A Crash Course in Perl5

Part 2: Patterns

Zeegee Software Inc.
http://www.zeegee.com/

Terms and Conditions

Road map

- Basics
  - Introduction
  - Perl syntax
  - Basic data types
  - Basic operators
- Patterns
  - Introduction
  - String matching and modifying
  - Pattern variables
- Data structures
  - LISTs and arrays
  - Context
  - Hashes
- Flow control
  - Program structures
  - Subroutines
  - References
  - Error handling
- Data
  - Input and output
  - Binary data
  - Special variables
- Object-oriented programming
  - Modules
  - Objects
  - Inheritance
  - Tying

Patterns

Introduction

What are they?

- A way of matching a text string ($_ by default) against some template, usually for the purpose of...
  - Extracting information from the string
  - "Editing" portions of the string
- Perl patterns are **regular-expressions**, which have a well-established syntax in the Unix world: used by **sed, awk, grep, Emacs**, etc...

Integrated right into the Perl language itself

What do they look like?

- In Perl, patterns usually delimited by `//`.
- Any single character matches itself, unless it is a **metacharacter** with a special meaning...

Metacharacter(s)	Meaning
`^`	Match beginning of line/string
`$`	Match end of line/string
`.`	Match any single character (except newline)
`\`	Quote the next metacharacter
`[]`	Character class...
`()`	Grouping...
`|`	Alternation ("or")...
`*, +, ?, {}`	Quantifiers ("repeat")...

Some simple examples

Expression	Matches...
`/log/`	All strings that contain "log", like `clogged` or `logarithm` or just plain `log`
`/^log/`	All strings that start with "log", like `logical` or `logarithm` or just plain `log`
`/log$/`	All strings that end with "log", like `starlog`
`/^Log$/`	The string `Log` (case-sensitive)
`/^bi.$/`	Three-letter strings beginning with `bi`, like `bit`
`/^w.r./`	Strings of at least 4 characters, with `w` as the first character and `r` as the third, like `worst` or `warp`

Character classes

- If you enclose a list of characters inside `[]`, the construct will match any single character in the list:

`/^b[aei]t$/`	Matches strings `bat`, `bet`, or `bit`

- Starting it with `[^` negates the class:

`/^b[^aei]t$/`	Matches `bXt`, `b9t`, and `b!t`... but *not* `bat`, `bet`, or `bit`

- Within the list, a `-` defines a range:

`/^b[A-Z1-9]t$/`	Matches `bAt`, `bBt`, ..., `b1t`, ..., `b9t`

EXERCISES

Write a pattern which only matches strings...

1. ...that contain at least one lowercase vowel (*a*, e, *i*, *o*, or *u*).
2. ...that contain at least one lowercase letter which is *not* a vowel.
3. ...which are 2-digit hexidecimal numbers.
4. ...which are five letters, beginning and ending in either *E* or *e*.

Grouping

- Patterns are made up of smaller *subpatterns*, to be matched from left to right.
- The smallest pattern is a single character or metacharacter:

  `/^W.r./` A pattern made up of 5 subpatterns

- **Grouping** defines subpatterns within the pattern that consist of more than one character. It is done with parentheses ():

  `/^(W.r)./` A pattern made up of 3 subpatterns

Alternation

- You can have one of your subpatterns specify several possible alternatives to match against, using the | :

  `/^f(ee|i|o|um)$/` Matches `fee`, `fi`, `fo`, or `fum`

- If the outermost subpattern is an alternation, you can drop the parentheses:

  `/(^A|A$)/` Matches strings that start or end in A
  `/^A|A$/` Ditto

  `/^(ho|hum)$/` Matches "ho" or "hum"
  `/^ho|hum$/` WRONG! Not the same!

