**Perl Part 3 Pattern Matching**

|  |  |
| --- | --- |
| Pattern Matching in Unix is based on regular expressions. There are several different regular expression engines causing differences in implementations. Perl uses a very powerful regex capability. Python's regex is very similar. | Perl has a powerful pattern matching capability:  **matching** match values and patterns  **substituting** substitute matched patterns with other values  **tokenize** pluck one or more values based on pattern matching |
| **Perl Pattern Matching**  **Matching:**  *string =~* m/*pattern*/ returns true if the pattern matched. Note that the "m" is optional.  *string !~* m/*pattern*/ returns true if the pattern was **not** matched. | **Example 3-1:** simple matches  $ vi example 31  #!/usr/bin/perl -w  # look for "four" any where in a string.  my $string1 = "John plays four square";  my $string2 = "four score and more";  if ($string1 =~ m/four/) {  print "'$string1' contains 'four'\n";  }  print "no match in '$string2' for 'eighty'\n"  if $string2 !~ /eighty/;  $ perl example31  'John plays four square' contains 'four'  no match in 'four score and more' for 'eighty' |
| **Substituting**  *string* =~ s/*pattern*/*replacement*/ substitutes once and returns 1 if it substituted  *string* =~ s/*pattern*/*replacement*/g substitutes globally (all occurrences) and returns the number of substitutions | **Example 3-2:** simple substitutions  $ vi example32  my $eight = "eight men lifted weights";  $string1 = $eight;  $res = $string1 =~ s/eight/seven/;  print "$res substitutions: '$string1' \n";  $string1 = $eight;  $res = $string1 =~ s/eight/seven/g;  print "$res substitutions: '$string1' \n";  $ perl example32  1 substitutions: 'seven men lifted weights'  2 substitutions: 'seven men lifted wsevens' |
| **Special Characters in Regular Expressions**  ^Anchors the match to the beginning of the string. Different meaning within [].  **\d** Matches any numeric digit 0 thru 9  **\D** Matches any character that is **not** a numeric digit  **\w** Matches any letter, numeric digit or underscore  **\W** Matches any character that is **not** a letter, numeric digit or underscore  **\s** Matches a space, tab or newline character  **\S** Matches any character that is **not** a space, tab or newline character  **. A dot** matches any character other than a newline character  **(*ex*)** Parentheses around an expression defines a **group**. The first matched group replaces $1, second matched group replaces $2, and so on. Groups are a powerful tokenizing mechanism.  **[*values*]** matches any character listed between the brackets  **[^*values*]** matches any character that is **not** listed between the brackets | **Example 3-3:** simple substitutions  $ vi example33  # define a pattern to match a abc123 id and use a group  # to get it.  my $string1 = "My ID is xyz123.";  if ($string1 =~ /([a-z][a-z][a-z]\d\d\d)/) {  print "ABC123 ID is $1\n";  }  # define a regular expression which matches Ho Ho Ho with  # any of these characters in between the words:  # comma or exclamation point  # It also has white space between those characters  # and the next Ho.  # The last Ho must be immediately followed by a !  my $string2 = 'Santa said, "Ho! Ho! Ho!"';  if ($string2 =~ /Ho[,!]\sHo[,!]\sHo!/) {  print "We found our real Santa\n";  } else {  print "Another fake ho ho ho'er\n";  }  $ perl example33  ABC123 ID is xyz123  We found our real Santa |
