PERL - REGULAR EXPRESSIONS

A regular expression is a string of characters that defines the pattern or patterns you are viewing. The syntax of regular expressions in Perl is very similar to what you will find within other regular expression.supporting programs, such as **sed**, **grep**, and **awk**.

The basic method for applying a regular expression is to use the pattern binding operators **=~** and **!~**. The first operator is a test and assignment operator.

There are three regular expression operators within Perl.

- Match Regular Expression - m//
- Substitute Regular Expression - s///
- Transliterate Regular Expression - tr///

The forward slashes in each case act as delimiters for the regular expression *regex* that you are specifying. If you are comfortable with any other delimiter, then you can use in place of forward slash.

The Match Operator

The match operator, m//, is used to match a string or statement to a regular expression. For example, to match the character sequence "foo" against the scalar $bar, you might use a statement like this −

```
#!/usr/bin/perl

$bar = "This is foo and again foo";
if ($bar =~ /foo/){
   print "First time is matching\n";
}else{
   print "First time is not matching\n";
}

$bar = "foo";
if ($bar =~ /foo/){
   print "Second time is matching\n";
}else{
   print "Second time is not matching\n";
}
```

When above program is executed, it produces the following result −

```
First time is matching
Second time is matching
```

The m// actually works in the same fashion as the q// operator series.you can use any combination of naturally matching characters to act as delimiters for the expression. For example, m{}, m, and m>< are all valid. So above example can be re-written as follows −

```
#!/usr/bin/perl

$bar = "This is foo and again foo";
if ($bar =~ m[foo]){
   print "First time is matching\n";
}else{
   print "First time is not matching\n";
}

$bar = "foo";
if ($bar =~ m{foo}){
   print "Second time is matching\n";
}else{
   print "Second time is not matching\n";
}
```

You can omit m from m// if the delimiters are forward slashes, but for all other delimiters you must use the m prefix.

Note that the entire match expression that is the expression on the left of =~ or !~ and the match operator, returns true *inascalarcontext* if the expression matches. Therefore the statement −

```
$true = ($foo =~ m/foo/);
```

will set *trueto1if*foo matches the regex, or 0 if the match fails. In a list context, the match returns the contents of any grouped expressions. For example, when extracting the hours, minutes, and seconds from a time string, we can use −

```
my ($hours, $minutes, $seconds) = ($time =~ m/(\d+):(\d+):(\d+)/);
```

Match Operator Modifiers

The match operator supports its own set of modifiers. The /g modifier allows for global matching. The /i modifier will make the match case insensitive. Here is the complete list of modifiers

Modifier	Description
i	Makes the match case insensitive.
m	Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary.
o	Evaluates the expression only once.
s	Allows use of . to match a newline character.
x	Allows you to use white space in the expression for clarity.
g	Globally finds all matches.
cg	Allows the search to continue even after a global match fails.

Matching Only Once

There is also a simpler version of the match operator - the ?PATTERN? operator. This is basically identical to the m// operator except that it only matches once within the string you are searching between each call to reset.

For example, you can use this to get the first and last elements within a list −

```
#!/usr/bin/perl

@list = qw/food foosball subeo footnote terfoot canic footbrdige/;

foreach (@list)
{
   $first = $1 if ?(foo.*)?;
   $last = $1 if /(foo.*)/;
}
print "First: $first, Last: $last\n";
```

When above program is executed, it produces the following result −

```
First: food, Last: footbrdige
```

Regular Expression Variables

Regular expression variables include **$**, which contains whatever the last grouping match matched; **$&**, which contains the entire matched string; **$`**, which contains everything before the matched string; and **$'**, which contains everything after the matched string. Following code demonstrates the result −

```
#!/usr/bin/perl

$string = "The food is in the salad bar";
$string =~ m/foo/;
print "Before: $`\n";
print "Matched: $&\n";
print "After: $'\n";
```

When above program is executed, it produces the following result −

```
Before: The
Matched: foo
After: d is in the salad bar
```

The Substitution Operator

The substitution operator, s///, is really just an extension of the match operator that allows you to replace the text matched with some new text. The basic form of the operator is −

```
s/PATTERN/REPLACEMENT/;
```

The PATTERN is the regular expression for the text that we are looking for. The REPLACEMENT is a specification for the text or regular expression that we want to use to replace the found text with. For example, we can replace all occurrences of **dog** with **cat** using the following regular expression −

```
#/user/bin/perl

$string = "The cat sat on the mat";
$string =~ s/cat/dog/;

print "$string\n";
```

When above program is executed, it produces the following result −

```
The dog sat on the mat
```

Substitution Operator Modifiers

Here is the list of all modifiers used with substitution operator.

Modifier	Description
i	Makes the match case insensitive.
m	Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary.
o	Evaluates the expression only once.
s	Allows use of . to match a newline character.
x	Allows you to use white space in the expression for clarity.
g	Replaces all occurrences of the found expression with the replacement text.

e	Evaluates the replacement as if it were a Perl statement, and uses its return value as the replacement text.

