

Introduction

Sunday, January 1, 2017 11:52 PM

WHAT ARE REGULAR EXPRESSIONS?

- Regular Expression
 - Symbols representing a text pattern
- Regular Expressions
 - Formal language interpreted by a regular expression processor
- Used for matching, searching, and replacing text
- "Regex" for short (sometimes "Regexp")
- Usage examples
 - Test if a phone number has the correct number of digits
 - Test if an email address is in a valid format
 - Search a document for either "color" or "colour"
 - Replace all occurrences of "Bob", "Bobby", or "B." with "Robert"
- Regular expression to match a year
 - `/\d{4}/` matches 2005, but also 0000-9999
 - `/(19|20)\d\d/` matches 1900-2099
 - `/(19[5-9]\d|20[0-4]\d)/` matches 1950-2049

WHAT IS MEAN BY MATCHES WITH REGULAR EXPRESSION?

- "matches"
 - A regular expression matches text if it correctly describes the text
- AND**
- Text matches a regular expression if it is correctly described by the expression

(for history and regex engines see videos)

Eager and Greedy

Regular expression engines are eager.

Regular expression engines are greedy.

- Regexp are eager to give us result so they try to reach to end as soon as possible and give us result.
- Regexp are greedy and they try to match as much as they can .

Notations convention and modes for regex

Monday, January 2, 2017 12:43 AM

NOTATION CONVENTIONS

- In general for differentiating a regular expression from string we write it between two slashes (/), Which is same as in 'ed' text editor created for unix. Which use following notation. obviously for using in program we do not use these slashes.

`g/re/p`

- We write text string inside quotes(" ") so that we can say that it is just string, but do not use it while using in the programs.

- Regular expression
 - /abc/
 - As in: g/re/p
 - Use without forward slashes
- Text string
 - "abc"
 - Use without quotes

MODES

- Regex comes in different modes like standard, global etc.(we learn later)
- It acts as modifier and comes after slashes(in notation, for just showing)
- In **regexpal** , we have checkboxes in top (for enabling these modes)
- Modes
 - Standard: /re/
 - Global: /re/g
 - Case-insensitive: /re/i
 - Multiline: /re/m
 - Dot-matches-all: /re/s

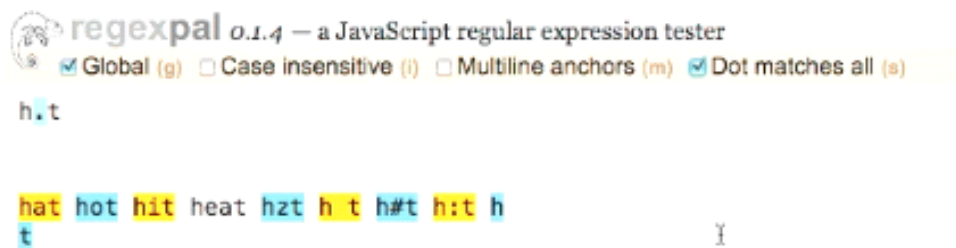
Standard vs Global(g) Matching

- In standard only first occurrence is matched for a text(full text).

- By default, standard matching is enabled in regex.
- Standard (non-global) matching
 - Earliest (leftmost) match is always preferred
 - /zz/ matches the first set of z's in "pizzazz"
- Global matching
 - All matches are found throughout the text
 - /zz/g matches the both sets of z's in "pizzazz"

Dot-matches-all(s)

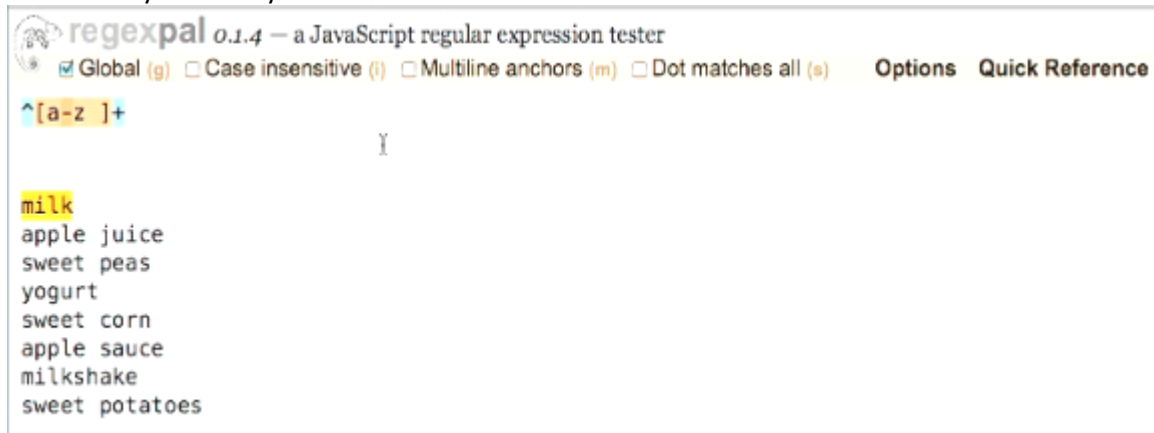
- Dot matches all option allows us to use wildcard character(dot) for matching new line also.



(h.t matches h\nt after checking 'Dot matches all')

Multiline-mode(m)

- As we can see in following example - full text is considered as 1 string (BY DEFAULT)
So here only first entry is matched.



- So if we wanted to apply regex to each and every line we can use **multiline mode(m)** ,
in regexpal it can be done by checking checkbox of 'Multiline anchors'

Precedence

Tuesday, January 31, 2017 10:41 PM

Precedence

Just as with mathematical formulas it's important to know the precedence of the regular expression elements. Precedence is as follows:



- **Elements:** like `a` are the basic building blocks of a regular expression.
- **Quantifiers:** like `+`, `*`, `?` and `{...}` bind tightly to the element on the left, for example `b+`.
- **Concatenation:** like `ab+c` binds after quantifiers.
- **Alternations:** like `|` binds as last.



For example, take the regular expression `ab+c|d`. This will match `abc`, `abbc`, `abbbc`, and so on and also `d`. Parentheses can be used to change these precedence rules. For example `ab+(c|d)` will match `abc`, `abbc`, `abbbc`, `...`, `abd`, `abbd`, `abbbd`, and so on. However, by using parentheses you also mark it as a sub-expression or capture group. It is possible to change the precedence rules without creating a new capture group by using `(?:...)`. For example `ab+(?:c|d)` matches the same as the preceding `ab+(c|d)` but does not create an additional capture group.

- MESSAGE 1 -same as regex in maths
 - 1-element
 - 2-closure(`*` and `+`)
 - 3.operator(concatination)
 - 4-or
- MESSAGE 2- Precedence will be used when paranthesis are not used properly.

