Perl 2: Intermediate Perl

- Alternation
- Complete Parsing
  - Quiz 1 Project 1

Lesson 9: **Introduction to One-Liners**

- Perl at the Command Line
- The -e Flag
- The -n Flag
- BEGIN and END blocks
  - Quiz 1 Project 1

Lesson 10: **Substitution and More One-Liners**

- Substitution
- Substitution in One-liners
- The -p Flag
- The -i Flag
  - Quiz 1 Project 1

Lesson 11: **More Perl Flags; More Perl Operators**

- The -w Flag
- The -Mstrict Flag
- The -c Flag
- The Trinary Operator
  - Nesting the Trinary Operator
  - The Trinary Operator as Lvalue
- The Scalar Range Operator
  - Quiz 1 Project 1

Lesson 12: **More on Numbers and Strings; More on Regular Expressions; More on Hashes**

- String and Number Literals
- Character Functions
- Number Functions
- split()
- Hashes: each()
- read()
  - Quiz 1 Project 1

Lesson 13: **For Loops and More Miscellaneous Topics**

- For Loops
- Loop Labels
- Unbuffered Output
  - Quiz 1 Project 1

Lesson 14: **Directory Reading Functions**

- opendir, readdir, closedir
- glob
  - Quiz 1 Project 1

Lesson 15: **More Math Functions and Course Wrap-Up**

- More Math Functions
- crypt()

Final Example Program

Quiz 1 Project 1

Getting Started

Welcome to the O'Reilly School of Technology's Perl 2—Intermediate Perl course!

Course Objectives

When you complete this course, you will be able to:

- create practical programs that interact with the user and the operating system.
- perform useful and important tasks without even writing programs, by calling Perl from the command line with brief "one-liners."
- match and change text using the powerful technology of regular expressions.
- enhance software quality with a larger repertoire of Perl operators, functions, and looping constructs.

Introduction

Hello, my name is Peter Scott, and I have been using and teaching Perl for over a dozen years. I'm the author of the books *Perl Debugged* and *Perl Medic* and the DVD/video *Perl Fundamentals*. I've taught Perl in person to hundreds of people and I am also the author of the Perl 1 course from OST. I love Perl, and I want you to enjoy your time spent learning more about Perl from this course. Please let your instructor know of any way in which you think this course could be improved, and also parts that you found particularly effective (so we know not to mess with those bits).

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have

some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove will look like this.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

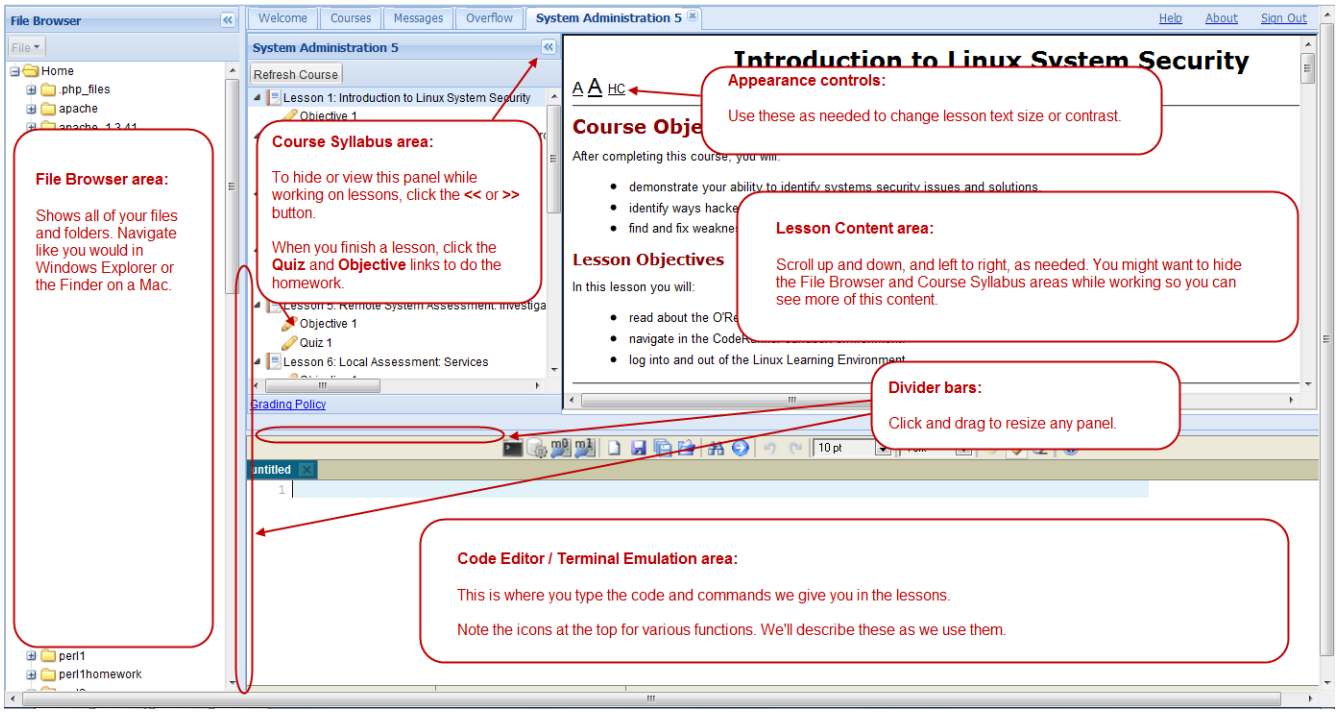
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

File Management Demo

Code Editor Demo

Coursework Demo

Prerequisites

We assume and require that you have either taken our Perl 1 course or have acquired the knowledge imparted by that course some other way. We'll spend some time in this lesson reviewing those prerequisites. (Of course, we can't review them completely; it took 16 lessons in Perl 1 to cover them. But we'll give it our best shot.)

The next section is a detailed review of the topics covered in the Perl 1 course; we recommend going through it as a refresher, but if you've taken Perl 1 recently or otherwise feel you don't need the review, you can skip to the next section, Postfixed Modifiers.

If you find as you review those prerequisites and the rest of this lesson that you're not sure you're ready for this course, don't worry—it happens. Don't give up, but don't try to struggle through the course if you're not sufficiently prepared. We're happy to transfer your registration to the Perl 1 course so you can get well-grounded in the fundamentals of Perl. Just contact us at info@oreillyschool.com.

Review

Here's what you should know already in order to complete this course successfully. For each of these topics, you need to know only the basics—we'll develop these topics in more depth during this course.

For instance, we require that you know how to declare a scalar variable and manipulate scalars containing strings and numbers, but this does not extend to handling pathologically long strings or understanding the internal structure of scalars:

OBSERVE: scalar initialization

```
my $dog = "Spot";
```

If you know what this code means, that's enough on the topic of initialization for this course.

Development and Execution: You understand how to create, edit, and run a Perl program. (We'll show you how the specialized training environment we use in this course works; but you should have developed and run Perl programs somewhere else at least once.) You know about the "shebang" line and you follow it with **use strict;** and **use warnings;** and you do not finalize any program that outputs errors or warnings.

You know that command line arguments arrive in the array **@ARGV**.

Documentation: You know about the *perldoc* program and how to find the documentation for any given Perl built-in function:

OBSERVE:

```
perldoc -f join
```

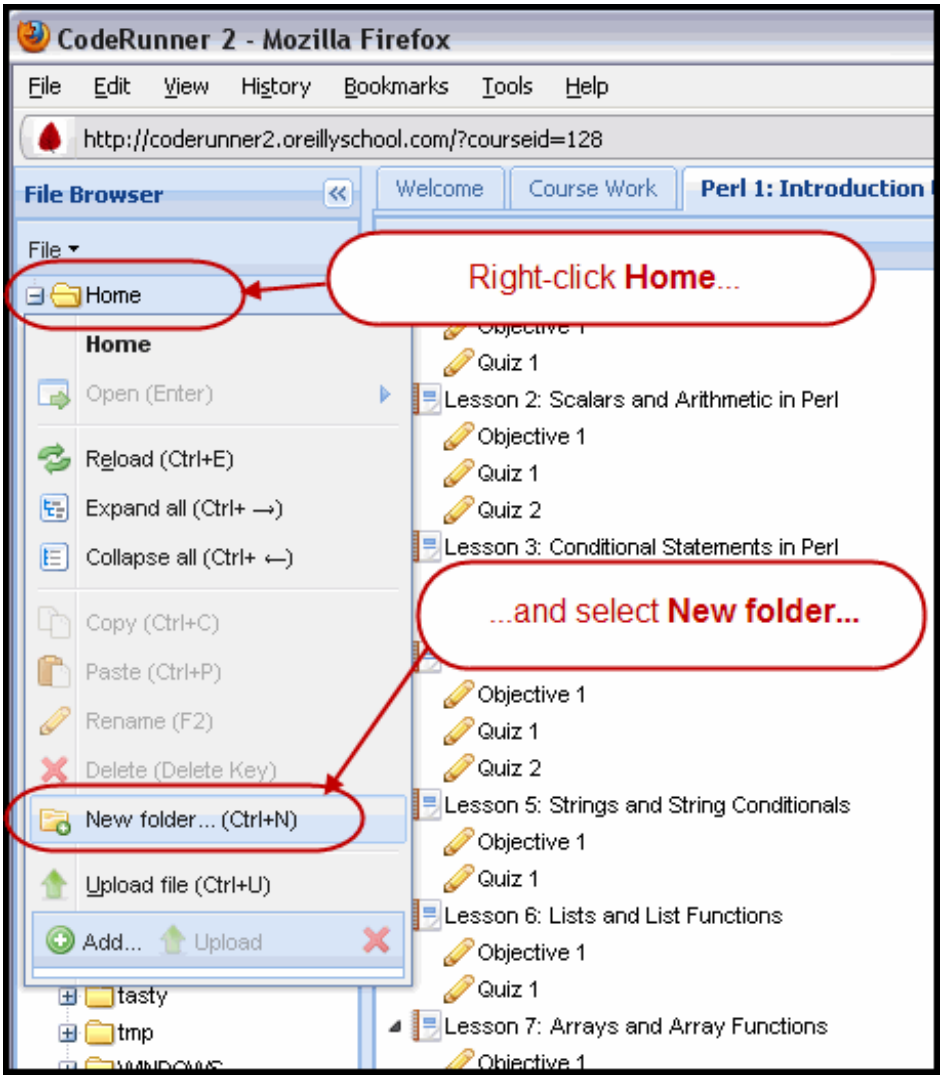
Output and errors: You have used the functions **print**, **warn**, and **die**; you can print to the standard output or error stream and you can make your program quit with an error message:

OBSERVE: output and basic error-handling

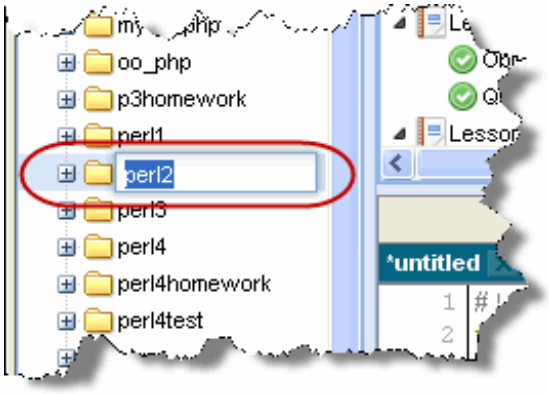
```
print "I have ", $song_count, " MP3 files";
die "Error in consistency check";
```

Let's start rolling these review points into a program now, so that you have something to do!

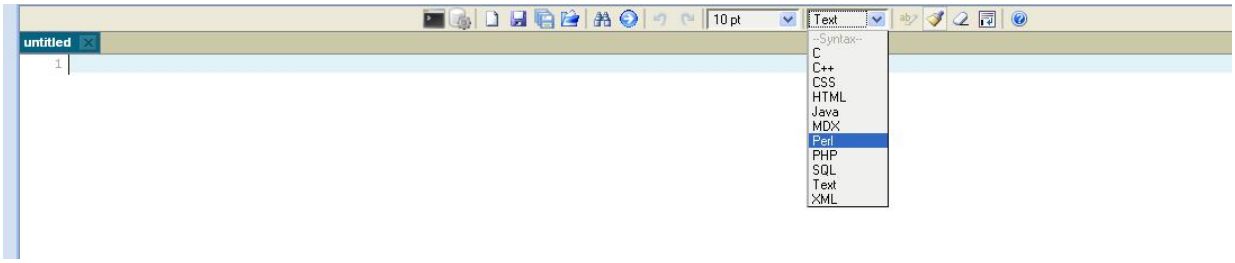
First, we'll create a folder to keep all of our Perl stuff organized. In the left panel of your CodeRunner window, right-click **Home**, and select **New folder...** as shown:



Name the folder **perl2** as shown:



Locate the drop-down menu in the CodeRunner toolbar. Be sure that **Perl** is the **Syntax** selected from this menu:



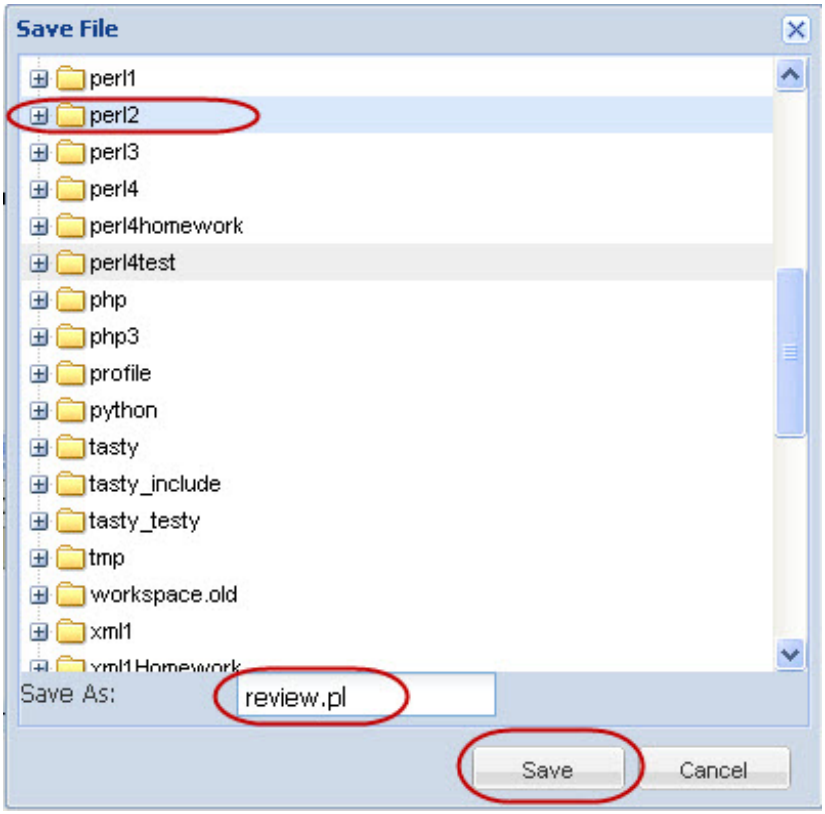
You should now see a Check Syntax icon on the left side of the toolbar.

In the editor, type (don't just cut and paste!) all the lines you see below so it looks like this:

CODE TO TYPE: review.pl

```
#!/usr/bin/perl
use strict;
use warnings;

die "This... is a very short program.  I only have time to... arrgh!";
```

Compile the program: Click Check Syntax. A new window will open where you can name and save the file. Select the **/perl2** folder you just created (save all your files here unless otherwise specified), enter the file name **review.pl**, and click **Save**:



(If the **/perl2** folder doesn't yet exist, right-click on the **Home** folder and create it by clicking **New Folder**.)

Click the **Save** button. You'll see a message confirming that the program compiled and syntax is OK. Go ahead and **Close** this window.

Now, let's run the program! Click the **New Terminal** button in the Toolbar:  .

Then, log in to the OST server. Be sure to replace **username** with your username. It should look something like this:

INTERACTIVE TERMINAL SESSION

```
login: username
password:
Last login: Mon Aug 16 11:36:41 from 63.171.219.110
cold:~$
```

Note When you enter your password, the cursor will not move. In fact, it will not appear as though anything is happening. Rest assured, it is. Just be sure to type your password carefully and hit Enter. If you are having trouble, please contact your mentor.

The server is named **cold**. All OST students have shells on this server. A **shell** is the place where you execute Unix commands on the server. The commands you execute on your shell will not effect any other shell.

cold:~$ is called a command prompt. If you see this, you're ready to execute Unix commands. To run your program, go to the Unix prompt and call it by name:

INTERACTIVE TERMINAL SESSION

```
cold:~$ cd perl2
cold:~/perl2$ ./review.pl
This... is a very short program.  I only have time to... arrgh!
```

Interpolation: You know about interpolating scalars and the more common escape sequences inside double-quoted strings:

OBSERVE:

```
print "I have $song_count MP3 files\n";
```

Formatted printing: You can print numbers and strings formatted to fit columns in tabular output using the **printf** function:

OBSERVE:

```
printf "I have %3d MP3 files\n", 13;
```

Variable declaration and initialization: You use **my** to declare your variables, and are able to initialize them at the same time:

OBSERVE:

```
my $song_count;
my $car_type = 'hybrid';
```

Modify **review.pl** as follows:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

die "This... is a very short program.  I only have time to... arrgh";
print "I have 3 MP3 files\nTheir lengths in minutes are:\n";
printf "#%2d: %-30s %5.2f\n", 1, "Fixing a Hole",     2.033333;
printf "#%2d: %-30s %5.2f\n", 2, "Lovely Rita",       2.07;
printf "#%2d: %-30s %5.2f\n", 3, "A Day in the Life", 5.55;
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./review.pl
I have 3 MP3 files
Their lengths in minutes are:
# 1: Fixing a Hole                2.03
# 2: Lovely Rita                  2.07
# 3: A Day in the Life            5.55
cold:~/perl2$
```

Scalar variables: You can use scalars to store numbers (integer and floating point) and strings:

OBSERVE:

```
$device_type = "cellphone";
$song_count = 42;
```

Arithmetic: You can perform basic arithmetic on numbers, and know that Perl has more complex arithmetic functions for everything from trigonometry to random numbers:

OBSERVE:

```
$area = $PI * $radius ** 2;
$song_count++;
```

Modify **review.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

print "I have 3 MP3 files\nTheir lengths in minutes are:\n";
printf "#%2d: %-30s %5.2f\n", 1, "Fixing a Hole",      2.033333;
printf "#%2d: %-30s %5.2f\n", 2, "Lovely Rita",        2.07;
printf "#%2d: %-30s %5.2f\n", 3, "A Day in the Life", 5.55;
my $song_count = 3;
my $song = 1;
print "I have $song_count MP3 files\nTheir lengths in minutes are:\n";
printf "#%2d: %-30s %5.2f\n", $song,     "Fixing a Hole",     2 +  2/60;
printf "#%2d: %-30s %5.2f\n", $song + 1, "Lovely Rita",       2 +  4/60;
printf "#%2d: %-30s %5.2f\n", $song + 2, "A Day in the Life", 5 + 33/60;
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./review.pl
I have 3 MP3 files
Their lengths in minutes are:
# 1: Fixing a Hole                  2.03
# 2: Lovely Rita                    2.07
# 3: A Day in the Life              5.55
cold:~/perl2$
```

String manipulation: You can truncate and lengthen strings, find their lengths, and test for substrings in them. Operators and functions used: **.** (concatenation), **substr**, **length**, and **index**:

OBSERVE:

```
$device_type = $portability . 'phone';
$area_code = substr $phone_number, 1, 3;
```

Lists: You understand the role of lists in Perl code and how they can be constructed with the range operator:

OBSERVE:

```
my @small_ints = 1 .. 10;
```

Array variables and array operations: You are able to use arrays to store multiple scalars; you can expand and contract them with the functions **push**, **pop**, **shift**, and **unshift**:

OBSERVE:

```
my @Beatles = qw(John Paul George Ringo);
push @Beatles, 'Pete';
$lead_singer = shift @Beatles;
```

Modify **review.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $song_count = 3;
my $song = 1;
printf "#%2d: %-30s %5.2f\n", $song,     "Fixing a Hole",      2 +  2/60;
printf "#%2d: %-30s %5.2f\n", $song + 1, "Lovely Rita",        2 +  4/60;
printf "#%2d: %-30s %5.2f\n", $song + 2, "A Day in the Life", 5 + 33/60;
my $song = 0;
my @songs = ("Fixing a Hole", "Lovely Rita", "A Day in the Life");
my @song_lengths = (2 + 2/60, 2 + 4/60, 5 + 33/60);
print "I have $song_count MP3 files\nTheir lengths in minutes are:\n";
printf "#%2d: %-30s %5.2f\n", $song + 1, $songs[$song], $song_lengths[$song];
$song = $song + 1;
printf "#%2d: %-30s %5.2f\n", $song + 1, $songs[$song], $song_lengths[$song];
$song = $song + 1;
printf "#%2d: %-30s %5.2f\n", $song + 1, $songs[$song], $song_lengths[$song];
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./review.pl
I have 3 MP3 files
Their lengths in minutes are:
# 1: Fixing a Hole                2.03
# 2: Lovely Rita                  2.07
# 3: A Day in the Life            5.55
cold:~/perl2$
```

Conditional statements: You know how to use the **if** and **unless** statements. You know about Perl's logical (boolean) operators and relational operators:

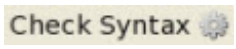
OBSERVE:

