Whenever I learn a new skill I concoct elaborate fantasy scenarios where it lets me save the day.

Oh no! The killer must have followed her on vacation!

But to find them we’d have to search through 200 MB of emails looking for something formatted like an address!

It’s hopeless!

Everybody stand back.

I know regular expressions.

https://xkcd.com
regular expressions

also known as regex or RE

used to find user-defined patterns in text

used in many programs

  SEXI (SNOBOL) => ed => grep => awk => perl

search patterns (usually) delimited by slashes e.g. /seek/
or almost any other non-word characters e.g. #seek# in Perl

most programs use Perl Compatible RE (PCRE)
pattern matching: tr///

good for ‘cleaning’ data
character based
  e.g. echo 'CBA' | perl -pe 'tr/ABC/abc/' # cba
[
] \

have a special meanings
use \ to get the literal meaning e.g. \ eq \n
can delimit ranges of characters e.g. [A-z], [0-9]
pattern matching: tr///

modifier c: complement
  e.g. echo 'ABC' | perl -pe 'tr/A/x/c' # Axxx

modifier d: delete
  e.g. echo 'ABC' | perl -pe 'tr/A//d' # BC

modifier s: squash
  e.g. echo 'ABB' | perl -pe 'tr/B//s' # AB

modifiers can be used in combination
  e.g. echo 'ABC' | perl -pe 'tr/A//cd' # A

can be used to count number of occurrences
  e.g. echo 'AABC' | perl -lane 'BEGIN{$A=0}
  ${A+=($_ =~ tr/A/A/)}END{print($A)}' # 2
pattern matching: m//

useful for finding text
returns 0 or 1 (true or false)
best used explicitly with =~ or !~
  e.g. if($x =~ m/A/) not if(/A/)
stops after finding the first match
complex patterns are slower than exact matches
pattern matching: m//

minimalism is good
capitalize on the smallest differentiating feature possible
fewer errors, much faster
patterns can be simple e.g. m/ABC/
or complex e.g. m/A[A-z]+C/
. * ? + [ ] ( ) { } ^ $ | \ / have special meanings
escape using slash \


pattern matching: m//

[] delimit sets of characters e.g. [ABC], [A-z]
[^] delimit exclusion sets e.g. [^ABC], [^A-z]
\d digit; \D non–digit
\w word character [a-zA-Z0-9_]; \W not word character
\s white space (\n, space, tab, etc); \S not white space
^ start of chunk
$ end of chunk
| or e.g. (A|B|C)
. (almost) anything
* zero or more matches
+ one or more matches
? zero or one match
{} number of matches e.g. {5}, {5,}, {5,10}
pattern matching: m//

modifiers: g, i, m, s, x

g: global
   find all occurrences, not just the first one
i: case insensitive
   treat capital and lower case letters as equivalent
m: multiline
   use \n to delimit ^ and $ matches
s: single line
   ignore \n when making matches
x: free form
   count white space when making matches
pattern matching: s///

replaces strings in text
  e.g. echo 'ABCD' | perl -pe 's/BC/x/' # AxD
replaces only the first occurrence (by default)
works like m// but finds and replaces
use built-in variable $1, $2, etc. to capture matches
built-in variables start with 1 (like in awk)
  e.g. echo 'ABC' | perl -pe 's/(A.C)/found: $1/'
    # found:ABC
() capturing group; (?:) non-capturing group
The regular expression `/^[A-Z]+[a-z]+\s+(?:[a-z]+)\$/gm` is used to match a string that starts with one or more uppercase letters, followed by one or more lowercase letters, and ends with an optional sequence of lowercase letters.

**EXPLANATION**

1. **1st Capturing Group `[A-Z]`**
   - Match a single character present in the list below:
     - `[A-Z]` a single character in the range between `A` (index 65) and `Z` (index 90) (case sensitive).

2. **Match a single character present in the list below**:
   - `[a-z]+`
   - **Quantifier** — Matches between one and unlimited times, as many times as possible, giving back as needed (greedy).
   - `[a-z]` a single character in the range between `a` and `z`.

