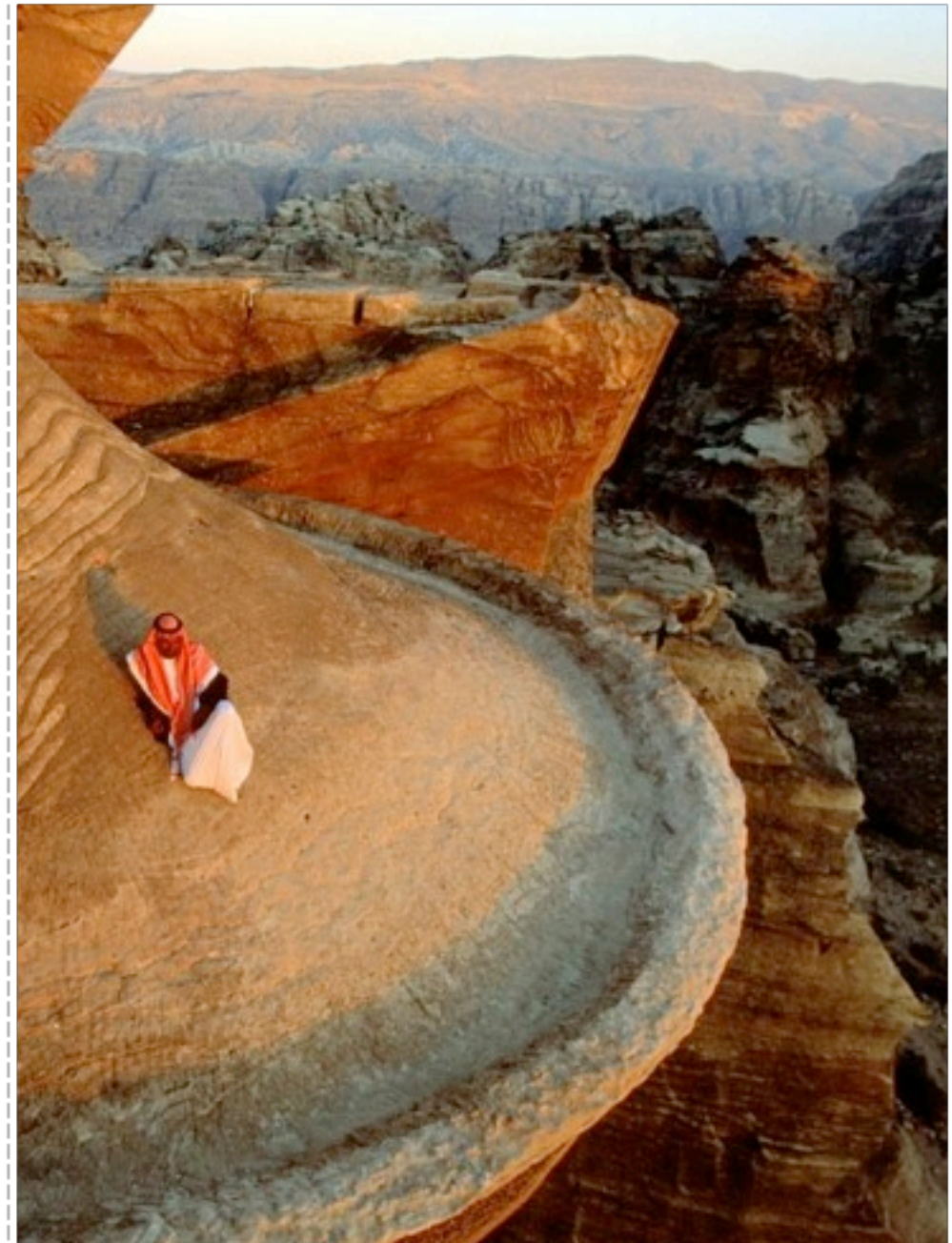
comments

?#

❖ Most regexes could definitely use some comments

❖ (?#...) specifies a comment

\d+(?# match some digits)

