printf

- `printf` in Perl is very similar to that of C.
- `printf` is most useful when when printing scalars. Its first (non-filehandle) argument is the format string, and any other arguments are treated as a list of scalars:

```
printf "%s %s %s %s", ("abc", "def") , ("ghi", "jkl");
# yields
abc def ghi jkl
```

printf

☞ Some of the common format attributes are

- ⇨ `%[-][N]s` → format a string scalar, N indicates maximum characters expected for justification, - indicates to left-justify rather than default right-justify.
- ⇨ `%[-|0][N]d` → format a numerical scalar as integer, N indicates maximum expected for justification, "-" indicates to left-justify, "0" indicates zero-fill (using both "-" and "0" results in left-justify, no zero-fill.)
- ⇨ `%[-|0]N.Mf` → format a numerical scalar as floating

point. "N" gives the total length of the output, and "M" give places after the decimal. After the decimal is usually zero-filled out (you can toggle this off by putting "0" before "M".) "0" before N will zero-fill the left-hand side; "-" will left-justify the expression.

Examples of `printf()`

```
printf "%7d\n", 123;
# yields
    123

printf "%10s %-10s\n","abc","def";
# yields
       abc def
```

Examples of `printf()`

```
printf "%10.5f %010.5f %-10.5f\n",12.1,12.1,12.1;
# yields
  12.10000 0012.10000 12.10000

$a = 10;
printf "%0${a}d\n", $a;
# yields
0000000010
```

Perl regular expressions

☞ Much information can be found at `man perlre`.

☞ Perl builds support for regular expressions as a part of the language like awk but to a greater degree. Most languages instead simply give access to a library of regular expressions (C, PHP, Javascript, and C++, for instance, all go this route.)

☞ Perl regular expressions can be used in conditionals,

where if you find a match then it evaluates to true, and if no match, false.

```
$_ = "howdy and hello are common";
if(/hello/)
{
  print "Hello was found!\n";
}
else
{
  print "Hello was NOT found\n";
}
# yields
Hello was found!
```

What do Perl patterns consist of?

- Literal characters to be matched directly
- "." (period, full stop) matches any one character (except newline unless coerced to do so)
- "*" (asterisk) matches the preceding item zero or more times
- "+" (plus) matches the preceding item one or more times

- "?" (question mark) matches the preceding item zero or one time
- "(" and ")" (parentheses) are used for grouping
- "|" (pipe) expresses alternation
- "[" and "]" (square brackets) express a range, match one character in that range

Examples of Perl patterns

`/abc/`	Matches "abc"
`/a.c/`	Matches "a" followed by any character (except newline) and then a "c"
`/ab?c/`	Matches "ac" or "abc"
`/ab*c/`	Matches "a" followed by zero or more "b" and then a "c"
`/ab|cd/`	Matches "abd" or "acd"
`/a(b|c)+d`	Matches "a" followed by one or more "b" or "c", and then a "d"
`/a[bcd]e/`	Matches "abe", "ace", or "ade"
`/a[a-zA-Z0-9]c/`	Matches "a" followed one alphanumeric followed by "c"
`/a[^a-zA-Z]/`	Matches "a" followed by anything other than alphabetic character

Character class shortcuts

You can use the following as shortcuts to represent character classes:

`\d`	A digit (i.e., `0-9`)
`\w`	A word character (i.e., `[0-9a-zA-Z_]`)
`\s`	A whitespace character (i.e., `[\f\t\n ]`)
`\D`	Not a digit (i.e., `[^0-9]`)
`\W`	Not a word (i.e., `[^0-9a-zA-Z_]`)
`\S`	Not whitespace

General quantification

You can specify numbers of repetitions using a curly bracket syntax:

```
a{1,3}        # ‘‘a’’, ‘‘aa’’, or ‘‘aaa’’
a{2}          # ‘‘aa’’
a{2,}         # two or more ‘‘a’’
```

Anchors

Perl regular expression syntax lets you work with context by defining a number of "anchors": \A, ^, \Z, $, \b.

/\ba/	Matches if "a" appears at the beginning of a word
/\Aa$/	Matches if "a" appears at the end of a line
/\Aa\Z/	Matches if a line is exactly "a"
/^Aa$/	Matches if a line is exactly "a"

Remembering substring matches

- Parentheses are also used to remember substring matches.
- Backreferences can be used within the pattern to refer to already matched bits.
- Memory variables can be used after the pattern has been matched against.

Backreferences

- A backreference looks like \1, \2, etc.
- It refers to an already matched memory reference.
- Count the left parentheses to determine the back reference number.

Backreference examples

```
/(a|b)\1/           # match ‘‘aa’’ or ‘‘bb’’
/((a|b)c)\1/        # match ‘‘acac’’ or ‘‘bcbc’’
/((a|b)c)\2/        # match ‘‘aba’’ or ‘‘bcb’’
/(.)\1/             # match any doubled characters except newline
/\b(\w+)\s+\b\1\s/ # match any doubled words
/(['"])(.*)\1/      # match strings enclosed by single or double quotes
```

Remember, perl matching is by default greedy

For example, consider the last backreference example:

```
$_ = "asfasdf 'asdlfkjasdf ' werklwerj'";
if(/(['"])(.*)\1/)
{
  print "matches $2\n";
}
# yields
matches asdlfkjasdf ' werklwerj
```

Memory variables

- A memory variable has the form $1, $2, etc.
- It indicates a match from a grouping operator, just as back reference does, but after the regular expression has been executed.

```
$_ = "  the larder  ";
if(/\s+(\w+)\s+/)
{
   print "match = '$1'\n";
}
# yields
match = 'the'
```

Regular expression "binding" operators

Up to this point, we have considered only operations against $_.

Any scalar can be tested against with the =~ and !~ operators.

```
"STRING" =~ /PATTERN/;

"STRING" !~ /PATTERN/;
```

Examples

```
$line = "not an exit line";
if($line !~ /^exit$/)
{
   print "$line\n";
}
# yields
not an exit line

# skip over blank lines...
if($line =~ /$^/)
{
  next;
}\
```

Automatic match variables

You don't have to necessarily use explicit backreferences and memory variables. Perl also gives you three default variables that can be used after the application of any regular expression; they refer to the portion of the string matched by the whole regular expression.

```
$`     refers to the portion of the string before the match
$&     refers to the match itself
$'     refers to the portion of the string after the match
```

Example of automatic match variables

```
$_ = "this is a test";
/is/;
print "before: < $` >  \n";
print "after:  < $' >  \n";
print "match:  < $& >  \n";
# yields
before: < th >
after:  <  is a test >
match:  < is >
```

Example of automatic match variables

```
#!/usr/bin/perl -w
# 2006 09 27 - rdl Script34.pl // change = to =:
use strict;
while(<>)
{
  /=/;
  print "$‘=:$’\n";
}
```

Other delimiters: Using the "m"

You can use other delimiters (some are paired items) rather than just a slash, but you must use the "m" to indicate this. (See `man perlop` for a good discussion.)

For instance:

```
m/.../  m{...}  m[...]  m(...)
m!...!  m,...,  m^...^  m#...#
```

Example

```
# not so readable way to look for a URL reference
if ($s =~ /http:\/\//)

# better
if ($s =~ m^http://^ )
```

Option modifiers

There are a number of modifiers that you can apply to your regular expression pattern:

Modifier	Description
i	case insensitive
s	treat string as a single line
g	find all occurrences