**MATCH INFORMATION**

- **Match 1**
  - **Full match** 0-18: Cupressus funebris
  - **Group 1**:
    - n/a: C
  - **Group 2**:
    - n/a: funebris
awk, the other useful scripting language

Alfred Aho, Peter Weinberger, and Brian Kernighan
AT&T Bell Labs (1970s)
many (very similar) versions (e.g. gawk, mawk, tawk)
scripts can be translated into C programs (e.g. awka)
standard tool in most (all?) POSIX operating systems
the limitations of awk inspired Larry Wall, in part, to write Perl (or so it has been written)
Welcome to 99 Bottles of Beer

This Website holds a collection of the Song 99 Bottles of Beer programmed in different programming languages. Actually the song is represented in 1500 different programming languages and variations. For more detailed information refer to historic information.

All these little programs generate the lyrics to the song 99 Bottles of Beer as an output. In case you do not know the song, you will find the lyrics to the song here.

Feel free to browse, to comment and to rate the different programming languages. In case your favourite programming language is missing, please submit your own piece of code. After a short review it will appear on the website.

For any comment, critic or praise concerning this website drop a message in our guestbook or contact one of the team members.

Have a lot of fun,
Oliver, Gregor and Stefan
99 bottles of beer on the wall, 99 bottles of beer.  
Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.  
Take one down and pass it around, 97 bottles of beer on the wall.

...

2 bottles of beer on the wall, 2 bottles of beer.  
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.  
Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.  
Go to the store and buy some more, 99 bottles of beer on the wall.
awk, its a language, really

http://99-bottles-of-beer.net/

#!/usr/bin/awk -f
BEGIN{
    split( "no mo"
    "rexxN"
    "o mor"
    "exsxx"
    "Take "
    "one dow"
    "n and pas"
    "s it around"
    "Go to the "
    "store and buy s"
    "ome more, x bot"
    "lex of beerx o"
    "n the wall" , s,
    "x"); for( i=99 ;
    i>=0; i--){ s[0]=
    s[2] = i ; print 
    s[2 + !(i) ] s[8]
    s[4+ !(i-1)] s[9]
    s[10]" , " s[!i(i)]
    s[8] s[4+ !(i-1)]
    s[9]".i?s[0]--:
    s[0] = 99; print 
    s[6+!i]s[!s[0]]
    s[8] s[4 +!(i-2)]
"
awk, its a language, really
http://99-bottles-of-beer.net/

#!/usr/bin/awk -f
# awk version of 99 bottles of beer
# by Whitey (whitey@netcom.com) - 06/05/95
# updated to create correct text including the ending
# by Osamu Aoki (osamu@debian.org) - 2007-11-04

BEGIN {
    for(i = 99; i >= 0; i--) {
        print ubottle(i), "on the wall," , lbottle(i) "."
        print action(i), lbottle(inext(i)), "on the wall."
        print
    }
}

function ubottle(n) {
    return sprintf("%s bottle%s of beer", n ? n : "No more", n - 1 ? "s" : ")
}

function lbottle(n) {
    return sprintf("%s bottle%s of beer", n ? n : "no more", n - 1 ? "s" : ")
}

function action(n) {
    return sprintf("%s", n ? "Take one down and pass it around," : \
                    "Go to the store and buy some more,"")
}

function inext(n) {
    return n ? n - 1 : 99
}
awk basics

-F to set field separator (FS)
-v to set your own variables
BEGIN{}{}END{}
built–in variables: NF, NR, FS, OFS, $1, $2, …
most useful functions/keywords:
  if/else    conditionally x==y; x!=y; x>=y; x<=y; x>y; x<y
  length     length of a string
  match      match a string with a RE
  print      print to stdout
  printf     print to stdout with formatting
  split      split a string into an array using a separator
Perl