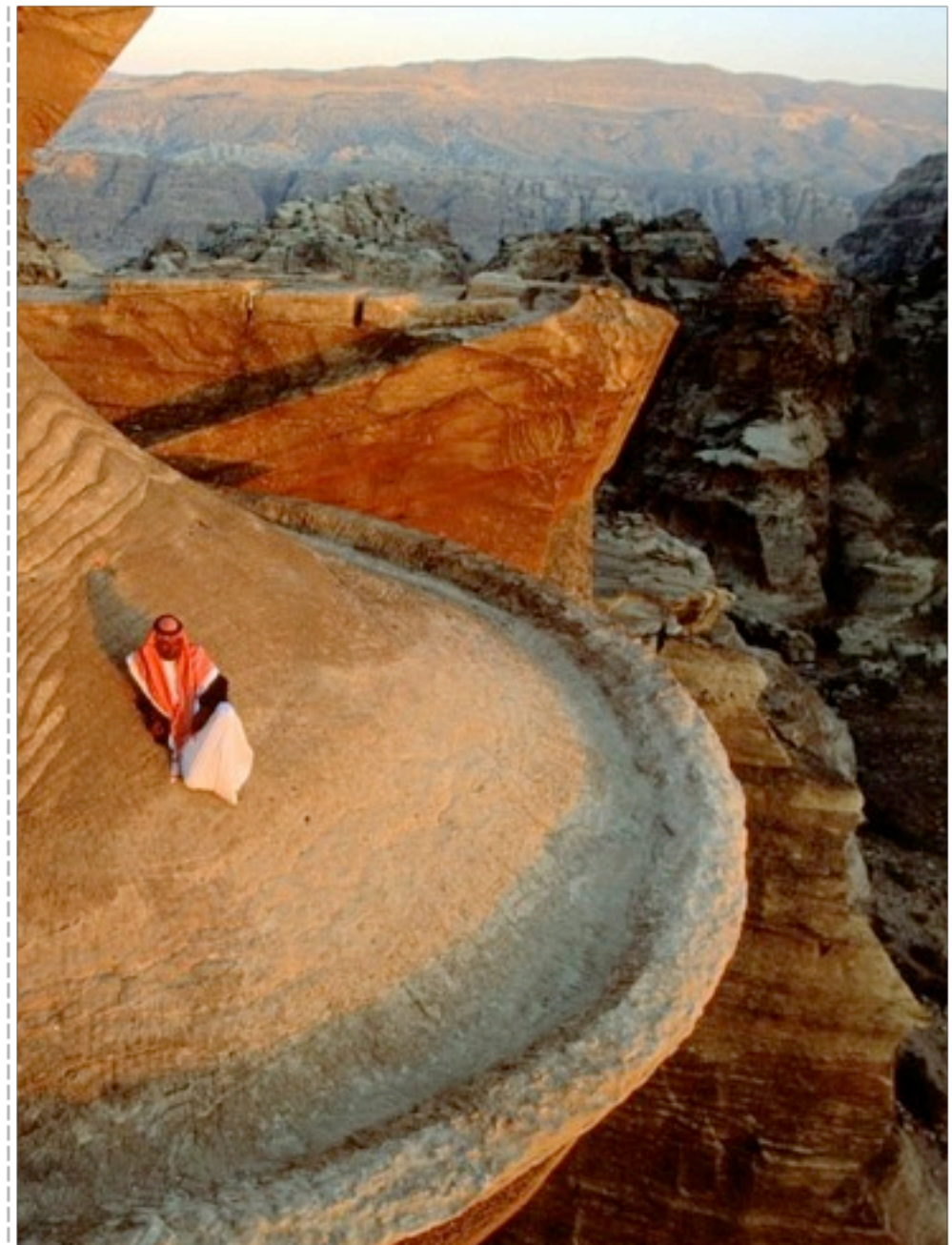


comments

?#

- ❖ If **(?x)** option is set, anything after **#** outside a character class and up to the next newline is considered a comment
- ❖ To match literal whitespace, escape it

(?x) \w+ # start with word characters
[?!] # and end with ? or !