Quantifiers

- **Quantifiers** are metacharacters indicating that text matching the previous subpattern must/may be repeated a given number of times:

Quantifier	Example	Meaning
`*`	`a*`	Match 0 or more times
`+`	`a+`	Match 1 or more times
`?`	`a?`	Match 0 or 1 times
`{`*n*`}`	`a{3}`	Match exactly *n* times
`{`*n*`,}`	`a{3,}`	Match at least *n* times
`{`*n*`,`*m*`}`	`a{3,5}`	Match at least *n* but no more than *m* times

Examples with quantifiers

Expression	Matches...
`/^smo+/`	All strings that begin with "sm" followed by 1 or more o's, like `smother` or `smooth` or `smoooooooooch!`
`/ma*il/`	All strings that contain an "m" followed by 0 or more a's and then "il", like `mil` or `email` or `maaaaail`
`/^ma?il$/`	The strings `mil` and `mail`
`/[^o](oo)+$/`	All strings (except `oo`) that end with an even number of o's, like `moo` or `moooo` or `wooo-hoooooo`
`/^mo{1,3}n$/`	The strings `mon`, `moon`, and `mooon`
`/^(Mo{2,})+$/`	Any strings consisting of one or more reptitions of `/Mo{2,}/`, like `MooMoooMooooMooMoooooo`

Non-greedy quantifiers

- How would you match a C-style comment, like `/* this */`? Here's a typical first try:

```
/\/\*.*\*\//
```

NO!

- That seems to work... but the .* is **greedy**: it matches as many characters as possible, which can be a problem:

```
/* do it */  x = 1;  /* done */
```

- **Non-greedy** quantifiers match *as few characters as possible*. Just put a "?" after the * or + quantifier:

```
/\/\*.*?\*\//
```

Escape sequences

- Patterns are **processed like double-quoted strings**, so all the normal \-escapes work: `\n`, `\t`, `\0377`, `\L`, etc.
- Additionally, Perl defines the following:

Metacharacter	Meaning
`\w`	Match a "word" character (alphanumeric plus "_")
`\W`	Match a non-word character
`\s`	Match a whitespace character
`\S`	Match a non-whitespace character
`\d`	Match a digit character (0-9)
`\D`	Match a non-digit character

- All the above may be used inside character classes

Escaping the //

- Remember that the // which delimits a pattern can create problems for you, since you can't have a bare "/" inside the pattern!
- One way around this is to escape every "/" as "\/".
- When matching against URLs or Unix filenames, this can result in "leaning toothpick syndrome..."

```
# Match files in /usr/local/bin...
/^\/usr\/local\/bin\/.+$/
```

- Later on we'll see a better way to write this...

Zero-width assertions

- Perl also defines the following escape sequences for **zero-width assertions** (meaning they don't match actual characters):

Metacharacter	Meaning
`\b`	Match a word-boundary (between a `\w` and a `\W`) *(within character classes, matches a backspace)*
`\B`	Match a non-word-boundary
`\A`	Match only at beginning of string (not at lines)
`\Z`	Match only at end of string (not at lines)
`\G`	Match only where previous `m//g` left off

Interpolating variables

- Variable interpolation works, just like in "-strings:

  `/ab${subpat}cd$/` As if `$subpat` were actually there

- The pattern must be legal after interpolation... if the variable can contain metacharacters, use `\Q` and `\E`:

```
$unsafe = "*You* + me???";
```

  `/^\Q$unsafe\E$/` Safely matches against `$unsafe`

Interpolation will cause the pattern to be recompiled each time it's encountered... very time consuming! You might want to use the /o modifier...

Modifiers

- To alter the way pattern matching is performed, you can suffix the pattern with any/many of these modifiers:

Modifier	Meaning
`i`	Do case-**i**nsensitive pattern matching
`m`	Treat string as **m**ultiple lines
`o`	Only compile **o**nce, after first interpolation
`s`	Treat string as a **s**ingle line
`x`	Use e**x**tended regular expressions (whitespace and commas allowed!)

- Example: `/$warp/iom`
- Let's examine them in detail...