```
if ( $lead_singer eq 'John' )
{
  print "In My Life...\n";
}
unless ( $temperature > 80 && $humidity < 0.25 )
{
  start_sprinklers();
}
```

Modify **review.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = 80;
my $song_count = 3;
my $song = 0;
my @songs = ("Fixing a Hole", "Lovely Rita", "A Day in the Life");
my @song_lengths = (2 + 2/60, 2 + 4/60, 5 + 33/60);
print "I have $song_count MP3 files\nTheir lengths in minutes are:\n";
printf "#%2d: %-30s %5.2f\n", $song + 1, $songs[$song], $song_lengths[$song];
$song = $song + 1;
printf "#%2d: %-30s %5.2f\n", $song + 1, $songs[$song], $song_lengths[$song];
$song = $song + 1;
printf "#%2d: %-30s %5.2f\n", $song + 1, $songs[$song], $song_lengths[$song];

if ( $song_lengths[0] + $song_lengths[1] + $song_lengths[2] < $CD_LENGTH )
{
  print "I can fit all of these on a CD.\n";
}
```



and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./review.pl
I have 3 MP3 files
Their lengths in minutes are:
# 1: Fixing a Hole                2.03
# 2: Lovely Rita                  2.07
# 3: A Day in the Life            5.55
I can fit all of these on a CD.
cold:~/perl2$
```

Because the combined length of the songs is not greater than 80 minutes (the space available on a standard 700M audio CD), you get a message that the songs can fit.

Looping statements: You know how to use the **while**, **until**, and **foreach** statements:

OBSERVE:

```
while ( my $arg = shift @ARGV )
{
  if ( $arg eq '--' )
  {
    last;    # Stop option processing
  }
  if ( index $arg, '-' == 0 )
  {
    push @options, substr $arg, 1;
  }
}
foreach my $option ( @options )
{
  unless ( is_valid( $option ) )
  {
    die "Invalid option $option\n";
  }
}
```

Modify **review.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = 80;
my $song_count = 3;
my $song = 0;
my $index = 0;
my @songs = ("Fixing a Hole", "Lovely Rita", "A Day in the Life");
my @song_lengths = (2 + 2/60, 2 + 4/60, 5 + 33/60);
print "I have $song_count MP3 files\nTheir lengths in minutes are:\n";
printf "#%2d: %-30s %5.2f\n", $song, $songs[$song], $song_lengths[$song];
$song = $song + 1;
printf "#%2d: %-30s %5.2f\n", $song, $songs[$song], $song_lengths[$song];
$song = $song + 1;
printf "#%2d: %-30s %5.2f\n", $song, $songs[$song], $song_lengths[$song];
foreach my $title ( @songs )
{
  printf "#%2d: %-30s %5.2f\n", $index + 1, $title, $song_lengths[$index];
  $index++;
}
if ( $song_lengths[0] + $song_lengths[1] + $song_lengths[2] < $CD_LENGTH )
{
  print "I can fit all of these on a CD.\n";
}
```

Check Syntax and run it

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./review.pl
I have 3 MP3 files
Their lengths in minutes are:
# 1: Fixing a Hole                2.03
# 2: Lovely Rita                  2.07
# 3: A Day in the Life            5.55
I can fit all of these on a CD.
cold:~/perl2$
```

Hash variables and operations: You know how to use hashes to map strings to values. You understand the applications of hashes to problems that require lookup tables, mapping of discrete values, existence checks, concordance-type counting, and other set operations including union and intersection. (Those are some fairly technical terms for some common situations; if you're not familiar with the terms, that's fine, as long as you've used hashes and understand that they're really important and useful in virtually every program.)

You know how to iterate through a hash with the **keys** function:

OBSERVE:

```
my %state = ( AK => "Alaska", AL => "Alabama", CO => "Colorado" );
foreach my $abbreviation ( keys %state )
{
  print "$abbreviation is $state{$abbreviation}\n";
}
while ( my $word = shift @tokens )
{
  $seen{$word}++;
}
```

Modify **review.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = 80;
my $song_count = 3;
my $index = 1;
my @songs = ("Fixing a Hole", "Lovely Rita", "A Day in the Life");
my @song_lengths = (2 + 2/60, 2 + 4/60, 5 + 33/60);
my %song_length = ("Fixing a Hole" => 2 + 2/60, "Lovely Rita" => 2 + 4/60, "A Da
y in the Life" => 5 + 33/60);
print "I have $song_count MP3 files\nTheir lengths in minutes are:\n";
my $total_length;
foreach my $title ( keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index + 1, $title, $song_lengths[$index];
  $index++;
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
  $total_length += $song_length{$title};
}
if ( $total_length < $CD_LENGTH )
{
  print "I can fit all of these on a CD.\n";
}
else
{
  print "I can't fit all of these on one CD!\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./review.pl
I have 3 MP3 files
Their lengths in minutes are:
# 1: A Day in the Life               5.55
# 2: Lovely Rita                     2.07
# 3: Fixing a Hole                   2.03
I can fit all of these on one CD.
cold:~/perl2$
```

Subroutines: You know how to declare and call subroutines, how to pass arbitrary arguments and return lists of values:

OBSERVE:

```
my ($name, $score) = get_score( extract_words( $input ) );
sub get_score
{
  my @words = @_;
  my $person = $words[3];
  my $score = $words[5];
  return ($person, $score);
}
sub extract_words
{
  my $input = shift;  # "The score for <person> is <score>
  return split ' ', $input;
}
```

Modify **review.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = 80;
my $song_count = 3;
my $index = 1;

my %song_length = ("Fixing a Hole" => 2 + 2/60, "Lovely Rita" => 2 + 4/60, "A Da
y in the Life" => 5 + 33/60);
print "I have $song_count MP3 files\nTheir lengths in minutes are:\n";
my $total_length;
foreach my $title ( keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
  $total_length += $song_length{$title};
}
if ( $total_length < $CD_LENGTH )
if ( total_length( %song_length ) < $CD_LENGTH )
{
  print "I can fit all of these on a CD.\n";
}
else
{
  print "I can't fit all of these on one CD!\n";
}

sub total_length
{
  my %media = @_;

  my $total = 0;
  foreach my $length ( values %media )
  {
    $total += $length;
  }
  return $total;
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./review.pl
I have 3 MP3 files
Their lengths in minutes are:
# 1: A Day in the Life              5.55
# 2: Lovely Rita                    2.07
# 3: Fixing a Hole                  2.03
I can fit all of these on a CD.
cold:~/perl2$
```

Context: You understand the difference between scalar and list context (bonus points if you also know about boolean and void contexts!). You know that context propagates through subroutine calls all the way down. And you know what an array evaluates to in scalar and list contexts:

OBSERVE:

```
print "There are " . @Beatles . " members of the band\n";
```

Modify **review.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = 80 * 60;
my $song_count = 3;
my $index = 1;

my %song_length = ("Fixing a Hole" => 2 + 2/60, "Lovely Rita" => 2 + 4/60, "A Da
y in the Life" => 5 + 33/60);
my @songs = keys %song_length;
print "I have $song_count MP3 files\nTheir lengths in minutes are:\n";
print "I have " . @songs . " MP3 files\nTheir lengths in minutes are:\n";
foreach my $title ( keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
}
if ( total_length( %song_length ) < $CD_LENGTH )
{
  print "I can fit all of these on a CD.\n";
}
else
{
  print "I can't fit all of these on one CD!\n";
}

sub total_length
{
  my %media = @_;

  my $total = 0;
  foreach my $length ( values %media )
  {
    $total += $length;
  }
  return $total;
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./review.pl
I have 3 MP3 files
Their lengths in minutes are:
# 1: A Day in the Life              5.55
# 2: Lovely Rita                    2.07
# 3: Fixing a Hole                  2.03
I can fit all of these on a CD.
cold:~/perl2$
```

Reading lines: You know about the <> ("readline") operator for getting lines from standard input, and how to use it in scalar and list contexts:

OBSERVE:

```
while ( defined( my $line = <> ) )
{
  chomp $line;
  if ( substr $line, 0 eq '<' )
  {
    push @old_file_lines, $line;
  }
  elsif ( substr $line, 0 eq '>' )
  {
    push @new_file_lines, $line;
  }
}
```

Sorting: You can sort strings:

OBSERVE:

```
for my $abbreviation ( sort keys %state )
{
  # ...
}
```

Wrapping Up the Review

Now let's look some of those features at work in an example program that acts as a calculator, processing directives that act on a running accumulator. (This also just happens to be an implementation of the project from the final assignment of the OST Perl 1 course.) Create **calculate.pl** in the CodeRunner editor as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $INVALID = "Invalid statement\n";
my $Accumulator;

while ( print( "> " ) && defined( my $line = <> ) )
{
  chomp $line;
  my $ok;
  ($ok, $Accumulator) = execute( $line, $Accumulator );
  if ( $ok )
  {
    print "OK\n";
  }
  else
  {
    warn $INVALID;
  }
}

sub execute
{
  my ($line, $current) = @_;

  if ( length $line == 0 )
  {
    return;
  }
  my $operator_length = index $line, ' ';
  if ( $operator_length < 0 )
  {
    return monadic( $line, $current );
  }
  else
  {
    my $operator = substr $line, 0, $operator_length;
    my $operand = substr $line, $operator_length + 1;
    return dyadic ( $operator, $operand, $current );
  }
}

sub monadic
{
  my ($operator, $current) = @_;

  if ( $operator eq 'EQUALS' )
  {
    if ( defined $current )
    {
      print " = $current\n";
    }
    else
    {
      print " (undefined)\n";
    }
    return ( 1, $current );
  }
  elsif ( $operator eq 'CLEAR' )
  {
    return ( 1, 0 );
  }
    elsif ( $operator eq 'EXIT' )
  {
    exit;
```

```
  }
  return;
}

sub dyadic
{
  my ( $operator, $operand, $current) = @_;

  if ( $operator eq 'PLUS' )
  {
    return ( 1, $current + $operand );
  }
  elsif ( $operator eq 'MINUS' )
  {
    return ( 1, $current - $operand );
  }
  elsif ( $operator eq 'TIMES' )
  {
    return ( 1, $current * $operand );
  }
  elsif ( $operator eq 'OVER' )
  {
    return ( 1, $current / $operand );
  }
  return;
}
```

Check Syntax and run it. Enter information as shown to make sure that it works:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./calculate.pl
> EQUALS
 (undefined)
OK
> CLEAR
OK
> EQUALS
 = 0
OK
> PLUS 42
OK
> EQUALS
 = 42
OK
> OVER 7
OK
> EQUALS
 = 6
OK
> TIMES 3
OK
> EQUALS
 = 18
OK
> MINUS 3
OK
> EQUALS
 = 15
OK
> explode
Invalid statement
> EXIT
```

This program demonstrates all of these topics:

- Proper preamble
- Scalar declaration and initialization
- Interpolation of scalars and interpretation of escape sequences in double-quoted strings
- Single-quoted strings
- Arithmetic operators and the numeric equality operator
- String testing with **eq**, **index()** and **length()**
- String manipulation with **substr()**
- Conditional statements and boolean operators
- Input from standard input and removing any trailing newline
- Defining, calling, and passing lists to and from subroutines
- Printing to standard output and standard error

Can you locate and understand each element within the code?

So now you've gotten a good sense of where we're starting from in this course. If you don't understand the topics we've gone over, contact your instructor to figure out which is the best route for you to take to become a Perl expert.

Postfixed Modifiers

Now on to something new! We'll start with **postfixed statement modifiers.** They fit into the category of what computer linguists call *syntactic sugar*, meaning that there's always another way of doing what they provide, but they can give you a more visually pleasing and succinct syntax for that operation...in other words, something sweeter.

In Perl, when only one statement is to be executed conditionally, you can not only leave off the curly braces, you can omit the parentheses around the condition. All you need to do is to put the **if** clause *after* the statement!

Whoa. That's a little weird if you've never encountered it before. What does it look like? Here's an example. Instead of writing:

OBSERVE: if statement

```
if ( $car_type eq 'hybrid' )
{
  print "Check remaining battery life\n";
}
```

You can write this:

OBSERVE: postfixed if modifier

```
print "Check remaining battery life\n" if $car_type eq 'hybrid';
```

You may be thinking to yourself, "But that's backwards! The computer can't run the **print** statement before it's tested the contents of **$car_type**!" *And you're right*; the computer doesn't do that. Perl transforms the postfixed **if** statement into the regular form of **if** statement (where the **if** comes first) before it executes the statement. This is why the postfixed form is called "syntactic sugar"... the lower levels of Perl aren't even aware that it exists. That's pretty sweet, huh?

You might now be thinking, "Okay, but it still doesn't make sense! Why write code in a form that pretends the computer can do things backwards? It's like looking at the code upside-down."

Well, we do it for the convenience of the people writing and reading the code. Convenience for the computer comes in a distant second to that of humans—power to the people! In fact, sometimes we do speak, or at least think like the postfixed **if** statement:

- "Go to the grocery store and get me a bag of flour to make cookies, if they also have chocolate chips."
- "Put the trash out tonight, if it's not raining."
- "The Federal Reserve is expected to raise interest rates tomorrow, unless the quarterly employment report is bad."

Sometimes the real world doesn't provide a postfixed conditional where it should. Consider the instructions on most shampoo bottles: "Lather, rinse, repeat." Nowhere does it tell you when to *stop*...

There is a postfixed **unless** modifier as well:

OBSERVE: postfixed unless modifier

```
die "Invalid location\n" unless $state{$abbreviation};
```

If the full statement would exceed the maximum line length you've allotted, then insert a line break and indent:

OBSERVE: postfixed if modifier

```
print "Check remaining battery life\n"
  if $car_type eq 'hybrid' || $device_type eq 'cellphone';
```

Let's try out some postfixed conditional modifiers on the **calculate.pl** program from our earlier Perl 1 review. Edit **calculate.pl** as shown (if you skipped the Perl 1 review, create the program as shown below):

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $INVALID = "Invalid statement\n";
my $Accumulator;

while ( print( "> " ) && defined( my $line = <> ) )
{
  chomp $line;
  my $ok;
  ($ok, $Accumulator) = execute( $line, $Accumulator );
  if ( $ok )
  {
    print "OK\n";
  }
  else
  {
    warn $INVALID;
  }
}

sub execute
{
  my ($line, $current) = @_;

  if ( length $line == 0 )
  {
    return;
  }
  return if length $line == 0;

  my $operator_length = index $line, ' ';
  if ( $operator_length < 0 )
  {
    return monadic( $line, $current );
  }
  else
  {
  return monadic( $line, $current ) if $operator_length < 0;

  my $operator = substr $line, 0, $operator_length;
  my $operand = substr $line, $operator_length + 1;
  return dyadic ( $operator, $operand, $current );
  }
}

sub monadic
{
  my ($operator, $current) = @_;

  if ( $operator eq 'EQUALS' )
  {
    if ( defined $current )
    {
      print " = $current\n";
    }
    else
    {
      print " (undefined)\n";
    }
    return ( 1, $current );
  }
  elsif ( $operator eq 'CLEAR' )
  {
    return ( 1, 0 );
```

```
  }
  elsif ( $operator eq 'EXIT' )
  {
    exit;
  }
return;
}

sub dyadic
{
  my ( $operator, $operand, $current) = @_;

  if ( $operator eq 'PLUS' )
  {
    return ( 1, $current + $operand );
  }
  elsif ( $operator eq 'MINUS' )
  {
    return ( 1, $current - $operand );
  }
  elsif ( $operator eq 'TIMES' )
  {
    return ( 1, $current * $operand );
  }
  elsif ( $operator eq 'OVER' )
  {
    return ( 1, $current / $operand );
  }
  return ( 1, $current + $operand ) if $operator eq 'PLUS';
  return ( 1, $current - $operand ) if $operator eq 'MINUS';
  return ( 1, $current * $operand ) if $operator eq 'TIMES';
  return ( 1, $current / $operand ) if $operator eq 'OVER';
  return;
}
```

Check Syntax and run it. Test it as shown:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./calculate.pl
> EQUALS
 (undefined)
OK
> CLEAR
OK
> EQUALS
 = 0
OK
> PLUS 42
OK
> EQUALS
 = 42
OK
> OVER 7
OK
> EQUALS
 = 6
OK
> TIMES 3
OK
> EQUALS
 = 18
OK
> MINUS 3
OK
> EQUALS
 = 15
OK
> explode
Invalid statement
> EXIT
```

Note

In general, you cannot replace an **if** block containing an **else** clause with a postfixed conditional; postfixed conditionals support only one outcome. We are able to replace many of the **else** and **elsif** clauses in the program with postfixed conditionals though, because the only statement currently present in their blocks is a **return** statement, which means that no additional code will be executed after the block anyway. It's up to you to decide whether you prefer this form to the previous one; remember, postfixed conditionals are optional and should only be used when they make the code clearer to you! I decided against using one in one instance above, because it would have spoiled the symmetry of the surrounding code.

Naked Blocks

Here's one more small new thing to introduce in this lesson: the (ahem) "naked block." You've seen blocks preceded by an **if** statement, blocks preceded by a **foreach** statement, and blocks preceded by a **while** statement. But a naked block is not preceded by *any* statement:

OBSERVE: naked block example

```
{
  my $label = get_label( $report_file );

  add_title( "Report for $date for region $label" );
}
```

Functionally, there is *no difference* between that code and the same code without the braces... so what's the point? One answer lies in *scoping*. Remember that the scope (the parts of a program where you can legally refer to something) of a lexical variable (one declared with **my**) is from the word 'my' until the next closing brace (or end of file if that comes first). An important goal of code development is to *make the scope of each variable as small as possible*. That reduces

the chance of accidentally using the wrong variable. See what happens with this example: create **naked.pl**:

naked.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $label = get_label( "report_file" );
add_title( $label );
sub get_label {
 return "imaginary label"
}
sub add_title
{
 print "Adding title of $_[0]\n";
}

# Pretend this is much later in the program...

$label = 'Nothing to do with the previous $label';
```

See how that program compiles and runs okay? Now change it:

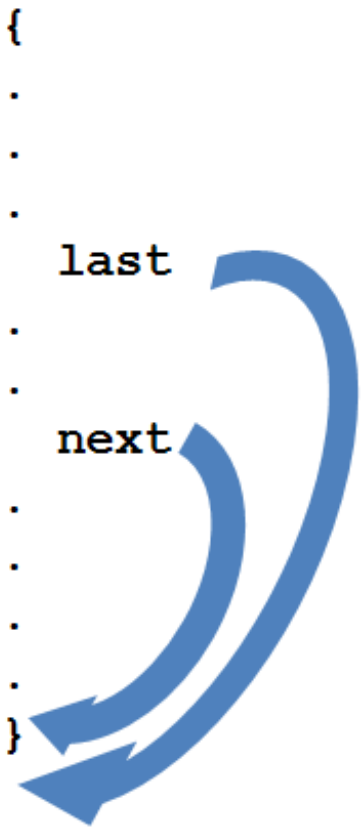
naked.pl

```
#!/usr/bin/perl
use strict;
use warnings;

{
 my $label = get_label( "report_file" );
 add_title( $label );
}
sub get_label {
 return "imaginary label"
}
sub add_title
{
 print "Adding title of $_[0]\n";
}

# Pretend this is much later in the program...

$label = 'Nothing to do with the previous $label';
```

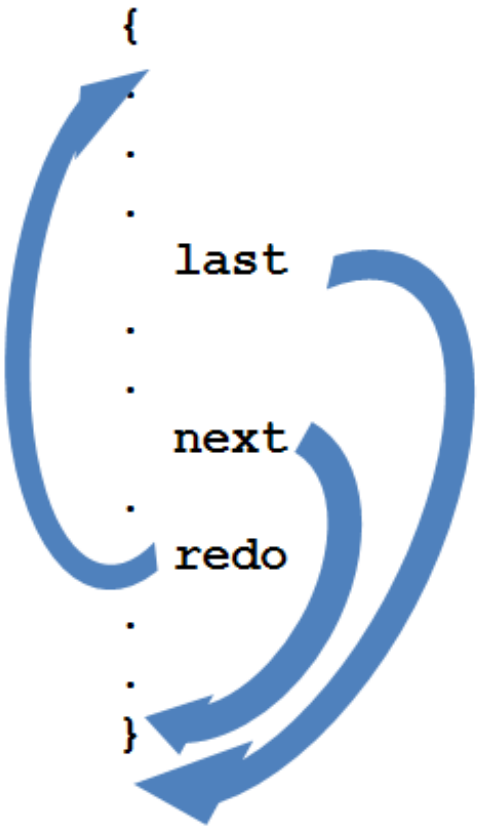
Save and run that program, or try to. See how it won't compile?

So a naked block lets you restrict the scope of a variable even when there's no **if**, **while**, or **foreach** involved. In the example above, the variable **$label** is needed only long enough to pass it to the routine **add_title**, so we placed it in a block that was only that long. The world is full of Perl programs that start out with immense lists of declarations of variables that only get used on a few lines much later in the code. These programs are hard to read and hard to maintain. Use scoping to make your programs better!

There's one more thing you can do with a naked block, although it's not very common in well-designed programs. Remember the loop control statements **next** and **last**? Well, they work inside a naked block as well. One way to remember them is that **last** jumps to right *after* the closing curly brackets, whereas **next** jumps to right *before* the closing curly brace:



In the case of a naked block, both **last** and **next** will have exactly the same effect, because the block is not a looping construct that will repeat. But there is a block control statement that *can* be useful in this case: **redo**. **redo** jumps back to right after the *opening* curly brace. Take a look:



It literally re-does the block. If you're in a looping block, it will cause the current iteration of the loop to be performed

again (without testing the condition in a **while** statement or taking the next list element in a **foreach** statement). In a naked block, **redo** will cause the block to be executed again. This doesn't provide any behavior you can't get with a **while** statement, but—remembering Perl's motto, "There's More Than One Way To Do It"—it can sometimes provide a more natural or intuitive way of solving a problem. Check out this example:

OBSERVE: redo in naked block

```
{
  print "Enter password: ";
  $password = read_password_from_terminal();
  redo unless length $password;
}
```

You *could* write that using a **while** or **until** loop, in fact, go ahead and try that right now. It's not necessarily worse than using **redo** like we did in the above example; just see which version you prefer. Perl is all about having choices!

Phew! That was a lot of information packed into one lesson, but we needed a healthy jump-start to get going. We have much to cover! See you in the next lesson!

Once you finish with the lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

Opening Filehandles

Lesson Objectives

When you complete this lesson, you will be able to:

- open specific files more efficiently using *short-circuiting*.
- open a file.
- write and overwrite a datafile.

Welcome to lesson two! In this lesson, we'll look at how to open specific files by a much more efficient mechanism than naming them on the command line or playing games with **@ARGV**. But first, we'll see how there is yet another way to express a conditional statement (don't worry, this is directly related to this lesson's topic, as you will soon see).

Yet Another Conditional

As we discussed in the last lesson, the postfixed conditional modifiers may *look* like they're the wrong way around, but they really aren't; and sometimes we talk that way anyway ("Make me a sandwich, if you have the time"), so postfixed modifiers can be a natural way of expressing yourself in Perl.

But if you still don't like the look of clauses in that order, you don't have to use postfixed modifiers. Remember Perl's motto: "There's More Than One Way To Do It" (TMTOWTDI). There is a *third* option. Remember the **&&** operator, meaning "and"?

&& example

```
if ( $count > 1 && $item eq "cereal" )
```

Let's investigate; create **test_and.pl** as shown:

CODE TO TYPE: test_and.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $count = 2;
if ( $count < 1 && item_test() )
{
  print "Don't expect to see this\n";
}
else
{
  print "Failed conditional\n";
}

sub item_test
{
  print "Don't expect to see this either\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~$ cd perl2
cold:~/perl2$ ./test_and.pl


Failed conditional
cold:~/perl2$
```

Perl didn't even *call* the **item_test** subroutine to find out its value—it didn't need to know!