Regex Toolkit



regex toolkit

- ✦ In your day-to-day development, you will frequently find yourself running into situations calling for regular expressions
- ✦ It is useful to have a toolkit from which you can quickly draw the solution
- ✦ It is also important to know how to avoid problems in the regexes themselves



matching vs. validation

- ❖ In matching (extraction) the regex must account for boundary conditions
- ❖ In validation your boundary conditions are known – the whole string



matching vs. validation

- ❖ Matching an English word starting with a capital letter

`\b[A-Z][a-zA-Z'-]*\b`

- ❖ Validating that a string fulfills the same condition

`^[A-Z][a-zA-Z'-]*$`

- ❖ Do not forget `^` and `$` anchors for validation!



using dot properly

- ❖ One of the most used operators
- ❖ One of the most misused
- ❖ Remember - dot is a shortcut for **[^\n]**
- ❖ May match more than you really want
- ❖ **<.>** will match **** but also **<!>**, **< >**, etc
- ❖ Be explicit about what you want
- ❖ **<[a-z]>** is better



using dot properly



- ❖ When dot is combined with quantifiers it becomes greedy
- ❖ `<.+>` will consume any characters between the first bracket in the line and the last one
- ❖ Including any other brackets!



using dot properly



❖ It's better to use negated character class instead

`<[^>]+>` if bracketed expression spans lines

`<[^>\r\n]+>` otherwise

❖ Lazy quantifiers can be used, but they are not as efficient, due to backtracking



optimizing unlimited repeats

- ❖ One of the most common problems is combining an inner repetition with an outer one
- ❖ If the initial match fails, the number of ways to split the string between the quantifiers grows exponentially
- ❖ The problem gets worse when the inner regex contains a dot, because it can match anything!

(regex1|regex2|..)*

(regex*)+

(regex+)*

(.*?bar)*



optimizing unlimited repeats

❖ PCRE has an optimization that helps in certain cases, and also has a hardcoded limit for the backtracking

❖ The best way to solve this is to prevent unnecessary backtracking in the first place via atomic grouping or possessive quantifiers

(regex1|regex2|..)*

(regex*)+

(regex+)*

(.*?bar)*



optimizing unlimited repeats

- ❖ Consider the expression that is supposed to match a sequence of words or spaces inside a quoted string

`[“”](\w+|\s{1,2})*[“”]`

- ❖ When applied to the string `“aaaaaaaaaaa”` (with final quote), it matches quickly
- ❖ When applied to the string `“aaaaaaaaaaa` (no final quote), it runs **35** times slower!



optimizing unlimited repeats

- ❖ We can prevent backtracking from going back to the matched portion by adding a possessive quantifier:

`[“”](\w+|\s{1,2})*[“”]`

- ❖ With nested unlimited repeats, you should lock up as much of the string as possible right away



extracting markup

- ❖ Possible to use `preg_match_all()` for grabbing marked up portions
- ❖ But for tokenizing approach, `preg_split()` is better

```
$s = 'a <b><I>test</I></b> of <br /> markup';
$tokens = preg_split(
    '!( < /? [a-zA-Z][a-zA-Z0-9]* [^/>]* /? > ) !x', $s, -1,
    PREG_SPLIT_NO_EMPTY | PREG_SPLIT_DELIM_CAPTURE);

result is array('a', '<b>', '<I>', 'test', '</I>',
                '</b>', 'of', '<br />', 'markup')
```



restricting markup

- ❖ Suppose you want to strip all markup except for some allowed subset. What are your possible approaches?
- ❖ Use `strip_tags()` - which has limited functionality
- ❖ Multiple invocations of `str_replace()` or `preg_replace()` to remove script blocks, etc
- ❖ Custom tokenizer and processor, or..



restricting markup

```
$s = preg_replace_callback(
    '! < (/?) ([a-zA-Z][a-zA-Z0-9]*) ([^/>]*) (/?) > !x',
    'my_strip', $s);

function my_strip($match) {
    static $allowed_tags = array('b', 'i', 'p', 'br', 'a');
    $tag = $match[2];
    $attrs = $match[3];
    if (!in_array($tag, $allowed_tags)) return '';
    if (!empty($match[1])) return "</$tag>";
    /* strip evil attributes here */
    if ($tag == 'a') { $attrs = ''; }
    /* any other kind of processing here */
    return "<$tag$attrs$match[4]>";
}
```


matching numbers

- ❖ Integers are easy: `\b\d+\b`
- ❖ Floating point numbers:
`integer.fractional`
`.fractional`
- ❖ Can be covered by `(\b\d+|\B)\.\d+\b`



matching numbers

- ❖ To match both integers and floating point numbers, either combine them with alternation or use:
`((\b\d+)?\.)?\b\d+\b`
- ❖ **`[+-]?`** can be prepended to any of these, if sign matching is needed
- ❖ **`\b`** can be substituted by more appropriate assertions based on the required delimiters



matching quoted strings

- ❖ A simple case is a string that does not contain escaped quotes inside it

- ❖ Matching a quoted string that spans lines:

