Regular Expressions

The construct called ‘regular expression’ is a method to isolate, add, remove, fold or replace certain patterns of text within other text. The term ‘regex’ is often used instead.

The simplest example is a Find command that exists in every text editor, for example: Find ‘abc’ in *text*  - In this case we crafted a string ‘abc’ to for inside the text that’s in the variable (or file*) text*.

Regular expressions are used as constructs by many languages like Java, Python, Perl, Ruby, Unix/Linux shell and many more. There are nuances and different ‘flavors’, but there are many common aspects.

A slightly more advanced example: We need to find all the words in a file of English words, that start with the letter ‘q’ followed by letter other than ‘u’. The file name is: wordlist.txt and we’ll use ‘egrep’ utility/command:

egrep ‘q[^u]’ wordlist.txt (In a few pages the explanation for this will come).

Regex is sort of a language, with rules and characters that mean different things when found in different contexts (called metacharacters)

Metacharacters  
Those are characters with special meaning when appear in a regex. To mark the beginning of a line, we use: ^ (caret), and the end of the line: $ (Dollar sign). So if my assignment is to find lines that contain only: ‘cat’ (with no spaces and no other characters on the line, my expression would be: ‘^cat$’

Dot (or period) means ‘any’ character (very powerful tool)

The vertical bar is used for alternation (or), for example: ‘cat|dog’ will match eiter ‘cat’ or ‘dog’.

Parenthesis are used for grouping and limiting scope of alternation, for example: ‘(T|t)he’ will match ‘The’ or ‘the’

\* (star), means repeating the character (or unit, or sub-expression) before it, 0 or more times.

+ (plus) means similarly but repeating at least once or more. ? (question mark means the previous sub-expression is optional. Collectively the star, plus and question mark are called ‘quantifiers’. We could also use interval quantifiers with minimum and maximum like this: {min,max} or example:

‘[a-zA-Z@#$]+([a-zA-Z0-9$#@\_]){0,30}’ match any legal identifier in PL/1. (the first character s alphabetic including the 3 extra-lingual, then any alphabetic, digit or underscore, up to 31 altogether.

Character class, are characters surrounded by square brackets, for instance: [abc] - what that means is that only one character of the class is used for a match (in this case either ‘a’, ‘b’ or ‘c’). Using this, for instance, to find the word ‘The’ or ‘the’, would be likethis: [Tt]he

In other words, the content of a character class is a list of characters, one of which can match at that point, so the implication is ‘or’

Inside a character class, some metacharaters lose their special meaning. For example the metacharater \* (star), and the + (plus) inside a character class they both simply mean themselves. The same is true for the ‘or’ | (vertical bar)

For example: ab\* can match: a, ab, abb, abbb, abbbb and so on. On the other hand, [ab\*] can match only: a, b or \*

A character class has a metacharater dash (-), only if it appears between characters, for example: [0-9a-z] mean all decimal digits and all lowercase English letters. If, however we write: [-abd], that means either -, a, b or d

Negation – Another important aspect of character class is matching everything ‘not’ in the class. For that we use the caret sign, as first in the class: [^abc] matches any character that is NOT a, b or c. Now we can understand why: ‘q[^u]’ matched every word starting with q, with a second letter different than u.

Some flavors of regex use \< and \> to mark the start and end of a WORD (similar to ^ and $ for line boundaries). egrep uses the switch –i to ignore letter case.

Back-reference – Another use of parenthesis is to refer back to text that matched an earlier sub-expression. In order to use it we deploy the sequence: \1 which means the group that matched. We can do that with more groups like \2 \3 etc. For example: ‘([a-z])([0-9])\1\2’ The \1 refers to the text matched by [a-z] and \2 refers to the text matched by [0-9]

Escape – In order to refer to metacharacters as regular ones, we can escape them by preceding them with a backslash. There for: ‘\.tr.\*’ will match text like .trash (the first dot is escaped so it means a ‘dot’ and the second means any character. Escape does not work in a character class. Another example, matching a word within parenthesis can be done by: ‘\([a-zA-Z]+\)’

When crafting a regex it’s important to know the data we’ll be working with, so we can find the balance between creating a perfect regex that always works, but will be very complex, vs. creating a quick regex that will give us good enough results. For example if we want to identify lines containing URL’s in a 50,000 line text, we can suggest using:

egreg –i ‘\<http:/[-a-z0-9\_.:]+/[-a-z0-9\_:@&?=+,.!/~\*’%$]\*\.tml?\>’ , however, this regex will match: ‘http://…./nada.html’ - which is certainly not a URL, but we can then filter it out ourselves.