‘Swiss-Army chainsaw’ (Henry Spencer)
  i.e. powerful but inelegant
  reclaimed as a complement by Perl enthusiasts
1987: Perl 1.0 released by Larry Wall
  combines the best of sed, awk, C, and sh
  multiple ways to do (al)most everything
1988: Perl 2.0 released
1989: Perl 3.0 released (GPL)
1991: Perl 4.0 released
1994: Perl 5.0 released
1995: CPAN founded
Perl (and 99 bottles of beer)...
#!/usr/bin/perl
$ARGV[0];
$nBottles = 100 if $nBottles eq '' || $nBottles < 0;

foreach (reverse(1 .. $nBottles)) {
    $s = ($_ == 1) ? "" : "s";
    $oneLessS = ($_ == 2) ? "" : "s";
    print "$_ bottle$s of beer on the wall,
    print "$_ bottle$s of beer,
    print "Take one down, pass it around,
    print $_ - 1, " bottle$oneLessS of beer on the wall\n};
print "\n*burp*\n";
Perl one-liners...

- `e '...'` == execute the code within quotes
- `-p` == print after processing each input chunk
- `-n` == do not print after processing each input chunk
- `-0777` == read the input all at once
- `-i.old` == edit file in-place (makes a copy file.old)
- `-l` == `chomp()` each chunk [remove \n | \r | \r\n]
- `-a` == `split(/ /, $_)` each line into @F
  - `-Fx` to use x instead of <space>

usually: perl -pe || perl -ne || perl -lane || perl -077 -le
…Perl one-liners

@ARGV == arguments used to start Perl
@F == input chunk split by the splitting scalar
default = <space> (-F flag)
$a, $b == used in sort()
$l, $2, … == used in regular expressions
Perl syntax basics

all stand-alone scripts begin with `#!/usr/bin/perl`
all stand-alone scripts (should) end with `exit();`
lines end with semicolons;
comments start with pound symbols `#`
double quotes are processed before the next action
single quotes are literal (no processing)
slashes escape special characters `\`
any pair of characters can be used for quotes
variables

$ == scalar
numbers, letters, strings, references (pointers)
size given by length($scalar)

@ == array
lists of numbers, letters, strings, references
accessed by index position e.g. $array[0]
size given by $#array

% == hash
lists of numbers, letters, strings, references
accessed by a key (a unique value) e.g. $hash{'key'}
size given by scalar(keys(%hash))
common numeric operators:

actions

\(x + y\)  
add

\(x - y\)  
subtract

\(x * y\)  
multiply

\(x / y\)  
divide

\(x % y\)  
modulus

\(x ** y\)  
exponent

\(x = y\)  
set to value

\(x++\)  
add one

\(x--\)  
subtract one

\(x + = y\)  
add to value

\(x - = y\)  
subtract from value

\(x *= y\)  
multiply value by

\(x /= y\)  
divide value by

\(x **= y\)  
exponent of value

\(x .. y\)  
rangle
common numeric operators: tests

$x == y$
equal
$x != y$
not equal
$x < y$
less than
$x > y$
greater than
$x >= y$
greater than or equal to
$x <= y$
less than or equal to
common text operators

**action**

\$x = \$y
- set
\$x .= \$y
- append
\$x \times \$y
- concatenate
\$x \times \$y
- repetition
\$x \ldots \$y
- range
\$x =~ s/old/new/
- replace

**test**

\$x =~ tr/0/1/
- replace
\$x =~ m/n/
- reports
\$x !~ m/n/
- reports
\$x eq \$y
- equal
\$x ne \$y
- not equal
common logical operators

&&
and

||
or

! not
arrays...  

arrays store ‘lists’ of values accessible by an index number
indices start with 0

@array = ('value0', 'value1', 'value2', 'value3');
to access an item use the index number

$array[2] = 'value2';
Perl converts or ignores out of range requests
print("$array[0], $array[1.9], $array[3], $array[10]\n");
  => value0, value1, value3,
the size (number of elements) of an array can be determined by calling the array in a scalar context

```perl
@array = ('value0', 'value1', 'value2', 'value3');
print("$#array\n"); => 3
```

to access a subset of the array use index numbers

```perl
@newArray = @array[1..2];
@newArray = @array[3, 2, 0];
```
...arrays: push...