Literal characters

Monday, January 2, 2017 12:58 AM

- Literal character means alphabet characters(a matches a , b matches b etc.)
- We can match literal characters(string) simple using same regular expression as the string itself.

- Examples-

`/car/` matches "car"

`/car/` matches the first three letters of "carnival"

- **Case-sensitive (by default)**

Note- this type of search is by default case-sensitive and it is advised to write all expression as case-sensitive and provide both type of character matching.

Ex- `/[a-zA-Z]/` this regular expression will match both types and we have written it in case-sensitive way.*

Metacharacters

Monday, January 2, 2017 1:15 AM

METACHARACTERS

- Characters with special meaning
 - Like mathematical operators
 - Transform literal characters into powerful expressions
- Only a few metacharacters to learn
 - \ . * + - { } [] ^ \$ | ? () : ! =
- Can have more than one meaning
 - Depends on how it is used in context
- Variation between regex engines

Learning Metacharacters

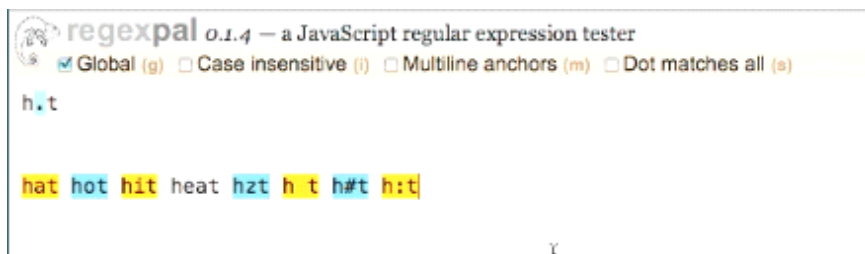
THE WILDCARD METACHARACTER

- Dot(.) acts as wildcard in regex, so it can be used to match anything except newline, It is because unix regex tools were line-based and they matches line by line.
(here we can use regex on multiple line)

Metacharacter	Meaning
.	Any character except newline

- Matches any one character except newline
 - Original Unix regex tools were line-based
 - /h.t/ matches "hat", "hot", and "hit", but not "heat"
- Broadest match possible
- Most common metacharacter
- Most common mistake
 - /9.00/ matches "9.00", "9500", and "9-00"

EXAMPLE-



(it didn't have matched newline, but we have been given a option, 'Dot matches all' through which we can match that too.)

Problems with Metacharacters

- Metacharacters are great but there is a problem , what if we wanted to matchdot(.) it self(wanted its literal meaning) , but then regex will think it is wildcard.
- For solving that problem we need to tell regex that we want its literal meaning, which can be done by escaping metacharacter.

ESCAPING METACHARACTERS

- Backslash(\) can be used for this purpose(same as c/c++, java)
- It can escape and metacharacter(so for \ we use \\)

Metacharacter	Meaning
\	Escape the next character

- Allows use of metacharacters as literal characters
 - Match a period with \.
 - /9\.00/ matches "9.00", but not "9500" or "9-00"
 - Match a backslash by escaping a backslash (\\)
- Only for metacharacters
 - Literal characters should never be escaped, gives them meaning
- Quotation marks are not metacharacters, do not need to be escaped

NOTE -1 only meta-characters should be escaped as literal character may have some special meaning with backward-slash(\).

Example - if we escape 't' it become '\t' which have special meaning to regex and used to represent 'TAB'

NOTE -2 while writing code in programming language we may need to escape Quotation marks(" ") because we write regular expression Quotation marks(so language will not be able to find which is starting and which is ending)

Ex- Java

```
String reg="hello\"world\"";
```


Defining Character set

Monday, January 2, 2017 4:49 PM

- Character set matches only a set of characters.
- Wild card character (dot) is also a character set which matches all characters.
- We make character set for matching some specific pattern.
example- `/[aeiou]/` matches any vowel.
- Character set is defined using meta-characters square brackets.

Metacharacter	Meaning
[Begin a character set
]	End a character set

- There is by default or between characters inside a regex.

String inside square brackets matches

- Any one of several characters
 - But **only** one character
 - Order of characters does not matter
- Examples
 - `/[aeiou]/` matches any one vowel
 - `/gr[ea]y/` matches "grey" and "gray"
 - `/gr[ea]t/` does not match "great"

Character Ranges

Monday, January 2, 2017 5:13 PM

CHARACTER RANGES

- Sometimes it becomes very tedious job to type a range fully.
- For example - if we wanted to have first letter of a word as upper case we have to type all characters in character set.

`[ABCDEFGHIJKLMNOPQRSTUVWXYZ]ello`

`Hello`

- We can use ranges to do the same work very easily like-

`[A-Z]ello`

`Hello`

- Dash (-) meta-character is used to define a range.

Metacharacter	Meaning
-	Range of characters

Syntax- `[start-end]`

- Range metacharacter
 - Represents all characters between two characters
 - Only a metacharacter inside a character set, a literal dash otherwise
- Examples
 - `[0-9]`
 - `[A-Za-z]`
 - `[a-ek-ou-y]`
- Caution
 - `[50-99]` is not all numbers from 50 to 99, it is the same as `[0-9]`
(here `[50-99]` represent `[5|0-9|9]` which becomes 0-9)

Negative character set

Monday, January 2, 2017 5:25 PM

NEGATIVE CHARACTER SETS

- Negative character set matches anything which do not match with what inside a character set.
- Carot(^) meta-character is used for this purpose.

Metacharacter	Meaning
<code>^</code>	Negate a character set

- This meaning is of carot(^) is only inside character set and as first character of character set.

- Examples

- `/[^aeiou]/` matches any one consonant (non-vowel)
- `/see[^mn]/` matches "seek" and "sees" but not "seem" or "seen"

- Caution

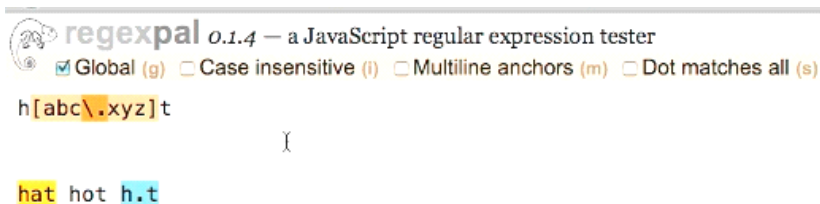
- `/see[^mn]/` matches "see " but not "see"

Meta-character inside character set

Monday, January 2, 2017 5:34 PM

METACHARACTERS INSIDE CHARACTER SETS

- Metacharacters inside character sets are already escaped
 - Do not need to escape them again
 - `/h[abc.xyz]t/` matches "hat" and "h.t", but not "hot"
- Exceptions
 - `] - ^ \`
- The exceptions are quite obvious and these are the meta-characters which have special meaning inside character sets (`\` is used for escaping), so we need to tell regex whether we want literal meaning of it or not.
Ex- we want to match `[or]`
then we can use `/[[\]]/`
We only need to escape ending square bracket as we have to tell that this is not ending of character set and we want literal meaning of that instead.
- If we escape that need not be escaped, there will be no problem as in following example we escaped dot(`.`) but need not to do so.