We call this *short-circuiting*; Perl does not evaluate the right side if it would make no difference to the result of the expression. If **$count** is not less than 1, the result of **item_test()** is irrelevant, so Perl saves time and doesn't bother; it just returns false as the result of the whole expression straight away.

Short-circuiting causes **&&** to behave just the same as an **if** statement would; if the left side (condition) is true, then evaluate the right side (conditionally executed block). Even though Perl computes a value for the **&&** expression, we don't have to use it. So we could write our postfixed example from the previous lesson as:

OBSERVE: && for control flow

```
$car_type eq 'hybrid' && print "Check remaining battery life\n";
```

How about that! But there is one change we should consider. There's another form of the **&&** operator that means *exactly* the same thing: **and**. The change might seem insignificant at first, but there is a crucial difference between the two options here. The **&&** operator has a relatively high *precedence*, which means that Perl will evaluate the **&&** operator before many other operators in an expression. (Precedence is the reason that **4 + 5 * 3** evaluates to 19 rather than 27; the ***** has a higher precedence than the **+** and hence **5 * 3** is computed before **4 +**.)

But the **and** operator has a very *low* precedence. In fact, nothing comes lower than **and**. So we'll want to use **and** instead of **&&** to control flow (that is, as an alternative to an **if** statement), otherwise we could get into trouble. Suppose we were writing an **&&** version of this statement:

OBSERVE: if example

```
if ( length $title > 1 )
{
  $print_title = $TRUE;
}
```

If we write that as:

OBSERVE: && example

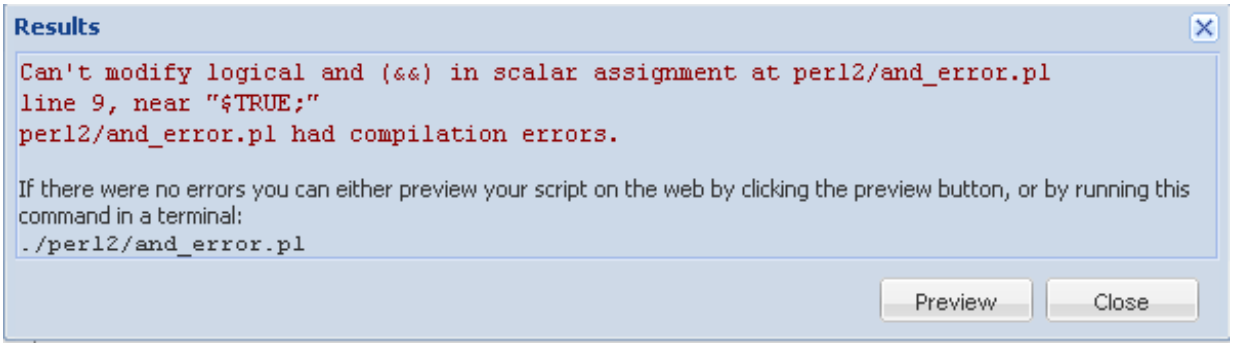
```
length $title > 1 && $print_title = $TRUE;
```

...we get a compilation error. Try it now to see what the error is; create **and_error.pl** as shown:

CODE TO TYPE: and_error.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $TRUE = 1;
my $title = "Intermediate Perl";
my $print_title;

length $title > 1 && $print_title = $TRUE;
```

Check Syntax, saving it in your **/perl2** folder as **and_error.pl**:



The problem lies in the high precedence of **&&**. (Of course, we could fix that problem by putting in parentheses, but if you follow the rule you're about to learn, you won't have to.) Modify **and_error.pl** as shown:

CODE TO EDIT: and_error.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $TRUE = 1;
my $title = "Intermediate Perl";
my $print_title;

length $title > 1 and $print_title = $TRUE;
```

Check Syntax again. Everything is fine now, because and gets evaluated "late." Now you can write the original example like this:

OBSERVE: and instead of postfixed if

```
$car_type eq 'hybrid' and print "Check remaining battery life\n";
```

The rule of thumb is, use **&&** and **||** in expressions (where a value is being calculated); use **and** and **or** for changing control flow (that is, instead of lengthier forms of **if** and **unless**).

To find out more about precedence, run **perldoc perlop** and look at the table in the section on Operator Precedence and Associativity.

open()

"You cannot open a book without learning something." -Confucius

Now we'll look at how to open a file. This is an important topic. Virtually every program of any utility will require that you read information into it from some source and write information from it to some destination. The function in Perl that lets us do both is called **open()**. I recommend that you do *not* read the **perldoc -f** documentation on **open()** yet, because it could confuse and overwhelm you. **open()** can be used for opening things other than files, and for opening things in multiple ways; the **perldoc -f** documentation covers *all* of them. Save that documentation for another time. The appropriate documentation section to read for this lesson is **perldoc perlopentut**.

Filehandles

Think about what it takes to read from a file. Every time you ask for more input, Perl has to know where it left off. Perl also has to remember other, more esoteric, information to do with things like buffers and such. You don't want to bother with all of that; you just want to get input. The way Perl remembers everything that it has to know, without bothering us with it, is to hide it all behind something called a *filehandle*. A filehandle *looks* like a scalar, but you can't do the usual scalar-ish kinds of things with it (well, you can try, but the results won't be helpful); you can only use it in file reading operations.

The same is true for outputting to a file, so we use filehandles for that as well. We create a filehandle with **open()**. There's another important reason for using **open()**; it gives us a convenient way to find out whether we're going to be successful at our task. If we're going to read from a file, we ought to find out first of all whether it exists. But suppose we checked for that, got the answer "yes," and then went to read from the file, but in between those two operations, someone deleted the file? Hmm. Fortunately, **open()** acts as a kind of contract with the computer; if it is successful,

you won't have to worry about someone deleting the file, because you got there first, and you'll be able to read from the file until you're done. Some operating systems achieve this by telling anyone who tries to delete a file that you've opened, that they're not allowed to delete it; some operating systems do it by deleting the file in name, but letting you continue to read the data from it until you're done, and only then releasing the space the file occupies. With me so far? Good! Similarly, if you're going to be writing to a file, you'd like to know in advance whether you have permission to do so, as that file could be located in a directory you're not allowed to change.

So that's why we have filehandles. We'll make some in a minute. First, let's go over the basic form of **open()**:

OBSERVE: generic form of open()

```
boolean = open filehandle, mode, filename
```

This is called the *three-argument* version of **open()**. There are other forms of the **open()** function that have two arguments or even one argument, but we won't be addressing them in depth in this course. The three-argument version is best for many reasons, primarily improved security. You'll still see programs using the two-argument form in particular, because that version was widely used before the three-argument version became available in 2000. We want to learn current techniques though, so we'll stick to the *three-argument* version of **open()**.

open() is a function like **print()**, where we typically leave the parentheses off of the arguments. Put the parentheses on if you like, but most people leave them off, most of the time.

Alright then, let's start creating an example and I'll explain it in detail as we go. Create **make_datafile.pl** in your /perl2 folder, as shown:

CODE TO TYPE: make_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

my $fh;
if ( open $fh, '>', $DATA_FILE )
{
  print {$fh} qq{2'02" Sgt. Pepper's Lonely Hearts Club Band\n};
  print {$fh} qq{2'44" With A Little Help From My Friends\n};
  print {$fh} qq{3'29" Lucy In The Sky With Diamonds\n};
  close $fh;
}
else
{
  die "Couldn't open $DATA_FILE: $!\n";
}
```

Note Those curly braces around **$fh** in the **print** statements are not typos! There's something unusual going on there that we'll explain fully later on.

Check Syntax and run it. A text file named **songs.data** appears in the folder (type the **ls** command to list the files there). To see the contents of the file, type this in the Unix shell:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ cat ./songs.data


2'02" Sgt. Pepper's Lonely Hearts Club Band
2'44" With A Little Help From My Friends
3'29" Lucy In The Sky With Diamonds
cold:~/perl2$
```

Let's take a closer look at this program:

OBSERVE: make_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

my $fh;
if ( open $fh, '>', $DATA_FILE )
{
  print {$fh} qq{2'02" Sgt. Pepper's Lonely Hearts Club Band\n};
  print {$fh} qq{2'44" With A Little Help From My Friends\n};
  print {$fh} qq{3'29" Lucy In The Sky With Diamonds\n};
  close $fh;
}
else
{
  die "Couldn't open $DATA_FILE: $!\n";
}
```

The **qq** operator is another way of writing double quotation marks ("**qq**" is supposed to remind you of that). The character immediately following **qq** is the delimiter for the string. If it's a left character of a mirrored pair (like **()**, or **<>**, **{}**), the closing delimiter for the string will be the right character. I used it in this example so I wouldn't have to escape the **"** character inside each string.

A filehandle is a scalar, but it's different from scalars we've encountered before. It's not a number or a string. You can't do anything with it except use it to read from files opened for reading or write to files opened for writing. Trying to increment one is pointless. The **open()** call will *assign* to the filehandle even though it's one of the arguments to **open()** (that's okay though; you can write subroutines that overwrite their arguments, although it's not usually good style). In our example, we've created the filehandle **$fh**.

A filehandle should be a *lexical variable* (one you declared with **my**) that is undefined. You can make sure of that and save a line of code at the same time by declaring the filehandle *inside* the **open()** call! This looks weird at first, but you can put **my** before the first appearance of a variable just about anywhere, not just on a line by itself. Let's try that. Modify **make_datafile.pl** as shown:

CODE TO EDIT: make_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

~~my $fh;~~
if ( open my $fh, '>', $DATA_FILE )
{
~~  print {$fh} qq{2'02" Sgt. Pepper's Lonely Hearts Club Band\n};~~
~~  print {$fh} qq{2'44" With A Little Help From My Friends\n};~~
~~  print {$fh} qq{3'29" Lucy In The Sky With Diamonds\n};~~
  print {$fh} << 'EOF';
2'02" Sgt. Pepper's Lonely Hearts Club Band
2'44" With A Little Help From My Friends
3'29" Lucy In The Sky With Diamonds
2'48" Getting Better
EOF
  close $fh;
}
else
{
  die "Couldn't open $DATA_FILE: $!\n";
}
```

Check Syntax and run it, and **cat ./songs.data** again. The previous data file was overwritten (with the same contents, plus the track we added). Here's what we did:

OBSERVE: make_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

if ( open my $fh, '>', $DATA_FILE )
{
  print {$fh} << 'EOF';
2'02" Sgt. Pepper's Lonely Hearts Club Band
2'44" With A Little Help From My Friends
3'29" Lucy In The Sky With Diamonds
2'48" Getting Better
EOF
  close $fh;
}
else
{
  die "Couldn't open $DATA_FILE: $!\n";
}
```

We collapsed all the **print** statements into a single one, using a heredoc (**<<**).

The result of the **open()** call is true if the file can be opened successfully. In this case, the file songs.data is opened for output (that's what the second parameter, the string **'>'** means; we'll see how to do input shortly). The third parameter (**$DATA_FILE**) refers to the file name.

I said I'd explain the curly braces around **{$fh}** in the **print** statements. The way we output to a filehandle (and it must be one that is open for writing, not reading, or we'll get an error message) is with the print statement. But because print sends everything to the standard output by default (technically, it sends it to the currently selected filehandle, which by default is the standard output), there must be a way to tell print to use a different filehandle. We do that with a special "argument," which is the filehandle inside curly braces (**{$fh}**). It's not an argument in the usual sense, because there's *no comma* after it; if you add a comma by mistake, Perl will just turn the filehandle into a (funny-looking) string and print it to the standard output. The braces are optional, by the way, but it's a best practice for readability to use them, so we always will.

The **$!** variable is a special variable in Perl, like **@ARGV**. If an **open()** call fails (returns false), **$!** is set to the text of the reason for the failure.

We'll almost always want to call **die** when an **open()** call fails; few programs can keep going constructively after being unable to open a file. You saw in the last lesson how to collapse a whole **elsif** clause using the **and** form of conditional when the statement in the block caused a change of control flow. We can do the same in this program. Modify **make_datafile.pl** as shown:

CODE TO EDIT: make_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

if ( open my $fh, '>', $DATA_FILE )
{
open my $fh, '>', $DATA_FILE or die "Couldn't open $DATA_FILE: $!\n";
print {$fh} <<'EOF';
2'02" Sgt. Pepper's Lonely Hearts Club Band
2'44" With A Little Help From My Friends
3'29" Lucy In The Sky With Diamonds
2'48" Getting Better
2'37" Fixing A Hole
EOF
close $fh;
}
else
{
  die "Couldn't open $DATA_FILE: $!\n";
}
```

Check Syntax and run it, and look at songs.data—again, the output is the same except for the new track.

We can do one more thing to make this program more concise. The **close()** function is the converse of **open()** (read the **perldoc -f** documentation on it if you like). As its name suggests, **close()** causes the file to close; you can no longer use the filehandle without getting an error message. In particular, all pending output to files opened for writing is flushed. (If you look at an output file as it's being written by another program running at the same time, you probably won't see the output show up until some time after the program has printed it. It's buffered by Perl or the operating system and written in chunks to improve efficiency.)

If the lexical variable holding the filehandle is destroyed, Perl calls **close()** right then and we don't need to call it explicitly. (After all, if the filehandle is no longer available to your program, you can't do anything more with it.) We can cause a lexical variable to be destroyed by limiting its scope. This is a good use for a naked block. Modify **make_datafile.pl** as shown:

CODE TO EDIT: make_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

{
  open my $fh, '>', $DATA_FILE or die "Couldn't open $DATA_FILE: $!\n";
  print {$fh} <<'EOF';
2'02" Sgt. Pepper's Lonely Hearts Club Band
2'44" With A Little Help From My Friends
3'29" Lucy In The Sky With Diamonds
2'48" Getting Better
2'37 Fixing A Hole
3'35" She's Leaving Home
EOF
}
close $fh;
```

Check Syntax and run it, and **cat ./songs.data**. Once again, the output is the same except for the new track. In this example, that makes no difference technically, because the program ends at that point and all open filehandles are closed upon program exit anyway. But it's still a good practice to limit the scope of a filehandle the same way you'd limit the scope of any other variable.

Every time you open a file, you should check to see whether the **open()** succeeded. Every Perl programmer does this

using **open ... or die ...**. So as you can see, the third alternative form of conditional syntax (... and/or ...) is part of what is likely the most common idiom in Perl.

Let's see what happens if the **open()** fails. At the Unix shell prompt, type the command shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ chmod -w ./songs.data
```

Now Check Syntax and run it and see what happens.

To change the rights back, type:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ chmod +w ./songs.data
```

Reading from filehandles

In the Perl 1 course, we discovered the **<>** operator, which reads from files named in **@ARGV**. But guess what? That's actually a *special case* of something more general that we'll learn about now. We'll write another program that will read the **songs.data** file and total the song lengths. Let's call this one **read_datafile.pl**. Type the code below as shown:

CODE TO TYPE: read_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

open my $fh, '<', $DATA_FILE or die "Couldn't open $DATA_FILE: $!\n";
while ( my $line = <$fh> )
{
  chomp $line;
  my $minute_pos = index $line, "'";
  my $second_pos = index $line, '"';
  my $minutes = substr $line, 0, $minute_pos;
  my $seconds = substr $line, $minute_pos + 1, $second_pos - $minute_pos - 1;
  my $running_time = $minutes + $seconds / 60;
  my $title = substr $line, $second_pos + 2;
  print "$title lasts $running_time minutes\n";
}
```

Check Syntax and run it. Now let's take a closer look:

OBSERVE: read_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

open my $fh, '<', $DATA_FILE or die "Couldn't open $DATA_FILE: $!\n";
while ( my $line = <$fh> )
{
  chomp $line;
  my $minute_pos = index $line, "'";
  my $second_pos = index $line, '"';
  my $minutes = substr $line, 0, $minute_pos;
  my $seconds = substr $line, $minute_pos + 1, $second_pos - $minute_pos - 1;
  my $running_time = $minutes + $seconds / 60;
  my $title = substr $line, $second_pos + 2;
  print "$title lasts $running_time minutes\n";
}
```

The **<>** operator (which we call the *readline* operator) in this program is taking an argument, namely, **$fh**. It works just like the empty form of **<>**, only without the magical behavior of opening successive files named on the command line. It returns the next line from the filehandle **$fh** in a scalar context, and all of the remaining lines in a list context.

If you took the Perl 1 course and you've been paying exceptionally close attention, you may wonder why the **while** line does not wrap the condition in a **defined** test. Isn't this asking for trouble, if the file contains a line with a string value that would be false? Granted, there is only one such possible value—a line containing precisely the character 0 (zero) and not even a newline on the end, which means it would have to be the last line in the file, but still, we don't want to miss any possible lines.

In fact, Perl looks at the condition of a **while** statement, and if it consists of an assignment from the readline operator, then it *wraps the condition in a call to **defined** for you.* Pretty helpful, huh? In a few lessons it will get even better!

Also, in this program we had to find the position of the single and double quote marks that mark the end of the minutes and seconds respectively in each line, and figure out how far away from those positions to extract the substrings we needed. (This is necessary because the input data is not in fixed-width columns.) That's clumsy, but there's a better way of doing it that we'll be learning about in this course.

Check out a common error condition now, so you know what it looks like. Delete the file **songs.data** (don't worry, you can get it back any time by rerunning **make_datafile.pl**) and then rerun **read_datafile.pl**. Make sure you get get an error and that you understand it.

open() modes

You've seen that to open a filehandle to read from, you specify a mode of **'<'**; and that to open a filehandle to write to, you specify a mode of **'>'**. There are many other possible modes, and they are detailed in the **perldoc -f open** documentation, but be forewarned—most of them are hard to understand and harder still to use correctly. One of the more accessible modes is *append*, which is denoted by **'>>'**. If the file does not already exist, it will be created; if it already exists, the output will be added to the end of its current contents.

Let's give that a test by modifying **make_datafile.pl** as shown:

CODE TO EDIT: make_datafile.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $DATA_FILE = 'songs.data';

{
  open my $fh, '>>', $DATA_FILE or die "Couldn't open $DATA_FILE: $!\n";
  print {$fh} <<'EOF';
2'02" Sgt. Pepper's Lonely Hearts Club Band
2'44" With A Little Help From My Friends
3'29" Lucy In The Sky With Diamonds
2'48" Getting Better
2'37" Fixing A Hole
3'35" She's Leaving Home
EOF
}
```

Check Syntax and run it a few times, and see what happens in **songs.data**.

Excellent. Congratulations on getting through this new and important topic! We'll build on what we've learned here as we go through this course. See you in a bit...

Once you finish the lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

Perl's Default Variable

Lesson Objectives

When you complete this lesson, you will be able to:

- use a *default* variable.
- determine whether a value is defined before you test its length.
- use the *predefined* filehandles.

The Wonderful World of $_

This morning, I had cereal for breakfast. First I got the cereal out, then I put the cereal in a bowl, then I added milk to the cereal, and then I put some fruit on top of the cereal. Finally, I ate the cereal.

What's wrong with that paragraph? It needlessly repeats the word "cereal" everywhere, as though it were an advertisement crafted by the Council on Carbohydrates. People don't talk like that. We'd be more likely to say something like "This morning, I had cereal for breakfast. First I got it out, then I put it in a bowl, then I added milk, and then I put some fruit on top. Finally, I ate it." In other words, we use the *pronoun* "it" to avoid repeating something that's understood, and sometimes we can even leave out the pronoun as well, where the context makes clear what we're referring to.

Perl has something similar to the pronoun "it"—a *default* variable that is the implied target of many operators and built-in functions in Perl. If you do not specify a variable, the functions and operators are executed on the default variable. That variable is **$_**, and it is the single most important variable in Perl. You *must* know explicitly which operators and functions use $_; do not assume or guess. We'll go over the most common uses of **$_** in this course.

Let's get started by taking a look at some examples of those operators that commonly use **$_**:

print(): The default thing to print is **$_**:

OBSERVE: print and $_

```
print;
print {$fh};
```

The first statement above will print whatever is in **$_** to the currently selected filehandle (by default, standard output). The second statement prints whatever is in **$_** to the filehandle **$fh**.

foreach(): The default loop variable in foreach() statements is **$_**:

OBSERVE: foreach and $_

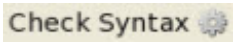
```
foreach ( @fruit )
{
  # Do something with $_
}
```

Technically, this is not the same as **foreach my $_ (@fruit)** because you can't say **my $_**—the variable does not belong to you, it belongs to Perl. But you can think of it that way. (Alright, in the very latest versions of Perl you *can* say **my $_**, but don't do it until you know a whole lot more about variables—for now just trust me on that.)