The Translation Operator

Translation is similar, but not identical, to the principles of substitution, but unlike substitution, translation *ortransliteration* does not use regular expressions for its search on replacement values. The translation operators are −

```
tr/SEARCHLIST/REPLACEMENTLIST/cds
y/SEARCHLIST/REPLACEMENTLIST/cds
```

The translation replaces all occurrences of the characters in SEARCHLIST with the corresponding characters in REPLACEMENTLIST. For example, using the "The cat sat on the mat." string we have been using in this chapter −

```
#/user/bin/perl

$string = 'The cat sat on the mat';
$string =~ tr/a/o/;

print "$string\n";
```

When above program is executed, it produces the following result −

```
The cot sot on the mot.
```

Standard Perl ranges can also be used, allowing you to specify ranges of characters either by letter or numerical value. To change the case of the string, you might use the following syntax in place of the **uc** function.

```
$string =~ tr/a-z/A-Z/;
```

Translation Operator Modifiers

Following is the list of operators related to translation.

Modifier	Description
c	Complements SEARCHLIST.
d	Deletes found but unreplaced characters.
s	Squashes duplicate replaced characters.

The /d modifier deletes the characters matching SEARCHLIST that do not have a corresponding entry in REPLACEMENTLIST. For example −

```
#!/usr/bin/perl

$string = 'the cat sat on the mat.';
$string =~ tr/a-z/b/d;

print "$string\n";
```

When above program is executed, it produces the following result −

```
b b   b.
```

The last modifier, /s, removes the duplicate sequences of characters that were replaced, so −

```
#!/usr/bin/perl

$string = 'food';
$string = 'food';
$string =~ tr/a-z/a-z/s;

print "$string\n";
```

When above program is executed, it produces the following result −

```
fod
```

More Complex Regular Expressions

You don't just have to match on fixed strings. In fact, you can match on just about anything you could dream of by using more complex regular expressions. Here's a quick cheat sheet −

Following table lists the regular expression syntax that is available in Python.

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
*	Matches 0 or more occurrences of preceding expression.
+	Matches 1 or more occurrence of preceding expression.
?	Matches 0 or 1 occurrence of preceding expression.
{ n}	Matches exactly n number of occurrences of preceding expression.
{ n,}	Matches n or more occurrences of preceding expression.
{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.

\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace $0x08$ when inside brackets.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.
[aeiou]	Matches a single character in the given set
[^aeiou]	Matches a single character outside the given set

The ^ metacharacter matches the beginning of the string and the $ metasymbol matches the end of the string. Here are some brief examples.

```
# nothing in the string (start and end are adjacent)
/^$/

# a three digits, each followed by a whitespace
# character (eg "3 4 5 ")
/(\d\s){3}/

# matches a string in which every
# odd-numbered letter is a (eg "abacadaf")
/(a.)+/

# string starts with one or more digits
/^\d+/

# string that ends with one or more digits
/\d+$/
```

Lets have a look at another example.

```
#!/usr/bin/perl

$string = "Cats go Catatonic\nWhen given Catnip";
($start) = ($string =~ /\A(.*?) /);
@lines = $string =~ /^(.*?) /gm;
print "First word: $start\n","Line starts: @lines\n";
```

When above program is executed, it produces the following result −

```
First word: Cats
Line starts: Cats When
```

Matching Boundaries

The **\b** matches at any word boundary, as defined by the difference between the \w class and the \W class. Because \w includes the characters for a word, and \W the opposite, this normally means the termination of a word. The **\B** assertion matches any position that is not a word boundary. For example −

```
/\bcat\b/ # Matches 'the cat sat' but not 'cat on the mat'
/\Bcat\B/ # Matches 'verification' but not 'the cat on the mat'
/\bcat\B/ # Matches 'catatonic' but not 'polecat'
/\Bcat\b/ # Matches 'polecat' but not 'catatonic'
```

Selecting Alternatives

The | character is just like the standard or bitwise OR within Perl. It specifies alternate matches within a regular expression or group. For example, to match "cat" or "dog" in an expression, you might use this −

```
if ($string =~ /cat|dog/)
```

You can group individual elements of an expression together in order to support complex matches. Searching for two peoples names could be achieved with two separate tests, like this −

```
if (($string =~ /Martin Brown/) ||  ($string =~ /Sharon Brown/))

This could be written as follows

if ($string =~ /(Martin|Sharon) Brown/)
```

Grouping Matching

From a regular-expression point of view, there is no difference between except, perhaps, that the former is slightly clearer.

```
$string =~ /(\S+)\s+(\S+)/;

and

$string =~ /\S+\s+\S+/;
```