- **RULE OF THUMB -**
we may not need to escape exceptions also sometimes, example when we have `']'` just after `'['` regex know it can't be ending `']'`, so it takes it as escaped, but it is proper and un-ambiguous to escape all exception meta-characters always, so it remains more readable.

Examples

- `/var[([0-9][\]])/`
- `/2003[-/]10[-/]05/` may not require escaping
- `/file[0-_]1/` does require escaping

* In second example we want to match date 2003-10-05 or having `'/'`, here we do not need to escape dash(`-`) as it is not between two, so it can't specify range, so regex takes it as escaped, but it is good practice to escape, so that we can provide unambiguous regex and more readable.

Shorthand character sets

Monday, January 2, 2017 6:09 PM

SHORTHAND CHARACTER SETS

- Shorthand character sets provide more compact notations for some generally used character sets. e.g. `\d` for `[0-9]`

Shorthand	Meaning	Equivalent
<code>\d</code>	Digit	<code>[0-9]</code>
<code>\w</code>	Word character	<code>[a-zA-Z0-9_]</code>
<code>\s</code>	Whitespace	<code>[\t\r\n]</code>
<code>\D</code>	Not digit	<code>[^0-9]</code>
<code>\W</code>	Not word character	<code>[^a-zA-Z0-9_]</code>
<code>\S</code>	Not whitespace	<code>[^\t\r\n]</code>

- It is to be noted that in word we have everything which can be used to create variables in C, This is because regex comes from UNIX which also use underscore(`_`) to name variable or file.

- Examples

- `/\d\d\d\d/` matches "1984", but not "text"
- `/\w\w\w/` matches "ABC", "123", and "1_A"
- `/\w\s\w\w/` matches "I am", but not "Am I"
- `/[\w-]/` matches as word character or hyphen (useful)
- `/[\d\s]/` matches any digit or whitespace character
- `/[^d]/` is the same as `/\D/` and `/[^0-9]/`

- Caution

- `/[^d\s]/` is not the same as `[\D\S]`
- `/[^d\s]/` = NOT digit OR space character
- `/[\D\S]/` = EITHER NOT digit OR NOT space character

- here `[/^d\s]` means anything 'not(digit or space)' which becomes '(not digit) and (not space)'

`[/^d\s]`

⌘

123 456 789 abc

- While `[\D\S]` means '(not digit) or (not space)'

`[\D\S]`

123 456 789 abc

POSIX BRACKET EXPRESSIONS

- Posix standardization took place and this has given following bracket shorthands.

Class	Meaning	Equivalent
<code>[:alpha:]</code>	Alphabetic characters	A-Za-z
<code>[:digit:]</code>	Numeric characters	0-9
<code>[:alnum:]</code>	Alphanumeric characters	A-Za-z0-9
<code>[:lower:]</code>	Lowercase alphabetic characters	a-z
<code>[:upper:]</code>	Uppercase alphabetic characters	A-Z
<code>[:punct:]</code>	Punctuation characters	
<code>[:space:]</code>	Space characters	\s
<code>[:blank:]</code>	Blank characters (space, tab)	
<code>[:print:]</code>	Printable characters, spaces	
<code>[:graph:]</code>	Printable characters, no spaces	
<code>[:cntrl:]</code>	Control characters (non-printable)	
<code>[:xdigit:]</code>	Hexadecimal characters	A-Fa-f0-9

- Working is little bit different from simple shorthands, these can be used only inside a character class *they can't be used stand alone.*

- Use inside a character class, not standalone

- Correct: `[:alpha:]` or `^[[:alpha:]]`

- Incorrect: `[:alpha:]`

Because if used standalone regex engine, would not know whether it is a character class or bracket shorthand.

Repetition Meta-character

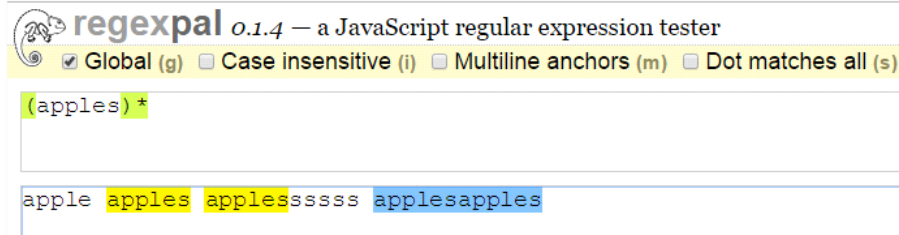
Wednesday, January 4, 2017 12:01 AM

REPETITION METACHARACTERS

- There are three meta-characters (*, + and ?) which can be used for repetition.

Metacharacter	Meaning
*	Preceding item zero or more times
+	Preceding item one or more times
?	Preceding item zero or one time

- Any operator will apply on only last characters so if we want to apply on bunch of characters use parenthesis.(we can do on character set)



Examples

- /apples*/ matches "apple", "apples", and "applesssssss"

(here it is to note that astrick(*) applies only on 's' , if we wants to repeat 'apples' then we have to use Paranthesis)

|

Examples

/apples+/ matches "apples" and "applesssssss", but not "apple"
/apples?/ matches "apple" and "apples", but not "applesssssss"
\d\d\d\d*/ matches numbers with three digits or more
\d\d\d\d+/ matches numbers with three digits or more
/colou?r/ matches "color" and "colour"

- Here one thing to note that in /\d\d\d\d*/ here '\d' is repeated which is digit class, so, last digit need not be repeated...
(as here we have '\d' we can be any digit after wards)
Eg. /\d\d\d\d*/ matches 1341 as well as 13412 ... (so 1 need not be repeated for match it could be any digit.)



