Perl and Regular Expressions

Regular Expressions are available as part of the programming languages Java, JScript, Visual Basic and VBScript, JavaScript, C, C++, C#, elisp, Perl, Python, Ruby, PHP, sed, awk, and in many applications, such as editors, grep, egrep.

Regular Expressions help you master your data.

What is a Regular Expression?

Powerful.

Low level description:

Describes some text

Can use to:

Verify a user's input

Sift through large amounts of data

High level description:

Allow you to master your data

Regular Expressions as a language

Can consider regular expressions as a language

Made of two types of characters:

Literal characters

Normal text characters

Like words of the program

Metacharacters

The special characters + **? . * ^ $ ( ) [{ | **

Act as the grammar that combines with the words according to a set of rules to create and expression that communicates an idea

How to use a Regular Expression

How to make a regular expression as part of your program

What do they look like?

In Perl, a regular expression begins and ends with /, like this: /**abc**/

/**abc**/ matches the string "abc"

Are these literal characters or metacharacters?

Returns true if matches, so often use as condition in an if statement

Example: searching for "Course:"

Problem: want to print all lines in all input files that contain the string "Course:"

```
while ( <> ) {
 my $line = $_;
 if ( $line =~ /Course:/ ) {
 print $line;
 }
}
```

Or more concisely:

```
while ( <> ) {

 print if $_ =~ /Course:/;

}
```

The "match operator" =~

If just use /Course:/, this returns true if $_ contains the string Course:

If want to test another string variable **$var** to see if it contains the regular expression, use

$var =~ /regular expression/

Under what condition is this true?

The "match operator" =~ 2

```
# sets the string to be searched:

$_ = "perl for Win32";

# is 'perl' inside $_?

if ( $_ =~ /perl/ ) { print "Found perl\n" };



# Same as the regex above.

# Don't need the =~ as we are testing $_:

if ( /perl/ ) { print "Found perl\n" };
```

/i Matching without case sensitivity

```
$_ = "perl for Win32";

# this will fail because the case doesn't match:

if ( /PeRl/ ) { print "Found PeRl\n" };

# this will match, because there is an 'er' in 'perl':

if ( /er/ ) { print "Found er\n" };

# this will match, because there is an 'n3' in 'Win32':

if ( /n3/ ) { print "Found n3\n" };

# this will fail because the case doesn't match:

if ( /win32/ ) { print "Found win32\n" };


# This matches because the /i at the end means

# "match without case sensitivity":

if ( /win32/i ) { print "Found win32 (i)\n" };
```

Using !~ instead of =~

```
# Looking for a space:

print "Found!\n" if / /;

# both these are the same, but reversing the logic with

# unless and !~

print "Found!!\n" unless $_ !~ / /;
```

```
print "Found!!\n" unless ! / /;
```

Embedding variables in regexps

```
# Create two variables containing regular expressions

# to search for:

my $find = 32;

my $find2 = " for ";


if ( /$find/ ) { print "Found '$find'\n" };

if ( /$find2/ ) { print "Found '$find2'\n" };


# different way to do the above:

print "Found $find2\n" if /$find2/;
```

The Metacharacters

The funny characters

What they do

How to use them

Character Classes [...]

```
my @names = ( "Nick", "Albert", "Alex", "Pick" );
```

```
foreach my $name ( @names ) {
 if ( $name =~ /[NP]ick/ ) {
 print "$name: Out for a Pick Nick\n";
 else {
 print "$name is not Pick or Nick\n";
 }
}
```

Square brackets *match a single character*

Examples of use of [...]

Match a capital letter:
[ABCDEFGHIJKLMNOPQRSTUVWXYZ]

Same thing: **[A-Z]**

Match a vowel: **[aeiou]**

Match a letter or digit: **[A-Za-z0-9]**

Negated character class: [^...]

Match any single character that is *not* a letter: **[^A-Za-z]**

Match any character that is not a space or a tab: **[^ \t]**

Example using [^...]

This simple program prints only lines that contain characters that are not a space:

```
while ( <> )

{

 print $_ if /[^ ]/;

}
```

This prints lines that *start with* a character that is not a space:

```
while ( <> )

{

 print $_ if /^[^ ]/;

}
```

Notice that ^ has two meanings: one inside **[...]**, the other outside.