Real life problem: Craft a regex to match any HTML tag. If you try: ‘<.\*>’ and your text is: ‘<I>short</I> it will match the entire thing and not the ‘<I>’ A better choice will be: ‘<[a-zA-Z]+>’

Parenthesis  
They are used for either grouping characters (to apply a quantifier on, or alteration with |), or for capturing the matching value. For example: /^\.([0-9]+)/ will capture the digits on the right of the decimal point, for strings that start with the decimal point, i.e : .345 🡪 $1 will be 345.  
The captured values (we can have multiple sets of parenthesis) are placed in special variables (Perl) named: $1, $2,…. (according to their placement in the expression from left to right).

None capturing parenthesis  
If we want to group items , but not create a ‘captured item’ (referred to with $1, $2 etc.), we can specify: (?:)  
Example:

if (/(bronto)?saurus (steak|burger)/){  
 Print “We’ll eat $1 \n”;   
} # will not work, because $1 will be the   
 # (bronto) part

We’ll need: if (/(?:bronto)?saurus (steak|burger)/)

Match variables ($1, $2 etc.) persist until the next successful match.  
$& - The whole part that matched  
$` - The whole part before the match in the string  
$’ - The whole part after the match in the string

(those three together will always be the whole string)

Back-reference is denoted by \1, \2 etc. and refer back to captured items number 1,2 etc. It is used to match some string with a repeated substring.   
For example:

(.)\1 Matches any two characters repeated (except newline), like: ‘aa’, ‘&&’ etc.

Later versions of Perl (5 an up) we can denote back-reference with: \g{N} N is the number of the group.

Summary

|  |  |  |
| --- | --- | --- |
| Metacharacter |  | Matches |
| .  [ ]  [^ ]  \*char* | Dot  Character class  Negated character  Escaped character | Any character except for a newline (\n)  Any one character inside  Any one character not in  Matches the literal *char* |
| Quantifiers for previous items | | |
| ?  \*  +  {min,max} | Question mark  Star  Plus sign  range | One allowed, but optional  Any number allowed including none  At least one required, more optional  Min required, max allowed |
| Items that match a position | | |
| ^  $  \<  \> | Caret  Dollar sign  Back slash + less than  Back slash + more than | Beginning of a line  End of a line  Position of word’s start  Position of word’s end |
| Other | | |
| |  ( )  \1, \2, …. | Or  Parenthesis  Back-reference | Either expression it separates  Grouping, limit scope of alternation, captures back-references.  Text previously matched by 1st, 2nd etc. group |
|  |  |  |

Python flavor

\s – whitespace, equivalent to [ \t\n\r\f\v]  
\S – anything but a whitespace (like [^\s]  
\d – any digit (like [0-9]  
\D – anything but a digit (like [^0-9]  
\b – whitespace around words (word boundary), backspace in character class  
\B – whitespace, but only not around words  
\w – Any alphanumeric character, including underscore (like [a-zA-Z0-9\_]  
\W – The complement of \w  
{n} – like a range with fixed number

import re, or: from re import \*  
str=”………….”  
result=re.findall(regex, str)

re functions (methods):  
match – matches a regex to the beginning of a string  
fullmatch - matches a regex to all of a string  
search – Searches a string for presence of the regex  
sub – substitute occurrences of a pattern (regex) found in a string  
subn – The same as sub, but also returns a number of substitutions made  
split – splits a string by a pattern  
findall(pattern,string,flags=I) – returns a list of all matches

Using flags: findall(pattern,string,flags=i) # i means: case insensitive.