Programming in Perl
Lecture 3 Examples

Formated print statement:

```
#!/usr/local/bin/perl -w

$string = "Hello world";
$integer = 785;
$float = 134.675;
printf("%20s %d %10.2f\n", $string, $integer, $float);
# full list of possible formats pp. 222-223 Programming Perl
```

Output:

```
         Hello world 785     134.68<newline>
```

Regular Expressions:

Predefined Character Classes:

```
                        Equivalent     Negated   Equivalent
Name        Construct     Class       Construct    Class
----        ---------   ----------    ---------  ----------
Digit          \d       [0-9]            \D      [^0-9]
Word char      \w       [a-zA-Z0-9_]     \W      [^a-zA-Z0-9_]
Space char     \s       [ \r\t\n\f]      \S      [^ \r\t\n\f]
```

Multipliers:

```
Construct              Allowed Range
---------              -------------
{n, m}                 Pattern must occur AT LEAST n times
                       but NO MORE than m times
{n,}                   Pattern must occur AT LEAST n times
{n}                    Pattern must occur EXACTLY n times
*                      0 or more times (same as {0,})
+                      1 or more times (same as {1,})
?                      0 or 1 times (same as {0,1})
```

Use of multipliers:

```
/a{4,6}/   # matches 4, 5, or 6 a's in string
/a{4,}/    # matches 4 or more a's in string
/a{4}/     # matches exactly 4 a's in string
/a{0,4}/   # matches 0 to 4 a's (4 or fewer) in string

/foo{3}/   # matches foooo (or longer) in string, for example,
           # if $_ = "foooooo", the match operator also returns a 1
           # Note : once foooo is matched and a 1 is returnd,
           # the entire string is sometimes said to be ``matched''
/(foo){3}/ # matches foofoofoo in string
```

```
        Read-Only Variables used with Regexes

Name        Perl Variable   Holds copy of
----        -------------   -------------
Match           $&          substring matched in string
Prematch        $`          portion of string before
                              matched substring
Postmatch       $'          portion of string after
                              matched substring

Note that these variables are modified each time a
regex search is done.
```

```
$_ = "baaad";

if (/a{4}/) {
  print("$&\n");
  print("$`\n");
  print("$'\n");
}
# nothing is printed; pattern
# doesn't match any portion
# of the string
```

```
$_ = "baaaad";

if (/a{4}/) {
  print("$&\n");
  print("$`\n");
  print("$'\n");
}
# prints:
# aaaa<newline>
# b<newline>
# d<newline>
```

```
$_ = "baaaaad";

if (/a{4}/) {
  print("$&\n");
  print("$`\n");
  print("$'\n");
}
# prints:
# aaaa<newline>
# b<newline>
# ad<newline>
```

```
$_ = "baaaaad";

if (/[^a]*a{4}[^a]*/) {
  print("$&\n");
  print("$`\n");
  print("$'\n");
}
# prints nothing
# baaaa<newline>
# <newline>
# ad<newline>
```

```
$_ = "baaaaad";

if (/[^a]+a{4}[^a]+/) {
  print("$&\n");
  print("$`\n");
  print("$'\n");
}
# prints:
# nothing is printed; pattern
# doesn't match any portion
# of the string
```

```
$_ = "baaaad";

if (/[^a]+a{4}[^a]+/) {
  print("$&\n");
  print("$`\n");
  print("$'\n");
}
# prints:
# baaaad<newline>
# <newline>
# <newline>
```

Greedy vs. Lazy Regex Evaluation:

(Greedy)

```
Given

  "barbarbarfoobarfoobarfoo"

  /\w*foo/ # matches entire barbarbarfoobarfoobarfoo
           # greedy evaluation by default; \w* goes
           # to the end of the string, then it has to
           # backtrack to check for 'f' followed by 'o'
           # followed by 'o'
Given

  "barbarbar!foobarfoobarfoo"

  /\w*foo/ # matches foobarfoobarfoo portion
           # the \w* was happy with the first
           # part of the string until it hit the !
           # then it had to restart processing
Given

  "a xxx c xxxxx c xxx d"
  /a.*c.*d/ # the first .* matches up to the
            # the SECOND 'c' ; leftmost is greediest
```

(Lazy)

```
Given

  "barbarbarfoobarfoobarfoo"

  /\w*?foo/ # matches barbarbarfoo
            # forces lazy evaluation; \w* matches
            # only to the first 'f', then it starts
            # checking for an 'o' followed by
            # another 'o'

Given

  "a xxx c xxxxx c xxx d"
  /a.*?c.*d/ # the first .*? now matches up to the
             # the FIRST 'c'; the rest is picked
             # up by the next .*
```