Support

- * is supported in all regex engines
- + and ? are not supported in BREs (i.e., old Unix programs)
- We can't use it with 'grep' we can use either 'egrep' or 'grep -e' (extended regex)

Quantified repetition

Wednesday, January 4, 2017 3:19 PM

QUANTIFIED REPETITION

- Sometime we may wanted to some limited times of repetition like 3 or 5, but the repetition meta-characters only provide 0 or 1 or unlimited repetition. For this we use Quantifier repetition.
- **Syntax**
 - `{min,max}`
 - *min* and *max* are positive numbers
 - *min* must always be included, can be zero
 - *max* is optional
 - Three syntaxes
 - `\d{4,8}` matches numbers with four to eight digits
 - `\d{4}` matches numbers with exactly four digits (*min* is *max*)
 - `\d{4,}` matches numbers with four or more digits (*max* is infinite)

EXAMPLES -

- Examples
 - `\d{0,}` is the same as `\d*`
 - `\d{1,}` is the same as `\d+`
 - `/\d{3}-\d{3}-\d{4}/` matches most U.S. phone numbers
 - `/A{1,2} bonds/` matches "A bonds" and "AA bonds", not "AAA bonds"

Greedy Expressions

Wednesday, January 4, 2017 4:04 PM

GREEDY EXPRESSIONS

- Greedy Expression is a concept on which Regular Expressions works.
- According to this behavior of Regex,
"Regex matches as much as they can according to their syntax"
- Standard repetition quantifiers are greedy
- Expression tries to match the longest possible string

- Example 1

- 01_FY_07_report_99.xls
- /\d+\w+\d+/\

- Now, in this example - regex matches (colored part)

01_FY_07_report_99.xls

and not only,

- 01_FY_07_report_99.xls

- Example 2

- "Milton", "Waddams", "Initech, Inc."
- /\",.+\", \",.+\"/\

- In Example-2, Let we have names surrounded by double Quotes in a file and we are matching using above regex.
- Now the problem is this will match entire file as 1 and not each name.
(because " match starting and then we match each and every word [including " comma] with . And get following

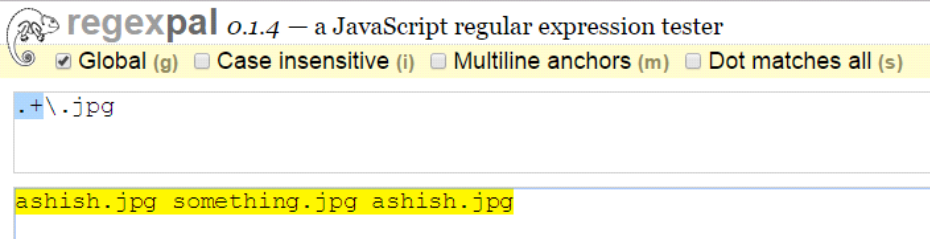
- "Milton", "Waddams", "Initech, Inc."
- /\",.+\", \",.+\"/\

- So problem is here that regex is greedy, and it tries to match as much as it can.

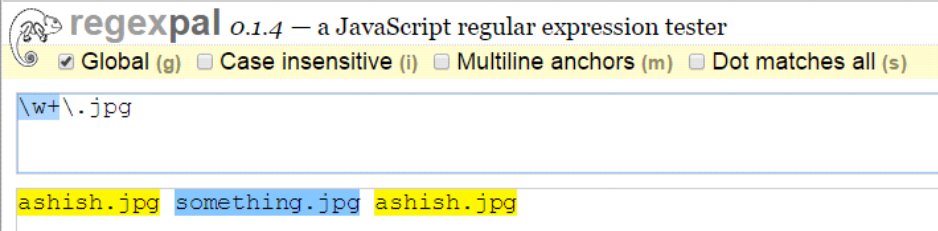
Rewinding or Backtracking-

- Although repetition quantifiers are greedy regex, they try to reduce their greedy ness by backtracking or rewinding, if they could not able to match by greedy.
 - Defers to achieving overall match
 - /\.+\.jpg/ matches "filename.jpg"
 - The + is greedy, but "gives back" the ".jpg" to make the match
 - Think of it as rewinding or backtracking
 - Gives back as little as possible
 - /\.*[0-9]+/ matches "Page 266"
 - /\.*/ matches "Page 26" while /[0-9]+/ matches "6"
- Example in following example(case 1) '.*' matches everything except last '.jpg'
 - because first it tries to match with full greedyness but then it could not as '.jpg' will not match if all go to '.*';
 - So it gives up 1 character and tries again and so on , and in similar way when he give 4 characters '.jpg' matched and we get a match.

Example - showing problem with Greedyness



- Clearly, as dot(.) is greedy it matched everything except last '.jpg'(backtracking) , so we have to prevent it to match space (which present in between of two names)



Lazy Expressions

Wednesday, January 4, 2017 8:45 PM

LAZY EXPRESSIONS

- By Default regex Quantifiers adopt greedy strategy, according to which
 - Greedy strategy
 - Match as much as possible before giving control to the next expression part
- But, we may tell Quantifiers to adopt lazy strategy, according to which
 - Lazy strategy
 - Match as little as possible before giving control to the next expression part
- Here we start by matching little as possible and then backtrack by increasing greedyness if not matched.

Making Lazy Expressions

- For making a quantifier lazy, we append '?' symbol after that which makes it to adopt lazy strategy.

Metacharacter	Meaning
?	Make preceding quantifier lazy

- Syntax
 - *?
 - +?
 - {*min*, *max*}?
 - ??
- Both are nearly same faster, but speed depend on the result...
- Examples
 - /\w*?\d{3}/
 - /[A-Za-z-]+?\./
 - /\. {4,8}? _ {4,8}/
 - /apples??/
- Support
 - Not supported in most Unix tools (BRE, ERE)
 - Supported in Perl compatible regex(in all languages)
(so not supported in Unix and Based OS)

LAZY EXPRESSION EXAMPLES -

- Example 1
 - 01_FY_07_report_99.xls
 - /\d+\w+\d+\/
- Example 2
 - "Milton", "Waddams" . "Initech, Inc."
 - /".+?", ".+?"/
- As we can see, the matched in different from what we are getting using Greedy expressions.

Caution-

- We might face a problem in Lazy expression, only when we have possible "epsilon" match or empty match, And we have all lazy repetitors. In this case all we match nothing and we match "**empty string**". here it is possible because no backtracking will happen as we have matched something.

/.*?[0-9]*?/

Page 266

Efficiency with repetition

Wednesday, January 4, 2017 9:44 PM

EFFICIENCY WHEN USING REPETITION

- Try to be as specific as possible.
 - Efficient matching + less backtracking = speedy results
 - Define the quantity of repeated expressions
 - `/.+/` is faster than `/.*/`
 - `/.{5}/` and `/.{3,7}/` are even faster
 - Narrow the scope of the repeated expression
 - `/.+/` can become `/[A-Za-z]+/`
 - Provide clearer starting and ending points
 - `/<.+>/` can become `/<[^>]+>/`
 - Use anchors and word boundaries
(anchors and boundaries tells the starting and ending point to word we have to match)
- Example
 - `/\w*s/` would be improved as `/\w+s/`
 - `/\w+s/` would be improved as `/[A-Za-z]+s/`
 - Perhaps as `/[a-z]+s/` or as `/[A-Z][a-z]+s/`

Grouping meta-characters

Wednesday, January 4, 2017 10:04 PM

GROUPING METACHARACTERS

- By default, operation apply only on the character after which operation is performed.

