

More Regular Expressions

List vs. Scalar Context for m//

- Last week, we said that `m//` returns 'true' or 'false' in scalar context. (really, 1 or "").
- In list context, returns list of all the numeric variables that have been captured (`$1`, `$2`, `$3`, etc)
 - ie, the parts of the pattern enclosed in parentheses
 - `$1`, `$2`, `$3`, etc are still set
- If no capturing parentheses, returns (`1`)
- If `m//` doesn't match, returns ()
- `my @pieces = ('518-276-6000' =~ /^(\d+)-(\d+)-(\d+)$/);`
 - `@pieces` → (518, 276, 6000);

Modifiers

- following the final delimiter, you can place one or more modifier characters. Each one modifies the regular expression and/or the matching operator
- Most modifiers apply to both `m//` and the search part of a `s///`
- full list of modifiers on pages 150 and 153 of Camel
 - or see the various `perldocs` mentioned at end of these slides

/i Modifier

- **/i** → case insensitive matching.
- Ordinarily, **m/hello/** would not match 'Hello.'
- any character that has a concept of "case" will match that character in either case.
- **m/hello/i** will match any of 'hello', 'HELLO', 'Hello', 'HeLlO', etc.

/s Modifier

- **/s** → Treat string as a single line
- Ordinarily, the **.** wildcard matches any character except the newline
- If the **/s** modifier is provided, Perl will treat the string your RegExp is matching as a single line, and therefore the **.** wildcard will match **\n** characters as well.
- **"Foo\nbar\nbaz" =~ m/F(.*)z/;**
– Match fails
- **"Foo\nbar\nbaz" =~ m/F(.*)z/s;**
– Match succeeds - **\$1** → "oo\nbar\nba"

/m Modifier

- **/m** → Treat string as containing multiple lines
- As we saw last week, **^** and **\$** match "beginning of string" and "end of string" respectively.
- if **/m** provided, **^** will also match right after a **\n**, and **\$** will match right before a **\n**
 - in effect, they match the beginning or end of a "line" rather than a "string"
 - **print \$1 if "Hello.\nWhat's up?" =~ /^(W\w+)/m;**
 - prints "What"
- despite the mnemonics, **/m** and **/s** are *not* mutually exclusive.

/x Modifier

- **/x** → Allow formatting of pattern match
- Ordinarily, whitespace (tabs, newlines, spaces) inside of a regular expression will match themselves.
- with **/x**, you can use whitespace to format the pattern match to look better
- **m/\w+:(\w+):\d{3}/;**
 - word, colon, word, colon, 3 digits
- **m/\w+ : (\w+) : \d{3}/;**
 - word, space, colon, space, word, space, colon, space, 3 digits
- **m/\w+ : (\w+) : \d{3}/x;**
 - word, colon, word, colon, 3 digits

More /x Fun

- **/x** also allows you to place comments in your regexp
- Comment extends from **#** to end of line
- **m/** **#begin match**
 \w+ : **#word, then colon**
 (\w+) **#word, saved in \$1**
 :\d{3} **#colon, and 3 digits**
 /x **#end match**
- Do not put end-delimiter in your comment
- To match actual **#**, must precede with backslash

m//g in Scalar Context

- 'progressive' match
- Perl will remember where the last (successful) pattern match for this string left off, and continue from there
- **\$s = "abc def ghi";**
 for (1..3){
 print "\$1 " if \$s =~ /(\w+)/;
 }
 - abc abc abc
- **for (1..3){**
 print "\$1 " if \$s =~ /(\w+)/g;
 }
 - abc def ghi
- Without knowing how many matches in advance...
- **while (\$s =~ /(\w+)/g){**
 print "\$1 ";
 }

m//g in List Context

- if there are no capturing parentheses, return all occurrences of match in the string
- **my \$nums = '1-518-276-6505';**
- **my @nums = \$nums =~ m/\d+/g;**
 - @nums → ('1', '518', '276', '6505')
- if there are any capturing parentheses, return all occurrences of those sub-matches only
 - **my \$string = 'ABC&123 DEF GHI%789';**
 - **my @foo = \$string =~ /([A-Z]+).\d+/g;**
 - @foo → ('ABC', '123', 'GHI', '789')
 - \$1 → 'GHI', \$2 → '789'

m// return value summary

	m//		m//g	
	no parens	parens	no parens	parens
list	(1)	captured submatches	all occurrences of entire match	all occurrences of submatches
scalar	1		1 (progressive match)	