To illustrate what you can expect in terms of the scoping of **$_** as the implicit variable in a foreach loop, create **foreach.pl**. Type the code below as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = "Outer\n";

print;

foreach ( "  Inner1\n", "  Inner2\n", "  Inner3\n" )
{
  print;
}

print;
```



and run it.

The output illustrates that the previous ("Outer") value of **$_** was restored after the loop.

defined() and length(): Two of the functions that operate on a single scalar and use **$_** by default.

It's not uncommon to use them both in a single expression. Suppose you want to know whether a variable (let's say, $input) is not the empty string. You might start out by writing **stringtest.pl** as shown:

CODE TO TYPE: stringtest.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $input = '';
if ( $input )
{
  print "\$input is not the empty string\n";
}
else
{
  print "\$input is the empty string\n";
}
```

But that's no good, because the string "0" has a positive length (of one), yet it is false. Modify **stringtest.pl** as shown:

CODE TO TYPE: stringtest.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $input = '0';
if ( $input )
{
  print "\$input is not the empty string\n";
}
else
{
  print "\$input is the empty string\n";
}
```

Whoops. So next you might try this modification to your **stringtest.pl** code:

CODE TO TYPE: stringtest.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $input = '0';
if ( length $input )
{
  print "\$input is not the empty string\n";
}
else
{
  print "\$input is the empty string\n";
}
```

Hmm. It's still not enough, because it's possible that **$input** has never been initialized and still contains **undef**. Okay then, modify **stringtest.pl** as shown:

CODE TO TYPE: stringtest.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $input~~ = '0'~~;
if ( length $input )
{
  print "\$input is not the empty string\n";
}
else
{
  print "\$input is the empty string\n";
}
```

That time you got a warning "Use of uninitialized value $input in length." You don't want any warnings, so you need to test to determine whether the value is defined before you test its length. You can do that by modifying your code as shown:

CODE TO TYPE: stringtest.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $input;
if ( defined( $input ) && length( $input )~~length $input~~ )
{
  print "\$input is not the empty string\n";
}
else
{
  print "\$input is the empty string\n";
}
```

(This time we added parentheses for readability, but it works just the same without them.)

If the variable you're testing is **$_**, you can write this:

OBSERVE: testing $_

```
if ( defined && length )
{
  do_something();
}
```

The alternative postfixed conditional form of that would be:

OBSERVE: testing $_

```
do_something() if defined && length;
defined && length and do_something();
```

More Postfixed Modifiers

Now that we've been introduced to **$_** and the postfixed conditionals, let's talk about a new postfixed statement modifier: **foreach**. Here's its general form:

OBSERVE: postfixed foreach

```
statement foreach list;
```

You don't need curly brackets or parentheses around the list. *But* you cannot specify a loop variable; it is forced to be $_. Let's start out with a quick example. Modify **foreach.pl** as shown:

CODE TO EDIT: foreach.pl

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = "Outer\n";

print;

foreach ( "  Inner1\n", "  Inner2\n", "  Inner3\n" )
{
  print;
}
print foreach "  Inner1\n", "  Inner2\n", "  Inner3\n";

print;
```

Check Syntax and run it.

You get the same result. Like the postfixed conditionals, this statement is *syntactic sugar* for the statement we had earlier that contained the curly braces and parentheses. You don't have to use this form of **foreach**; you can always stick to the longer one if you want, even when there's only one statement in the block. Here's another example:

OBSERVE: postfixed foreach

```
print "$_: $count{$_}\n" foreach sort keys %count;
```

That one is so common that it's an idiom in its own right, and you'll use it in a program later in this lesson. The most common thing to want to do with a hash is to dump it out in some predictable ordering so you can look at its contents. You'll likely use a line like that frequently (if only during development) so you can verify the contents of a hash. (The predictability comes from using **sort** on the list of keys, which are otherwise returned in an effectively random order from **keys**.)

While we're learning the postfixed **foreach**, let's also look at the postfixed **while**. Its general form is:

OBSERVE: postfixed while

```
statement while condition;
```

Again, no braces, and no parentheses necessary around the condition. As you might have guessed, this also is syntactic sugar for the longer form. There is also a postfixed **until**:

OBSERVE: postfixed until

```
statement until condition;
```

Let's look at an example that copies the standard input to the standard output, prefixed with line numbers. Create **number.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line;
my $count = 1;
printf "%5d %s", $count++, $line while defined( $line = <STDIN> );
```

Check Syntax and run it. Now type the code below into the Unix Window that's running **number.pl**:

INTERACTIVE TERMINAL SESSION

```
cold:~$ cd perl2
cold:~/perl2$ ./number.pl < ./number.pl
```

Most programs of this type (a text *filter*) use **<>** to read input, but now I want to introduce you to a new, *predefined*, filehandle: **STDIN**. Perl sets this up for you, open to the standard input, so you can read from it. You may wonder why it's a word instead of a scalar, like, for example, $STDIN? It's because STDIN originates from a much earlier version of Perl that didn't have filehandles in scalars. When filehandles in scalars were introduced, the Perl maintainers decided to keep the old name of **STDIN** around so as not to break older programs. They also decided not to introduce new scalar variables like $STDIN, because doing so might break older programs that used that variable for something else.

There are two predefined output filehandles, named **STDOUT** and **STDERR**, opened to the standard output and standard error streams respectively.

Let's wrap this up with one more example. First, copy the file gettysburg to your current directory. This contains the text of Lincoln's *Gettysburg Address*, and it will be interesting input to the program we are about to write. Then, create **word_counter.pl** by typing the code below as shown:

CODE TO TYPE: word_counter.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my %count;
while ( $_ = <> )    # See note below
{
  my @words = split;
  $count{$_}++ foreach @words;
}

print "$_: $count{$_}\n" foreach sort keys %count;
```

Check Syntax and run it in the terminal window as shown below:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./word_counter.pl ./gettysburg.txt
```

Note This program explicitly assigns to $_ in the **while** condition. Don't start using that idiom in your own programs! Within a few lessons, you'll see how to replace that with a much better alternative.

The program counts the number of occurrences of every word in the files named on the command line. You can run it on any text file; *The Gettysburg Address* just happens to provide a data set that's interesting and not too big for our purposes.

Another new element in this example is that there are *no* arguments to the **split** function. The second argument (what to split) defaults to, you guessed it, $_. And the first argument (the token to split on) defaults to white space. So this makes a decent first approximation of a word finder. You wouldn't want to use it in any application that took a linguistic view of the concept of "word," because it does nothing to eliminate punctuation, for example.

And if you think *that* program was succinct, it can be made even shorter by eliminating a temporary variable. Modify your code as shown:

CODE TO EDIT: word_counter.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my %count;
while ( $_ = <> )    # See note above
{
  my @words = split;
  $count{$_}++ foreach @words;
  $count{$_}++ foreach split;
}

print "$_: $count{$_}\n" foreach sort keys %count;
```

Check Syntax and run it in the terminal window as shown:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./word_counter.pl ./gettysburg.txt
```

You'll see the same results.

Some programmers don't like **$_** because they believe that every variable in a program should be visible and explicit. And certainly if you're using **$_**, the big advantage in doing so *is* gained by using it implicitly (in functions and operators that use it as a default). That makes for code that is more readable by virtue of its succinctness (think back to the paragraph about cereal). It defeats the usefulness of **$_** to mention it explicitly, over and over. Still, you usually can't avoid mentioning it here and there. A good rule of thumb is that the number of implicit uses of **$_** in a section of code should exceed the number of explicit uses.

If you don't like the idea of **$_** to begin with, then you don't need to use it. But because **$_** is a very *Perlish* notion (based on patterns of natural language), you *do* need to understand it because you will encounter it in virtually every Perl program ever written. And we will be using it a lot throughout the rest of this course! If you think I'm joking about how Perlish **$_** is, check out this T-shirt design the Portland Perl Mongers considered for 2010:

Once you finish the lesson, go back to the syllabus page by clicking on the page tab above and do the assignments. Good work so far. See you in the next lesson!

Regular Expressions: Introduction

Lesson Objectives

When you complete this lesson, you will be able to:

- embed regular expressions in a Perl program.
- use regular expressions or "regexes."
- integrate *regex metacharacters*.

Introduction to Regular Expressions

"The way is long if one follows precepts, but short... if one follows patterns.">
-Lucius Annaeus Seneca (4 BC - 65 AD)

In this lesson we'll learn about one of the most important fundamental operations in Perl—*regular expressions*. "Perl" doesn't actually officially stand for anything, it's not an acronym. That's why it's not written as "PERL". Though Perl's author, Larry Wall, did once suggest using the acronym "Practical Extraction and Reporting Language." That acronym fits with Perl's reputation for being a great language for parsing and manipulating text and unstructured data. In that respect, the regular expression in Perl and how well it is integrated into the rest of the language, is crucial.

First of all, don't let the phrase "regular expression" scare you. The term was invented by mathematicians before computers ever existed. We may be stuck with the label but not with the dry mathematical theory. We'll leave that part out of our discussion of regular expressions and explain them in such a way that mortals without math degrees can understand.

Secondly, in the same way that mathematicians and physicists like to create formulas where every character stands for something different, like $\mathbf{E = mc^2}$, regular expressions are their own separate language, where every character represents something. You write what amounts to a little program in the regular expression language, which gets embedded in a Perl program, and when Perl runs it, it fires up the *regular expression engine* to decipher and run your regular expressions.

But what do regular expressions *do?* They're used to find patterns in text. You can also use them to find patterns in general data that you wouldn't consider text, but it's easier to talk about regular expressions in terms of parsing text; that's mostly what they're used for anyway.

Speaking of simplifying our discussion, "regular expressions" is a bit of a mouthful. Since we're going to use the term over and over, we'll use the common contraction *regex*.

We'll start out with tasks like determining whether the string "apple" occurs in the text, or whether there's a vowel or a telephone number in it. Later, we'll look at more complicated examples. The regex language includes more features than we will cover in this course, and more than one book has been written just on that language. With all of those powerful features, the regex language can even be used to parse programs written in languages as complicated as Perl.

Literal Matching

Let's get started with the basic type of regex, and see how to embed it in a Perl program. Create **reg_literal.pl** as shown:

CODE TO TYPE : reg_literal.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $text = "I have three apples and four oranges";
$text =~ /apple/ and print "Found an apple\n";

$text = "I have two tomatoes and one zucchini";
$text =~ /apple/ or print "Didn't find an apple\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~$ cd perl2
cold:~/perl2$ ./reg_literal.pl
Found an apple
Didn't find an apple
cold:~/perl2$
```

Our program found an apple in the first string we put in **$text** ("I have three **apple**s and four oranges") and didn't find an apple in the second string we put in **$text** ("I have two tomatoes and one zucchini").

Let's take a closer look at this program:

OBSERVE: reg_literal.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $text = "I have three apples and four oranges";
$text =~ /apple/ and print "Found an apple\n";

$text = "I have two tomatoes and one zucchini";
$text =~ /apple/ or print "Didn't find an apple\n";
```

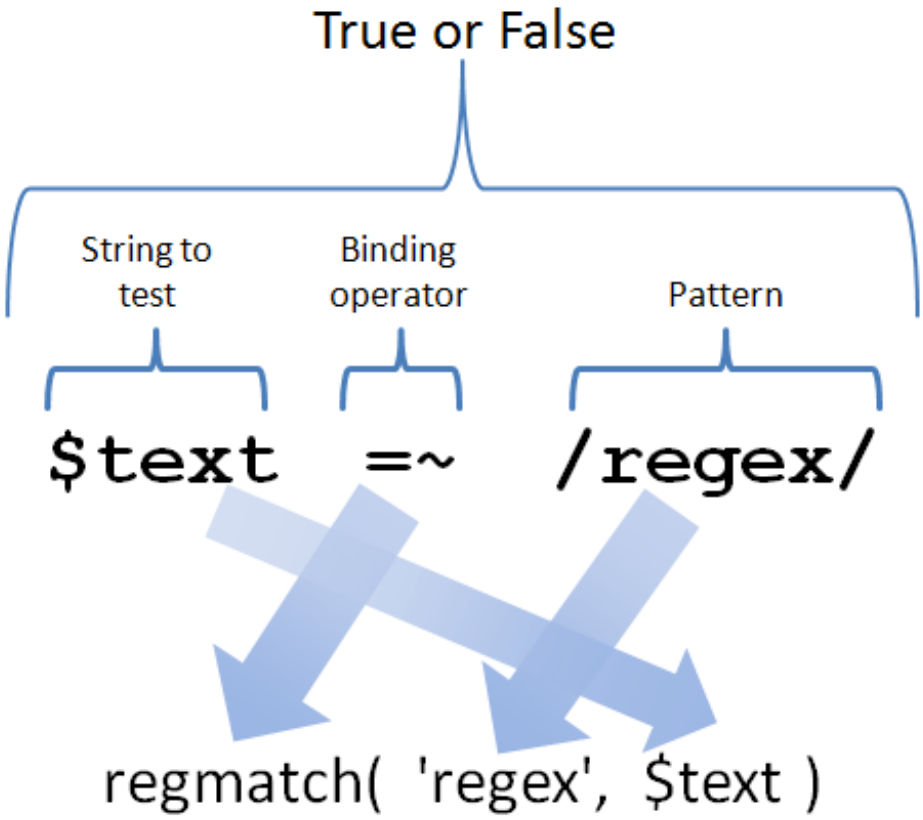
See how we used **and** to mean *if* and **or** to mean *unless*?

Now, imagine that Perl has a function named say, **regmatch()**, that returns true or false, and it works like this:

OBSERVE: hypothetical regmatch() example

```
if ( regmatch( 'apple', $text ) )
{
  # ...
}
```

The regex would be in the first argument, and the string to match it against would be in the second one. That makes sense, right? Many languages implement regular expressions just like this. Perl, however, introduces new syntax so that regexes can appear in the code without the clutter of words like "regmatch" making it hard to read. That funny expression breaks down like this:



The **/apple/** is the regex, and the **=~** is the *binding operator*, which tells Perl that the regex is to be tested against whatever appears to the left of it, in this case, **$text**. It looks like some sort of assignment because of the equals sign, but trust me, there's no assignment going on. **=~** simply means, "I have a regex to my right which is to be matched against the string to my left, and the value of the whole expression **STRING =~ REGEX** is true if the regex matched the string, and false if it didn't."

WARNING	There is no actual **regmatch** function in Perl! I made that up as a hypothetical equivalent to the new syntax for the sake of explanation.

To be ruthlessly precise, technically the regular expression is **apple**. The slashes are delimiters *around* the regular expression. But even with the power of font coloring at our disposal, it's still awkward to refer to a regex in documentation without delimiters around it; without them, readers would have a hard time separating the regex (which usually contains punctuation) from the surrounding text. So now when I refer to "the regular expression, **/apple/**," I trust that you understand that the regex itself is located between the slashes.

Now, what if we want to match a slash? Modify reg_literal.pl as shown:

CODE TO EDIT: reg_literal.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $text = "I have three apples/four oranges";
$text =~ /apples// and print "Found apples/\n";

$text = "I have two tomatoes and one zucchini";
$text =~ /apple/ or print "Didn't find an apple\n";
```

Check Syntax, and you get compilation errors. Because the slash is treated as a delimiter, you can't match on it as well.

You can use different delimiters if you like, delimiters that follow the same rules as **qq**, but if you do, you *must* prefix the first delimiter with the letter **m**. Modify **reg_literal.pl** and see this for yourself:

CODE TO EDIT: reg_literal.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $text = "I have three apples/four oranges";
$text =~ m!apples/! and print "Found apples/\n";

$text = "I have two tomatoes and one zucchini";
$text =~ m(apple) or print "Didn't find an apple\n";
```

Check Syntax and run it.

INTERACTIVE SESSION:

```
cold:~/perl2$ ./reg_literal.pl


Found apples/
Didn't find an apple
cold:~/perl2$
```

This alternate syntax is useful when a regular expression contains a slash; using this approach, the slash does not need to be escaped.

Structure and Execution of Regular Expressions

Now that you've seen a basic example, let's deconstruct it. Remember I said that in a regular expression, every character had a meaning? Most characters—the vast majority—mean something very simple: "Match this character." That's what **/apple/** means: "Match an **a**, then match a **p**, then match another **p**, then match an **l**, then match an **e**." You might be tempted to think that this explanation is overkill, and I could just have said it means, "Match **apple**." But it's *good* to be meticulous when explaining how regular expressions work. If you're going to be a great Perl programmer, you need to learn the basic rules of regexes inside and out.

So **/apple/** causes Perl to look in the string that is bound to the regex via **=~** first for an **a**, scanning each character in the string from left to right (beginning to end), until it finds an **a**. Looking at the first test (against the string **"I have three apples and four oranges"**), the regex engine finds one when it gets to the **a** in **have**, at which point it declares success on that character, and goes on to the next character in the regex, which says that the next thing in the string must be a **p**. But it isn't. So Perl *undoes* the match of the **a** (this action is known as **backtracking**), and moves forward through the string again, looking for another **a**. When it next finds one (in **apples**), it tests each succeeding character in the regex against the next character in the string, and if each of them match in turn, Perl gets to the end of the regex, and is able to return a true value from the expression.

Some characters do not match literally in regexes, which make regexes even more interesting and powerful. Such characters are known as *regex metacharacters*:

OBSERVE: Regex Metacharacters

```
* ( ) + [ ] { } | . ? ^
```

If you want to match any of those literally, you have to escape them with a backslash. You'll learn what they do later, but for now, look at that list closely, and remember that those characters are special. In addition to those characters, the **$** character has special meaning when it's the last character of a regular expression. A regex is like a double-quoted string, so variables interpolate within it. So a **$** that *isn't* at the end of the regex is assumed to be the start of a scalar variable, and an **@** is assumed to be the start of an array variable.

A slash (*/*) character in the middle of a regex needs to be escaped with a backslash (\) if the regex delimiter is a slash. Otherwise, backslashes obey the same rules that they do in double-quoted strings, with a few exceptions that we will discuss later.

That information dump about metacharacters was for those of you who like to know the rules up front. If you're fretting about how to remember all of that, relax; we'll do plenty of examples with explanations to make sure that you get (and

keep) all of it.

The /i Modifier

So far, we haven't done anything with regular expressions that couldn't be done with the **index()** function. That's about to change. Certain letters typed in *after* a regex (after the last slash or delimiter) are called *modifiers* because they modify how the regex engine interprets the regex. The first modifier we'll look at is the **/i** modifier. (Technically, it's the **i** modifier; the **/** is the delimiter, but we'll stick with the convention and call it the **/i** modifier.)

The **i** modifier tells the regex engine to **i**gnore the case of letters in the regex. So **/apple/i** will still match **"apple"**, but will also match **"APPLE"**, **"Apple"**, **"ApPlE"** and all the other combinations of upper and lower case.

Another reason we usually write regexes with the delimiters around them here is to make it clear that we are applying a modifier like **/i** to the regex. (There is, in fact, a way to place something *between* the slashes that has the same effect as **/i**, but it is so hideous that I am not going to show it.)

Modify **reg_literal.pl** as shown:

CODE TO EDIT: reg_literal.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $text = "I HAVE THREE APPLES AND FOUR ORANGES";
$text =~ /apple/i and print "Found an apple\n";

$text = "I have two tomatoes and one zucchini";
$text =~ m{apple}i or print "Didn't find an apple\n";
```

Check Syntax and run it. Look over the use and effect of a modifier with the default delimiters and different delimiters.

That program didn't require the use of **{}** alternate delimiter syntax, but some programmers do so routinely because they like the way it looks. The alternate delimiter syntax is handy when your regex contains slashes. Modify **reg_literal.pl** as follows:

CODE TO EDIT: reg_literal.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $text = "I HAVE THREE APPLES AND FOUR ORANGES";
~~$text =~ /apple/i and print "Found an apple\n";~~
$text =~ /24\/5\/1819/i and print "Found Queen Victoria's birthday\n";

$text = "I have two tomatoes and one zucchini";
~~$text =~ m{apple}i or print "Didn't find an apple\n";~~
$text =~ m{7/7/1776}i or print "Didn't find American Independence Day\n";
```

Do you see how using the default delimiters of **//** requires escaping slashes inside of the regular expression, which makes it harder to read? We call that *leaning toothpick syndrome*.