The /i modifier

- Putting /i after a pattern makes all matching case-insensitive:

`/^warp$/`	Matches `warp` only
`/^warp$/i`	Matches `warp` or `WARP` or `wArP`

- This even extends to variables which were interpolated to make the pattern:

```
$var = "WarP";
/^$var$/i
```

Matches `warp` or `WARP` or `wArP`

The /o modifier

Patterns which contain a variable interpolation are compiled each time they're encountered, since the variable might have changed. This is expensive, and may be needless!

```
$name = "Fred";
while (<STDIN>) {          # for each input line
   print "Match!" if /$name/o;
}
```

- Putting **/o** after a pattern means tells Perl to **only compile it once**, the first time it is encountered. This is *much* more efficient!

The /s modifier

- The **.** (dot) metacharacter never matches a newline unless you use the ***/s*** (*single-line*) modifier.
- This tells Perl to pretend that the string is a single line, so "dot" will match *any* character, *including* newline:

```
$_ = "does\n before\n precede\n after?\n";

if (/before.*after/)  { ... }  # fails
if (/before.*after/s) { ... }  # succeeds
```

The /m modifiers

- Normally, we think of a string as a single line.
- By default, ^ only matches at the beginning of a string, and $ at the end of a string (or before a newline at the end). Embedded newlines will *not* be matched.
- The **/m** (*multi-line*) modifier says that ^ should also match *after any newline*, and $ should also match *before any newline*.

```
$_ = "From: me\nSubject: Hi\nTo: you\n";
if (/^Subject: /m) {       # succeeds
   print "I found a Subject: line!\n";
}
```

EXERCISES

1. Write a pattern that matches any strings which break the rule "*i* before *e*, except after *c*".
2. Write a pattern which matches all strings that contain at least one double-letter; e.g., *book*, *crass*, *llama*, etc.
3. Write a pattern that matches strings which are FTP or HTTP URLs for the hostname in $host.
4. Same as above, but only match ones which point to files with a `.htm` or `.html` extension.

Patterns

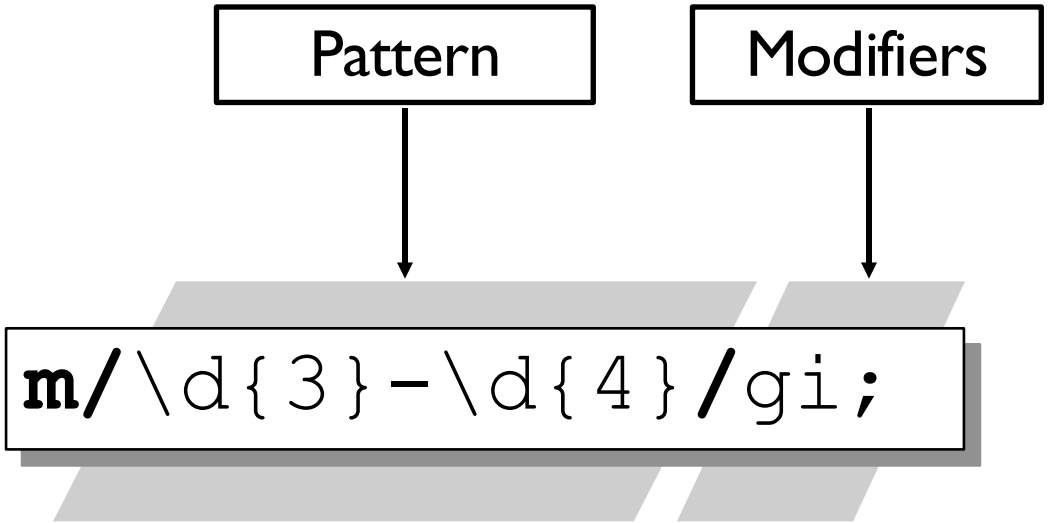
String match & modify

"Binding" operators

- Certain operations match or modify the scalar `$_` by default. The binary **binding operators** make them work on another string:
  - **Left argument** is the scalar to be matched/modified
  - **Right argument** is the pattern/substitution/translation
  - **Return value** indicates success of operation

