



The Kalmanovitz Library and
The Center for Knowledge Management

Regular Expressions in Perl

Gilberto da Gente
Bioinformatics Resource Specialist
Tuesday, April 8th 2008
University of California, San Francisco



UCSF

Regular Expressions

- "A *regular expression*, often called a *pattern* in Perl, is a template that either matches or doesn't match a given string."
- By default, regular expressions are strings that are bounded by slashes, e.g., `/cat/`
- By default, the string that will be searched is `$_`
- The delimiter can be changed to virtually any nonalphanumeric character by preceding the first occurrence of the new delimiter with an `m`, e.g., `m#cat#`.

Literal Patterns

- The simplest form of a pattern is a literal string. Thus, one can search for `/cat/`, as shown in the previous slide
- Normally, such an expression would appear in some conditional context, such as an if statement.

```
if (/cat/) {  
    print "cat found in $_\n";  
}
```

Special Characters

- The period (.) stands for any single character except a new line.

```
/.at/ # matches "cat," "bat", but not "at"
```

- An explicit *category* or *class* of characters can be specified by placing the characters in square brackets(`[]`).

```
/[0123456789]/
```

- Ranges of characters can also be specified:

```
/[0-9]/  
/[a-z]/  
/[A-Z]/  
/[0-9a-zA-Z]/
```

Special Characters

- Several predefined categories are available

<code>\d</code>	# digits
<code>\w</code>	# words
<code>\s</code>	# space
<code>\D</code>	# not digits
<code>\W</code>	# not words
<code>\S</code>	# not space

<code>\t</code>	#tab
<code>\n</code>	#newline
<code>\l</code>	#lowercase next char
<code>\u</code>	#uppercase next char
<code>\L</code>	#lowercase till end
<code>\U</code>	#uppercase till end

- Any character or range can be turned into a not condition by placing a carat (`^`) in front of it.

<code>/[^0-9]/</code>	# not a digit
-----------------------	---------------

Multipliers

- The question mark (?) indicates zero or one of the preceding character.

`/a?t/` # zero or one a followed by t

- An Asterisk (*) indicates any number of occurrences of any character that occurs in the position where the asterisk occurs in the pattern.

`/a*t/` # any number of a's followed by t

- A plus sign (+) indicates one or more of the preceding character.

`/a+t/` # one or more a's followed by t

Multipliers

- The concept of multiplier can be generalized by placing curly braces around a minimum and a maximum number of occurrences of the preceding character.

`/a{2,4}t/` # between 2 and 4 a's followed by t
`/a{2,}t/` # 2 or more a's followed by t
`/a{2}t/` # exactly 2 a's followed by t

- Pattern matching is greedy, meaning that if a pattern can be found at more than one place in the string but one instance is longer than the others, the longest match will be identified.

Memory

- The portion of the string that matches a pattern can be assigned to a variable for use later in the statement or in subsequent statements.
- This is done by placing the portion to be *remembered* in parentheses (`()`).
- Multiple segments, specified by multiple occurrences of parentheses through the pattern, are available in variables, `\1`, `\2`, `\3`, etc. in the order corresponding to the different parenthesized components.
- Beyond the scope of the statement, these stored segments are available in the variables, `$1`, `$2`, `$3`, etc.

Memory

- Other information available in variables include
 - `$&`, the sequence that matched;
 - `$``, everything in the string up to the match;
 - and `$'`, everything in the string beyond the match.

```
/c(.*)t/
```

```
# in caaat,
```

```
  \1 is "aaa";
```

```
  $1 has the same value
```

```
  $& is "aaa"
```

```
  $` is "c"
```

```
  $' is "t"
```

Anchors

- The pattern that is searched for in the string can be restricted to several specified locations, such as the beginnings and endings of words or the beginnings and endings of the string.

`\b` indicates a word boundary.

`\B` indicates any place but a word boundary.

Carat (`^`) restricts the pattern to the beginning of the string.

Dollar sign (`$`) specifies the end of the string.