Shorthand for Common Character Classes

Since matching a digit is very common, Perl provides **\d** as a short way of writing
[0-9]

\D matches a non-digit: **[^0-9]**

\s matches any whitespace character; shorthand for **[\t\n\r\f]**

\S non-whitespace, **[^ \t\n\r\f]**

\w word character, **[a-zA-Z0-9_]**

\W non-word character, **[^ a-zA-Z0-9_]**

Matching any character

The dot matches any character except a newline

This matches any line with at least 5 characters:

print if /...../;

Matching the beginning or end

to match a line that contains exactly five characters:

print if /^.....$/;

the ^ matches the beginning of the line.

the $ matches at the end of the line

Matching Repetitions: * + ? {n,m}

To match zero or more:

/a*/ will match zero or more letter a, so matches "", "a", "aaaa", "qwereqwqwer", or the nothing in front of *anything*!

to match at least one:

/a+/ matches at least one "a"

/a?/ matches zero or one "a"

/a{3,5}/ matches between 3 and 5 "a"s.

Example using .*

```
$_ = 'Nick Urbanik <nicku@nicku.org>';

print "found something in <>\n" if /<.*>/;


# Find everything between quotes:
```

```
$_ = 'He said, "Hi there!", and then "What\'s up?"';

print "quoted!\n" if /"[^"]*"/;

print "too much!\n" if /".*"/;
```

Capturing the Match with (...)

Often want to scan large amounts of data, extracting important items

Use parentheses and regular expressions

Silly example of capturing an email address:

```
$_ = 'Nick Urbanik <nicku@nicku.org>';

print "found $1 in <>\n" if /<(.*)>/;
```

Capturing the match: greediness

Look at this example:

```
$_ = 'He said, "Hi there!", and then "What\'s up?"';

print "$1\n" if /"([^"]*)"/;

print "$1\n" if /"(.*)"/;
```

What will each print?

The first one works; the second one prints:
Hi there!", and then "What's up?

Why?

Because *, ?, +, {m,n} are *greedy*!

They match as much as they possibly can!

Being Stingy (not Greedy): ?

Usually greedy matching is what we want, but not always

How can we match as little as possible?

Put a ? after the quantifier:

***?** Match 0 or more times

+? Match 1 or more times

?? Match 0 or 1 time

{n,}? Match at least n times

{n,m}? Match at least n, but no more than m times

Being Less Greedy: Example

We can solve the problem we saw earlier using non-greedy matching:

```
$_ = 'He said, "Hi there!", and then "What\'s up?"';

print "$1\n" if /"([^"]*)"/;

print "$1\n" if /"(.*?)"/;
```

These both work, and match only
Hi there!

Sifting through large amounts of data

Imagine you need to create computing accounts for thousands of students

As input, you have data of the form:

Some heading on the top of each page

More headings with other content, including blank lines

A tab character separates the columns

123456789 H123456(1)

234567890 I234567(2)

345678901 J345678(3)

... ...

987654321 A123456(1)

Capturing the Match: (...)

```
# useradd() is a function defined elsewhere
# that creates a computer account with
# username as first parameter, password as
# the second parameter

while ( <> ) {
 if ( /^(\d{9})\t([A-Z]\d{6}\([\dA]\))/ ) {
 my $student_id = $1;
 my $hk_id = $2;
 useradd( $student_id, $hk_id );
}
```

The Substitution Operator s///

Sometimes want to replace one string with another (editing)

Example: want to replace Nicholas with Nick on input files:

```
while ( <> )
{
 $_ =~ s/Nicholas/Nick/;
 print $_;
}
```

Avoiding leaning toothpicks: /\/\/

Want to change a filename, edit the directory in the path from, say /usr/local/bin/filename to /usr/bin/filename

Could do like this:

s/\/usr\/local\/bin\//\/usr/\bin\//;

but this makes me dizzy!

We can do this instead:

s!\/usr/local/bin/!/usr/bin/!;

Can use any character instead of / in **s///**

For *matches*, can put **m//**, and use any char instead of /

Can also use parentheses or braces:

s{...}{...} or **m{...}**

Substitution and the /g modifier

If an input line contains:

Nicholas Urbanik read "Nicholas Nickleby"

then the output is:

Nick Urbanik read "Nicholas Nickleby"

How change all the Nicholas in one line?

Use the /**g** (global) modifier:

```
while ( <> )
{
 $_ =~ s/Nicholas/Nick/g;
 print $_;
}
```

Making regular expressions readable: /x modifier

Sometimes regular expressions can get long, and need comments inside so others (or you later!) understand

Use /x at the end of s///x or m//x

Allows white space, newlines, comments