`$a` **=~** `EXPR`	Indicates that you want to match/modify string *$a* against the given EXPRession
`$a` **!~** `EXPR`	Same, but the return value is logically negated

Anatomy of an m// command




```
m{\d{3}-\d{4}}gi;
```

Matching with m//

- To see if a scalar matches a given PATTERN, use the `m//` operator:

```
$blank = 1  if  $x =~ m/^\s*$/;
```

- The leading `m` is optional if using `//` as delimiters:

```
$blank = 1  if  $x =~ /^\s*$/;
```

- Remember, the default string is `$_`:

```
$blank = 1  if  /^\s*$/;
```

Extracting subpattern matches

- When using `m//`, text matching any ()-delimited subpatterns is placed in the variables `$1`, `$2`, ..., where each `$n` corresponds to the group beginning with the *n*th left-paren:

```
#  Is $phone of the form "912-867-5309"?
if ($phone =~ /^(\d{3})-(\d{3}-\d{4})$/) {
    ($areacode, $number) = ($1, $2);
}
```

- Since `m//` returns `($1,$2,..,$n)`, we can shorten that to:

```
($areacode, $number) =
    ($phone =~ /^(\d{3})-(\d{3}-\d{4})$/);
```

Options to m//

- To alter the way pattern matching is performed, you can suffix the m// pattern with any/many of these modifiers:

Modifier	Meaning
`i,m,o,s,x`	As previously discussed
`g`	Match globally; e.g., find all occurrences

- For example:

```
$MaxWarp = 9;          # assume this is a "constant"
…
if ($speed =~ /^warp $MaxWarp$/io) {
        # could be "WARP 9" or "wArP 9" or ...
}
```

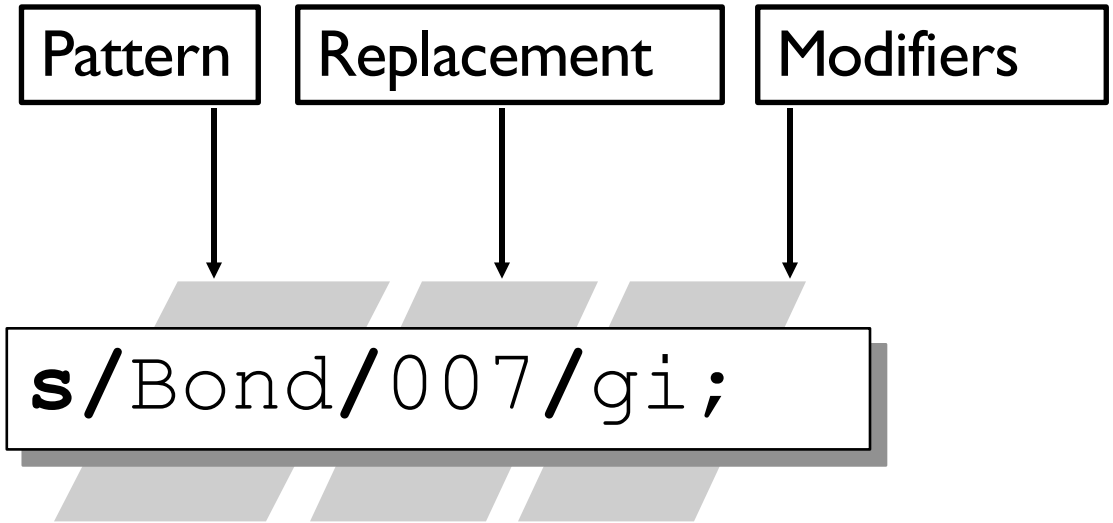
The m//g modifier

- Putting **/g** (*global*) after a m// pattern allows you to match inside a while loop, until done:

```
# Extract all integers from $_:
while (m/(\d+)/g) {
   print "found $1 before pos ", pos($_), "\n";
}
```

- Expression returns true if there was a match, false if not, and moves the "*match position*" along.
- You can use **pos()** to explicitly get/set the "match position" in the string being matched against.