/c Modifier (for m//)

- **/c** → continue progressive match
- Used only in conjunction with **/g**
- When **m//g** finally fails, if **/c** used, don't reset position pointer
- **\$s = "Billy Bob Daisy";**
- **print "\$1 " while \$s =~ /(B\w+)/g;**
 - Billy Bob
- **print \$1 if \$s =~ /(\w+i\w+)/g;**
 - Billy
- **print "\$1 " while \$s =~ /(B\w+)/gc;**
 - Billy Bob
- **print \$1 if \$s =~ /(\w+i\w+)/gc;**
 - Daisy

/g Modifier (s/// version)

- /g → global replacement
- Ordinarily, only replaces first instance of PATTERN with REPLACEMENT
- with /g, replace all instances at once.
- \$a = '\$a / has / many / slashes /';
\$a =~ s#/#\\#g;
- \$a now → '\$a \ has \ many \ slashes \'

Return Value of s///

- Regardless of context, s/// always returns the number of times it successfully search-and-replaced
- If search fails, didn't succeed at all, so returns 0, which is equivalent to false
- unless /g modifier is used, s/// will always return 0 or 1.
- with /g, returns total number of search-and-replaces it did

/e Modifier (s/// only)

- /e → Evaluate Perl code in replacement
- REPLACEMENT is no longer a simple string. Instead, it is the return value of a piece of Perl code.
- s/(\d+)/\$1 + 10/ge;
– search for all integers in string. Replace each one with 10 greater.
- "42 100 18" becomes "52 110 28"
- without /e, would become "42 + 10 100 + 10 18 + 10"

Modifier notes

- Modifiers can be used alone, or with any other modifiers.
- Order of multiple modifiers has no effect
- **s/\$a .*? c/\$b/gixs;**
 - search \$a, ignoring case, for all instances of \$a, followed by anything – including a newline – followed by 'c'. Replace each instance with \$b.

Pre-compiling a Regexp

- It may be useful to break up a regexp into component parts.
- The **qr//** operator is used to compile a regexp for later use
- **my \$capWord = qr/[A-Z][a-z]*/;**
- **my \$word = qr/[a-z]+/;**
- **\$stm =~ /^\$capWord(?:\s\$word)+\.\$/**
- If any modifiers are used, apply for that part only:
 - **my \$search = qr/hello/i;**
 - **\$x =~ /word: \$search/**
 - matches "word: hello", "word: HeLLo", but not "WORD: hello"

A Bit More on Capturing

- So far, we know that after a pattern match, \$1, \$2, etc contain sub-matches.
- What if we want to use the sub-matches while still in the pattern match?
- If we're in the replacement part of **s///**, no problem – go ahead and use them:
- **s/(\w+) (\w+)/\$2 \$1/; # swap two words**
- if still in match, however....

Capturing Within Pattern

- to find another copy of something you've already matched, you cannot use \$1, \$2, etc...
 - operation passed to variable interpolation *first*, then to regexp parser
- instead, use \1, \2, \3, etc...
 - "back references"
- `m/(\w+) .* \1/;`
- Find a word, followed by a space, followed by anything, followed by a space, followed by that same word.

Look(ahead|behind)

- Four operations let you "peek" into other parts of the pattern match without actually trying to match.
- Positive lookahead: `(?=PATTERN)`
- Negative lookahead: `(?!PATTERN)`
- Positive lookbehind: `(?<=PATTERN)`
- Negative lookbehind: `(?<!PATTERN)`

Positive lookahead

- We want to remove duplicate words from a string:
 - "Have you seen this this movie?"
- Could try:
 - `s/(\w+)\s\1/$1/g;`
 - This won't work for everything. Why not?
 - Hint: "what about this this this string?"