adds a value to the end of an array
  push(@array, $value);
the same (but faster than):
  $array[$#array + 1] = $value;
...arrays: sort...

sorts arrays alphabetically (case sensitive)
   @newArray = sort(@array);
   @newArray = sort({$a cmp $b} @array);
sorts arrays reverse alphabetically (case sensitive)
   @newArray = sort({$b cmp $a} @array);
sorts arrays alphabetically (case insensitive)
   @newArray = sort({uc($a) cmp uc($b)} @array);
sorts arrays numerically (ascending)
   @newArray = sort({$a <=> $b} @array);
sorts arrays numerically (descending)
   @newArray = sort({$b <=> $a} @array);
...arrays: split...

converts a string into an array
strings and regex are allowed

$x = 'this is a string';
@array = split(/ /, $x); => [this] [is] [a] [string]
($zero, $one, $two, $three) = split(/ /, $x);

@array = split(/,/, $x); => [this is a string]
@array = split(/i|s/, $x); => [th] [] [] [] [a] [tr] [ng]
@array = split(/is/, $x); => [th] [ ] [a string]
arrays: join

converts an array into a string

```perl
$array = ('value0', 'value1', 'value2', 'value3');
$string = join(' ', @array);
print("$string\n");  # => value0 value1 value2 value3

$string = join(' x ', @array);
print("$string\n");  # => value0 x value1 x value2 x value3
```
hashes...

a hash stores a series of values accessed by a unique key

a random entry order list (i.e. a very fast database)

key uniqueness is strictly enforced

%hash = ('key0' => 'value0', 'key1' => 'value1', 'key2' => 'value2', 'key3' => 'value3');

to access a value use its key

() is returned if the key does not exist

print("$hash{"key0"}, $hash{"key3"}, $hash{"key10"}\n");

=> value0, value3,
...hashes...

good for making lists with unique values (keys)
  no sort required (random entry order)
  @unique = keys %{{ map {$_ => 1} @array}};

good for finding data (a hard–coded database)
  extremely fast, but only works with exact matches

good for counting the number of each item
  $hash{$_}++ for @array;
deletes elements from a hash

delete($hash['key']);
returns 0/1 (false/true) if a hash element exists

if(exists($hash['key'])){  
print("it exists\n");}

} else {  
print("it does not exist\n");}

}
returns an array of keys from a hash
in ‘random’ order (may be different each time)
@@array = keys(%hash);
or returns a scalar of hash size
$number = keys(%hash);
returns an array of values from a hash
  in ‘random’ order (may be different each time)
@array = values(%hash);

or returns a scalar of hash size
$number = values(%hash);
...hashes: sort

can sort keys by values alphabetically (case sensitive):
@array = sort({$hash{$a} cmp $hash{$b}} keys(%hash));
numerically (ascending):
@array = sort({$hash{$a} <=> $hash{$b}} keys(%hash));
Perl functions: substr

extracts a string from another string
useful in combination with index

$fragment = substr($string, start)
$fragment = substr($string, start, length)

optionally replaces

$fragment = substr($string, start, length, $new)
Perl functions: length

counts the number of characters in a string

length($string)

starts from 0 (i.e. zero indicates one character)

if NULL, returns -1
Perl functions: reverse

reverses the order of letters in a string

$gnirts = reverse($string)

or reverses the order of elements in an array

@yarra = reverse(@array)
Perl functions: uc

converting strings (or character) to UPPERCASE

$STRING = uc($string)

$STRING = uc($STRING)
Perl functions: lc

converts strings (or character) to lowercase

$string = lc($STRING)

$string = lc($string)
Perl functions: print

prints strings
default to STDOUT (screen)
use sparingly (it can be slow)

print('text')
print("$string")
print(STDOUT "text")
print(OUTFILE "text")
Perl functions: sprintf

formats data (numbers)
useful to making results look pretty
useful for rounding numbers

$number = sprintf("%.3f", $number)

%d == signed integer
%u == unsigned integer
%e == floating point number in scientific notation
%f == floating point number in fixed decimal notation