Anatomy of an s/// command




```
s/Bond/007/gi;
```

```
s{Bond}{007}gi;
```

Search/replace with s///

- To modify a scalar, use the `s///` operator:

```
s/\s//                  # Remove first whitespace char
s/\s//g                 # Remove all whitespace chars
s/^\s+|\s+$//g          # Remove lead/trail whitespace

s/0/1/                  # Change first 0 to a 1
s/0/1/g                 # Change all 0s to 1s
s/\bteh\b/the/ig        # Fix all occurrences of word "teh"

s/Dr. (\S+)/$1, M.D./g;   # Turn "Dr. X" into "X, M.D."
```

- Returns number of substitutions made (true), or 0 (false) if no matches were found.

Using subpatterns in s///

- As in `m//`, text matching any ()-delimited subpattern of the pattern is placed in variables `$1`, `$2`, ..., where each `$`*n* corresponds to the group beginning with the *n*th left-paren.
- Those variables can be used the in the replacement to switch things around:

```
# Change "x loves y" to "y loves x"...
s/(\S+) loves (\S+)/$2 loves $1/g;
```

Options to s///

- To alter the way matching/replacement is performed, you can suffix the s/// pattern with any/many of these modifiers:

Modifier	Meaning
`i,m,o,s,x`	As previously discussed
`g`	Replace globally; e.g., find all occurrences
`e`	Evaluate replacement as expression

The s///g modifier

- Putting **/g** (*global*) after a s/// pattern-replace causes it to keep matching and replacing until no more matches remain:

```
# Replace the first digit with an X:
$_ = 'a-1 b-2 c-3';
s/\d/X/;        # $_ is now 'a-X b-2 c-3'
```

```
# Replace all digits with X's:
$_ = 'a-1 b-2 c-3';
s/\d/X/g;       # $_ is now 'a-X b-X c-X'
```

The s///e modifier

- Normally, right-hand-side is interpreted as a double-quoted string.
- Putting **/e** (*eval*) after a s/// pattern-replace causes the right-hand-side to be evaluated as a Perl expression (returning a string):

```
# Replace any integer i with i-times-2:
$_  = "2 + 3 = 5";
s/(\d+)/$1 * 2/ge;   # becomes "4 + 6 = 10"
```

