Getting started with Perl

- Rather than looking at syntax, we'll study some "programming idioms" and see how to implement them in Perl

- An "idiom" is a "characteristic mode of expression", or a way of doing something ...

- Perl Idiom #1 - Processing Text Files

Our First Perl Script

```
#!/usr/bin/perl

while ( <> )
{
 print;
}
```

A Variation on our First Perl Script

```
#!/usr/bin/perl

while ( <> )
{
 # Note how the next line includes an
 # optional clause placing a condition
 # on the print command.

 print if /barryp/;
}
```

Regular Expressions - The Heart of Perl

- Regular Expressions are used as the basis of patterns in Perl
- Using a special notation, we state the pattern of text that we are interested in finding within our data, a process referred to as "pattern matching"
- Perl has four regular expression operators:
    - *alternation*, a choice, written as an in-fix |
    - *concatenation*, a collection, written as a series of characters
    - *repetition*, written as a post-fix *
    - *option*, written as a post-fix ?

Alternation: Making Choices

```
P|J|B
```

- We use the | symbol to indicate that we wish to match either the letter "P", "J", or "B"
- If we wanted either the pattern "PJ" or "B", we would write:

```
PJ|B
```

- "PJ" or "PB" can be matched by using brackets to bind the alternation:

```
P(J|B)
```

Concatenation: Matching Characters

- We have already seen this a couple of times:

```
PJ
```

- is a concatenation, as is:

```
barryp
```

- Concatenation is simply any combination of characters from a particular character set

- Concatenation binds more tightly than Alternation, so

```
Apple|Sun|Motorola
```

- is not the same as (and does not mean):

```
Appl(e|S)u(n|M)otorola
```

Repetition: Repeating Patterns

- It is often useful to match a repeating pattern, and we can do this in Perl using the * symbol:

```
x*
```

- matches an arbitrary number of x characters (zero or more)
- Note that * binds more tightly than alternation and concatenation so:

```
PJB*
```

- is not the same as (and does not mean):

```
(PJB)*
```

- Here's an interesting pattern, modified from Chapman's "Perl: The Programmer's Companion", page 13:

```
((Buy|Sell) (ten|twenty|fifty|a hundred) Eircom Shares!)*
```

Option: Maybe, Maybe Not

- Like *, options in Perl regular expressions are post-fix, and we use the ? character:

```
PJ?B
```

- will match PJB and PB

- The binding power of ? is equal to *, so it's greater than alternation and concatenation, so:

```
PJB?
```

- is not the same (and does not mean):

```
(PJB)?
```

Specifying Patterns

- When we take a regular expression and place it between two slash characters, we have a pattern:

```
/barryp/
```

- matches any line in our input that has the sequence of characters "barryp" in it

```
/bash/
```

- looks for the sequence "bash"

More (Powerful) Regular Expressions

- Perl provides various extensions to the notation seen so far:

```
A|E|I|O|U
```

- can be written as:

```
[AEIOU]
```

- This notation is refered to as a "character class"
- "Everything but" is represented ^ (i.e., inverse):

```
[^AEIOU]
```

More Character Classes

- "Ranges" are represented by -

```
[0-9]
```

- is the same as:

```
0|1|2|3|4|5|6|7|8|9
```

- We can combine character class and operators as follows:

```
[A-Za-z_][A-Za-z0-9_]*
```

- Note: in the previous example, [] *binds* just like () when using character classes, so we match multiple characters from the second character class (zero or more)

Special Character Classes

- Perl shorthand for frequently used character classes includes:

```
\d      a digit, i.e., [0-9]

\s      a "space" character, i.e., [ 
\n\r\t\f]

\w      a word character, i.e., [A-Za-z0-9_]

\D      is the inverse of \d

\S      is the inverse of \s

\W      is the inverse of \w
```

Shorthand Examples

- We could have written:

```
[A-Za-z_]\w*
```

- instead of:

```
[A-Za-z_][A-Za-z0-9_]*
```

- In Perl, There’s More Than One Way To Do It ... so, pick one that works for you!

```
[\d,]
```

- refers to any digit or a comma

Dot

- The full-stop (or period or dot) character has significiant meaning
- It represents all characters (except newline)

```
.*
```

- means any combination of characters which *does not include* newline
- Note: the "*" means "zero or more"
- Note: within [], the "." loses its special meaning, so:

```
[\w.]
```

- refers to any word character or a dot (full-stop/period)

More Shorthand (Examples)

```
\w\w*
```

- refers to one or more word characters, which is ok, but looks a little strange

- Again, in Perl, There's More Than One Way To Do It, so:

```
\w+
```

- is equivalent, and reads "at least one or more" word characters

- If we wanted to look for exactly 6 word characters we could write:

```
\w\w\w\w\w\w
```

- but we'd rather use:

Even More Shorthand (Examples)

- The following might be useful at GAA All-Ireland Finals:

```
((Hip! ){2}Hooray!){3}
```

- What do you think the following means?

```
[1-9]\d{2,4}
```

- Any number that matches from 100 - 99999
- If the second number is missing, it is taken to be infinity, so we have:

```
[1-9]\d{2,}
```

- which is 100 to a really big number!