Parentheses as memory:

```
$_ = "apples pears peaches plums";
/(\w*)\s+(\w*)\s+(\w*)/;
# matches 0 or more word characters followed by
# 1 or more space characters, etc.
print ("$1\n$2\n$3\n");
```

Output:

```
apples<newline>
pears<newline>
peaches<newline>
```

```
$_ = "apples pears peaches plums";
/(\w*)(\w*)(\w*)/;
# matches 0 or more word characters followed by
# 0 or more word characters, etc.
print ("$1\n$2\n$3\n");
```

Output:

```
apples<newline>
<newline>
<newline>
```

Reusing a previously matched pattern:

```
$_= "axxyxxb";
if (/a.{5}b/) {
  print("any five: $&\n");
}
# condition will evaluate to false
if (/a(.)\1{4}b/) {
  print("same five: $&\n");
}

$_ = "<B> bold text </B>";
if (/<([a-zA-Z0-9]*?)>.*?<\/\1>/) {
  print("bold: $&\n");
}
$_ = "<i> italic text </i>";
if (/<([a-zA-Z0-9]*?)>.*?<\/\1>/) {
  print("italic: $&\n");
}
```

( ) vs (?:):

```
$_ = "apples pears peaches plums";
/(?:\w*)\s(\w*)\s(\w*)/;
# matches 0 or more word characters followed by
# 0 or more word characters, etc.
print ("$1\n$2\n$3\n");
# $1 is assigned to by the first (),
# not the (?:); prints
Use of uninitialized value at ./parens.plx line 7.
pears<newline>
peaches<newline>
<newline>
```

Anchors:

```
Anchor Pattern      Meaning
--------------      -------
^                   Matches pattern only at beginning of string
$                   Matches pattern only at end of string
\b                  Matches pattern at a word boundary
                    (between characters that match \w and \W)
\B                  Matches pattern except at a word boundary
(?=regex)           Matches pattern if engine would match <regex> next
(?!regex)           Matches pattern if engine wouldn't match <regex> next


Examples:

 /^Al/;   # matches Al iff Al is at start of string
          # "Al said hi" match
          # "Hi Al" no match
 /Al$/;   # matches Al iff Al is at end of string
          # "Al said hi" no match
          # "Hi Al" match "Hi Al\n" match as well

 # word boundaries ; note : hello_there is a word
 #                          9.9 are two different words

 /Al\b/;   # "Al said hi" match
           # "Albert said hi" no match
 /\bAl/;   # "Hi Allen" match
           # "Mr. vanAllen" no match
 /\bAl\b/; # "Al" match
           # "Albert said hi" no match
           # "Mr. vanAllen" no match
 /\bAl\B/; # "Albert said hi" match
           # "Al said hi" no match

 # last two called lookahead anchors

 /Bill (?=The Cat|Gates)/; # matches substring only in strings
                           # in which "The Cat" or "Gates" comes
                           # after "Bill "
```

```
/Bill (?!The Cat|Gates)/; # matches substring only in strings
                          # in which neither "The Cat" nor "Gates"
                          # comes after "Bill "
```

Precedence:

```
regex Grouping Precedence

Name                      Representation
----                      --------------
Parentheses               ( ) (?: )
Multipliers               ? + * {m,n} ?? +? *? {m,n}?
Sequence & anchoring      abc ^ $ (?= ) (?! )
Alternation               |

Examples of use of parentheses:

hi*                 # matches "h", "hi", "hii", "hiii", ...
                    # within a string
(hi)*               # matches "", "hi", "hihi", ...
                    # within a string

/^fee|fie|foe$/;    # matches "fee" at beginning of the string
                    # "fie" anywhere OR "foe" at the end of the
                    # string
/^(fee|fie|foe)$/;  # matches a string consisting solely of
                    # "fee" , "fie" or "foe"

/to(nite|night)/    # matches 'tonite' or 'tonight'
/toni(te|ght)/      # same as above; more efficient
```

Pattern Binding Operator (=~):

```
$name = "Joe Smith";
if ($name =~ /Ren|Stimpy/) {
  print ("Goodnight $&\n");
}

# can be used on anything that yields a scalar value
do {
  # stuff
  print("Continue (y/n)?");
} until (<STDIN> =~ /^[nN]/);

-- ignoring case (case insensitivity)

do {
  # stuff
  print("Continue (y/n)?");
} until (<STDIN> =~ /^n/i);
```