More Regex Examples

Let's get comfortable with what we've learned so far. Create **reg_literal2.pl** as shown:

CODE TO TYPE: reg_literal2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

foreach my $text ( qw(Matt bats the ball at Atticus) )
{
  print qq{"$text" };
  if ( $text =~ /at/ )
  {
    print "matches";
  }
  else
  {
    print "does not match";
  }
  print " /at/\n";
}
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./reg_literal2.pl
"Matt" matches /at/
"bats" matches /at/
"the" does not match /at/
"ball" does not match /at/
"at" matches /at/
"Atticus" does not match /at/
cold:~/perl2$
```

Do you understand why the program produces that output?

For another example, create **reg_literal3.pl** as shown:

CODE TO TYPE: reg_literal3.pl

```
#!/usr/bin/perl
use strict;
use warnings;

foreach my $text ( "Stoplight", "Red Light", "Green means go" )
{
  print qq{"$text": };
  if ( $text =~ /sto/i && $text =~ /top/ )
  {
    print "matches AND clause\n";
  }
  elsif ( $text =~ /light/ || $text =~ /light/i )
  {
    print "matches OR clause\n";
  }
  else
  {
    print "matches neither clause\n";
  }
}
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./reg_literal3.pl
"Stoplight": matches AND clause
"Red Light": matches OR clause
"Green means go": matches neither clause
cold:~/perl2$
```

Do you understand why that output is printed?

Finally, let's try an example of a primitive implementation of the popular Unix **grep** program. Create **reg_grep.pl** as shown:

CODE TO TYPE: reg_grep.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $regex = shift;

while ( $_ = <> )
{
  print if $_ =~ /$regex/;
}
```

Check Syntax and run it, using one of your source files as input:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl2$ ./reg_grep.pl light ./reg_literal3.pl
foreach my $text ( "Stoplight", "Red Light", "Green means go" )
  elsif ( $text =~ /light/ || $text =~ /light/i )
cold:~/perl2$
```

This program takes, as its first argument, a regular expression. We can interpolate a scalar inside a regex (as I said, a regex behaves like a double-quoted string for the most part); the result is used as the regex. The remaining arguments are used as filenames to open (if we didn't supply any, it would work on standard input). The matching lines are printed. We use the default argument ($_) to print.

Congratulations, you've just gotten started on possibly the single most important topic in Perl! And there's a lot more to come!

Once you finish with the lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

Regular Expressions: Character Classes

Lesson Objectives

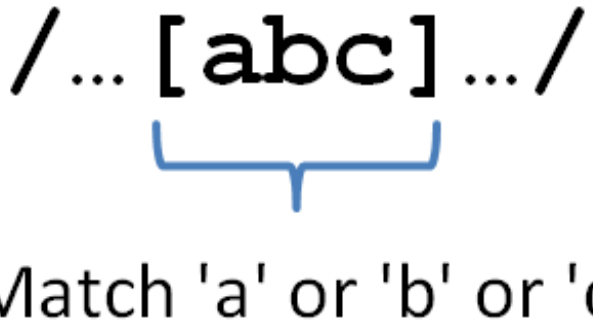
When you complete this lesson, you will be able to:

- use regex character classes.

Character Classes

Welcome to the second lesson in this course on regular expressions! If this course is your first encounter with regexes, be very excited! You're learning a brand new language *and* a whole new way that computers get the job done. If you don't yet feel your brain expanding, check in with it again in a lesson or two.

Time to go beyond the basic literal matching we've done so far with regexes. You remember that square brackets are special inside a regex, right? Now you'll find out why. Square brackets in a regex indicate that the regex engine should match *any one of the characters* between the square brackets. This is called a *character class*.



Match 'a' or 'b' or 'c'

Let's create a quick program to help you learn by example! Create **reg_charclass.pl** as shown:

CODE TO TYPE: reg_charclass.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @strings = qw(pot pout pet sip nit snip spit);
my @regexes = qw(p[aeiou]t [snp]i[tp]);

printf "%10s%13s%13s\n", 'Regex:', @regexes;
foreach my $text ( @strings )
{
  printf "%10s", $text;
  foreach my $regex ( @regexes )
  {
    if ( $text =~ /$regex/ )
    {
      printf "%13s", "  X  ";
    }
    else
    {
      printf "%13s", "";
    }
  }
  print "\n";
}
```

Check Syntax and run it. The output should look like a text version of this:

Regex:	p[aeiou]t	[snp]i[tp]
pot	X	
pout		
pet	X	
sip		X
nit		X
snip		X
spit	X	X

Let's go through each test in that table in turn:

- **pot** matches **/p[aeiou]t/** because the **p** matches literally, the **o** matches the **o** in the character class **[aeiou]**, and the **t** matches literally.
- **pout** does not match **/p[aeiou]t/** because the regex requires that after matching the literal **p** and *one* of the characters **a**, **e**:, **i**, **o**, or **u**, that we then match a **t**. But we don't have a **t** in the input; we have a **u**. Remember, a character class matches *exactly* one of the characters between the square brackets.
- **pet** matches **/p[aeiou]t/** the same way that **pot** did.
- **sip** matches **/[snp]i[tp]/** by matching the **s** in **[snp]**, then the literal **i**, then the **p** in **[tp]**.
- **nit** matches **/[snp]i[tp]/** by matching the **n** in **[snp]**, then the literal **i**, then the **t** in **[tp]**.
- **snip** matches **/[snp]i[tp]/** *starting* at the **n**, matching [snp], then the literal **i**, then the **p** in **[tp]**. (In fact, the regex engine will first match the **s** with **[snp]**, but when the next character is not a literal **i**, it will *backtrack* and start from the beginning of the pattern again.)
- **spit** matches **/[snp]i[tp]/** by a similar process. It also matches **/p[aeiou]t/**.

Check out these useful character classes:

- **[aeiou]**—a vowel. (Or **[aeiouy]** if you prefer.)
- **[0123456789]**—a digit.
- **[abcdefghijklmnopqrstuvwxyz]**—a lower-case letter.

Those last two examples are a bit long, don't you think? Something that useful shouldn't be that hard to type. And if I'd left out the digit **6** or the letter **q**, I bet you wouldn't have noticed. Perl offers a shortcut for making ranges of letters or digits in a character class: a hyphen between two letters or two digits means *pretend that all the letters or digits in between had been typed there*. So the last two examples above would come down to this:

- **[0-9]**
- **[a-z]**
- And another example: **[A-Z]** (uppercase letters)

Much better! For now you only want to use this when the characters on each side are either both letters (of the same case) or both digits. (Full details are available in **perldoc perlre**, but you have to wade through a lot to find it—definitely not for the faint of heart.) If you want a literal hyphen in a character class, make it the last character before the closing square bracket; then you don't have to worry about it being interpreted as a range.

But wait, it gets better (and by "better", I mean "more complicated...but also more useful")! If the caret symbol is the *first* thing in a character class after the opening square bracket, it *negates* the class, meaning that the class now matches any character *except* the ones listed. So now we have this:

- **[^0-9]**—any character that is *not* a digit
- **[^a-zA-Z]**—any character that is *not* an upper- or lower-case letter

The caret symbol has several different meanings in regular expressions depending on where it's used, so pay careful attention each time you encounter a new use for it.

Character Class Shortcuts

If you thought **[0-9]** was a short idiom for matching a digit, you haven't fully appreciated just how *lazy* Perl programmers can be! (Laziness—the good kind—is one of the three principal virtues of a Perl programmer as defined by Larry Wall. The other two are *hubris* and *impatience*.) Perl defines some convenient shortcuts for certain common

character classes. Because you'll use them so often, they're just two characters long (with one exception that's even shorter).

Instead of	You can type
[0-9]	\d
[^0-9]	\D
[a-zA-Z0-9_]	\w
[^a-zA-Z0-9_]	\W
[\n\t\f\r\v]	\s
[^ \n\t\f\r\v]	\S
[^\n]	.

Let's take these one at a time!

- \d matches any digit.
- \D matches any character that is *not* a digit.
- \w stands for "word character"; it matches any upper- or lower-case letter, digit, or underscore.
- \W matches any character that is *not* a "word" character.
- \s matches any character that is "white space." The escape sequences after the space character in the character class [\n\t\f\r] stand for newline, tab (you know those already), form feed, and carriage return. Those are some odd characters, but they all have the distinction of not putting any actual ink on a page or pixels on a screen. You will use this shortcut a lot.
- \S matches any character that is not white space. (That's backslash capital S.)
- . is a special shortcut that matches any character except for a newline. That one is used so often in regular expressions that it was worthy of being only one character long. (Remember in the last lesson that the "period" was one of the special characters in a regex? Now you know why. If you want to match a literal period, backslash it.)

Now it's time to see some of those shortcuts in action! Create **reg_charclass2.pl** as shown:

CODE TO TYPE: reg_charclass2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

# 0, 1, I, and O are not used in airline ticket locators to avoid ambiguity in print
my $pc = '[A-HJ-NP-Z2-9]';

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains a date\n"
    if $line =~ m{\d\d/\d\d/\d\d};
  print "'$line' contains a zip code\n"
    if $line =~ /\d\d\d\d\d/;
  print "'$line' contains a variable declaration\n"
    if $line =~ /my\s[\$\@\%][A-Za-z]/;
  print "'$line' contains a air ticket locator\n"
    if $line =~ /$pc$pc$pc$pc$pc$pc/;
}

__END__
This line shouldn't match anything
Easter falls on 24/04/11 next year
O'Reilly School of Technology, Sebastopol, CA 95472
Pi is approximately 3.14159265
foreach my $pet ( @animals )
My reservation is N2QVYX
Not a valid reservation code: ABC10I
```

Check Syntax and run it. Now type the code below at the Unix prompt and observe output from reg_charclass2.pl:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd  perl2
cold:~/perl2$ ./reg_charclass2.pl
'Easter falls on 24/04/11 next year' contains a date
'O'Reilly School of Technology, Sebastopol, CA 95472' contains a zip code
'Pi is approximately 3.14159265' contains a zip code
'foreach my $pet ( @animals )' contains a variable declaration
'My reservation is N2QVYX' contains a air ticket locator
cold:~/perl2$
```

When you're matching something like a zip code, you might think that a pattern of **/\d\d\d\d\d/** (like we have above) is adequate under all circumstances, and then be surprised when it matches an input like **"Pi is approximately 3.14159265"**. When you step back and look at the input, you see that the regex is matching **14159**. But you know that zip codes are only (in this example, anyway) five characters long, and you might think that the regex should reject any input with more than five consecutive digits. Of course, it doesn't. In order to do that, you'll need a feature that we'll introduce later.

Most of the time, you won't have to worry about that though, because your input will not be completely unpredictable and unstructured. It's usually only in class examples that you see matching done against conversational text like in the above code. Generally, in real-world problems, the input data is more constrained; for instance, any group of five consecutive digits in this particular input data could only be a zip code.

The pattern matching the declaration of a variable only needs to match the first part of a variable name. We don't know how long a variable name will be, but we know it's going to start with a letter after the sigil (that is, a symbol created for a specific "magical purpose," like $, @, or %), so as long as we match that much, we're fine. We backslashed the **$** to avoid having Perl think we were trying to interpolate a variable in the character class at that point, and then we backslashed the **@** and **%** sign purely for the sake of symmetry, so readers of our code don't wonder about the different treatment—in fact, the **@** and **%** do not need to be backslashed. We need to put **A-Za-z** in the character class because if we leave one range out and use the **/i** modifier instead, Perl would accept **MY** as a match for the **my** earlier in the regex.

And there's something more in that program—it contains its own input data! We did that so we wouldn't have to create a separate data file for input, and to show the special filehandle **DATA**. That input filehandle is automatically opened by Perl when you have a line in your program consisting precisely of the seven characters **__END__** (underscore, underscore, E, N, D, underscore, underscore) followed by a newline. At that point, Perl stops compiling your program, and everything after that point will be available through the **DATA** filehandle. (Of course, you don't *have* to read it. Usually people put the documentation for their code there so it doesn't get compiled.)

Note The actual definitions of **\d** and **\w** are broader than is stated above, but that difference only matters when you are matching against characters outside the standard ASCII set. Certain Unicode characters also match those shortcuts in recent versions of Perl. For the sake of simplicity, we will continue to assume that **\d** is equivalent to **[0-9]**, but in the event that your input string contains Unicode and you intend to use the match in a numeric context, you should use **[0-9]** instead.

Substitution

When I said that the pattern **/\d\d\d\d\d/** would match **"Pi is approximately 3.14159265"**, you could tell why that was true, but you had to work through it some, didn't you? Wouldn't it be great if Perl could show us what it matched? Well, it can, if we use the *substitution operator* instead of the match operator. Here's what it looks like:

OBSERVE: substitution operator

```
s/regex/replacement/
```

You use it just like the match operator. We'll explore that in more depth in a future lesson; for now, just be aware that it *changes* the variable it is bound to by swapping whatever *regex* matched with *replacement*. (If you try using it on a constant string like **"Hello world"**, Perl will tell you that you can't do that.)

Change **reg_charclass2.pl** as follows (deleting the code that looks ~~like this~~ and adding/changing the code that looks like this):

CODE TO EDIT: reg_charclass2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

# 0, 1, I, and O are not used in airline ticket locators to avoid ambiguity in print
my $pc = '[A-HJ-NP-Z2-9]';

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains a date\n"
    if $line =~ ms{\d\d/\d\d/\d\d}(*****);
  print "'$line' contains a zip code\n"
    if $line =~ s/\d\d\d\d\d/*****/;
  print "'$line' contains a variable declaration\n"
    if $line =~ s/my\s[\$\@\%][A-Za-z]/*****/;
  print "'$line' contains a air ticket locator\n"
    if $line =~ s/$pc$pc$pc$pc$pc$pc/*****/;
}

__END__
This line shouldn't match anything
Easter falls on 24/04/11 next year
O'Reilly School of Technology, Sebastopol, CA 95472
Pi is approximately 3.14159265
foreach my $pet ( @animals )
My reservation is N2QVYX
Not a valid reservation code: ABC10I
```

Check Syntax and run it. Now type the code below at the Unix prompt and observe output:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./reg_charclass2.pl
'Easter falls on ***** next year' contains a date
'O'Reilly School of Technology, Sebastopol, CA *****' contains a zip code
'Pi is approximately 3.*****265' contains a zip code
'foreach *****et ( @animals )' contains a variable declaration
'My reservation is *****' contains a air ticket locator
cold:~/perl2$
```

The regular expression matched was turned into five asterisks by the **s///** operator. Just like **m//**, it returns true if the regex matched, otherwise it returns false. Although it's not our intention to change the string we're matching, we'll use the substitution operator to do it now, so that you can see just what the regex matched.

At this point, you can tell that regular expressions are capable of much more than literal matching. And we've only just started—wait until you see what comes next!

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above, and do the assignments.

Regular Expressions: Quantifiers

Lesson Objectives

When you complete this lesson, you will be able to:

- use regex quantifiers.

Regular Expressions: Quantifiers

"You can't quantify love." -Emily Giffin

If you've ever been frustrated when you run into the gap between what you've wanted to do with regular expressions and what you can do, that's about to change. This lesson introduces the concept of the regex *quantifier*, which lets you vary the *number* of things you are able to match.

So far, everything you've seen that can go into a regex, matches one, and only one, character, or the regex fails to match. When the regex does match, you've had ordinary characters like **'Q'** and **'!'** that match literally, or you have a character class in square brackets, or a shortcut for a character class.

The *quantifier* is not a new kind of thing to match. (By the way, the technical term for a unit of matching in a regex is not "thing" but an "atom"; I'll use that term from now on because I think you're ready for advanced terminology now.) A quantifier modifies an atom in a regex by saying that you want to match some number of that atom (occurring consecutively at that point in the input) that is not necessarily exactly one.

We learned earlier that regular expressions were invented by mathematicians, so it comes as no surprise now to learn that the majority of quantifiers are expressed with single (and cryptic) characters. Before I introduce them, let's take a quick look at some examples of where and why you might want to use quantifiers.

- **Optional presence:** You are matching a word that may or may not be present in the plural form by the addition of the letter **'s'** (example inputs: **'apple'** and **'apples'**). **Or**: You are matching a number that may or may not have a decimal point in the middle (examples: **3.14159** and **42**).
- **Zero or more:** You are matching a number again, and after the decimal point there may be any number of digits, including zero (examples: **17.** and **101.7**). **Or**: You are picking apart text containing mathematical expressions and there may or may not be white space characters surrounding binary operators like **=** and **+**; in fact, it could be any amount of white space, or none at all (examples: **E = m * c**2** and **s=d*t**2/2**).
- **One or more:** You are matching a number again. This one has at least one digit before the decimal point, possibly more (for example, **2.71828** and **98.4**). **Or**: You are matching a word; it contains one or more letters, but you see no need to go to the Guinness Book of Records to find out the length of the longest word in existence just so you can limit the range of the match; you know that the input is well-behaved enough that "infinite" will do as a maximum length specification.
- **Exact number:** You're matching an airline ticket locator code like we did in the last lesson. Rather than repeat the specification for each character 6 times, you want to say, "Match this (character specification) 6 times." **Or**: You have a report containing FedEx tracking numbers that are exactly 22 digits long.
- **Length range:** You're creating a crossword puzzle and you want to select words from the dictionary that are between 3 and 10 letters long. **Or**: You are matching an IP address (version 4) and each octet is a decimal number one to three digits long (examples: **63.171.219.89** and **208.201.239.101**).

And now...here are the quantifiers themselves. They are placed *after* the atom that they quantify in a regex. So instead of a regex reading like the (more intuitive) "*this many* of *this*," it calls for, "*this thing*, *this many* times."

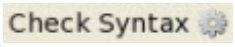
Quantifier	Interpretation	Example	Meaning
?	Optional (zero or one)	**/apples? and bananas?/**	Match **'apple and banana'** or **'apples and banana'**or **'apple and bananas'** or **'apples and bananas'**.
*****	Zero or more	**/\d\.\d*/**	Match a single digit, a period, and then zero or more digits.
+	One or more	**/\w+/**	Match one or more consecutive "word" characters.
{*n*}	Exact match (*n* times)	**/[A-HJ-NP-Z2-9]{6}/**	Same as the airline locator code regex in the previous lesson.
	Range match (from *m*		

{*m*,*n*}	Range match (from *m* to *n* times inclusive)	**/\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/**	Match an IPV4 address.
{,*n*}	Range match (from 0 to *n* times)	**/,\d{,6},/**	A comma, optionally followed by a number up to 999999, followed by another comma.
{*m*,}	Range match (*m* or more times)	**/\n{2,}/**	Require a gap of at least two blank lines.

Okay, enough lecture; it's time to try some of these out! Create **reg_quantifiers.pl** by typing the code below as shown:

CODE TO TYPE: reg_quantifiers.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains a zip code\n"
    if $line =~ s/\d{5}/*****/;
  print "'$line' contains an air ticket locator\n"
    if $line =~ s/[A-HJ-NP-Z2-9]{6}/*****/;
  print "'$line' contains a floating point number\n"
    if $line =~ s/\d+\.\d+/*****/;
  print "'$line' contains an IP address\n"
    if $line =~ s/\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/*****/;
  print "'$line' contains fruit\n"
    if $line =~ s/\d+\s+apples?.*\d+\s+bananas?/*****/;
}

__END__
Not a float by this standard: 12.
Neither is this: .34
But this is: 12.34
O'Reilly School of Technology, Sebastopol, CA 95472
My reservation is N2QVYX
www.oreillyschool.com is at 63.171.219.89
Not an IP address: 12.123.1234.12
Not a valid reservation code: ABC10I
I have 1 apple and 12 bananas
```



and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl2
cold:~/perl2$ ./reg_quantifiers.pl
'But this is: *****' contains a floating point number
'O'Reilly School of Technology, Sebastopol, CA *****' contains a zip code
'My reservation is *****' contains an air ticket locator
'www.oreillyschool.com is at *****.219.89' contains a floating point number
'Not an IP address: *****.1234.12' contains a floating point number
'I have *****' contains fruit
cold:~/perl2$
```

The lines that contain either correct and incorrect IP addresses each match the pattern for a floating point number, because they contain digits followed by a period, followed by digits. But **12.123.1234.12** does *not* match the pattern for an IP address, because there are four digits in the third octet, while the pattern says there can be at most 3. Because the pattern must still match something more (a period) after that third octet, the match fails.

The quantifiers are what we call *greedy*: they will match as much as they can. As long as they can keep finding something in the input to match, they will keep matching. People often think that means the quantifiers find the longest possible match in the input. That is *not* true. As long as a successful match can be found, the regex will match the *first*

one that it finds, scanning from left to right. For instance, the regex:

OBSERVE: Regex
`/[A-Z]+\d*/`

...when applied to the input:

OBSERVE: Input
`"I am THX1138"`

...will match **'I'**, *not* **'THX1138'**. It needs at least one upper-case letter; **'I'** qualifies, so it matches; there are no more letters, so next it needs zero or more digits; there are no digits, but that's okay, because zero is a valid number to match, and so it stops.

This is an important point and worth exploring in depth. Suppose the regex had been:

OBSERVE: Regex
`/\w+\d+/`

...and the input had been:

OBSERVE: Input
`"Robert Duvall is THX1138"`

In this case, the regex engine starts matching with the **'R'**, and keep matching letters greedily, until it comes to the first space, at which point it can no longer match **\w+**, so it looks to see whether it can match the *next* thing, which is **\d+**. That requires *at least* one digit, and there isn't one. The regex engine then *backtracks,* unwinding each character it matched, looking for a digit, but does not find one. It backtracks all the way to the beginning without success, and then it moves the pointer forward by one character and starts again. (This may sound inefficient, but the regex engine is full of optimizations to make this process faster.)

The regex engine continues in that way (reading **'Duvall'** and **'is'** as matches against **\w+** but not finding a match for **\d+**) until it matches **\w+** with **'THX1138'**. Why does that whole string match **\w+**? Because **\w** *includes* digits in its specification! At this point, the regex engine sees that it has reached the end of the string, so it starts backtracking. It unwinds by one character—**'8'**—and tries again to see if it has a match for **\d+**. It *does*. Then it tries matching more digits greedily, but it has reached the end of the input. It looks at the regex and finds that there is nothing more to be matched. It has found a match—the **\w+** matched **'THX113'**, and the **\d+** matched **'8'**—so it returns success.

Okay, so you can tell *whether* an input string is matched by a regex, but not exactly *what* matched that regex. And just knowing that there's (for example) a part number or a zip code in a string may not be good enough; you may want to know the specific part number, without resorting to using **index()** and **substr()**. If the regex engine could find the match for you, shouldn't it be able to tell you which part matched? Yes, it should, and it will. That's coming in the next lesson!

Regex Crafting

Picking the right regex for a particular problem is something of an art. Rarely is it necessary to craft a regex that matches just what you need and nothing else. For instance, the pattern we used above for an IP address can still match an invalid address like **1.2.999.3** (it matches because each octet has 1 to 3 digits; it is invalid because the biggest octet should be 254), but usually when you are looking for an IP address the input will be sufficiently constrained. Under some circumstances, any input containing just digits will be an IP address because all other possibilities *in the problem domain you are solving* are exhausted. The complete regex that matches *only* valid IPV4 addresses is much more complicated. However, there are modules that you can use, notably **Regexp::Common::net**, that encapsulate that complexity for you. Once you learn how to download and use modules, you can choose whether you want to adopt that approach for regexes for common patterns. There's nothing to be gained from using a regex that's longer than necessary, and you will lose readability, so think carefully before making a complex regex just for the sake of it.

Take a look at the last regex in **reg_quantifiers.pl**:

OBSERVE: Regex

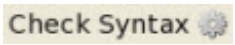
```
/\d+\s+apples?.*\d+\s+bananas?/
```

That's a common idiom: match something you want, followed by some arbitrary number of characters that you don't care about, followed by something else that you want. You'll end up using this pattern a lot. Now, I know that **.** matches any character *except* **\n**, and you might wonder why it doesn't just match any character. Most of the time you'll be matching against a single line that you've read in and **chomp**ed, so it won't mater. Also, when matching against multiline text, it's far more common to want to constrain your matches to within a single line than not, so this implementation of **.** is intentional.