| **Special Characters Applied to a Preceding Expression**  The following special characters are applied to a preceding expression:  **?** matches **zero** or **one** of the preceding expression  **\*** matches **zero** or **more** of the preceding expression  **+** matches **one** or **more** of the preceding expression  **{*n*}** matches **exactly *n*** of the preceding expression  **{*n,*}** matches ***n* or more** of the preceding expression  **{*,m*}** matches ***0* to *m*** of the preceding expression  **{*n,m*}** matches **at least *n* and at most *m*** of the preceding expression  **$** preceding expression must match at the end of the string  **| *ex*** matches the preceding expression or the following one | **Example 3-4:** alternate values and optional groups  $ vi example34  #!/usr/bin/perl -w  # define a regex that matches Mickey Mouse or Minnie Mouse  # Mouse is optional  $one = "Goofy yelled at Mickey Mouse";  $res = $one =~ /(Mickey|Minnie)\s\*(Mouse)?/;  print "Result: $res, first: '$1', second: '$2'\n";  $two = "Pluto licked Minnie's hand";  $res = $two =~ /(Mickey|Minnie)\s\*(Mouse)?/;  print "Result: $res, first: '$1', second: '$2'\n";  $ perl example34  Result: 1, first: 'Mickey', second: 'Mouse'  Use of uninitialized value $2 in concatenation (.) or string at example34 line 10.  Result: 1, first: 'Minnie', second: ''  Why did $two match the pattern even though it didn't match "Mouse"?  Because the ? says it has to match 0 or 1 times so 0 is valid.  **Example 3-5:** Assigning group values into variables  $ vi example35  #!/usr/bin/perl -w  # define a regex that matches Mickey Mouse or Minnie Mouse  # Mouse is optional  my $first;  my $last;  $one = "Goofy yelled at Mickey Mouse";  ($first, $last) = $one =~ /(Mickey|Minnie)\s\*(Mouse)?/;  print "first: '$first'" if defined($first);  print " second: '$last'" if defined( $last);  print "\n";  $two = "Pluto licked Minnie's hand";  ($first, $last) = $two =~ /(Mickey|Minnie)\s\*(Mouse)?/;  print "first: '$first'" if defined($first);  print " second: '$last'" if defined( $last);  print "\n";  $ perl example35  first: 'Mickey' second: 'Mouse'  first: 'Minnie' |
| **Values that Must be Escaped to be Matched**  There are several symbols that have special meanings to regular expressions. To match the literal values, these must be escaped:  **\.** matches a period  **\\** matches a backslash  **\(** matches a left parenthesis  **\)** matches a right parenthesis  **\[** matches a left bracket  **\]** matches a right bracket  **\{** matches a left curly brace  **\}** matches a right curly brace  **\^** matches a carat  **\$** matches a dollar sign  **\|** matches a vertical bar  **\+** matches a plus sign  **\\*** matches an asterisk | **Example 3-6:** regex for a phone number  $ vi example36  #!/usr/bin/perl -w  # define a regular expression for a phone number (999)-999-9999  # saving the area code and number  # Note than the parentheses highlighted in blue are for groups  $text1 = "Please call me at (210)-555-1234";  ($area, $phone) = $text1 =~ /\((\d{3})\)-(\d{3}-\d{4})/;  if (defined ($phone)) {  print "Area Code= $area, Phone= $phone\n";  }  $ perl example36  Area Code= 210 Phone= 555-1234 |
| **Tokenizing:**  *string =~* m/*patternWithParens*/ if the specified pattern matches, it sets special variables named $1, $2, $3, …  (*variableList*) = *string =~* m/*patternWithParens*/  if the specified pattern matches, it sets the variables in the *variableList* with the values between parentheses.  Also see the **split** function for more examples of tokenizing. | **Example 3-7:**  In these examples, assume string is "we went home"   |  |  |  |  | | --- | --- | --- | --- | | **Match Pattern** | **Explanation** | **Matched** | **$1** | | m/we/ | Looks for 'we' | 1 | n/a | | m/(we)/ | Looks for we, if found populates $1 | 1 | 'we' | | m/(we.\*)t/ | Looks for 'we', followed by zero or more of any character, and followed by a 't'. | 1 | 'we wen' | | m/ (we.\*)t/ | Looks for blank, followed by 'we', followed by zero or more of any character, and followed by a 't'. | 1 | 'wen' | | m/([^aeiou]{3})/ | Looks for three adjacent non-vowels | 1 | 'nt ' | |  |  |  |  | |