- If a literal dollars sign occurs in the pattern, mark it with the backslash.

Anchor Examples

```
/\bat/ # matches "at" and "attention", but not "bat"  
/at\b/ # matches "at" and "bat", but not "attention"  
/at\B/ # matches "attention" but not "at" and "bat"  
/^\at/ # matches "at $5.00, it is a bargain" but not  
        "where you are at"  
/at$/ # matches "where you are at" but not "at  
        $5.00, it is a bargain"  
/\$/ # matches "at $5.00, it is a bargain"
```

Variable Interpolation

- Variables are interpolated. Since the dollar sign is used to mark ends of strings it should not conflict with interpolation of scalar variables that begin with a dollar sign.

```
$word = "cat";
```

```
/$word/ # matches strings that contain "cat"
```

Explicit Target String

- By default, the string that is searched is `$_`.
- Instead of searching in the string contained in the default variable, `$_`, the search can also be performed on the string specified on the left by using the `(=~)` operator .
- The `(=~)` operator takes two arguments: a string on the left and a regular expression pattern on the right.

```
$a =~ /cat/
```

```
# does the content of $a contain "cat"?
```

```
<LTSTDIN> =~ /cat/
```

```
# does the next line of input contain "cat"?
```


Regular Expression Operators

- Regular expression operators include a regular expression as an argument but instead of just looking for the pattern and returning a truth value, as in the examples above, they perform some action on the string.

Substitution

- Looks for the specified pattern and replaces it with the specified string.
- By default, it does this for only the first occurrence found in the string.
- Appending a (**g**) to the end of the expression tells the operator to make the substitution for all occurrences.

Form:

```
s/pattern/replacement/  
s/pattern/replacement/g  
$var =~ s/pattern/replacement/
```

Substitution Examples

```
s/cat/dog/
```

```
# replaces "cat" with "dog" in $_
```

```
s/cat/dog/gi
```

```
# same thing, but applies to "CAT", "Cat"  
wherever they appear
```

```
$a =~ s/cat/dog/
```

```
# applies the operation to $a
```

Translation

- Scans a string character by character, and replaces all occurrences of the characters found in the SEARCHLIST with the corresponding character in the REPLACEMENTLIST.

Form:

```
tr/SEARCHLIST/REPLACEMENTLIST/[optional characters]
```

Examples

Change everything to lower case

```
$string =~ tr/[A-Z]/[a-z]/;
```

Change all vowels to numbers

```
$string =~ tr/[A,E,I,O,U,Y]/[1,2,3,4,5]/;
```

Split

- Split searches for a pattern in a specified string and, if it finds it, throws away the portion that matched and returns the "before" and "after" substrings, as a list.

Form:

```
@var = split(/pattern/, string);  
@var = split(/pattern/);
```

Examples:

```
@a = split(/cat/, $aString);  
@a = split(/cat/);
```

- If no string is specified, the operator is applied to \$_.
◦ The default for split is to break up \$_ on whitespace:

Join

- Approximately the opposite of split. Takes a list of values, concatenates them, and returns the resulting string.

Form:

```
$var = join("item_1", $item2, . . .);
```

Example:

```
$a = join("cat", "dog", "bird");  
# returns "catdogbird"
```

Exercises

```
#change fred and barney to all caps
$_ = "I saw Barney with Fred.";
s/(fred | barney)/\U$1/gi;
# $_ is now "I saw BARNEY with FRED."
```

```
# check for Y2K date (4-digit year specified)
# format exactly: nn/nn/nnnn or nn-nn-nnnn
($line1 =~ m/[0-9]{2}[\ /-][0-9]{2}[\ /-][0-9]{4}/)
# returns true if proper format
```

```
# check for non-DNA characters
my $string =
"CCACACCACACCCACACaCCCaCaCATCACAC
CACACACCACACTACACCCA*CACACACA";
if( $string =~ m/([ ^ATCG])/i) {
    print "Warning! Found: $1 in the string";}

```