Let's do another example! Create **reg_email.pl**:

CODE TO TYPE: reg_email.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains an email address\n"
    if $line =~ s/\w+\@\w+\.\w\w\w/*****/;   # Won't match all possible email addresses
!
}

__END__
From: peter@psdt.com
To: scott@oreilly.com
Subject: Intermediate Perl lessons
Cc: president@whitehouse.gov

Hi Scott!  I've just uploaded the latest content for the
Intermediate Perl course.  Barack, I think you'll have fun
with the regular expression lessons.  By the way, can we use
your private address of yeswecan2008@h0tmail.com for all
future correspondence?  We're having trouble getting past
the government spam filters.  Thanks!
```



and run it, and see what lines match. No surprises there. But that regular expression is intentionally crippled so you're not tempted to try to use it for matching *any* email address. It only matches one or more word characters, followed by an **@** sign, followed by one or more word characters, a period, and three word characters. It won't match, for instance, **'fred@slateco.cc'**, or **'wilma-@bedrock.edu'** (why?). Verify that for yourself:

CODE TO EDIT: reg_email.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains an email address\n"
    if $line =~ s/\w+\@\w+\.\w\w\w/*****/;    # Won't match all possible email addresse
s!
}

__END__
From: peter@psdt.com
To: scott@oreilly.com
Subject: Intermediate Perl lessons
Cc: president@whitehouse.gov

Hi Scott!  I've just uploaded the latest content for the
Intermediate Perl course.  Barack, I think you'll have fun
with the regular expression lessons.  By the way, can we use
your private address of yeswecan2008@h0tmail.com for all
future correspondence?  We're having trouble getting past
the government spam filters.  Thanks!
the government spam filters.  We don't have any problem reaching,
for instance, fred@slateco.cc, or wilma-@bedrock.edu.  Thanks!
```

The email address **'*@example.com'** is perfectly legal; some people have addresses like that. Still, they are rejected by most of the email validation code on the web. Before you decide on a regex for matching email addresses, you need to think about what you're trying to accomplish and what the tradeoffs are for your choices. The regular expression that *technically* matches the legal specification for an email address is a *page* long, and it probably isn't what you really want anyway. To get the current best advice on solving this problem, run **perldoc perlfaq9** and see the section titled "How do I check a valid mail address?"

Note The **@** in the regex does not *need* to be escaped, because it is followed by a backslash. But it's worth doing consistently anyway, because if it were followed by, for example, a letter, let's say **'q'**, Perl would think that we were trying to interpolate an array named **@q**, and we'd get into trouble.

Food for Thought

Using regular expressions requires understanding lots of subtle details—you're learning a whole new language after all! For instance, matching something optional—for example, the **s?** in **/apples?/**—only makes sense if you're matching something else after that. Otherwise, it's pointless, because you've already found a match with everything that came before the optional part; you get exactly the same outcome regardless of whether the optional part is present, because it can always match nothing.

Creating regexes that can match nothing is a common mistake. Let's say a programmer (not you, of course) tries to write a regex that will match any kind of number at all, and thinks to himself, "it can have digits before and after a decimal point, like **12.34**, so I'll have a regex of **/\d+\.\d+/**. But wait, the digits before the decimal point are optional: I could have a number like **.12**, so my regex is really **/\d*\.\d+/**. No, wait, the digits after the decimal point are optional too, because I could have a number like **12.**, so my regex is really **/\d*\.\d*/**. But wait, I could have a number like **1234** so the decimal point is optional there too, and that means my regex is really **/\d*\.?\d*/**."

There's the problem with that logic, and it's worth seeing it in action. Create **reg_number.pl** as shown:

CODE TO TYPE: reg_number.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains a number\n"
    if $line =~ s/\d*\.?\d*/*****/;
}
__END__
One kind of number: 12.
Another kind of number: .34
A third kind of number: 12.34
A plain integer: 1234
No number in this line, not even a period
Or in this one
```

Check Syntax and run it. What is the output? Why do you think that happened?

The problem is that this regex can match *nothing:* no digits, followed by no period, followed by no digits. And every possible input can always satisfy that immediately; each one has 'nothing' before the first character ever appears. Add a blank line in the inputs:

CODE TO EDIT: reg_number.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains a number\n"
    if $line =~ s/\d*\.?\d*/*****/;
}
__END__
One kind of number: 12.
Another kind of number: .34
A third kind of number: 12.34
A plain integer: 1234
(Insert a blank line here.)
No number in this line, not even a period
Or in this one
```

Check Syntax and run it.

Even the blank line matches! Because we have **chomp**ed off the newline character, we really *are* matching against the empty string (zero length), and Perl can always find the empty string at the beginning of every input.

The easiest way to fix this particular case is with multiple regexes. There are several ways to do that; here's one:

CODE TO EDIT: reg_number.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains a number\n"
    if $line =~ s/\d*\.?\d*/*****/;
    if $line =~ s/\d+\.\d*/*****/
    || $line =~ s/\d*\.\d+/#####/
    || $line =~ s/\d+/~~~~~/;
}
__END__
One kind of number: 12.
Another kind of number: .34
A third kind of number: 12.34
A plain integer: 1234

No number in this line, not even a period
Or in this one
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./reg_number.pl
'One kind of number: *****' contains a number
'Another kind of number: #####' contains a number
'A third kind of number: *****' contains a number
'A plain integer: ~~~~~' contains a number
cold:~/perl2$
```

The *ordering* of the regex tests is important here. What would have happened if the third test had been tried first? Try it:

CODE TO EDIT: reg_number.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  print "'$line' contains a number\n"
    if $line =~ s/\d+\.\d*/*****/s/\d+/~~~~~/
    || $line =~ s/\d*\.\d+/#####/
    || $line =~ s/\d+/~~~~~/s/\d+\.\d*/*****/;
}
__END__
One kind of number: 12.
Another kind of number: .34
A third kind of number: 12.34
A plain integer: 1234

No number in this line, not even a period
Or in this one
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./reg_number.pl
'One kind of number: ~~~~~.' contains a number
'Another kind of number: .~~~~~' contains a number
'A third kind of number: ~~~~~.34' contains a number
'A plain integer: ~~~~~' contains a number
cold:~/perl2$
```

Why did that happen? Think it over.

The quantifier is both concise and powerful, and provides the foundation for regular expressions. If your understanding of any part of this lesson is still fuzzy, go back and study it some more. Play with the examples, modify them, and ask your instructor for clarification where necessary.

In the next lesson, we'll build on what you've learned so far to quantify a literal character or something matching a single character, and eventually quantify something even longer!

Once you finish the lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

Regular Expressions: Anchors and Captures

Lesson Objectives

When you complete this lesson, you will be able to:

- use regex anchors to mark the boundary between a string of digits and something that isn't a string of digits.
- use regex captures.

The key concepts we need to learn here are *anchors* (how to match a *place* rather than a *thing*), and *captures* (how to get at what part of a regex matched).

Anchors

Until now, we've had difficulty telling Perl where to stop matching. For instance, looking for a US zip code with the regex **/\d{5}** worked fine for input like **"Sebastopol, CA 94572"**, but it also matched when the input was **"Pi is about 3.14159265"**. That may not matter—it could be that for the problem you're solving, all inputs that contain five consecutive digits will always contain only zip codes, so your regex is adequate. But what if that's not the case? Suppose you want to "match five consecutive digits, but no more"?

We can solve this problem already—kind of. Consider the regex **/\D\d{5}\D/**. "Match a non-digit, followed by 5 digits, followed by a non-digit." That'll successfully discriminate against our Pi example above, because every group of five digits is either preceded or succeeded by a digit. But what about certain inputs that *should* match, like **"Sebastopol, CA 94572"**? That no longer matches, because the five digits are not followed by a non-digit—they're not followed by anything at all!

Instead, we want to say, "Match five digits that are not preceded by a digit and not followed by a digit." Do you see the difference between that and the regex we had before? We need to allow for the possibility that the string will start with the five digits or end with the five digits, or maybe even be composed exactly of five digits.

In short, we need to match a *place*, rather than a character, on each end of our five digits; we want to indicate that a specific place marks the boundary between a string of digits and something that isn't a string of digits. Perl lets us do that in the form of an *anchor* called the *word boundary*, expressed with the sequence **\b**. Let's try an example to see this in action. Create **reg_boundary.pl** as shown:

CODE TO TYPE: reg_boundary.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @regexes = qw( \d{5} \D\d{5}\D \b\d{5}\b );
printf "%40s%10s%10s%10s\n", 'Regex:', @regexes;

while ( my $line = <DATA> )
{
  chomp $line;
  printf "%40s", $line;
  foreach my $regex ( @regexes )
  {
    if ( $line =~ /$regex/ )
    {
      printf "%10s", "   X   ";
    }
    else
    {
      printf "%10s", "";
    }
  }
  print "\n";
}

__END__
Sebastopol, CA 94572
20500 is the zip code of the White House
Pi is about 3.14159265, give or take
The Statue of Liberty is at 10004-1467
2147483647 is MAXINT on my machine
```

Check Syntax and run it. You should see something like this:

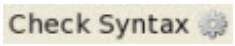
Regex:	\d{5}	\D\d{5}\D	\b\d{5}\b
Sebastopol, CA 94572	X		X
20500 is the zip code of the White House	X		X
Pi is about 3.14159265, give or take	X		
The Statue of Liberty is at 10004-1467	X	X	X
2147483647 is MAXINT on my machine	X		

The last regex is the only one to correctly match what we want and nothing more, in all cases. The **\b** *word boundary* anchor matches a *place* (something that has no width, contains no characters) that can be described like this: "On one side is a word character, and on the other side is not a word character." Keep in mind that "not a word character" is *not* the same as "a character that isn't a word character." Also, remember that a word character is the character class of letters, digits, and the underscore character. Look over the results in the table, make sure you understand them, and play with changing the program until you're comfortable with it.

There isn't a special anchor that only matches the boundary between digits and not-digits, so we used the closest thing, which worked for all our above examples. Just to make sure you understand its limitations as well, make this change to the code:

CODE TO EDIT: reg_boundary.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @regexes = qw( \d{5} \D\d{5}\D \b\d{5}\b );
printf "%40s%10s%10s%10s\n", 'Regex:', @regexes;

while ( my $line = <DATA> )
{
  chomp $line;
  printf "%40s", $line;
  foreach my $regex ( @regexes )
  {
    if ( $line =~ /$regex/ )
    {
      printf "%10s", "   X   ";
    }
    else
    {
      printf "%10s", "";
    }
  }
  print "\n";
}

__END__
Sebastopol, CA 94572
20500 is the zip code of the White House
Pi is about 3.14159265, give or take
The Statue of Liberty is at 10004-1467
2147483647 is MAXINT on my machine
My zip is _98362_
```



and run it.

/\b\d{5}\b/ does *not* match the last input. Because underscore is a word character, we do not have a word boundary on either side of the five digits; **_9** is a word character (_) followed by another word character (**9**), and **2_** is a word character (**2**) followed by another word character (_). Either one of those is sufficient for the regex to fail to match.

All of these failing examples may make you wonder whether you could ever write a regex that will match right. Don't worry. Our exploration of regexes and consequences is meant to get you in the habit of thinking about them correctly, that way, when you have a real problem to solve, you'll be able to craft an appropriate regex. Most real problems are not as vexing as the examples we've come up with for you to wrestle with in these lessons—after all, how often do you encounter zip codes with underscores on each end?

Also, you've seen the substitution operator in our examples of regular expressions so far, but it isn't always going to be there. It's actually more common to match than to substitute; we've used substitutions primarily as a teaching aid, but that's going to change. Substituting wouldn't be as instructive where anchors are involved because they match *places*, not *characters*.

Alright then. Here are two anchors that are likely to be even more useful to you than **\b**.

- Match beginning of string: **\A**. This sequence matches the place that is the beginning of the string. Whatever comes next in the regex will only match successfully if it is at the beginning of the string. **\A** should always be at the beginning of a regex (or at the beginning of an alternation group, which we'll get into shortly) to be meaningful.
- Match ending of string: **\z**. Yes, that's a *little* z. This sequence matches the place that is the ending of the string. Whatever comes before it in the regex will only match successfully if it is at the ending of the string. **\z** should always be at the end of a regex (or at the end of an alternation group) to be meaningful.

Let's see these in action! Create **reg_edges.pl**:

CODE TO TYPE: reg_edges.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @regexes = qw( \A[aeiou] [aeiou]\z \A[aeiou]+\z );
printf "%40s%13s%13s%13s\n", 'Regex:', @regexes;

while ( my $line = <DATA> )
{
  chomp $line;
  printf "%40s", $line;
  foreach my $regex ( @regexes )
  {
    if ( $line =~ /$regex/i )
    {
      printf "%13s", "   X   ";
    }
    else
    {
      printf "%13s", "";
    }
  }
  print "\n";
}

__END__
Neither beginning nor ending in a vowel
A line that starts in a vowel
There's a vowel on the end of this line
A vowel at the start and end here
IOU
```

Check Syntax and run it. You should see something like this:

Regex:	\A[aeiou]	[aeiou]\z	\A[aeiou]+\z
Neither beginning nor ending in a vowel			
A line that starts in a vowel	X		
This line has a vowel here		X	
A vowel at the start and end here	X	X	
IOU	X	X	X

The line that starts and ends with a vowel does *not* match **/\A[aeiou]+\z/** because that regex reads, "Match start of string, one or more vowels, end of string", and after matching the start of the string and the first vowel, what follows is neither a vowel nor the end of the string. That regex will only match strings that consist only of vowels (the **+** quantifier means that there must be at least one vowel, so an empty string will not count).

We're using the **/i** modifier so that the vowel character class matches the capital **'A'**.

Remember that scalars interpolate inside regexes, so if **$regex** is **'\A[aeiou]'**, then **/$regex/** is the same as typing **/\A[aeiou]/**. It's pretty uncommon to interpolate scalars inside of regular expressions; I'm doing it here to produce those nice tables. But eventually, when you work your way up to building complex regular expressions from smaller pieces, you may choose to put those pieces in scalars for much the same reason you would use Perl code inside a named subroutine.

Captures

We've devoted lots of time and energy learning how to determine whether an input matches a regular expression, but we still haven't learned how to identify the part of the input that's been matched. We'll definitely want to be able to do that. And we can, using one of the most useful features of regular expressions: **captures**. Here's how:

Note Place parentheses around the part of the regex that has the match you want to identify. If the match succeeds, the part of the input that matched the part of the regex in parentheses will be placed in the special variable **$1**.

You may remember that the list of "special" characters in a regex includes **(** and **)**. Now you're going to find out why. Let's learn by doing! Create **reg_capture.pl** as shown:

CODE TO TYPE: reg_capture.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( my $line = <DATA> )
{
  chomp $line;
  if ( $line =~ /\b(\d{5})\b/ )
  {
    print "Captured: $1\n";
  }
  else
  {
    print "No match for '$line'\n";
  }
}
__END__
Sebastopol, CA 94572
20500 is the zip code of the White House
Pi is about 3.14159265, give or take
The Statue of Liberty is at 10004-1467
2147483647 is MAXINT on my machine
```

Check Syntax and run it.

Now you can see not only *when* the regex matched, but *what* it matched! For the sake of clarity, the **\b** anchors were placed outside of the capturing parentheses, but since they match a place (which has zero width), it would have made no difference if we had put them inside. Usually you will only have capturing parentheses around part of a regex, while the rest of the regex sets up the context for that part.

Capturing is a tremendously useful feature of regular expressions. It doesn't change *what* you match, it makes available the parts of the input that *did* match.

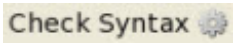
You may be wondering what happens if you have more than one set of parentheses, or whether there's a **$2** variable. Good! I'm glad you're curious. In fact, the capturing feature is much more general than **$1**: You can have multiple sets of capturing parentheses, and they will capture to variables named **$2**, **$3**, and so on, for as many capturing parentheses as you have.

You can even *nest* the parentheses so that one match is a subset of another. (It's not too common in practice, but I know you're wondering about it!) If you do that, you'll want to be able to determine which parentheses capture to **$1** and which capture to **$2**. Here's the way to remember: The first (leftmost) left parenthesis starts a group that captures to **$1**. The second left parenthesis captures a group that captures to **$2**, and so on.

So, let's do another example that shows us the awesome power of multiple capturing parentheses! Create **reg_capture2.pl** as shown:

CODE TO TYPE: reg_capture2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  if ( $line =~ /((\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3}))/ )
  {
    print "Found an IP address: $1\n";
    print "First  octet: $2\n";
    print "Second octet: $3\n";
    print "Third  octet: $4\n";
    print "Fourth octet: $5\n";
  }
  else
  {
    print "No match for '$line'\n";
  }
}

__END__
www.oreillyschool.com is 63.171.219.89
This line won't match
208.201.239.101 is www.perl.com
```



and run it.

Look it over and make sure you understand where **$1**, **$2**, and the other capture variables came from.

Now let's take a quick detour into system programming to demonstrate some *real* Perl power. Create **reg_capture3.pl** as shown:

CODE TO TYPE: reg_capture3.pl

```
#!/usr/bin/perl
use strict;
use warnings;

use Socket;

while ( my $line = <DATA> )
{
  chomp $line;
  if ( $line =~ /(\S+)\s+is\s+(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/ )
  {
    my ($name, $ip) = ($1, $2);
    my $cname = gethostbyaddr( inet_aton( $ip ), AF_INET );
    print "$name -> $ip -> $cname\n";
  }
  elsif ( $line =~ /(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\s+is\s+(\S+)/ )
  {
    my ($ip, $name) = ($1, $2);
    my $lookup;
    if ( defined( $lookup = gethostbyname( $name ) ) )
    {
      $lookup = inet_ntoa( $lookup );
    }
    else
    {
      $lookup = "Couldn't resolve!";
    }
    print "$ip -> $name -> $lookup\n";
  }
  else
  {
    print "No match for '$line'\n";
  }
}

__END__
www.oreillyschool.com is 63.171.219.89
This line won't match
208.201.239.101 is www.perl.com
157.166.226.25 is cnn.com
```

Check Syntax and run it.

You're executing forward and reverse DNS lookups on data parsed from text! The **gethostbyname()** and **gethostbyaddr()** functions are built-in to Perl. You can read the **perldoc -f** documentation for those functions and copy what it says to do, even though you may not know what statements like **use Socket** do. In fact, that's how I devised the program in our last example.

You can read from the variables like **$1**, but you are not allowed to write to them. Don't try to declare them with **my**, because they don't belong to you; Perl creates them. A good practice is to copy those numbered variables into named variables like we did in our example, and then use the named variables.

Note the *list assignment* **my ($ip, $name)** = (**$1, $2**), which assigns multiple corresponding variables.

Okay, one more example! Create **reg_capture4.pl** as shown:

CODE TO TYPE: reg_capture4.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( my $line = <DATA> )
{
  chomp $line;
  if ( $line =~ /(.*),\s+(\w\w)\s+(\d{5})/)
  {
    my ($city, $state, $zip) = ($1, $2, $3);
    printf "City: %-20s; State: $state; Zip: $zip\n", $city;
  }
}
__END__
O'Reilly School of Technology
1005 Gravenstein Highway
Sebastopol, CA 95472

O'Reilly Media
10 Fawcett Street
Cambridge, MA  02138

PSDT
136 E. 8th Street #232
Port Angeles, WA 98362
```

Check Syntax and run it.

If we assume that in any address line that contains a city, that city's name will be followed by a comma, we can match the city using the trivial regex **.***. The regex engine will start matching by swallowing all of the characters in the line for that component, but when it reaches the end of the string and then sees that the regex requires the next character to be a comma, it backtracks until it has unswallowed the comma. A benefit of using **.*** is that we can match city names containing any characters (except commas); if we'd used a regex like **\S+**, it wouldn't have matched **"Port Angeles"**, which contains a space.

Note It's a common mistake to perform a regex match and then use the numbered variables *without checking to see if the match was successful*. If you do that and the regex match was *not* successful, but you use **$1** anyway, its value will be whatever it had from the last successful match with capturing parentheses. This will cause chaos in your programs. *Always* check to see whether a match succeeded (using any of the three forms of conditional statements you have learned) before using **$1** or other numbered variables.

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above and do the assignments. See you in the next lesson!

Grouping, Alternation, and Complete Parsing

Lesson Objectives

When you complete this lesson, you will be able to:

- use **$_** with regexes.
- *group* (how to treat several atoms as though they were a single atom.
- use alternation (to match one of several choices).
- parse with regexes.

Hi and welcome back! In this lesson, you'll learn about using **$_** with regexes, *grouping* (how to treat several atoms as though they were a single atom; you'd think the term for this would be "molecule," but it isn't), and alternation (how to match one of several choices). Then we'll try some examples of parsing with regexes.

$_ and regexes

Our previous examples have been more long-winded than they needed to be. So far we've always matched a regex against an explicitly named input using the binding operator (**=~**). But there is a default variable available for regex matching—no prizes for guessing what it is! When you have a regex that is not bound to any variable via **=~**, it will match against **$_**. We introduced regexes in the course by incorporating a hypothetical function called **regmatch**. When you call **regmatch** with only one argument, **regmatch($regex)**, instead of two, **regmatch($regex, $input)**, it's as though **$_** is the second argument, **regmatch($regex, $_)**.

What does that look like? Let's do an example! Copy **reg_capture4.pl** from the previous lesson and modify it:

CODE TO EDIT: reg_capture4.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( my $line = <DATA> )
{
  chomp $line;
  if ( $line =~ /(.*),\s+(\w\w)\s+(\d{5})/ )
while ( $_ = <DATA> )
{
  chomp;
  if ( /(.*),\s+(\w\w)\s+(\d{5})/ )
  {
    my ($city, $state, $zip) = ($1, $2, $3);
    printf "City: %-20s; State: $state; Zip: $zip\n", $city;
  }
}

__END__
O'Reilly School of Technology
1005 Gravenstein Highway
Sebastopol, CA 95472

O'Reilly Media
10 Fawcett Street
Cambridge, MA  02138

PSDT
136 E. 8th Street #232
Port Angeles, WA 98362
```

Check Syntax, saving it as **reg_capture5.pl**. Then, run it to make sure it produces the same output as before:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl2
cold:~/perl2$ ./reg_capture5.pl
City: Sebastopol          ; State: CA; Zip: 95472
City: Cambridge           ; State: MA; Zip: 02138
City: Port Angeles        ; State: WA; Zip: 98362
cold:~/perl2$
```

Again, we don't declare **$_** with **my**; it doesn't belong to us, it belongs to Perl. And the default variable for **chomp()** is **$_**.

We'll make even better use of **$_** as we proceed in this lesson.

Groups

One problem we've run into when applying quantifiers is that we can only apply them to something that matches a single character. When I introduced the **?** quantifier with the example **apples?**, you may have thought, "Hmm, there are lots of plural forms that won't be able to use this quantifier. What about words that are made plural by adding 'es'?"