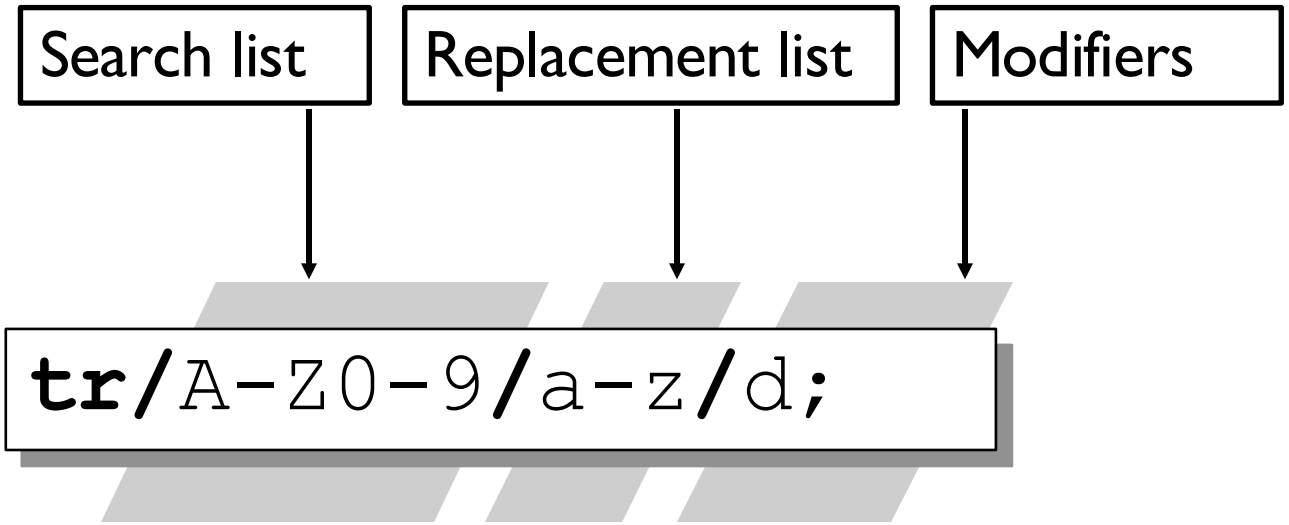
The s///x modifier

- Putting **/x** (e*Xtended*) after a s/// pattern-replace allows you to use whitespace and comments for readability:

```
# Delete C-style comments from $code:
$code =~ s{
    /\*      # Match opening /* delimiter
    .*?      # Match minimal no. of chars
    \*/      # Match closing */ delimiter
}{}gsx;
```

g = global search/replace
s = dot matches newline
x = allow comments/whitespace

Anatomy of a tr/// command




```
tr{A-Z0-9}{a-z}d;
```

Translation with tr///

- To translate or count characters, use the `tr///` operator (also called `y///`):

```
tr/A-Z/a-z/          # Convert to lowercase
tr/a-z/A-Z/          # Convert to uppercase
```

- If no replacement list, just counts the characters:

```
tr/*//               # Count stars
tr/*0-9//            # Count stars and digits
```

- Returns number of characters translated or counted.

tr/// is not s///!!!

- The search list in tr/// is *not* a regular expression pattern!

  You can use \ in the search list for special characters (like `\000`), and - to denote ranges (like `a-z`)... but that's it!

- The replacement list in tr/// does *not* interpolate!

```
tr/([a-z])/$stuff/;
```

The tr///c modifier

- A */c* *complements* the search list:

```
# Change anything alphabetic to a "?":
tr/a-zA-Z/?/;

# Change anything NOT alphabetic to a "?":
tr/a-zA-Z/?/c;
```

The tr///d modifier

- Normally, if replacement list is shorter than the search list, the last char of the replacement list is duplicated. However, a **/d** says to just *delete* anything not given:

```
# Downcase uppers, and replace digits with z's:
tr/A-Z0-9/a-z/;     # "THX-1138" -> "thx-zzzz"

# Downcase uppers, and delete digits:
tr/A-Z0-9/a-z/d;    # "THX-1138" -> "thx-"
```

- For the purposes of the returned value, deletion is regarded as translation, so both of the above return 7.

The tr///s modifier

- A */s* *squashes* duplicate matches:

```
# Change each digit to a single x:
tr/0-9/x/;        # "NCC-1701D" -> "NCC-xxxxD"

# Change runs of digits to a single x:
tr/0-9/x/s;       # "NCC-1701D" -> "NCC-xD"

# Change "yeeeeoooww" to "yeow":
tr/a-zA-Z//s;
```

Pattern variables

Patterns / Pattern-related variables

$1 ... $9

- As discussed already, each *$n* holds the text matched in group *n* of the last pattern match.
- We define group *n* as the one beginning with the *nth* left parenthesis.

```
/^(http|ftp)://([\w\.]+):(\d+)(/.*)$/
  1            2          3    4
```

- If you want to make use of $*n*, be sure to store it in a normal variable before you do your next pattern match!

Patterns / Pattern-related variables

$& ($MATCH)

- The string matched by the last successful pattern match.

```
if (m{\bHTTP://\S+}i) {
        print "Line contains a URL: $&\n";
}
```

- Read-only
- **Mnemonic** (*Larry's*)**:** like & in some editors
- **Mnemonic** (*mine*)**:** "...aaaaand, the match was...?"

$`/$' ($PREMATCH/$POSTMATCH)

- The strings preceding/following whatever was matched by the last successful pattern match.

```
$_ = 'abcdefghi';
/def/;
print "($`)($&)($')\n";
```

```
(abc)(def)(ghi)
```

- Read-only
- **Mnemonic:** ` precedes a quoted string, ' follows it