Lookaheads to the rescue

- The problem is that the regular expression is "eating up" too much of the string.
- We instead just want to check if a duplicate word exists, but not actually match it.
- Instead of checking for a pair of duplicate words and replacing with first instance, delete any word if it is followed by a duplicate
- `s/(\w+) \s (?= \1)//gx;`
- "Search for any word (and save it) followed by a space, then *check to see* if it's followed by the same word, and replace the word and space with nothing"

Negative Lookahead

- `(?!PATTERN)`
- Same concept. This time, *check to see* if PATTERN does NOT come next in the string.
- Replace all peanuts with almonds (but not almond butter, it's yucky)
- `s/peanut(?! butter)/almond/g;`

Lookbehind

- Positive: `(?<=PATTERN)`
- Negative: `(?<!PATTERN)`
- Same concept as look-ahead. This time, ensure that PATTERN did or did not occur *before* current position.
- ex: `s/(?<!c)ei/ie/g;`
 - Search string for all "ei" not preceded by a 'c' and replace with "ie"
 - "i before e except after c"
- NOTE: only 'fixed-length' assertions can be used for look-behind (for example, `c*` doesn't work)

Find specific patterns

- Last week, we looked for three-digit numbers:
- `/(?:^\d|\D)\d{3}(?:\D|$)/`
 - search for (beginning of string OR non-digit), followed by 3 digits, followed by (non-digit OR end of string)
- With look-arounds, slightly easier:
- `/(?!\d)\d{3}(?!\d)/`
 - Search for three digits that are not preceded by a digit and not followed by a digit.
- Look for a word in the middle of a sentence (ie, spaces on either side)
 - `/(?<=\s)(\w+)(?=\s)/`

Transliteration Operator

- `tr///` → does not use regular expressions.
 - Probably shouldn't be in RegExp section of book
 - Authors couldn't find a better place for it.
 - Neither can I
- `tr///` does, however, use the binding operators `=~` and `!~`
- formally:
- `tr/SEARCHLIST/REPLACEMENTLIST/;`
 - search for characters in SEARCHLIST, replace with corresponding characters in REPLACEMENTLIST

What to Search, What to Replace?

- Much like character classes (from last week), `tr///` takes a list or range of characters.
- `tr/a-z/A-Z/;`
 - replace any lowercase characters with corresponding capital character.
- TAKE NOTE: SearchList and ReplacementList are NOT REGULAR EXPRESSIONS
 - attempting to use RegExps here will give you errors
- Also, no variable interpolation is done in either list

tr// Notes

- In either context, **tr///** returns the number of characters it modified.
- if no binding string given, **tr///** operates on **\$_**, just like **m//** and **s///**
- **tr///** has an alias, **y///**. It's depreciated, but you may see it in old code.

tr// Notes

- if Replacement list is shorter than Search list, final character repeated until it's long enough
 - **tr/a-z/A-N/;**
 - replace a-m with A-M.
 - replace n-z with N
- if Replacement list is null, repeat Search list
 - unless **/d** is used (see next slide)
 - useful to count characters, or squash with **/s**
- if Search list is shorter than Replacement list, ignore 'extra' characters in Replacement

tr// Modifiers

- **/c** → Compliment the search list
 - 'real' search list contains all characters *not* in given searchlist
- **/d** → Delete characters with no corresponding characters in the replacement
 - **tr/a-z/A-N/d;**
 - replace a-n with A-N. Delete o-z.
- **/s** → Squash duplicate replaced characters
 - sequences of characters replaced by same character are 'squashed' to single instance of character

`tr///` vs `s///`

- `tr///` does not use `[]`
- `s///` does not allow character classes (or any special regular expression formation) within the replacement
- `tr/[a-z]/[A-Z]/; #WRONG`
- `s/[0-9]*/[a-j]*/; #WRONG`

Where to get help

- `perldoc perlrequick`
 - Quick Start guide to RegExps
- `perldoc perlretut`
 - Tutorial for RegExps
- `perldoc perlref`
 - Reference Guide to RegExps
- `perldoc perlop`
 - `m//`, `s///`, and `??` in "Regex Quote-Like Operators"
- `perldoc perlre`
 - Full Guide to Regular Expressions

Enough!

- I ***strongly*** suggest you take the time to understand all of these regular expressions.
- Homework 2 due Wednesday. Homework 3 will be given out next week. It will require more complicated RegExps than HW2.
- Any questions about any of this material, ***please*** email us (`perl_TA@cs.rpi.edu`)