Fortunately, capturing parentheses have a helpful side-effect that we can take advantage of here. They *group* everything between them so that it effectively becomes an atom. That means that if you put a quantifier after a parenthesis group, it applies to everything in the group. So to match **"wrench"** or its plural, you can write **/wrench(es)?/** (or, if you also want to match serving maids, **/wr?ench(es)?/**).

There's one disadvantage here though. If we are capturing parts of the input to use in **$1** or some other quantifier as well, then introducing a parenthesis group solely for the purpose of forming an atom to be quantified, will throw off our counting: **$1** may become **$2**, for instance. We need *noncapturing* parentheses: parentheses that will form a group, but not capture to a numbered variable. Perl does have them, but they look a bit weird. After the opening parenthesis we have to add **?:**, like this:

OBSERVE: noncapturing parentheses

```
(?:regex)
```

Let's try an example. Suppose you're capturing US zip codes, some of which might be five-digit codes like **95472**, and some of which might be ZIP+4 codes like **95472-2811**. Create **reg_zip.pl** as shown:

CODE TO TYPE: reg_zip.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( $_ = <DATA> )
{
  chomp;
  /(\d{5}(?:-\d{4})?)/ and print "Found zip: $1\n";
}

__END__
O'Reilly School of Technology
1005 Gravenstein Highway
Sebastopol, CA 95472-2811

O'Reilly Media
10 Fawcett Street
Cambridge, MA  02138

PSDT
136 E. 8th Street #232
Port Angeles, WA 98362-6129
```

Check Syntax and run it. See how it finds and captures both types of zip code? Let's dissect the regex **/(\d{5}(?:-\d{4})?)/>**:

(	Begin a group capturing to **$1**
\d{5}	Match five digits
(?:	Begin a noncapturing group
-	Match a hyphen
\d{4}	Match four digits
)	Close noncapturing group
?	Quantify noncapturing group as optional
)	End capturing to **$1**

The inner parentheses are noncapturing, so no **$2** is created. This could be especially helpful later on if we have inputs with text occurring after the zip code that we want to capture, because it makes counting the numbered capture variables easier.

Alternation

Sometimes we want to match one thing *or* another. Suppose we want to match one of a number of fruit. We could use Perl's Boolean || operator; create **reg_or.pl** as follows:

CODE TO TYPE: reg_or.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( $_ = <DATA> )
{
  chomp;
  print "Captured: $1\n" if /(apple)/ || /(banana)/ || /(cherry)/;
}

__END__
I have apple in stock
You are the apple of my eye
Time flies like an arrow; fruit flies like a banana
I have cherry in stock
I have kiwi fruit in stock; no banana, sorry
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./reg_or.pl
Captured: apple
Captured: apple
Captured: banana
Captured: cherry
Captured: banana
cold:~/perl2$
```

This is not a bad approach, but its capabilities are limited.

Suppose we wanted to match a phrase of the form "I have <fruit> in stock," and capture the fruit; now we have to write **/I have (apple) in stock/ || /I have (banana) in stock/ || /I have (cherry) in stock/**.

The repetition here makes the expression nearly unreadable. Try it in the program and see:

CODE TO EDIT: reg_or.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( $_ = <DATA> )
{
  chomp;
  print "Captured: $1\n" if /(apple)/ || /(banana)/ || /(cherry)/I have (apple) in sto
ck/ || /I have (banana) in stock/ || /I have (cherry) in stock/;
}

__END__
I have apple in stock
You are the apple of my eye
Time flies like an arrow; fruit flies like a banana
I have cherry in stock
I have kiwi fruit in stock; no banana, sorry
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./reg_or.pl
Captured: apple
Captured: cherry
cold:~/perl2$
```

Note This also changes the regex to match against the complete phrase, "I have *x* in stock"; we find matches now on only the first and fourth lines.

Let's see how to make it easier with alternation. Create **reg_alt.pl** as shown:

CODE TO TYPE: reg_alt.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( $_ = <DATA> )
{
  chomp;
  print "Captured: $1\n" if /(apple|banana|cherry)/;
}

__END__
I have apple in stock
You are the apple of my eye
Time flies like an arrow; fruit flies like a banana
I have cherry in stock
I have kiwi fruit in stock; no banana, sorry
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./reg_alt.pl
Captured: apple
Captured: apple
Captured: banana
Captured: cherry
Captured: banana
cold:~/perl2$
```

The regular expression language provides a solution with the *alternation* operator. It looks familiar; it's a single vertical bar (**|**) and it means "or." Remember, this vertical bar is being used *inside* a regular expression; even though it is being used to mean "or," it's *not* the same as the boolean operator we used in **reg_or.pl**!
Now rewrite the program to handle the second match we were trying to make:

CODE TO EDIT: reg_alt.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( $_ = <DATA> )
{
  chomp;
  print "Captured: $1\n" if /(apple|banana|cherry)/I have (apple|banana|cherry) in sto
ck/;
}

__END__
I have apple in stock
You are the apple of my eye
Time flies like an arrow; fruit flies like a banana
I have cherry in stock
I have kiwi fruit in stock; no banana, sorry
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./reg_alt.pl
Captured: apple
Captured: cherry
cold:~/perl2$
```

Our examples show that the *scope* of the alternation operator is to the nearest enclosing parentheses, or the edges of the regex if there are no enclosing parentheses.

If we didn't have the **\b** anchor available to us, we might rewrite **/\b\d{5}\b/** as **/(?:\A\d{5}\D\D|\d{5}\z|\A\d{5}\z)/**. But I think you'll agree that **\b** is a lot better!

Let's do another example. Create **reg_alternate.pl** as shown:

CODE TO TYPE: reg_alternate.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( $_ = <DATA> )
{
  chomp;
  /\b(\S+berr(?:y|ies))/ and print "Enfruitened: $1\n";
}

__END__
We have strawberries,
and "whateverberries,"
and a giant gooseberry
on a big plate with a lingonberry pie!
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./reg_alternate.pl
Enfruitened: strawberries
Enfruitened: whateverberries
Enfruitened: gooseberry
Enfruitened: lingonberry
cold:~/perl2$
```

The inner parenthesis group in the regex—**(?:y|ies)**—does not *need* to be noncapturing (take out the **?:** and rerun the program and see for yourself). But it is good *practice* to make sure you *never* use capturing parentheses, unless you use the numbered variable that is created. Many programmers, upon reading a regex and encountering a capturing group, automatically look for the use of the corresponding numbered variable and become frustrated if they don't find it.

WARNING Perl tests alternation possibilities from left to right, and stops at the first match. If a later alternative would produce a longer match, it won't be found after an earlier match has already succeeded. So the regex **/\d{5}|\d{5}-\d{4}/** would *never* match a ZIP+4 zip code, because the first alternative (**\d{5}**) will always succeed first. Switch the order of the alternatives!

Complete Parsing

Now we'll use your new regex skills to parse text files with some of Perl's most common idioms.

$_ and <> Remember the programmer's motto: Don't Repeat Yourself. You've already seen code in these forms:

OBSERVE: line input idiom

```
while ( defined( my $line = <DATA> ) )
while ( defined( $_ = <DATA> ) )
```

And we've learned that this code can be shortened using a special case of the **while** statement. If the condition contains an assignment from the readline operator, then Perl will *implicitly* wrap the assignment in **defined()** for you. So you can write the lines above like this:

OBSERVE: line input idiom

```
while ( my $line = <DATA> )
while ( $_ = <DATA> )
```

But wait—it gets better! If there is a readline operator, but *no* assignment to a variable, Perl will *implicitly* assign to **$_** for you. So the second line above can be written just like this:

OBSERVE: line input idiom

```
while ( <DATA> )
```

Now *that's* succinct. And it applies to the use of the readline operator on *any* filehandle, including the magic behavior with *no* filehandle:

OBSERVE: line input idiom

```
while ( <> )
```

A *very* common pattern in Perl programs is:

OBSERVE: Perl filter pattern

```
while ( <> )
{
  chomp;
  if ( /regex/ )
   {
     # Do something with match
   }
}
```

Let's try an example. Create **grep.pl** as show:

CODE TO TYPE: grep.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $regex = shift;

while ( <> )
{
  print if /$regex/;
}
```

Our examples are getting *shorter*. That is very cool.

Check Syntax and run it. This is the world's shortest implementation of the famous Unix *grep* program. You have *grep* available in your shell; try it out if you're not already familiar with it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ grep perl ./*.pl
```

This is the synopsis of how to run our version:

OBSERVE: invocation

```
./grep.pl regex files
```

The first argument is the regex to match and the remaining arguments are the search files (omit them if you want to search standard input, for instance, if this is part of a shell pipeline). Okay, now try this:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ ./grep.pl '\bw\w' ./*.pl
```

It prints lines from the code we've entered in this lesson that contain words starting with "w." Have fun and experiment with other regexes!

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

Introduction to One-Liners

Lesson Objectives

When you complete this lesson, you will be able to:

- experience the power of one-liners.

"Right now I'm having amnesia and deja vu at the same time. I think I've forgotten this before." -Steven Wright

Perl at the Command Line

Picture this scene: It's five minutes before the company van pool is about to leave and your officemate comes to you in a panic. "Help!" he says, "I have these text files and I need to extract last year's sales data from them for tomorrow morning's management meeting. I need the data tonight so I can draw up my conclusions. Can you stay late and write a program that will retrieve my data and email the program to me at home later?" You study the file format sheet he's given you for a few seconds, and then reply casually, "No." But as he starts to bristle, you say, "I don't need to write a program or miss my ride home. One moment, please." Then, at a shell window, you change to the directory containing the text files and type a single command taking up the width of the terminal that baffles your colleague. A second later, you show him the output and pipe it into an email message to him. As you leave for the van pool with a minute to spare, he is left looking at the screen in shock.

That scenario is the power and possibility of the Perl one-liner, and if you think it's an exaggeration, I myself have been on the programmer's end of that transaction many times. In this lesson, we'll learn how to run Perl code without ever having to write a program. In fact, we won't write a single program during this lesson!

The relevant documentation for his lesson is **perldoc perlrun**. You might want to take a look at it now. Until now, we've only invoked Perl by writing a program, making it executable, and running it; the **#!** line at the beginning tells the operating system to run the program by passing it as input to another program called **perl**. Perl's path is on the **#!** line (except on the Windows operating system, where the **.pl** file extension does the same job).

But we can execute Perl code at the command line without writing a program. Because this code is provided in a single line (the only place you type **<RETURN>** is once at the end), we call it a *one-liner*. There are many short one-liners that perform identically to whole, specialized programs. You can find entire web pages dedicated to lists of useful one-liners.

Open a new terminal where you can type commands at the cold~$ prompt.

The usual Unix utilities are available, like **ls**, **chmod**, and so forth. You can also type **perl**, but if you do that without any arguments, it'll wait for you to type a program at the terminal, but nothing will happen (aside from the reporting of syntax errors) until you press **Ctrl-D** (signifying end-of-file) at the beginning of a line. There are all kinds of options we can pass to **perl** to change that behavior.

> **Note** The name of the program you run is **perl**, not **Perl**. Whenever I refer to the program itself, I'll use a lowercase **p**. When I'm talking about the language, I'll continue to use an uppercase **P**.

Go ahead and check the version of Perl. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -v

This is perl, v5.10.1 (*) built for x86_64-linux-thread-multi

Copyright 1987-2009, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl". If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.

cold:~$
```

That code reported on the version of Perl you're running. If you'd like to see a *lot* more detail, use an uppercase V instead. Whenever you see a one-liner like that in this lesson, type it at the command line to see what happens, and then experiment with changing it.

The -e Flag

Now let's see how we can execute code from the command line. This is done with the **-e** (**e**xecute) option. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -e


No code specified for -e.
cold:~$
```

That gives us an error. We have to provide code to execute! I guess Perl can't read our minds. Okay then, type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -e 'print "Hello, world!"'


Hello, world!cold:~$
```

I think you already know how to fix that output.

Because we're entering this command to a shell (the program that printed the prompt), we need to surround the Perl code with single quotation marks. That protects nearly all of the shell metacharacters that we may have put in the Perl code from being interpreted by the shell. The text that may still cause problems will be determined by the shell we're using. **!** is the character that most often needs to be backslashed in Perl code entered this way. But even then, that depends on what follows the **!**. Try this example. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -e 'print "Hello, world!\n"'
Hello, world!
cold:~$
```

See? No problem at all! And we added the newline (**\n**) just as we would in a program. But Perl supplies a handy option to save us from typing **\n** at the end of a **print** statement. It's the **-l** (that's a lowercase **l** for "**l**ine feed") option. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -le 'print "Hello, world!"'


Hello, world!
cold:~$
```

Run that, and you'll see Perl inserts a newline after **Hello, world!**. The **-l** flag causes Perl to behave as if every **print** statement has an extra final **"\n"** argument.

I've combined the two flags **-l** and **-e** into one option in the customary Unix way. Typing **-le** is identical to typing **-l -e**.

If you don't have a calculator handy, you can use Perl instead. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -le 'print 1/sqrt(1 - 0.9**2)' # Relativistic time dilation factor at nine
-tenths the speed of light

2.29415733870562
cold:~$
```

Using Perl as a calculator lets you assign variables as well. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ # Relativistic time dilation factor at 100,000 km/h:


cold:~$ perl -le '$c = 3E8; $v = 100_000*1E3/3600; print 1/sqrt(1 - ($v/$c) **2)'
1.00000000428669
cold:~$
```

Something about that code looks wrong. Where is the **my** keyword that declares **$c** and **$v**? Hmm. Well, is anything else missing? Yes: **use strict** and **use warnings**. But this isn't the way we've learned to write Perl. What's going on?

The **use strict** pragma (compiler directive) causes Perl to require variables to be declared with **my**; this is essential when writing programs to protect against accidental typos. A one-liner is short enough that we don't have to worry about typos. So we don't need to **use strict** and therefore don't have to declare our variables with **my**, which helps keep the one-liners shorter and more readable.

Note Let me reiterate: You can leave out **use strict** and **use warnings** and **my** from one-liners, but *not* from programs. You'll encounter programs written by other folks who have left them out, which causes all kinds of problems. Don't be that programmer.

The -n Flag

I think you're ready to experience the real power of one-liners. The perl program can take some options that cause it to behave as if code is present, when it actually isn't. For instance, **while (<>)** is "magic" code. When it is inside Perl, **while (<>)** expands to about a dozen lines that shift filenames off **@ARGV**, open them, read a line into **$_**, and then test it to see if the line is defined.

The **-n** flag is "magic" also. It causes Perl to assume that the code you provide is surrounded by a loop reading from

inputs. If you type **perl -n -e '*code*'**, it is roughly equivalent to this program:

OBSERVE: -n equivalent

```
while ( <> )
{
  code
}
```

That magic readline operator is going to expect you to provide filenames on the command line or it will read from standard input. So the filenames have to follow the code you provide using **-e**. (Technically, you can pass the arguments to **perl** in any order, and you could follow the **-e *code*** option with another option such as **-n**. But it is a universal convention that **-e** is the last option provided to perl, and it makes one-liners easier to read.)

We can (sort of) translate one of our last lesson's examples into this new format. Remember **grep.pl**? Try it as a one-liner (changing the matching character to "z" in order to get less output). Type the code below into the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl2
cold:~/perl2$ perl -ne 'print if /\bz\w/' ./*.pl
20500 is the zip code of the White House
My zip is _98362_    my ($city, $state, $zip) = ($1, $2, $3);
    printf "City: %-20s; State: $state; Zip: $zip\n", $city;
    my ($city, $state, $zip) = ($1, $2, $3);
    printf "City: %-20s; State: $state; Zip: $zip\n", $city;
20500 is the zip code of the White House
  print "'$line' contains a zip code\n"
$text = "I have two tomatoes and one zucchini";
  print "'$line' contains a zip code\n"
  /(\d{5}(?:-\d{4})?)/ and print "Found zip: $1\n";
my $zip;
$zip = "98362";
if ( length( $zip ) != 5 )
  die "$zip isn't a standard zip code\n";
elsif ( length( $zip ) != 9 && length( $zip ) != 10 )  # May have hyphen between parts
  die "$zip isn't a ZIP+4 code\n";
  print "$zip could be a zip code\n";
cold:~/perl2$
```

In this version we put the actual regex in explicitly instead of using **/$regex/**. But think about it; where would **$regex** come from if we had not entered the regex implicitly? The command line, right? And that's where our entire program is to begin with!

Now try adding in the **-l** flag as well. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ perl -nle 'print if /\bz\w/' ./*.pl


20500 is the zip code of the White House
My zip is _98362_
    my ($city, $state, $zip) = ($1, $2, $3);
    printf "City: %-20s; State: $state; Zip: $zip\n", $city;
    my ($city, $state, $zip) = ($1, $2, $3);
    printf "City: %-20s; State: $state; Zip: $zip\n", $city;
20500 is the zip code of the White House
  print "'$line' contains a zip code\n"
$text = "I have two tomatoes and one zucchini";
  print "'$line' contains a zip code\n"
  /(\d{5}(?:-\d{4})?)/ and print "Found zip: $1\n";
my $zip;
$zip = "98362";
if ( length( $zip ) != 5 )
  die "$zip isn't a standard zip code\n";
elsif ( length( $zip ) != 9 && length( $zip ) != 10 )  # May have hyphen between parts
  die "$zip isn't a ZIP+4 code\n";
  print "$zip could be a zip code\n";
cold:~/perl2$
```

Now wait just a minute! Why did that produce the same output? If the **-l** flag causes **print()** to add a newline to the end of what it prints, shouldn't we see a blank line between every line of output?

Let's back up a minute and think about why the output looked right when we had just the **-n** and **-e** flags. Perl read each line into **$_** and—if it matched the regex—printed it. The readline operator left the trailing **\n** in **$_** and **print()** printed it.

So why isn't there an extra newline with **-l**? Because in addition to attaching the newline to the end of **print()** statements, the **-l** flag performs an *implicit* **chomp** on lines read in through the magic **<>** readline operator.

That feature of **-l** may not have made a difference in this example, but it will come in handy often in the future, so when we use the **-n** flag, we'll also use the **-l** flag. You'll use the **-nle** flag combination often.

That's a nice example, but we already have the **grep** program on our system. (Unless you're on Windows, in which case Perl can save you from having to get the Unix utilities.) Let's get Perl to do something that we don't have a common program to do. Suppose you have text files containing stock inventory reports that have lines that look like this:

OBSERVE: stock report

```
Item    Quantity        Unit Price
Direhorse Reins         21   199
Ikran Saddle    12      341
Thanator Shield 41      99
Fan Lizard Reflectors   350     15
Sturmbeest Corral       1       32000
```

The fields are separated by tabs. If we were dumping the first line in Perl format, it would look like this: **"Item\tQuantity\tUnit Price\n"**. Tab-separated value format is a good choice for human-readable text files like this because it is easy to parse, does not require fixed-width fields, and permits spaces in field values. Use **wget** to copy the stock_report file to your /perl folder. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ wget "http://courses.oreillyschool.com/Perl2/stock_report"
--07:42:48--  http://courses.oreillyschool.com/Perl2/stock_report
           => `stock_report'
Resolving www.oreillyschool.com... 63.171.219.110
Connecting to www.oreillyschool.com|63.171.219.110|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 148 [text/plain]

100%[================================================================================
==================================================>] 148           --.--K/s


07:42:48 (5.65 MB/s) - `stock_report' saved [148/148]

cold:~/perl2$
```

Now, let's say that you want to compute the total value of each item in inventory, obtained by multiplying the unit cost by the quantity on hand. You could run a one-liner. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ perl -nle '/(.*)\t(\d+)\t(\d+)/ and print "$1: ", $2*$3' ./stock_report
Direhorse Reins: 4179
Ikran Saddle: 4092
Thanator Shield: 4059
Fan Lizard Reflectors: 5250
Sturmbeest Corral: 32000
cold:~/perl2$
```

We make sure the match succeeds before using **$1** and friends; we don't just multiply the second field by the third blindly, because that effort would be foiled at the first line. Sometimes people write code to omit the first line because it's a header, but that approach fails when the header line is either missing or duplicated later. Our approach avoids those problems by only processing valid lines.

Take a look at the regular expression and make sure you understand how it works for this input data. The **.*** will initially match the whole line, then it will backtrack as far as the last tab character in search of a match. Then it will backtrack again to the previous tab character, so that it can match one or more digits after that, followed by a tab, which is followed by one or more digits. There is no need for anything more specific in this regex, which will not match the header line because the second and third fields are not composed of digits.

Mastering the one-liner can make the scenario I described at the beginning of this lesson a reality. Now, *that's* capability worth the price of admission! And there's more to come!

BEGIN and END blocks

Suppose instead of printing the total value of *each* item in inventory, we want to print the total value of *all* items in inventory. We would still use **-n** to loop over all lines of input, adding the value to a running total; we'd also want to print the total. But how can we do that when all the code we put in the **-e** argument is executed inside the implicit **while** block generated by **-n**?

The answer lies in a special capability of Perl called the **END** block. The **END** block declares a special subroutine (but without the word **sub**) that Perl will execute on your behalf when all other code has finished and the program is about to exit. Run the one-liner below in the Unix window as shown:

INTERACTIVE SESSION:

```
cold:~/perl2$ perl -nle '/\t(\d+)\t(\d+)/ and $total += $1*$2; END{ print "Total: $tota
l" }' ./stock_report

Total: 45401
cold:~/perl2$
```

The block of code following **END** will execute after all other code has finished, which means that the **while** loop will have finished executing, and so the **END** block will be executed only once.

The variable **$total** isn't declared with **my**, but that's okay, because we don't have **use strict**. We just have to make sure that we type **$total** correctly both times. Since the output will be zero if we get it wrong, that shouldn't be difficult.

Notice that *this* regex neither matches, nor captures the first field. It doesn't have to because as long as we match the two tab characters, we will capture the second and third fields that we want.

There is a counterpart to **END** called **BEGIN**, which declares code to be run before all other code starts. As is the case with **END**, **BEGIN** will be run only once, even if we specify the **-n** flag, because it is run before the **while** loop *starts*.