`/apples+/` matches "apples" and "applesssssss", but not "apple"

(in example above '+' applies on 's' character only after which operation is applied)

- We sometimes want to apply some operation(repetition metacharacters) on full word, we can group that using paranthesis.

Metacharacter	Meaning
(Start grouped expression
)	End grouped expression

USES -

- Group portions of the expression
 - Apply repetition operators to a group
 - Makes expressions easier to read
 - Captures group for use in matching and replacing

*capturing and backrefrence we will learn later(store the group to use later)

- some people also group single character to make it more readable.

`/run(s)?/` is the same as `/runs?/`

RESTRICTION -

- Cannot be used inside character set

- As paranthesis do not comes in exception meta-characters for character-set, they have their literal meaning inside [], so we can't use them inside character set.

- Examples

- `/(abc)+/` matches "abc" and "abcabcabc"
- `/(in)?dependent/` matches "independent" and "dependent"
- `/run(s)?/` is the same as `/runs?/`

Alterations

Wednesday, January 4, 2017 10:18 PM

ALTERNATION METACHARACTER

Metacharacter	Meaning
	Match previous or next expression

- | is an OR operator
 - Either match expression on the left or match expression on the right
 - Ordered, leftmost expression gets precedence
 - Multiple choices can be daisy-chained
 - Group alternation expressions to keep them distinct

- daisy chain means concatenated ORs , like

`/abc|def|ghi|jkl/` matches "abc", "def", "ghi", and "jkl"

- Working of OR(|) is same as C, so regex engine took the string's letter and try to match it with all options of regex OR and if it find it breaks.

- Group 'alternation expressions' as below

`/apple(juice|sauce)/` is not the same as `/applejuice|sauce/`

(we haven't used grouping then it will be second case)

Examples

- `/apple|orange/` matches "apple" and "orange"
- `/abc|def|ghi|jkl/` matches "abc", "def", "ghi", and "jkl"
- `/apple(juice|sauce)/` is not the same as `/applejuice|sauce/`
- `/w(ei|ie)rd/` matches "weird" and "wierd"

Example -1 using | character of OR

`apple|orange|`

apple orange appleorange apple|orange

Example -2 using literal meaning

`apple\|orange`

apple orange appleorange apple|orange

WRITING LOGICAL AND EFFICIENT ALTERNATIONS

- See video for more detail on ' the working of OR '

Repeating and nesting Alternation

Wednesday, January 4, 2017 10:42 PM

REPEATING AND NESTING ALTERNATIONS

- Repeating
 - First matched alternation does not effect the next matches
 - `/(AA|BB|CC){6}/` matches "AABBAACCAABB"
- Nesting
 - Check nesting carefully
 - `/(\d{2}([A-Z]{2}|- \d\w\d\w) | \d{4}(- \d{2}- [A-Z]{2,8} | _x[A-F]))/`
 - Trade-off between precision, readability, and efficiency
(nesting makes speed slow as now we have multiple loops matching)

Start and end anchors

Friday, January 13, 2017 7:14 PM

START AND END ANCHORS

Metacharacter	Meaning
^	Start of string/line
\$	End of string/line
\A	Start of string, never end of line
\Z	End of string, never end of line


- Reference a position, not an actual character
 - Zero-width
- Here we can see Carot(^) is used here as the start of a string/line.
- Examples
 - /^apple/ or /\Aapple/
 - finding apple at the beginning at the string.

```
^apple  
  
apple is the apple
```

Example-2

/apple\$/ or /apple\Z/

- Find apple at the end of the string.

 **regexpal** 0.1.4 — a JavaScript regular expression tester

☒ Global (g) ☐ Case insensitive (i) ☒ Multiline anchors (m) ☐ Dot match

```
apple$  
  
apple  
apple  
and we can do this using a n apple
```

Note - we have checked Multiline anchors here ,so different lines will not be considered as single string, similarly for next example.

Example-3

`/^apple$/ or /\Apple\Z/`

- It means we have defined everything from starting to end of string/line, so match only "apple".

```
^apple$
```

```
apple
```

```
apple
```

```
and we can do this using a n apple
```

- Support
 - ^ and \$ are supported in all regex engines
 - \A and \Z are supported in Java, .NET, Perl, PHP, Python, Ruby

Line Breaks and multiline mode

Friday, January 13, 2017 9:06 PM

- As we already know the following meta-characters

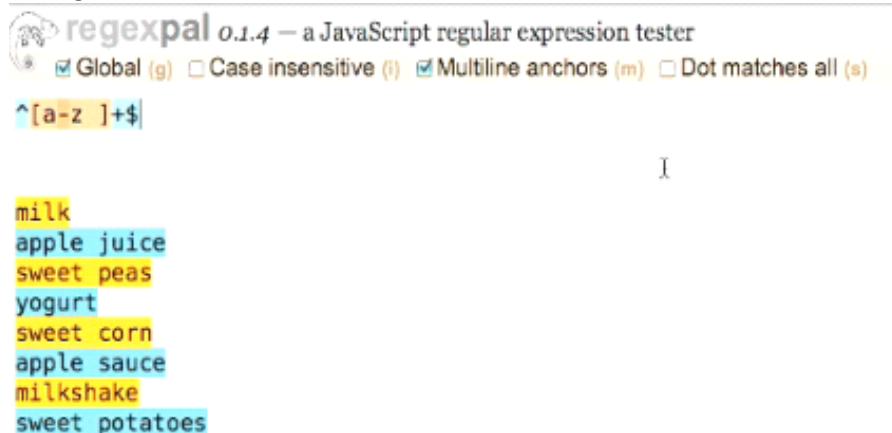
Metacharacter	Meaning
^	Start of string/line
\$	End of string/line
\A	Start of string, never end of line
\Z	End of string, never end of line

- ^ and \A and \$ and \Z handles newline differently.

- **Single-line mode**

- ^ and \$ do not match at line breaks
- \A and \Z do not match at line breaks
- Many Unix tools support only single line

- In single-line mode both works the same.



- **Multiline mode**

- ^ and \$ will match at the start and end of lines
- \A and \Z do not match at line breaks

- In multiline mode \A and \Z retain their functionality, but ^ and \$ start to match starting and end of each line differently.
- Will not work in regexpal as JS donot supports it.