However, the benefit of grouping is that it allows us to extract a sequence from a regular expression. Groupings are returned as a list in the order in which they appear in the original. For example, in the following fragment we have pulled out the hours, minutes, and seconds from a string.

```
my ($hours, $minutes, $seconds) = ($time =~ m/(\d+):(\d+):(\d+)/);
```

As well as this direct method, matched groups are also available within the special $x variables, where x is the number of the group within the regular expression. We could therefore rewrite the preceding example as follows −

```
#!/usr/bin/perl

$time = "12:05:30";

$time =~ m/(\d+):(\d+):(\d+)/;
my ($hours, $minutes, $seconds) = ($1, $2, $3);

print "Hours : $hours, Minutes: $minutes, Second: $seconds\n";
```

When above program is executed, it produces the following result −

```
Hours : 12, Minutes: 05, Second: 30
```

When groups are used in substitution expressions, the $x syntax can be used in the replacement text. Thus, we could reformat a date string using this −

```
#!/usr/bin/perl

$date = '03/26/1999';
$date =~ s#(\d+)/(\d+)/(\d+)#$3/$1/$2#;

print "$date\n";
```

When above program is executed, it produces the following result −

```
1999/03/26
```

The \G Assertion

The \G assertion allows you to continue searching from the point where the last match occurred. For example, in the following code, we have used \G so that we can search to the correct position and then extract some information, without having to create a more complex, single regular expression −

```
#!/usr/bin/perl

$string = "The time is: 12:31:02 on 4/12/00";

$string =~ /:\s+/g;
($time) = ($string =~ /\G(\d+:\d+:\d+)/);
$string =~ /.+\s+/g;
($date) = ($string =~ m{\G(\d+/\d+/\d+)});

print "Time: $time, Date: $date\n";
```

When above program is executed, it produces the following result −

```
Time: 12:31:02, Date: 4/12/00
```

The \G assertion is actually just the metasymbol equivalent of the pos function, so between regular expression calls you can continue to use pos, and even modify the value of pos *andtherefore*\G by using pos as an lvalue subroutine.

Regular-expression Examples

Literal characters

Example	Description
Perl	Match "Perl".

Character Classes

Example	Description
[Pp]ython	Matches "Python" or "python"
rub[ye]	Matches "ruby" or "rube"
[aeiou]	Matches any one lowercase vowel
[0-9]	Matches any digit; same as [0123456789]
[a-z]	Matches any lowercase ASCII letter
[A-Z]	Matches any uppercase ASCII letter
[a-zA-Z0-9]	Matches any of the above
[^aeiou]	Matches anything other than a lowercase vowel
[^0-9]	Matches anything other than a digit

Special Character Classes

Example	Description
.	Matches any character except newline
\d	Matches a digit: [0-9]
\D	Matches a nondigit: [^0-9]
\s	Matches a whitespace character: [\t\r\n\f]
\S	Matches nonwhitespace: [^ \t\r\n\f]
\w	Matches a single word character: [A-Za-z0-9_]
\W	Matches a nonword character: [^A-Za-z0-9_]

Repetition Cases

Example	Description
ruby?	Matches "rub" or "ruby": the y is optional
ruby*	Matches "rub" plus 0 or more ys
ruby+	Matches "rub" plus 1 or more ys
\d{3}	Matches exactly 3 digits
\d{3,}	Matches 3 or more digits
\d{3,5}	Matches 3, 4, or 5 digits

Nongreedy Repetition

This matches the smallest number of repetitions −

Example	Description
<.*>	Greedy repetition: matches "<python>perl>"
<.*?>	Nongreedy: matches "<python>" in "<python>perl>"

Grouping with Parentheses

Example	Description
\D\d+	No group: + repeats \d
\D\d+	Grouped: + repeats \D\d pair
[*Pp*]*ython*(, ?)+	Match "Python", "Python, python, python", etc.

Backreferences

This matches a previously matched group again −

Example	Description
[*Pp*]ython&\1ails	Matches python&pails or Python&Pails
[′ "][^\1]*\1	Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

Example	Description
python|perl	Matches "python" or "perl"
rub*y*|*le*)	Matches "ruby" or "ruble"
Python! + |\?	"Python" followed by one or more ! or one ?

Anchors

This need to specify match positions.

Example	Description
^Python	Matches "Python" at the start of a string or internal line
Python$	Matches "Python" at the end of a string or line
\APython	Matches "Python" at the start of a string
Python\Z	Matches "Python" at the end of a string
\bPython\b	Matches "Python" at a word boundary
\brub\B	\B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
Python? = !	Matches "Python", if followed by an exclamation point
Python?!!	Matches "Python", if not followed by an exclamation point

Special Syntax with Parentheses

Example	Description
R?#comment	Matches "R". All the rest is a comment
R?*i*uby	Case-insensitive while matching "uby"
R?*i*:*uby*	Same as above
rub?:*y*|*le*)	Group only without creating \1 backreference

Processing math: 100%