`"[^"]*"`

- ❖ Matching a quoted string that does not span lines:

`"[^"]*\r\n*"`



matching quoted strings

“
(
[[^]"]+
|
(?^{<=}\\^{\\})"
)*+
“

Matching a string with escaped quotes inside

“([[^]"]+|(?^{<=}\\^{\\})")*+”

opening quote

a component that is

a segment without any quotes

or

a quote preceded by a backslash

component repeated zero or more times
without backtracking

closing quote



matching e-mail addresses

- ❖ Yeah, right
- ❖ The complete regex is about as long as a book page in 10-point type
- ❖ Buy a copy of Jeffrey Friedl's book and steal it from there



matching phone numbers

❖ Assuming we want to match US/Canada-style phone numbers

800-555-1212

1-800-555-1212

800.555.1212

1.800.555.1212

(800) 555-1212

1 (800) 555-1212

❖ How would we do it?



matching phone numbers

❖ The simplistic approach could be:

`(1[.-])? \(? \d{3} \)? [.-] \d{3} [.-] \d{4}`

❖ But this would result in a lot of false positives:

1. (800) - 555 1212 800) . 555 - 1212

1 - 800 555 - 1212 (800 555 - 1212



matching phone numbers

^(?:
 (?:(?
 \d{3}
 ((?
 \1 |
 [.-]))
 \d{3}
 \2
 \d{4}
 |
 1[]?\\(\\d{3}\\\\[]\\d{3}-\\d{4}
)\$

anchor to the start of the string

may have 1. or 1- (remember the separator)

three digits

if we had a separator

match the same (and remember), otherwise

match . or - as a separator (and remember)

