Regular Expressions: The Power of Perl

1. What is a regular expression (regex) ?

- it is a ‘description’ for a group of characters you want to search for in a string, a file, a website, etc...
- think of the group of characters as a ‘pattern’ that you want to find within a string.
- Use regular expressions to search text quickly and accurately.

Pattern Matching Syntax in Perl

- modifers go here (more on this later)

Syntax:

```
$variable_name =~ /pattern/;
```

- this is the variable containing the string you want to search

- this is the pattern you want to search for within the variable

-this operator is used only for testing regular expression

- letters in this location denote what type of regular expression search this is (more on this later)

regex example

- Suppose we have a text file containing this text:

```
Ah, everybody get on the floor, let's dance
Don't fight the feeling give yourself a chance

Shake, shake, shake - shake, shake, shake
Shake your booty, shake your booty
Oh, shake, shake, shake - shake, shake, shake
Shake your booty, shake your booty

Oh, you can, you can do well, very well
Your love can stand the world, I can tell

Oh, shake, shake, shake - shake, shake, shake
Shake your booty, shake your booty
Oh, shake, shake, shake - shake, shake, shake
Shake your booty, shake your booty, wow wow, yeah!
```

--(Shake, Shake, Shake) Shake Your Booty   KC & The Sunshine Band

- On which lines does the word ‘shake’ occur?
- Using regular expressions we can find out.

```
#!/usr/bin/perl -w

open(TEXT, "<shake.txt") or die "Can't open file.\n";

$lineNum = 1;  ## holds the number of the line
$line    = 0;  ## holds each line as its read in
$count   = 0;  ## keeps count of 'shake' frequency
@lineAry = (); ## holds line numbers

while($line = <TEXT>)
{
  chomp($line); ## remove invisible '\n' character

  if($line =~ /shake/)
  {
    ## count frequency of 'shake' in current line.
    while($line =~ /shake/ig) { $count++; }

    ## store current line number in array
    push(@lineAry, $lineNum);
    $lineNum++;
  }
}
close(TEXT); ## close file

print "\n\n'Shake' occurs $count times in file.\n";
print "It occurs on the following lines: ";
foreach $e (@lineAry) { print "$e, "; }

print "\n\n";
exit;
```

Checking the current line to see if ‘shake’ occurs even once in it.

Counting frequency of ‘shake’ in the current line.

Output

```
'Shake' occurs 32 times in file.
It occurs on the following lines: 1, 2, 3, 4, 5, 6, 7, 8,
```

Special Regular Expressions

- These are regular expression syntaxes which do different things when they find the a given pattern…

`$v =~ m/pattern/`	find a match to ‘pattern’ in $v
`$v =~ s/p1/p2/`	find pattern1 and SUBSTITUTE it with pattern2
`$v =~ tr/abc/xyz/`	find **a** REPLACE with **x**, find **b** REPLACE with **y**, and find **c** REPLACE with **z**

m/pattern/ ← m = match

s/p1/p2/ ← s = substitute

tr/a/z/ ← t = transliteration

Modifiers for regex's

-these modifiers go just after the last '/' of the regular expression statement.

ex: $v1 =~ /pattern/

here!

`$v =~ /pattern/i`	case **i**nsensitive	will match: july and July and JULY
`$v =~ /pattern/g`	do a **g**lobal search	search a string for ALL instances of 'pattern'not just the first one

Special Characters in regex

-There are a series of special regex terms called ‘metacharacters’ used to encompass/describe a variety of string characters

-The metacharacters are meant to decrease your typing and increase the efficiency of a regular expression search

Syntax	Meaning in regex
\d, [0-9]	match a digit
\w, [a-zA-Z_0-9]	match a character
\D	match a non-digit character
\W	match a non-word character
\s, [\t\n\r\f]	match a whitespace character
\S	match a non-whitespace character

\n	match a newline character
\r	match a carriage return
\t	match a tab
\f	match a formfeed
.	match any SINGLE character

- There are other metacharacters, check the web for a complete list.

regex Quantifiers

-these syntax structures allow you to specify how long a regular expression pattern match should be

```
$v = "What's perl good for? agtttgggaaccctaattgaa 934553";
```

Syntax	Meaning (red means greedy)	Example
`*`	match 0 or more times	`$v =~ m/\w*/`
`+`	match 1 or more times	`$v =~ m/\w+/`
`?`	match 1 or 0 times	`$v =~ m/\w?/`
`{n}`	match exactly n times	`$v =~ m/\w{6}/`
`{n,}`	match at least n times	`$v =~ m/\w{50,}/`
`{n,m}`	match at least n, but not more than m times	`$v =~ m/\w{10,30}/`

Greediness in regex

-the quantifiers given in red on the previous slide are said to be “greedy”

- this means that they will ALWAYS attempt to match the given pattern as much as possible. (ie: they’re not satisfied with just one match, they’ll match’em all!)

```
$v = “Here is a sequence: aacctaggcctttacaacgggtta”;

if($v =~ m/Here is a (.*)/i)
{
  print “>> $1\n”;
}
```

This is what perl would print:

```
>> sequence: aacctaggcctttacaacgggtta
```

Adding ‘Fuzziness’ to your Search

-sometimes you’re not sure exactly what your pattern will look like

- ex: `aaattgcc` vs `atatagcc` ← same # of letters, different order

- You can tell perl to search for varying patterns

Syntax	Meaning	Example
`^`	Match at the beginning of the line	`$v =~ m/^at/i;`
`$`	Match at the end of the line (just before \n character)	`$v =~ m/$cc/i;`

`[ ]`	Match anything which contains the characters given within [ ]	`$v = 2008;`
		`$v =~ m/200[0-9]/`

`|`	Used to separate multiple string patterns	`$v =~ m/[aaa|ata]/i`

Grabbing text from a regex

```
$v = "We're going to party like it's 1999! (  Prince?)";
```

How perl sees this text:

```
We're\sgoing\sto\sparty\slike\sit's\s
1999!\s\(\s\sPrince\?\)\n
```

- If you only wanted to assign '1999' to a variable how could you do it?