| **Exercise:** same as example 3-6, but remove the "-' from the phone.  There are many solutions. | #!/usr/bin/perl -w  $text1 = "Please call me at (210)-555-1234";  # 1. Use substr to remove the dash  ($area, $phone) = $text1 =~ /\((\d{3})\)-(\d{3}-\d{4})/;  if (defined($phone)) {  $phone = substr($phone,0,3) . substr($phone,4)  print “Area Code= $area, Phone= $phone\n”; }  # 2. Don't include the dash originally  my ($phonep1, $phonep2);  ($area, $phone1, $phone2) = $text1 =~ /\((\d{3})\)-(\d{3})-(\d{4})/; if (defined($phone2)) {  $phone = $phone1 . $phone2  print “Area Code= $area, Phone= $phone”; } |
| **Exercise:** input line contains:  first name, last name and major separated by commas.  Parsing the input line to get $first, $last, $major using:  #1 split  #2 pattern matching  After completing those, how would you also allow optional white space after the commas, but not include the white space in our variables?  ??  ?? | $ vi Exer2.pl  #!/usr/bin/perl -w  print "What is your first name, last name, and major\n",  "(separate with commas):";  defined($\_ = <STDIN>)  or die "input error from STDIN (likely EOF)\n";  my ($first, $last, $major);  chomp $\_; # remove the \n  # 1. Using split  ??  # 2. Using pattern matching  ??  $ perl exer2.pl  What is your first name, last name, and major  (separate with commas):bob,wire,AGR  Using split: 'bob', 'wire', 'AGR'  Using matching: 'bob', 'wire', 'AGR' |
| **Greedy vs Nongreedy Matches**  Perl uses a greedy matching mode for multi-occurring patterns unless told to be nongreedy. With **greedy matching mode**, it tries to match as much text as possible.  In the first example on the right, our pattern is matching anything followed by a space. The second group then matches anything.  Because the ".\*\s" in the first group is greedy, it matched  "hello there " instead of "hello ".  Placing a ? after a multi-occurring pattern, causes **re** to use a nongreedy matching mode. This is not the same meaning of ? as above. With **nongreedy matching mode**, it tries to match as little text as possible (but it **begins with the left side if possible**). | **Example 3-8:** Some examples using greedy and non-greedy matches  $str = "hello there world";   |  |  |  | | --- | --- | --- | | **pattern** | **$1** | **$2** | | m/(.\*\s)(.\*)/ | 'hello there ' | 'world' | | m/(.\*?\s)(.\*)/ | 'hello ' | 'there world' |   $str = "Ho! Ho! Ho! Ho! Ho! ";   |  |  | | --- | --- | | **pattern** | **$1** | | m/((Ho!\s){2,5})/ | 'Ho! Ho! Ho! Ho! Ho! ' | | m/((Ho!\s){2,5}?)/ | 'Ho! Ho! ' | |
|  |  |
| **Exercise: fill in the chart** | |  |  |  |  | | --- | --- | --- | --- | | **#** | **String** | **Pattern** | **$1** | | 1 | xaaab | /(xa\*)/ |  | | 2 | xaaab | /(xa\*?)/ |  | | 3 | xaaab | /(xa\*?)b/ |  | | 4 | xabxaab | /(x.\*)b/ |  | | 5 | xabxaab | /(x.\*?)b/ |  | | 6 | xabxaab | /(xa\*)b/ |  | |
| **Pattern Modifiers**  **/g** matches multiple occurrences. This is used with substitution.  **/i** Ignore case  **/m** Causes ^ and $ to match newlines inside your string. This is useful when multiple text lines are in a variable and you want to have ^ apply to multiple locations not just the beginning of the string. This is used with substitution.  $one = "Goofy yelled at Mickey Mouse";  ($first, $last) = $one =~ /(mickey|minnie)\s\*(mouse)?/i;  print "first: '$first'" if defined($first);  print " second: '$last'" if defined( $last);  print "\n"; | **Example 3-9: pattern modifiers**  $ vi example39  #!/usr/bin/perl -w  my $str = "Although Unix is well known today, Unix was not known\n"  . "outside of technical circles in the 1980s. I worked\n"  . "for a company that was hiring technical writers in\n"  . "the mid 1980s. An interviewer asked a candidate, who\n"  . "was a high school English teacher, if she knew Unix.