ACTIVATING MULTILINE MODE IN DIFFERENT LANGUAGES

- Java: `Pattern.compile("^regex$", Pattern.MULTILINE)`
- JavaScript: `/^regex$/m`
- .NET: `Regex.Match("string", "^regex$", RegexOptions.Multiline)`
- Perl: `m/^regex$/m`
- PHP: `preg_match(/^regex$/m, "string")`
- Python: `re.search("^regex$", "string", re.MULTILINE)`
- Ruby: `string.match(/^regex$/m)`
 - As we can see we have to use constant or place `\m`

Word boundaries

Saturday, January 14, 2017 12:18 AM

WORD BOUNDARIES

- Word boundary present boundary for a word, and word may have '\w' characters([a-zA-Z0-9_])
- Reference a position, not an actual character
(so this not an actual word it just represent position(like \$ and ^))

Metacharacter	Meaning
\b	Word boundary (start/end of word)
\B	Not a word boundary

- Conditions for matching
 - Before the first word character in the string
 - After the last word character in the string
 - Between a word character and a non-word character
- Word characters: [A-Za-z0-9_]
(word characters can also be represented by \w)
- Support
 - Most regex engines, not in early Unix tools (BREs)
(so we can use in EBREs(egrep or grep -E) but not BREs (grep))
- Boundary examples
 - /\b\w+\b/ finds four matches in "This is a test."
 - /\b\w+\b/ matches all of "abc_123" but only part of "top-notch"

```
\b\w+\b
```

```
ashish;  
abc_123  
abc$asdf  
top-notch
```

- Not a boundary examples

- `/\BThis/` does not match "This is a test."
- `/\B\w+\B/` finds two matches in "This is a test." ("hi" and "es")

(`\Bw+\B` means it will match anything which matches anything without having word boundaries.

So in 'test' first and last t both are word boundaries so it will match 'es' where 'e' and 's' are referred as non-word boundaries.

- For matching words it is better to always use word-boundaries as it will speed up the search.

- Caution

- A space is not a word boundary
- Word boundaries reference a position
 - Not an actual character
 - Zero-length

- Examples

- String: "apples and oranges"
- No match: `/apples\band\boranges/`
- Match: `/apples\b \band\b \boranges/`

(obviously in example boundaries reference to 's' of 'apples' and so on...

Back-references

Wednesday, January 18, 2017 1:27 AM

- In back-references grouped expressions, grouped expression are captured/saved to be used later.

BACKREFERENCES

- Grouped expressions are captured
 - Stores the matched portion in parentheses
 - `/a(p{2}l)e/` matches "apple" and stores "ppl"
 - Stores the data matched, not the expression
 - Automatically, by default
- Backreferences allow access to captured data
 - Refer to first backreference with `\1`
- here as 'ppl' is stored as it was in group and remember for later.
- So the regex engine stores 'ppl' not 'p{2}l' regex.

Metacharacter	Meaning
<code>\1 through \9</code>	Backreference for positions 1 to 9

- Usage
 - Can be used in the same expression as the group
 - (we can use previous stored backreference using `\n` and use it as group later)

Can be accessed after the match is complete

(we can use it after match complete in programming language)

Cannot be used inside character classes

- Support
 - Most regex engines support `\1 through \9`
 - Some regex engines support `\10 through \99`
 - Some regex engines use `$1 through $9` instead

- Examples

- `/(apples) to \1/` matches "apples to apples"
- `/(ab)(cd)(ef)\3\2\1/` matches "abcdefefcdab"
- `/<(i|em)>.+?</\1>/` matches "<i>Hello</i>" and "Hello"
 - Does not match "<i>Hello" or "Hello</i>"

(it doesn't match above because it stores the value `i` or `em` not `(i|em)`)

Example - 1

```
(nit[bkd]) is the best \1.  
  
nitb is the best nitb.  
nitk is the best nitk.  
nitd is the best nitd.
```

(it has stored whatever matched and then we can use it later, so for first example 'nitb' matched, so it stores that which can be used later.

Example - 2 finding consecutive same words.

```
\b(\w+)\b\s+\b\1\b
```

```
paris is the  
the best place place in in world.
```

Back-references to optional expression

Wednesday, January 18, 2017 1:57 AM

- Back-references stores whatever is matched by group expression.

- Optional elements

- `/A?B/` matches "AB" and "B"

- Captures occur on zero-width matches

- `/(A?)B/` matches "AB" and captures "A"
- `/(A?)B/` matches "B" and captures ""

(here in second case as `/(A?B)/` matches "B" it means group matches nothing so, `\1` will be "", so back-references become of zero-width.

- Backreferences become zero-width too

- `/(A?)B\1/` matches "ABA" and "B"
- `/(A?)B\1C/` matches "ABAC" and "BC"

OPTIONAL GROUPS -

- If groups are made optional(using ?) then groups are called option groups.

- Captures do not always occur on optional groups

- `/(A)?B/` matches "AB" and captures "A"
- `/(A)?B/` matches "B" and does not capture anything

- In second case, group '(A)' do not matches with anything so capturing will not be there. and first case , 'A' will be captured because group (A) matches 'A'.

GROUP FAILURE -

- Group is said to be failed when, it do not match anything and we have used it backreference. so in next example, we have use `\1` assuming A will match,so when group (A) will not match, `\1` FAILS, thus group failed.
- Backreferences is to a group that failed to match
 - `/(A)?B\1/` matches "ABA" but not "B"
 - Except in JavaScript
- In JS or ActionScript (both based on ECMAScript) it also matches both ABA and B, but in all other if fails to matches on group reference failure(when do not match anything).

Example- 1

JavaScript example -



regexpal 0.1.4 — a JavaScript regular expression tester

☒ Global (g) ☐ Case insensitive (i) ☒ Multiline anchors (m) ☒ Dot matches all (s)



☒ Global (g) ☐ Case insensitive (i) ☒ Multiline anchors (m) ☒ Dot matches all (s)

(A)?B\1

ABA
B

REASON - so In java script it will ignore the backrefrence of group if group do not match anything and match), so here because group-1 { having (A) } do not matches, we ignore \1 , but id do not happen in others)

- **Java example -**

ABA
B

(A)?B\1

#

Ignore case Dot-all Multi-line UNIX-lines Comments Unicode case

ABA
B

B

ABDAD

(A)?B(C)?(D)?\1\3

#

☐ Ignore case
 ☐ Dot-all
 ☐ Multi-line
 ☐ UNIX-lines
 ☐ Comments
 ☐ Unicode case

ABDAD

#

- Note - always number will be assigned to groups form left to right , so we here have 3 groups (1,2 and 3). And here it matches accordingly.