Perl Metacharacters

- We hit a problem when we want to include a *metacharacter* in a pattern match
- The metacharacters we've seen so far include: [,],*,?,{,}, etc., etc.

```
/What is you name?/
```

- may not give us what we want, whereas:

```
/What is your name\?/
```

- will work as we expect it to
- This process is refered to as "escaping" the character

Escaping Characters

- Inside [], only ^, -, and] need to be escaped, as they have special meaning

- We need to be very careful with the "/" character - for example, we may try this while processing the /etc/passwd file on Linux:

```
print if /bin/bash/;
```

- which will screw-up - we should have used:

```
print if /bin\/bash/;
```

- For short examples, this is ok, but what if we were matching the following:

```
http://elmo.itcarlow.ie/booklist.html
```

The Match Operator

- We could write something like the following:

```
print if /http:\/\/elmo\.itcarlow\.ie\/booklist\.html/;
```

- which will work, but looks disgusting!

- Again, with Perl, There's More Than One Way To Do It, and by pre-fixing the pattern we wish to match with a "m" (the match operator) we can adjust the delimiting character, which is a "/" by default:

```
print if m!http://elmo\.itcarlow\.ie/booklist\.html!;
```

- or we could use any *bracket pairing*, for example:

```
print if m{http://elmo\.itcarlow\.ie/booklist\.html};
```

Shorthand For Certain Character Escapes

```
\t        tab
\n        newline (system-dependent)
\r        carriage return
\f        formfeed
\b        backspace (special case)
\a        alarm (bell)
\e        escape
\cx       control-x (x is any key)
\0xxx     character code xxx in octal
\xyy      character code yy in
hexidecimal
```

Matching Discrete Words

```
/bash/
```

- matches lines with "bash", "bashing", "bashed", "non-bash", etc., etc., which may or may not be what we want

```
/\bbash\b/
```

- matches just the word "bash", surrounded by an "empty string"
- Note that \b is not the same as \s in this context
- Here's a even better way to write the pattern:

```
/\b[Bb]ash\b/
```

- which matches "bash" and "Bash"

Matching At Start/End Of Lines

```
print if /^barryp/;
```

- will match if "barryp" is at the start of the line

```
print if /bash$/;
```

- will match if "bash" is at the end of the line
- As we learn more about Linux/UNIX, you will see that ^ and $ are used in this context elsewhere (for an example, review your *vi Quick Reference*)
- What about this pattern?

```
print if /^barryp.*bash$/;
```

If or Unless

- Using "if", we can indicate that we want to include a match, as we have already seen:

```
print if /^barryp.*bash$/;
```

- Using "unless" we can indicate that we want to include everything but the match:

```
print unless /^barryp.*bash$/;
```

- This use of unless can sometimes prove very handy indeed

Substitutions and Translations

- It's nice to be able to search text files for patterns
- It would be nicer if we could do something to the matched patterns once found
- Perl provides such a facility via Substitutions and Translations
- Substituting text with s:

```
while (<>)
{
 s/barryp/Paul Barry/;
 print;
}
```

- replaces "barryp" when matched with "Paul Barry"

Multiple Substitutions

- Simply place the substitutions on separate lines:

```
while (<>)
{
 s/barryp/Paul Barry/;
 s/kinsella/Austin Kinsella/;
 s/varleyj/Joe Varley/;
 print;
}
```

- Although this works, only the first occurrence of the matched pattern on each line is substituted
- To indicate that all occurrences on the line should be changed, use a post-fixed g:

```
s/barryp/Paul Barry/g
```

- The g stands for "global"

Referring to Matched Patterns

- It is sometimes useful to refer to whatever was found within the substituted string:

```
s/barryp/$& is the id for Paul
Barry/;
```

- will replace “barryp” with “barryp is the id for Paul Barry”
- $& is the *match variable*
- As this is Perl, There’s More Than One Way To Do It, so we can replace the rather cryptic $& with $MATCH which can be easier to read
- Note: to use $MATCH, your Perl script must state “use English;” near the top of the source file

More Than One Match

- What do you think the following does?

```
s/(\w+) and (\w+)/$2 and $1/;
print;
```

- Two matched words separated by the word “and” are reversed

- Here’s another variation:

```
s/(\w+) and \1/$1 twice/;
print;
```

- If the string “Barry and Barry” was matched, we would substitute “Barry twice” instead

- So, $1, $2, $3, and so on, refer to matches found

Translation

- Sometimes we want to *translate* characters instead of *substitute*, so we have the tr operator

```
tr/a-z/A-Z/
```

- Will convert ever lowercase letter into the UPPERCASE equivalent

- Here’s a very simple *rot13* translator:

```
while (<>)
{
    tr/A-Za-z/N-ZA-Mn-za-m/;
    print;
}
```

Translation Qualifiers

- If you append a “c” to the tr line, we *complement* the translation, i.e., it is applied to any character not in the string

```
tr/.;?!,: \t\n/x/c
```

- Replaces every character except those matched with the letter x
- *Squashing* is also possible with the “s” qualifier:

```
tr/ \t/ /s;
```

- “squashes” runs of spaces and tabs into a single space
- Deletion is performed by the “d” qualifier:

```
tr/0-9/0-7/d;
```

- will remove any 8’s and 9’s from the input stream

Filehandles

- So far, we have relied on Perl's default behaviour to process files:

```
while (<>)
{   # Do your processing here ...  }
```

- In actual fact, we are using the STDIN filehandle, which is automatically set up for us by the Perl environment
- Other standard filehandles exist: STDOUT, STDERR, and DATA
- And, of course, we can declare our own filehandles:

```
open MYFILE, 'data.txt';
while (<MYFILE>)
{
    print;
}
```

```
close MYFILE;
```

What's This "DATA" Thing?

```
while (<DATA>)
{
     print if /data/;
}
__END__
This is the data this program will use.
As we are using the DATA filehandle, Perl looks
to
the end of the script, represented by __END__,
and
starts reading data from there, i.e., after
__END__, as if it was an input file.
This can be really handy when testing a script.
We will use it a lot.
```