Substitution:

```
        # basic format:
        s/regex-old-string/regex-new-string/

        $_ = "foot fool buffoon";
        s/foo/bar/;
        # $_ is now "bart fool buffon"
        # changes first match encountered

        $_ = "foot fool buffoon";
        s/foo/bar/g;
        # $_ is now "bart barl bufbarn"
        # changes all matches encountered (global)

        $_ = "foOt Fool buffOon";
        s/foo/bar/gi;
        # $_ is now "bart barl bufbarn"
        # changes all matches encountered (global)
        # case insensitive

        $_ = "hello world";
        $new = "goodbye";
        s/hello/$new/;
        # replaces hello with goodbye

# use of x to allow commenting
$number = 9999999999;
$number =~ s/
    (\d{1,3})        # before a comma: one to three digits
    (?=              # followed by, but not part of what's matched
       (?:\d\d\d)+   #    some number of triplets...
       (?!\d)        #    ...not followed by another digit
    )                # (which ends the number)
    /$1,/gx;         # x allows regular expression to be broken
                     # across lines and comments to be inserted
print("$number\n");
# the result printed is 9,999,999,999
```

Variable interpolation and regexes:

```
$none_found = 1;
@lines = ("toy joy", "camel llama", "susan roy");

print("Enter a word to search for: ");
chomp($search_str = <STDIN>);

# continued on next page ...
```

```
# search through array for $search_str
# as if it were a single word
print("Lines containing $search_str:\n");
foreach $line (@lines) {
  if ($line =~ /\b$search_str\b/) {
    print ("$line\n");
    $none_found = 0;
  }
}
if($none_found) {
  printf("(no lines contained $search_str)\n");
}
```

Use of Quote Escape to Backslash Escape Regex Pattern Characters:

```
  -- \Q : the quote escape

  $what = "[box]";
  foreach (qw (in[box] out[box] white[sox])) {
    if (/\Q$what\E/) {
      print ("$_ matched!\n");
    }
  }
  # matches [box] of 'in[box]' and 'out[box]'
  # without \Q, would match  b  of 'in[box]',  o  of 'out[box]'
  # and  o  of 'white[sox]'
```

Example regex (recognizes complex numbers) :

```
$digit = '\d';
$digits = "$digit+";
# omitting definition of float
$int = "[+-]?$digit+";
$real = "(?:$float|$int)";
$imag = "(?:${real})?i";
$opt_spaces = '\s*';
# qq double quotes everything between < and >;
# this is always treated as a single string
# characters other than <> may be used
$complex = qq<
    $real           # real part
    $opt_spaces     # 0 or more spaces
    [+-]            # + or -
    $opt_spaces     # 0 or more spaces
    $imag           # imaginary part
>;

# strip comments out of string
$complex =~ s/#.*//g;
# strip spaces and newlines out of string
$complex =~ s/\s+//g;

# continued on the next page ...
```

```
print("Enter a string to test: ");
chomp($test = <STDIN>);
while ($test ne "") {
  if ( $test =~ /$complex/ ) {
    print $test, " contains the complex number $& \n";
  }  else {
    print $test, " does NOT contain a complex number\n";
  }
  print("Enter a string to test: ");
  chomp($test = <STDIN>);
}
```

Split:

```
$line = "Betty Boop:555-5555:1 Boop Lane::100000";
# split line use : as delimiter
@fields = split(/:/,$line);
# @fields is ("Betty Boop", "555-5555", "1 Boop Lane",
#             "", "100000")

@fields = split(/:+/,$line);
# @fields is ("Betty Boop", "555-5555", "1 Boop Lane",
#             "100000")

# empty trailing fields are ignored
$line = "Betty Boop:555-5555:1 Boop Lane:100000:";
# split line use : as delimiter
($name,$phone,$address,$salary,$dob) = split(/:/, $line);
# $dob is undef
```

Join:

```
# @fields is ("Betty Boop", "555-5555", "1 Boop Lane",
#             "100000")
# put $line back together
$gluedline = join(":", @fields);
# "Betty Boop:555-5555:1 Boop Lane:100000"

# note: the glue string is just a string NOT a regex

# to get glue in front of a list as well
$result = join("+", "", @fields);
# "" is treated as the empty element to be glued
# with the first data element of @fields
# $result is "+Betty Boop+555-5555+1 Boop Lane+100000";

# to get glue in back of a list as well
$result = join("+", @fields, "");
# "" is treated as the empty element to be glued
# with the last  data element of @fields
# $result is "Betty Boop+555-5555+1 Boop Lane+100000+";
```