Example- 2

(A)?B(C)?(D)?\1\3

BDD

- Here as group 1, and 2 do not matches \1 get ignored, so regex- B(D)?\1, which get matched.

BDD

(A)?B(C)?(D)?\1\3

#

☐ Ignore case
 ☐ Dot-all
 ☐ Multi-line
 ☐ UNIX-lines
 ☐ Comments
 ☐ Unicode case

BDD

BDD

- Element is optional, group/capture is not optional
 - `/(A?)B/` matches "B" and captures ""
- Element is not optional, group/capture is optional
 - `/(A)?B/` matches "B" and does not capture anything

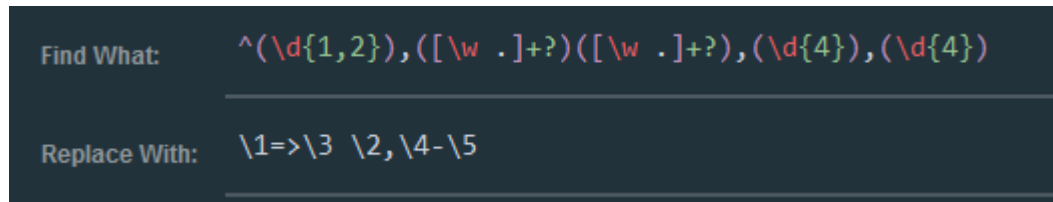
(so if capture depend upon group, so in first case group is not optional so, always something will be captured, and in second case group is optional and when it will not be used we get nothing captured.)

Find and Replace using Backreference

Monday, January 30, 2017 7:52 PM

FIND AND REPLACE USING BACKREFERENCES

- We can use backreferences in replace to replace with the value we found.
- This can't be done regex-pal but can be done in programming languages and Text-editors



The screenshot shows a text editor interface with a dark background. It has two input fields. The first field, labeled 'Find What:', contains the regular expression `^(\d{1,2}),([\w .]+?)([\w .]+?),(\d{4}),(\d{4})`. The second field, labeled 'Replace With:', contains the replacement string `\1=>\3 \2,\4-\5`. The backslashes in the replacement string are highlighted in blue.

- So here in above example in sublime text editor \1 and so on are backreferences which are captured. We may have to use \$1 and so on in some other text-editors or Programming languages.

PROCESS -

- Create a regular expression that matches target data
- Test regular expression and revise as needed
 - Use anchors and specificity to narrow scope
- Add capturing groups
 - Capture anything that varies row-to-row
- Write the replacement string
 - Use all captures
 - Add back anything not captured but still needed
 - May need to use \$1 instead of \1

Non-capturing Group expression

Monday, January 30, 2017 8:14 PM

NON-CAPTURING GROUP EXPRESSIONS

- By default all groups are captured by regex engines.
- We may tell the engine not to capture a group by following metacharacter.

Metacharacter	Meaning
<code>?:</code>	Specify a non-capturing group

- Use that meta-character in the starting of group to tell not to capture a group.

SYNTAX MEANING -

- `/(?:regex)/`
 - `?` = "Give this group a different meaning"
 - `:` = "The meaning is non-capturing"
 - Syntax
 - `/(\w+)/` becomes `/(?:\w+)/`
 - Turns off capture and backreferences
 - Optimize for speed
 - Preserve space for more captures
 - Support
 - Most regex engines except Unix tools
- Only older unix tool do not support it , the invention to this comes from PERL compatible languages.

Example -



regexpal 0.1.4 — a JavaScript regular expression tester

☒ Global (g) ☐ Case insensitive (i) ☐ Multiline anchors (m) ☐ Dot matches all (s)

(?:oranges) and (apples) to \1

oranges and apples to oranges

oranges and apples to apples

- So here (apples) become \1

LOOKAROUND Assertions Introduction

Tuesday, January 31, 2017 1:46 AM

- LOOKAROUND Assertions allows us to look around to match some condition for matching regex.
- LOOKAROUND Assertions are of two types
 - LOOKAHEAD
 - POSITIVE LOOKAHEAD
 - NEGATIVE LOOKAHEAD
 - LOOKBEHIND
 - POSITIVE LOOKBEHIND
 - NEGATIVE LOOKBEHIND
- Allows testing of a regular expression apart from matching
 - Peek forwards or backwards
 - `/sea(?=shore)/`
 - Match a string using multiple expressions
 - `/^(?=.*\d)(?=.*[A-Z]).{8,15}$/`
 - Define rejection expressions
 - `/online(?! training)/`
 - Find last occurrence
 - `/(\item)(?!.*\1)/`

Positive Lookahead Assertions

Monday, January 30, 2017 8:36 PM

POSITIVE LOOKAHEAD ASSERTIONS

Metacharacter	Meaning
<code>?=</code>	Positive lookahead assertion

- Assertion of what ought to be ahead
 - If lookahead expression fails, the match fails
 - Any valid regular expression can be used
 - Zero-width, does not include group in the match
- They are zero-width like word boundaries(`\b`), same like word boundaries which refers to word boundaries and not actually have any width.
Assertions only tells if the assertion success or fails. They store true/false accordingly.
- Syntax
 - `/(?=regex)/`
 - Like non-capturing group expression here in syntax `?=`
 - `?` Means this group has something unusual
 - `=` means it is look around assertion (waha `:` means non-capturing)
 - They will not match anything itself but they just indicate some assertions.
- Examples
 - `/(?=seashore)sea/` matches "sea" in "seashore" but not "seaside"
 - Same as `/sea(?=shore)/`
- In `/(?=seashore)sea/` it say assert 'seashore' if there, then match 'sea'
- In `/sea(?=shore)/` it says to match sea only when it has shore after it,
So we have to make sure 'shore' is after 'sea' to match it.
 - This is same as first one because there we have 'sea' common which we can take before.

Example -2 Let we wanted to match all words having , after them.

WITH OUT ASSERTION

```
\b[A-Za-z ']+\b,
```

```
Self-Reliance  
By Ralph Waldo Emerson, 1841
```

```
"Ne te quæsiueris extra."
```

- But there is a problem , it also match comma itself, but we do not wanted to do this.
We can use assertion here.

WITH ASSERTION

```
\b[A-Za-z ']+\b(?:=,)
```

```
Self-Reliance  
By Ralph Waldo Emerson, 1841
```

Double-testing with LOOKAHEAD

Tuesday, January 31, 2017 12:22 AM

- **Examples**

- `/(?=seashore)sea/` matches "sea" in "seashore" but not "seaside"
- Same as `/sea(?=shore)/`

- Although both matches same thing there is **2 important difference** between them
 - **Order of execution** - In first it checks assertion first and then try to match and in Second it matches then checks assertion, we can optimize a/c to our text.
 - **Double checking** - in second first type it checks assertion if it passes then checks word itself, so we are doing 2 different types of checking it, this property is used very much
 - We can also stack to check 3 times and so on.

