Regular expressions and case insensitivity

As previously mentioned, you can make matching case insensitive with the `i` flag:

```
/\b[Uu][Nn][Ii][Xx]\b/;          # explicitly giving case folding

/\bunix\b/i;                     # using ‘‘i’’ flag to fold code
```

Really matching any character with "."

As mentioned before, usually the "." (dot, period, full stop) matches any character except newline. You make it match newline with the `s` flag:

```
/"(.|\n)*"/;                 # match any quoted string, even with newlines embedded
/"(.*)"/s;                   # same meaning, using ``s'' flag
```

N.B. – I like to use the flags `///six;` as a personal default set of flags with Perl regular expressions.

Going global with the ‘‘g’’ flag

You can make your matching global with the g flag. For ordinary matches, this means making them stateful: Perl will remember where you left off with each reinvocation of the match unless you change the value of the variable, which will reset the match.

Going global with the ‘‘g’’ flag

```
#!/usr/bin/perl -w
# 2006 09 29 - rdl Script36.pl
# shows the //g as stateful...
while(<>)
{
    while(/[A-Z]{2,}/g)
    {
        print "$&\n" if (defined($&));
    }
}
```

Interpolating variables in patterns

You can even specify a variable inside of a pattern – but you want to make sure that it gives a legitimate regular expression.

Interpolating variables in patterns

```
my $var1 = "[A-Z]*";
if( "AB" =~ /$var1/ )
{
  print "$&";
}
else
{
  print "nopers";
}
# yields
AB
```

Regular expressions and substitution

☞ The s/.../.../ form can be used to make substitutions in the specified string.

☞ If paired delimiters are used, then you have to use two pairs of the delimiters.

☞ g after the last delimiter indicates to replace more than just the first occurrence.

☞ The substitution can be bound to a string using =~ . Otherwise it makes the substitutions in $_.

☞ The operation returns the number of replacements performed, which can be more than one with the 'g' option.

Examples

```
#!/usr/bin/perl -w
# 2006 09 29 - rdl Script37.pl
# shows s///g... by removing acronyms
use strict;
while(<>)
{

    s/([A-Z]{2,})//g;
    print;
}
```

Examples

```
s/\bfigure (\d+)/Figure $1/  # capitalize references to figures
s{//(.*)}{/\*$1\*/}          # use old style C comments
s!\bif(!if (!                # put a blank between  if and  (
s(!)(.)                      # tone down that message
s[!][.]g                     # replace all occurrences of '!' with '.'
```

Case shifting

You can use `\U` and `\L` to change follows them to upper and lower case:

Case shifting

```
$text = " the acm and the ieee are the best! ";
$text =~ s/acm|ieee/\U$&/g;
print "$text\n";
# yields
 the ACM and the IEEE are the best!
```

Case shifting

```
$text = "CDA 1001 and COP 3101

are good classes, but COP 4342 is better!";
$text =~ s/\b(COP|CDA) \d+/\L$&/g;
print "$text\n";
# yields
cda 1001 and cop 3101

are good classes, but cop 4342 is better!
```

Using `tr///` (also known as `y///`)

☞ In Perl you can also convert one set of characters to another using the `tr/.../.../` form. (Or if you like, you can use `y///`.)

☞ Much like the program `tr`, you specify two lists of characters, the first to be substituted, and the second what to substitute.

- ☞ `tr` returns the number of items substituted (or deleted.)
- ☞ The modifer `d` deletes characters not replaced.
- ☞ The modifer `s` “squashes” any repeated characters.

Examples (from the `perlop` man page)

```
$ARGV[1] =~ tr/A-Z/a-z/;    # canonicalize to lower case
$cnt = tr/*/*/;             # count the stars in $_
$cnt = $sky =~ tr/*/*/;     # count the stars in $sky
$cnt = tr/0-9//;            # count the digits in $_
```

More examples

```
# get rid of redundant blanks in $_
tr/ //s;

# replace [ and { with ( in $text
$text =~ tr/[{/(/;
```

Using `split`

The `split` function breaks up a string according to a specified separator pattern and generates a list of the substrings.

Using `substring`

For example:

```
$line = " This sentence contains five words. ";
@fields = split / /, $line;
map {  print "$count --> $fields[$count]\n"; $count++; } @fields;
# yields
 -->
1 --> This
2 --> sentence
3 --> contains
4 --> five
5 --> words.
```

Using the `join` function

The `join` function does the reverse of the `split` function: it takes a list and converts to a string.

However, it is different in that it doesn't take a pattern as its first argument, it just takes a string:

```
@fields = qw/ apples pears cantaloupes cherries /;
$line = join "<-->", @fields;
print "$line\n";
# yields
apples<-->pears<-->cantaloupes<-->cherries
```

Filehandles

[Also see `man perlfaq5` for more detail on this subject.]

A filehandle is an I/O connection between your process and some device or file. Perl output is buffered.

Perl has three predefined filehandles: `STDIN`, `STDOUT`, and `STDERR`.

Filehandles

Unlike other variables, you don't declare filehandles. The convention is to use all uppercase letters for filehandle names. (Especially important if you deal with anonymous filehandles!)

The open operator takes two arguments, a filehandle name and a connection (e.g. filename). The connection can start with "< , > , or ">> to indicate read, write, and append access.

Examples

```
open IN,  in.dat ;      # open  in.dat for input
open IN2,  <$file ;     # open filename in $file for input
open OUT,  >out.dat ;   # open  out.dat for output
open LOG,  >>log.txt ; # open  log.txt to append output
```

Closing filehandles

The close operator closes a filehandle. This causes any remaining output data associated with this filehandle to be flushed to the file.

Perl automatically closes filehandles at the end of a process, or if you reopen it.

Examples

```
close IN;  # closes the IN filehandle
close OUT; # closes the OUT filehandle
close LOG; # closes the LOG filehandle
```

Testing `open`

You can check the status of opening a file by examining the result of the open operation. It returns a true value if it succeeded, and a false one if it failed.

```
if (!open OUT,  >out.dat ) {
      die  Could not open out.dat. ;
}
```

Using a filehandle

```
Open IN,  <in.dat ;
Open OUT,  >out.dat ;
$i = 1;
while ($line = <IN>) {
     printf OUT  %d: $line , $i;
}
```

Note that a comma is not used after the filehandle in a `print` or `printf` statement.

Reopening a filehandle

You can reopen a standard filename. This allows you to perform input or output in a normal fashion, but to redirect the I/O from/to a file within the Perl program.

Examples of reopening a filehandle

```
# redirect standard output to out.txt
open STDOUT, >out.txt ;
printf Hello world!\n ;
# redirect standard error to append to log.txt
open STDERR, >>log.txt ;
```