another three digits

same separator as before

final four digits

or

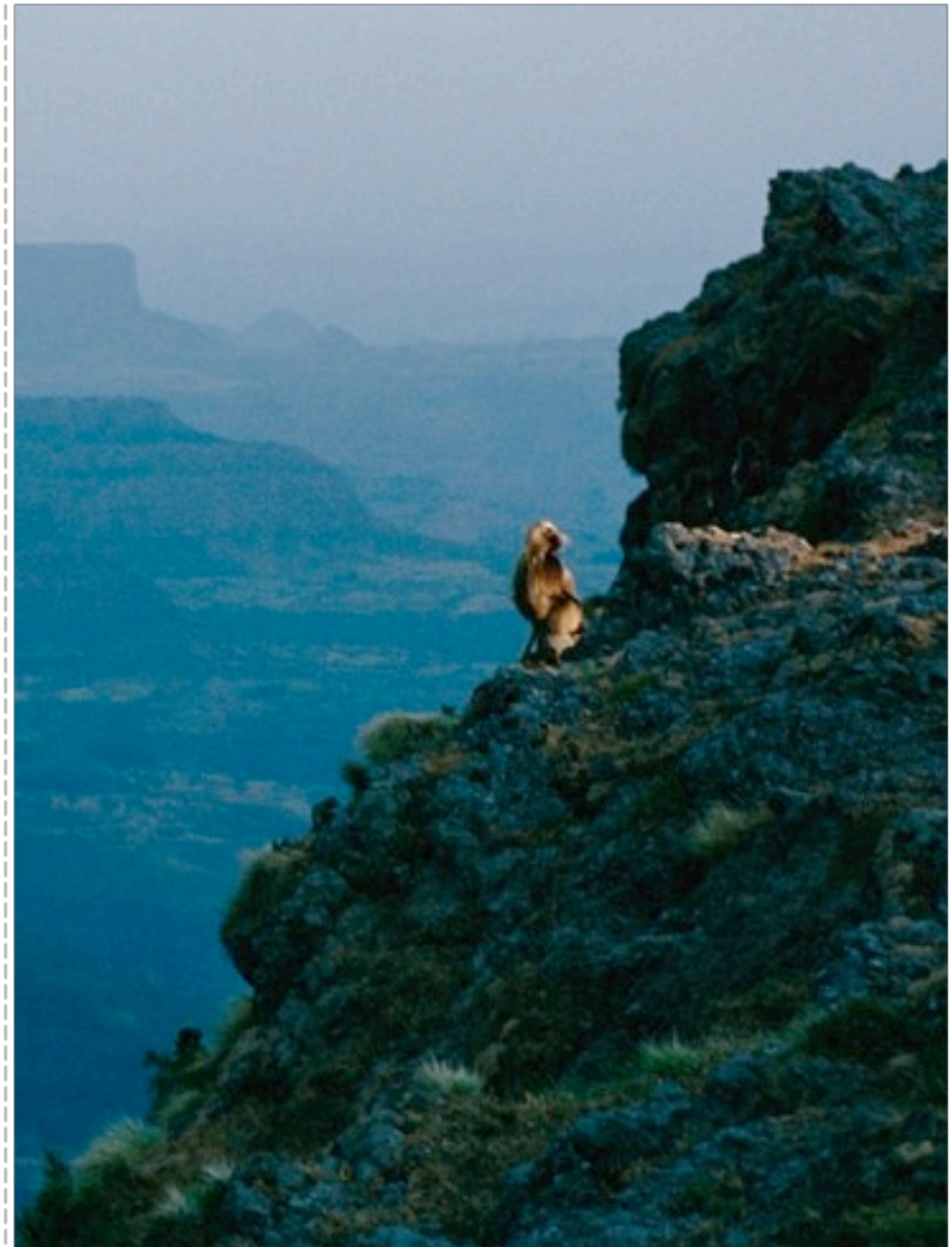
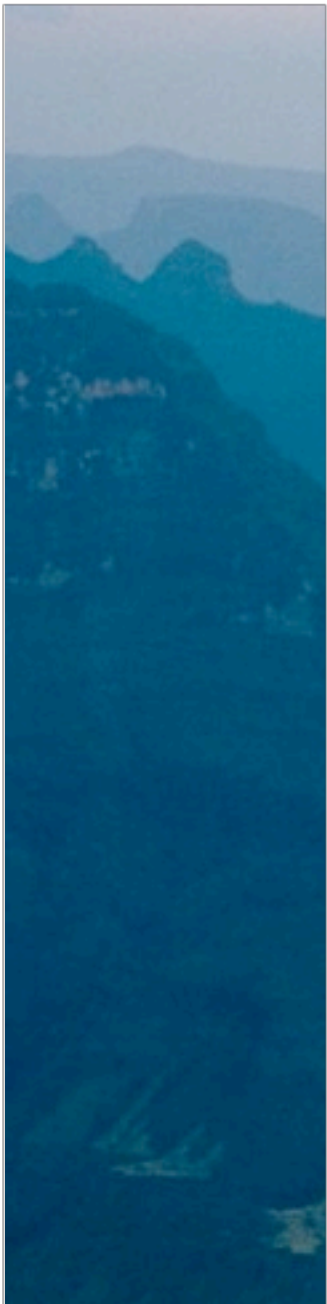
just match the third format

anchor to the end of the string



tips

- ❖ Don't do everything in regex – a lot of tasks are best left to PHP
- ❖ Use string functions for simple tasks
- ❖ Make sure you know how backtracking works



tips



- ❖ Be aware of the context
- ❖ Capture only what you intend to use
- ❖ Don't use single-character classes



tips

- ❖ Lazy vs. greedy, be specific
- ❖ Put most likely alternative first in the alternation list
- ❖ Think!



regex tools

- ❖ Rubular.com
- ❖ Regex buddy
- ❖ Komodo Regex tester (Rx toolkit)
- ❖ Reggy (reggyapp.com)
- ❖ RegExr (<http://www.gskinner.com/RegExr/>)
- ❖ <http://www.spaweditor.com/scripts/regex/index.php>
- ❖ <http://regex.larsolavtorvik.com/>





Thank You!

Questions?

<http://zmievski.org/talks>