- Perl allows you to "grab" just the text your interested in from a match to a regular expression. This is done using parantheses ( ).

```
$v =~ m/\D+(\d+)/g;
$newVar = $1;
```

collects ONLY 1999 and assigns it the unseen variable $1

```
#!/usr/bin/perl -w

$v = "We're going to party like it's 1999!
(  Prince?)\n";

print "\n$v";

if($v =~ m/\D+(\d+)/g)
{
  $newVar = $1;
  print "\nnewVar contains:  $newVar\n";
}

## don't use 'g' modifier here!
if($v =~ m/(\d+)/)
{
  $newVar2 = $1;
  print "\nnewVar2 contains: $newVar2\n";
}
exit;
```

Output

```
We're going to party like
it's 1999! (  Prince?)

newVar contains:  1999

newVar2 contains: 1999
```

```
$v = "We're going to party like it's 1999! (  Prince?)";
```

put into $1

put into $2

put into $3

```
$v =~ m/(.{5}) \w+\s\w+\s(\w+)\D+(\d+)/ig;
```

```
print "$1\n";

print "$2\n";

print "$3\n";
```

This would output this

```
We're

party

1999
```

Some websites on Regular Expressions

http://www.perldoc.com/perl5.6.1/pod/perlre.html

http://www.troubleshooters.com/codecorn/littperl/perlreg.htm

http://www.devshed.com/Server_Side/Administration/RegExp/page2.html

http://www.javaworld.com/javaworld/jw-07-2001/jw-0713-regex.html

Is everyone still with me?

Subroutines (aka: Functions)

What is a subroutine?

- it is a unique function that you create to perform some action
- they save you a lot of retyping
- they make for “neater” looking programs

Syntax:

```
sub subroutineNameOfYourChoice
{
   …block of code…
}
```

- Place your subroutines either at the beginning or end of a script.

Passing Arguments to a Subroutine

-when arguments (variables) are passed to a subroutine they are stored in the magical array called `@_`

- the individual arguments are accessed by the magical variable `$_[index]` (where index = 0,1,2,3…)

example: `$_[0], $_[1], $_[2]`…

-This is how you pass a variable to a subroutine you are calling from within the main body of your script:

```
subroutineName($var1, $var2);
```

Passing Arguments Continued…

-Once the arguments have been passed they now have to be assigned to NEW variables within the subroutine.

-There are two ways of assigning data to the subroutine variables:

1. Assign all of the variables in one shot:

```
sub subroutineName
{
  my($var1, $var2, $var3) = @_;
  ..rest of subroutine..
}
```

Now:

```
$var1 = $_[0];
$var2 = $_[1];
$var3 = $_[2];
```

2. Sequentially "pop off" each passed value from `@_` into a new subroutine variable.

```
#!/usr/bin/perl -w

my @ary1 = (1,2,3);
print "\n";
printThis($ary1[0], $ary1[1], $ary1[2]);

exit;

##========== Subroutines ============

sub printThis

{
  my $var1 = shift;
  my $var2 = shift;
  my $var3 = shift;

  print "\$var1 = $var1\n";
  print "\$var2 = $var2\n";
  print "\$var3 = $var3\n";

}
```

Output:

```

$var1 = 1
$var2 = 2
$var3 = 3
```

Returning Values from a Subroutine

```
#!/usr/bin/perl -w

my @ary1 = (1,2,3);

print "\n";
$returnedVal = printThis($ary1[0], $ary1[1], $ary1[2]);
print "The returned value is: $returnedVal\n";

exit;
##========== Subroutines ============

sub printThis
{
  my ($var1, $var2, $var3) = @_;

  print "\$var1 = $var1\n";
  print "\$var2 = $var2\n";
  print "\$var3 = $var3\n";

  $sumOfVars = $var1 + $var2 + $var3;

  return($sumOfVars);
}
```

```
$var1 = 1
$var2 = 2
$var3 = 3
The returned value is: 6
```

Localizing Variables

- You can make variables have a limited “range of influence” within your program by preceding their initial declaration with the word ‘my’

-red $var1 and green $var1 each have different values

```
#!/usr/bin/perl -w

my $var1 = "Apples";

print "\n\$var1 in the main body of\n";
print "script contains: $var1\n";
printOtherVar(); ## calls subroutine
print "\$var1 in main body again: $var1\n";

exit;

##========== Subroutines =================
sub printOtherVar
{
  my $var1 = 0.439871;

  print "\n\$var1 in 'printOtherVar()'\n";
  print "function contains: $var1\n";
}
```

Output:

```
$var1 in the main body of
script contains: Apples

$var1 in 'printOtherVar()`
$var1 in main body again: Apples
function contains: 0.439871
```

Top 10 Perl Websites

0	http://www.google.com/
1	http://www.perldoc.com/perl5.6.1/
2	http://perl.about.com/mbody.htm
3	http://www.perlfaq.com/
4	http://www.xav.com/perl/
5	http://www.cpan.org/
6	http://use.perl.org/
7	http://www.tek-tips.com/ (click on ‘Perl’)
8	http://www.bioperl.org/
9	The man pages! On a Unix prompt type: `man perlre`, or `man perlsub`.