\n";  my $s1 = $str;  # Substitue every occ of Unix with Linux  $s1 =~ s/Unix/Linux/g;  print "s1: $s1\n";  # Put "Fun- " at the front of each line.  my $s2 = $str;  $s2 =~ s/^/Fun- /gm;  print "s2: $s2\n";  # Put " -End" at the end of each line  my $s3 = $str;  $s3 =~ s/$/ -End/gm;  print "s3: $s3";  $ perl example39  s1: Although Linux is well known today, Linux was not known  outside of technical circles in the 1980s. I worked  for a company that was hiring technical writers in  the mid 1980s. An interviewer asked a candidate, who  was a high school English teacher, if she knew Linux.  s2: Fun- Although Unix is well known today, Unix was not known  Fun- outside of technical circles in the 1980s. I worked  Fun- for a company that was hiring technical writers in  Fun- the mid 1980s. An interviewer asked a candidate, who  Fun- was a high school English teacher, if she knew Unix.  s3: Although Unix is well known today, Unix was not known -End  outside of technical circles in the 1980s. I worked -End  for a company that was hiring technical writers in -End  the mid 1980s. An interviewer asked a candidate, who -End  was a high school English teacher, if she knew Unix. -End |
| **Back References**  It may be necessary within a pattern to reference something that was already matched. This is when we use back references: \*k* where *k* is an integer. This is similar to $*k* for group values, but \*k* can be referenced within the pattern.  In the example to the right, we want to match lines in code where the variable on the left side of an assignment is used **immediately** on the right side.  dTotal = dTotal + iQty \* dUnitPrice;// match  dTot = dTotal \* 2; // don’t match  Why doesn't this pattern work correctly?  /^\s\*(\w\*)\s\*=\s\*\1/  ??.  Does this one solve the problem above?  /^\s\*(\w\*)\s\*=\s\*\1\W/  ??    Are there other problems? Suppose we want to match it anywhere on the right side.  dTotal = iQty \* dUnitPrice + dTotal;// match  What would you use for the pattern?  ?? | **Example 3-10: back references**  $ vi example310  #!/usr/bin/perl -w  my $str1 = " dTotal = dTotal + iQty \* dUnitPrice;"  my $str2 = "dTot = dTotal \*2;  if ($str1 =~ m/^\s\*(\w\*)\s\*=\s\*\1/ ){  print "1. '$1'\n";  }  if ($str2 =~ m/^\s\*(\w\*)\s\*=\s\*\1/ ){  print "2. '$1'\n";  }  if ($str1 =~ m/^\s\*(\w\*)\s\*=\s\*\1\W/ ){  print "3. '$1'\n";  }  if ($str2 =~ m/^\s\*(\w\*)\s\*=\s\*\1\W/ ){  print "4. '$1'\n";  }  $ perl example310  1. 'dTotal'  2. 'dTot'  3. 'dTotal' |
| **Creating a Pattern Which Uses Variable References**  It is sometimes necessary to use a pattern that is in a variable. Suppose a user provides a pattern that is used against a list of courses. Instead of hard-coding the pattern, it can come from a variable.  Some implementations provide **qr//** to define a regular expression that can be compiled so that it doesn't have to be completely re-compiled on every use. | **Example 3-11: variable references**  $ vi example311  my $query = "CS3..3";  my @courses = ("CS2123", "CS2233", "CS3333", "CS3343",  "CS3341", "CS3423", "CS3421", "CS3723", "CS3843",  "CS3841");  foreach $course (@courses) {  if ($course =~ /$query/){  print "$course ";  }  }  print "\n";  $ perl example311  CS3333 CS3343 CS3423 CS3723 CS3843  **Example 3-12: qr//**  $ vi example312  my $query = qr/CS3..3/;  my @courses = ("CS2123", "CS2233", "CS3333", "CS3343",  "CS3341", "CS3423", "CS3421", "CS3723", "CS3843",  "CS3841");  foreach $course (@courses) {  if ($course =~ /$query/){  print "$course ";  }  }  print "\n";  $ perl example312  CS3333 CS3343 CS3423 CS3723 CS3843 |
| There are websites which can help you properly construct regex expressions. | Try regex101.com |

**©2017 Larry W. Clark,** UTSA CS students may make copies for their personal use