- **Match a pattern that also matches another pattern**

- `/\d{3}-\d{3}-\d{4}/` matches "555-302-4321" and "555-781-6978"
- `/^[0-5\ -]+$` matches "555-302-4321" and "23140-5"
- Now we wanted to check a number to match both conditions (first and second above), we can stack them using assertion, we will make 1 condition assertion and then apply other.
RULE - make n-1 assertions and 1 main regex

`/(?=[0-5\ -]+\d{3}-\d{3}-\d{4})/` matches "555-302-4321"

- We can also stack to check more times and so on.

`/(?=[0-5\ -]+\d{3}-\d{3}-\d{4})(?=.*4321)\d{3}-\d{3}-\d{4}/` matches "555-302-4321"

WORKING -

- The regex engine first matches look ahead assertion, if then it goes to check second and so on., If any thing is false then it comes back.

Example - Matching a password having at least 1 digit and capital letter.

`^(?=.*\d)(?=.*[A-Z]).{8,15}$`

`sword42Fish`

- Step 1 - First assertion matches to 'sword42', so it goes further(it matches something we haven't used ^ and ? To match full))
- Step 2 - Second assertion Matches 'sword42F'
- Now, regex matches full , regex is `/'^.{8,15}$/'` here because `(?=regex)` has no width, they are just reference.

Negative lookAhead Assertion

Tuesday, January 31, 2017 12:46 AM

- Opposite to Positive look ahead assertion.
- They return true if assertion not matches regex.

Metacharacter	Meaning
?!	Negative lookahead assertion

• Syntax

- `/(!regex)/`

• Examples

- `/(!seashore)sea/` matches "sea" in "seaside" but not "seashore"
- Same as `/sea(!shore)/`

- It finds if there is 'seashore' , if yes then it will not match sea.

- `/online(! training)/` does not match "online training"

Example - it will match all number

- 1- Having 0-5 digits
- 2- Not having 4321
- 3- Having pattern 3-3-4

```
(?=^[0-5\-\-]+$)(?!.*4321)\d{3}-\d{3}-\d{4}
```

555-302-4321

555-781-6978

555-245-1312

Example - it will match last 'black' in a string.

```
(\bbblack\b)(?!.*\1)
```

The black dog followed the black car into the black night.

Lookbehind Assertions

Tuesday, January 31, 2017 12:55 AM

LOOKBEHIND ASSERTIONS

- We use less than (<) sign to tell negative assertion.
 - Syntax
 - `/(?<=regex)/`
 - `/(?<! regex)/`
 - Examples
 - `/(?<=base)ball/` matches the "ball" in "baseball" but not "football"
 - Same as `/ball(?<=baseball)/`
- Here it say look behind 'base' and if it is present then match ball.
- In LOOKAHEAD we say, look ahead 'seashore' if is present then match 'sea'.

`/(?=seashore)sea/` matches "sea" in "seashore" but not "seaside"

LOOK BEHIND NEGATIVE ASSERTIONS

`/(?<!base)ball/` matches the "ball" in "football" but not "baseball"

- Support
 - Simple expressions in .NET, Java, Perl, PHP, Python, Ruby 1.9
 - Not supported in JavaScript, Ruby 1.8, Unix
- Only simple expressions are allowed and these are
 - Simple expressions means fixed length
 - Literal text
 - Character classes
 - No repetition or optional expressions
 - Alternation only with fixed-length items
 - Allowed: `(?<=cat|dog|rat)`
 - Not allowed: `(?<=apple|banana|plum)`

POWER OF POSITIONS

Tuesday, January 31, 2017 1:45 AM

THE POWER OF POSITIONS

- It is related to find and replace , see video

About UNICODE

Tuesday, January 31, 2017 1:56 AM

ABOUT UNICODE

- Initially we have single byte = $2^8=256$ characters and then comes 2 bytes and so on.
- Single byte
 - Uses one byte (eight bits) to represent a character
 - Allows for 256 characters
 - A-Z, a-z, 0-9, punctuation, common symbols
- Double byte
 - Uses two bytes (16 bits) to represent a character
 - Allows for 65,536 characters
- More space needed because of following
 - Many more characters than English alphabet
 - Latin: à á â ã ä å
 - Symbols: ≤ ≥ ≠ € £
 - Arabic, Chinese, Greek, Hebrew, Korean, Thai, ...
 - Over 109,000 characters
- Some other system is needed so comes Unicode comes.
 - Unicode
 - Variable byte size
 - Maintains compatibility with one- and two-byte encoding
 - Allows for over one million characters
- Unicode is variable size it automatically assign required bytes for a character.
 - for english it gives 1 byte and so on.

- Unicode
 - Mapping between a character and a number
 - "U+" followed by a four-digit hexadecimal number
 - ∞ is written as U+221E
 - Combinations
 - é can be U+00E9 or U+0065 U+0301
 - Can combine more than two

UNICODE IN REGULAR EXPRESSIONS

- Complications for regular expressions
 - Words can be spelled multiple ways
 - "cafe", "café"
 - Words can be encoded multiple ways
 - "café" can be encoded as four or five characters
 - Wildcard matching
 - Backtracking
 - Unicode is relatively new

SOLUTION -

- Unicode indicator: \u
 - \u followed by a four-digit hexadecimal number (0000-FFFF)
 - /caf\u00E9/ matches "café" but not "cafe"
- Support
 - Java, JavaScript, .NET, Python, Ruby
 - Perl and PHP use \x instead
 - Not supported in older Unix tools

Unicode Wildcard and Property

Tuesday, January 31, 2017 2:05 AM

UNICODE WILDCARD

- Unicode wildcard: `\X`
 - Matches any single character
 - Always matches line breaks (like `./s`)
 - `/caf\X/` matches "café" and "cafe"
- Support
 - Only supported in Perl and PHP

UNICODE PROPERTIES

- Unicode property: `\p{property}`
 - Matches characters that have a property
 - `/\p{Mark}/` or `/\p{M}/` matches any "mark" (accents)
 - `/\p{Letter}/` or `/\p{L}/` matches any letter

Unicode property	Abbreviation
Letter	L
Mark	M
Separator	Z
Symbol	S
Number	N
Punctuation	P
Other	C

Task -Understand these

- Support
 - Java, .NET, Perl, PHP, Ruby
 - Not JavaScript, Python, and Unix tools