Now, suppose we want to write a one-liner that really works just like our **grep.pl** program, and take the regex as the first argument on the command line. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ perl -ne 'BEGIN{ $regex = shift } print if /$regex/' '\bz\w' ./*.pl
20500 is the zip code of the White House
My zip is _98362_
  my ($city, $state, $zip) = ($1, $2, $3);
  printf "City: %-20s; State: $state; Zip: $zip\n", $city;
  my ($city, $state, $zip) = ($1, $2, $3);
  printf "City: %-20s; State: $state; Zip: $zip\n", $city;
20500 is the zip code of the White House
 print "'$line' contains a zip code\n"
$text = "I have two tomatoes and one zucchini";
 print "'$line' contains a zip code\n"
 /(\d{5}(?:-\d{4})?)/ and print "Found zip: $1\n";
my $zip;
$zip = "98362";
if ( length( $zip ) != 5 )
   die "$zip isn't a standard zip code\n";
elsif ( length( $zip ) != 9 && length( $zip ) != 10 ) # May have hyphen between parts
  die "$zip isn't a ZIP+4 code\n";
print "$zip could be a zip code\n";
cold:~/perl2$
```

Do you understand how that works? There are many other uses for a **BEGIN** block in a one-liner, such as opening an output file used to print calculations from within the implicit **while** loop.

Now I'd like to introduce quite possibly the most useful one-liner you will ever see. I have used variations of it countless times. Ladies and gentlemen—the *concordance* one-liner! A one-liner for producing a count of items seen in the input, by counting them in a hash. (In my examples below I use a hash named **%h**, but there's nothing special about that name whatsoever, and you could use a hash of any name you want.) It usually contains an **END** block of the form: **END{ print "$_: $h{$_}" foreach keys %h }**. It can use the postfixed **foreach**, but any of the following forms may be used instead:

OBSERVE: END block variations

```
END{ print "$_: $h{$_}" foreach sort keys %h }
END{ print "$_: $h{$_}" foreach sort { $h{$a} cmp $h{$b} } keys %h }
END{ print "$_: $h{$_}" foreach sort { $a <=> $b } keys %h }
END{ print "$_: $h{$_}" foreach sort { $h{$b} cmp $h{$a} } keys %h }
END{ print "$_: $h{$_}" foreach sort { $h{$a} <=> $h{$b} } keys %h }
END{ print "$_: $h{$_}" foreach sort { $h{$b} <=> $h{$a} } keys %h }
END{ print "$_: $h{$_}" foreach keys %h }
```

Take a look at those variations. They represent various ways of printing a list sorted "asciibetically" or numerically, either by key or by value from the hash. Can you assign the correct description to each block?

The way we populate **%h** varies, but whatever way we choose, it's going to happen inside the implicit **while** loop from the **-n** flag. Here's an example; say we have a lengthy input file containing IP addresses of machines that have attempted to connect to our machine at given times. The file starts like this:

OBSERVE: connection report

```
Time Address
00:12:58 29.77.211.85
00:15:48 253.173.225.68
00:18:17 156.79.243.123
00:21:10 36.134.160.68
```

A single space character separates the fields. You have been asked to find out which addresses have attempted the most connections, and how many attempts have been made. Use **wget** to copy the **connections** file into your /perl2 directory. Type the code below in the Unix window as shown:

INTERACTIVE SESSION:

```
cold:~/perl2$ wget "http://courses.oreillyschool.com/Perl2/connections"
--08:32:47--  http://courses.oreillyschool.com/Perl2/connections
          => `connections'
Resolving www.oreillyschool.com... 63.171.219.110
Connecting to www.oreillyschool.com|63.171.219.110|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3,879 (3.8K) [text/plain]

100%[=====================================================================================
=================================================>] 3,879         --.--K/s


08:32:47 (49.99 MB/s) - `connections' saved [3879/3879]
cold:~/perl2$
```

Then, type this one-liner into the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ perl -nle '/ (\d.*)/ and $h{$1}++; END{ print "$_: $h{$_}" foreach sort {
 $h{$b} <=> $h{$a} } keys %h }' ./connections
(some output removed for the sake of brevity; your output will differ)
130.247.245.62: 1
22.127.4.96: 1
47.68.249.66: 1
91.68.249.66: 1
159.150.244.67: 1
7.3.203.82: 1
28.171.37.72: 1
190.199.46.207: 1
72.243.168.209: 1
178.124.131.63: 1
27.9.161.165: 1
cold:~/perl2$
```

Wow. Now, to help management read this (because invariably their response to that report will be, "Too much information! I just want to see the top offenders"), let's pipe it through the Unix **head** command (of course, you don't need to type the entire line again; just press the up arrow key, then press the **End** key to move the cursor to the end of the line, and type the additional code as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ perl -nle '/ (\d.*)/ and $h{$1}++; END{ print "$_: $h{$_}" foreach sort {
 $h{$b} <=> $h{$a} } keys %h }' ./connections | head -5
cold:~/perl2$
```

head is a program that just prints out the first few lines of its input, in this case, the first five lines (because of the **-5** argument).

Let's dissect this one-liner. It's the longest one we've written to date, so I'll turn it into the equivalent program for the purpose of understanding it:

Program Equivalent

```
while ( <> )
{
  / (\d.*)/ and $h{$1}++;
}

foreach ( sort { $h{$b} <=> $h{$a} } keys %h } )
{
  print "$_: $h{$_}";
}
```

Let's look at the regex first. Remember that spaces match literally inside regexes, so that leading space (between the first "/" and the "(") forces the parentheses to capture after the end of the first field (the timestamp); the \d at the start of the capture group means the heading line ("Time Address") won't match. We're matching a space followed by a digit and then as many more characters as exist in the line (.*).

The parentheses around \d.* mean that the input that matches that part (the IP address) will be saved in **$1**. We use that as a key in a hash **%h**—*we have not declared that hash;* again, you don't need to declare variables in one-liners (there's no **use strict** statement).

We postincrement the hash value $h{**$1**} with **++**. Perl will create each hash entry the first time we reference it, so the first time we capture any particular address in **$1**, this statement will attempt to increment the corresponding hash entry, find that it's not in the hash, automatically create it with a value of undef, and then increment it, turning the value into 1. This all occurs inside the while loop (which is explicit in the equivalent program but *implicit in the one-liner)*.

Then, at the end of the program, we loop over all of the keys of the hash, sorted by descending numerical order of the corresponding value (the number of times each IP address was seen), printing out the IP address and the number of

times we saw it.

Just so you understand that there's nothing sacred about the name **%h**, try it again with a hash **%d**:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ perl -nle '/ (\d.*)/ and $d{$1}++; END{ print "$_: $d{$_}" foreach sort {
 $d{$b} <=> $d{$a} } keys %d }' ./connections | head -5
cold:~/perl2$
```

The point here is that in typing a one-liner, we tend to use one-letter variable names to keep the lines short.

I get asked for that kind of information all the time, and using a one-liner like that, I am able to provide it. You're looking at seriously useful Perl code for solving real-world problems quickly; learn it, live it, love it!

The principal value of the one-liner is as a one-off piece of code that you literally type just once. You might find yourself saving a one-liner in a shell alias, but that would be about the extent of reuse.

The possible uses of the match operator are endless. I used one a few minutes ago while starting the next lesson. I had stored my code examples in subdirectories named **Lesson1**, **Lesson2**...**Lesson9**, and was about to create **Lesson10** when I realized that the directories wouldn't sort in the order I wanted when I did an **ls** in the parent directory. The solution was to rename those directories to **Lesson01**, **Lesson02**...**Lesson09**. But that's nine **mv** commands to type—so tedious! Instead, I enlisted the help of Perl's **rename** function and typed this:

OBSERVE: rename pipeline

```
ls | perl -nle '/(Lesson)(\d)/ and rename $_, "${1}0$2"'
```

The **ls** command produces a list of every file in the current directory (I could do this from within Perl, but this is a simpler approach), and because **stdout** is a pipe, it will have one file per line. Perl's **-n** flag processes that list one file at a time, and the **-l** flag autochomps each line. The match operator looks for files of the form **Lesson** followed by a digit (there might be other files in the directory), and when it matches, **rename**s the file to the same name with a zero inserted before the digit. Be aware that this would *not* have worked if I had executed it *after* creating the **Lesson10** directory. (Why? And how would you fix it?)

Oh, and I snuck something new in there (more is more!). I needed to construct a string that contained the variable **$1** followed by the digit 0 and then the variable **$2**. But if I just typed **"$10$2"**, Perl would think I was referring to the variable **$10**. Of course, I could use the concatenation operator to get around this: **$1** . **'0'** . **$2**. But instead I used an alternative syntax for Perl variable names that tells Perl exactly where the variable name begins and ends—with the curly brackets. So, you can write strings like **"I have $count ${fruit}s"** to get a result like **"I have 4 apples"**.

You're really getting the hang of this stuff now, huh? Good job! After you've finish your assignments, I'll see you in the next lesson...

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

Substitution and More One-Liners

Lesson Objectives

When you complete this lesson, you will be able to:

- use the substitution operator in more diverse situations.
- use substitution in one liners.

We'll take a diversion from our discussion of one-liners for the first part of this lesson to discuss Perl's substitution operator. Then we'll go over how it can be used in more one-liners!

Substitution

We've used the **substitution operator** (**s///**) in this course already, but without understanding it fully. The **s///** operator combines the regular expression matching operator with the ability to change the part of the input that was matched to some double quoted string, all in a minimalist syntax. Here's what it looks like:

OBSERVE: Substitution Operator
`s/regex/replacement/`

Just like the match operator, you can use other delimiters if you want; unlike the match operator, the **s** at the beginning cannot be omitted when the delimiter is **/**. If you want the regex to be case-insensitive, add the **/i** modifier at the end:

OBSERVE: Substitution Operator
`s/regex/replacement/i`

So how do you use it? Just like the match operator; either bound to an explicit variable:

OBSERVE: Bound Substitution
`$text =~ s{cat}{dog}  # Note use of paired delimiters`

...or acting implicitly on **$_**:

OBSERVE: Substitution on $_
`s/p(?:oo|[uiea])ddle/fiddle/`

In the end, whatever was matched by the regex between the first set of delimiters, is changed to whatever is between the second set of delimiters. This change takes place in the variable explicitly or it is implicitly bound to the substitution operator; that is, that variable is modified in place. The *result* of the **s///** is true if the regex matched. It's false if it didn't.

Okay, let's see it in action! Create **substitute.pl** as shown:

CODE TO TYPE: substitute.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $date = localtime();
$date =~ s/ (\d\d:\d\d:\d\d)// or die "$date not in expected format!\n";
my $time = $1;
while ( defined( my $line = <DATA> ) )
{
  chomp $line;
  $line =~ s/X_DATE_X/$date/;
  $line =~ s/X_TIME_X/$time/;
  print "$line\n";
}

__END__
The date is X_DATE_X today
and the time is X_TIME_X;
i.e., it is X_TIME_X on X_DATE_X
```

Check Syntax and run it. This is actually a common pattern, it's called *templating*. It looks for special markers in text and turns them into computed values at run-time. I'll walk you through each of the substitutions in this program:

- **$date** =~ **s/(\d\d:\d\d:\d\d)//**
  $date is set to the scalar context result of **localtime**, which is something of the form **"Day Mon DD HH:MI:SS YYYY"**; for example, **Tue May 11 21:41:40 2010** (It might be different if you run this on a system with a non-English locale). I want to extract the **HH:MI:SS** part and call that the time, and then call the rest of the string in **$date** after **HH:MI:SS** is removed. So the date in this example would be **Tue May 11 2010** and the time would be **21:41:40**. This first substitution removes the time, including the leading space, from **$date**. It also saves the time in a capturing parenthesis group—the regex part of a substitution is a proper regex, so if it matches, capturing groups will save to **$1**, **$2**, and so on. The next line of code copies **$1** to **$time**. (The **die** execution path will only be taken if the return from **localtime** is incorrect; unlikely, but we always want to follow the good practice of testing for success before using **$1**.)
- **$line** =~ s**/X_DATE_X/$date/**
  Replace **X_DATE_X** in **$line** with the current date.
- **$line** =~ s**/X_TIME_X/$time/**
  Replace **X_TIME_X** in **$line** with the current time.

See how variables get interpolated in the replacement string? That string is interpreted as a double-quoted string. Let's do another example. Create **dedup.pl** as shown:

CODE TO TYPE: dedup.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  s/([A-Z]{3})\s+\1/$1/;
  print;
}

__END__
 32   1.334   DEX FDU
968  95.348   FTY JOS
874   2.230   KLS KLS
348 237.849   RTR ISH
742  83.018   FTW FTW
```

Check Syntax and run it. It parses a report format with lines that consist of an integer, a floating point number, and then two acronyms, which are sometimes the same. Don't try to figure out what this data means (you won't get far; I made it up!), or why your goal is to eliminate duplicated acronyms from the report; for now, let's just see how this program works.

All of the program's logic is contained within the substitution that takes place on **$_**. The regex part matches three consecutive uppercase letters (saved into **$1**), followed by mandatory whitespace, followed by—wait a minute, what's this **\1** doing there?

\1 tells Perl that you want to match *whatever the first capturing group* just matched (so there'd better be capturing parentheses before that **\1**). You can not get that effect in the regex by using **$1** because **$1** isn't populated by Perl until the whole regex match has completed. Instead, you would get the *previous* value of **$1** (which would be **undef** the first time out, but after any successful match and capture, it would be whatever was left over from the last regex). That would be a hard bug to find, so for now, avoid using **$1** *inside a regex*.

The replacement string in **s///** is not in the regex, so we can use **$1** there. So our substitution operator reads like this: "Start matching; capture three uppercase letters, then whitespace followed by the same letters, and substitute everything you matched with the letters you captured." That's how we remove duplicates!

Substitution in One-liners

The substitution operator is a favorite tool used in one-liners. For instance:

OBSERVE: s/// one-liner

```
perl -nle 's/(\w{3,})\s+\1/$1/;print' input_file...
```

That's a one-liner version of the program **dedup.pl**. Do you see how the one-liner helps you save time and work? Well, hold on, because we can save even more.

The -p Flag

It's so common to want to do a one-liner of the form we just saw—read lines in a loop into **$_**, do some transformation on **$_** and then print it—that Perl has a special flag you can use instead of **-n**; it's the **-p** flag (the "p" stands for "print" and the "n" stands for "no printing").

using **-p**, our previous one-liner can be shortened further to this:

OBSERVE: -p one-liner

```
perl -ple 's/(\w{3,})\s+\1/$1/' input_file...
```

It's really common to transform a text file (or files) by performing a substitution on each line, that's wrapped up in the one-liner with **-ple** tags. You may think that we've just reinvented the **sed** program, but Perl's regular expressions are a lot more powerful than **sed**'s. And, if you ever want to add more transformational code using the substitution for extra lines of computation, you can do that.

Try that one-liner right now. Edit **dedup.pl** to extract just the text that comes after the **__END__** line into a file named **./perl2/dup_data.txt**, and then run it:

INTERACTIVE TERMINAL SESSION

```
cold:~$ cd perl2
cold:~/perl2$ perl -ple 's/(\w{3,})\s+\1/$1/' ./dup_data.txt
```

To recap, the **-p** flag assumes a loop of:

OBSERVE: Implicit -p code

```
while ( <> )
{
   # code
   print;
}
```

...around your **-e** *code*. It does not work with **-n**; using both together will result in an error.

The -i Flag

In another common scenario, you are given one or more text files and then asked to transform them in some way expressed in terms of a substitution, like this: "Take these files and change the date format from MM/DD/YY to YYYY-MM-DD," or "Give me a new set of files just like these, but with all the names of the staff changed from uppercase to title case," or "Change the name and model number of the product in these press releases to the latest version and send them over to marketing." The task of changing text in files comes up so frequently that Perl has a special flag to facilitate it: the **-i** flag.

-i is always used in conjunction with **-p**; it modifies the destination of **print** statements so that they write back to the files you already read from within the **-p** loop.

That's a lot to wrap your brain around in one go, so let me break it down. The **-i** flag takes an argument that is an extension used to rename the original files so that you have a backup copy. When our program invokes **perl -p -i.bak -e '*code*' file**, it means this:

1. Rename file to file.bak.
2. Open file.bak for reading.
3. Open file for writing.
4. For each line in the input:
    1. Read it into **$_**.
    2. Run *code*.
    3. Print **$_** to the output.
5. Close the input and output files.

And if there are any more files named on the command line, we'd do the same for them as well. (The **-i** is ignored if there are no files named and input comes from standard input.)

That's a *lot* of power to pack into a little letter! And **-i** has one other trick available: if you don't provide an extension argument for it, then it will modify the input files *in place,*; that is, with no backup. You need to be really confident in your Perl skills before you use that mode.

Let's look at a few examples of the **-i** flag:

OBSERVE: dos2unix

```
perl -pi.bak -e 's/\r\Z//' file...
```

This little gem performs the same function as the Unix program dos2unix: it converts text files that came from a Windows system (where the line separators are, in a Unix Perl terminology, **\r\n**) to Unix format (where the line separators are just **\n**). To do that, we need to get rid of any **\r** that's followed by a **\n**. The **\Z** anchor is closely related to the one we've seen already: **\z**, but where **\z** matches the end of the string, **\Z** matches the end of the string **or** a newline (**\n**) that is followed by the end of the string.

You can put the **-l** flag in as well, but it's more common for people to leave it out when using the **-i** flag. Using **-l** in this case means that the newline gets stripped off during input and put back in at the end of the implicit **print** to the output; it's not going to have any effect unless for some bizarre reason we put **print()** statements in our **-e** code.

Let's try this example now:

INTERACTIVE SESSION:

```
cold:~/perl2$ perl -pi.bak -e 's/(\w{3,})\s+\1/$1/' dup_data.txt
```

Verify that it has updated **dup_data.txt** and that the original file is in **dup_data.txt.bak**.

Often you'll need to do some reformatting. Use **wget** to copy the **cd_data** file to your /perl folder as shown. Type the code below in the Unix window as shown:

INTERACTIVE SESSION:

```
cold:~/perl2$ wget "http://courses.oreillyschool.com/Perl2/cd_data"
--14:58:39--  http://courses.oreillyschool.com/Perl2/cd_data
           => `cd_data'
Resolving www.oreillyschool.com... 63.171.219.110
Connecting to www.oreillyschool.com|63.171.219.110|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 906 [text/plain]

100%[=====================================================================================
==================================================>] 906           --.--K/s


14:58:39 (13.50 MB/s) - `cd_data' saved [906/906]
cold:~/perl2$
```

Then, try this example. Type the code below in the Unix window as shown:

INTERACTIVE SESSION:

```
cold:~/perl2$ perl -pi.bak -e 's/\A(\d+)\.\s+/$1:\n\t/' ./cd_data
```

That code performs the task of indenting the rest of the information about each track on a line after the track number. Let's try another one. Put the original file back and type the code below in the Unix window as shown:

INTERACTIVE SESSION:

```
cold:~/perl2$ mv ./cd_data.bak ./cd_data
cold:~/perl2$ perl -pi.bak -e 's/\A(\d+)\.\s+(.*)\s+([\d:]+)\Z/$1 ($3)\n\t$2/' ./cd_dat
a
```

You can embed character class shortcuts inside character classes, so for instance, in our example **[\d:]** is a character class for digits and the colon.

To make sure you understand the effect of **-l** and the difference between **\z** and **\Z**, try that example with the other possible combinations of the different flags, and also using **\z** instead of **\Z**.

Go over that last example, carefully inspecting each part of the regex and substitution, until you are sure you understand how it works.

Note If you are uncertain about the effects of the **-i** operation, you can test it without having to rename the backup file after each attempt. Just leave out the **-i** flag and get standard output instead of overwriting the input file(s).

I'll bet you are going to see *lots* of ways you can use substitution and the **-i** flag in your everyday work and leisure coding from now on!

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

More Perl Flags; More Perl Operators

Lesson Objectives

When you complete this lesson, you will be able to:

- use the -w Flag.
- use the -Mstrict Flag.
- use the -c Flag.
- use the Trinary Operator.
- use the Scalar Range Operator.

The -w Flag

If you get ambitious with the **-e** flag, you'll appreciate the benefits of turning on warnings. Instead of having to type **use warnings** at the beginning of the **-e** argument, you can include **-w** in the list of perl flags.

A handy use of **-e** is to test out some feature of Perl quickly, before using it in a program you're developing. Suppose that you want to perform host name to IP address resolution and you think you heard somewhere that Perl has the **gethostbyname** function. You confirm this with **perldoc -f gethostbyname**, and see the basic syntax for using it. But you want to be sure you're using it properly before you put it in your program, and the documentation you see describes over a dozen other functions at the same time. A quick and dirty one-liner is a perfect tool to make sure you're using the function the right way. Since whatever you learn is going to be used in a real program, you may want to make sure that warnings are enabled in your one-liner as well. No problem! Even a typo that results in a nonexistent host is easier to diagnose with **-w**. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -MSocket -le 'print inet_ntoa scalar gethostbyname "www.perl.org"'
cold:~$ perl -MSocket -le 'print inet_ntoa scalar gethostbyname "www.perl.orc"'
Bad arg length for Socket::inet_ntoa, length is 0, should be 4 at -e line 1.
cold:~$
```

Now try it with **-w**. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -MSocket -wle 'print inet_ntoa scalar gethostbyname "www.perl.org"'
cold:~$ perl -MSocket -wle 'print inet_ntoa scalar gethostbyname "www.perl.orc"'
Use of uninitialized value in subroutine entry at -e line 1.
Bad arg length for Socket::inet_ntoa, length is 0, should be 4 at -e line 1.
cold:~$
```

This tells us that we should check the return from **gethostbyname** before deciding whether to pass it to **inet_ntoa**. This is in agreement with the documentation. So we come up with this version:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -MSocket -wle '$packed_addr = gethostbyname "www.perl.org"; defined $packe
d_addr and print inet_ntoa $packed_addr'
```

It's almost ready to paste into a program. We'd just need to add in **my** to get it to work with **use strict**. Which brings us to our next topic...

The -Mstrict Flag

So you've seen that **-w** invokes **use warnings** in a one-liner; now what about **use strict**? There isn't a single letter flag that gives you the same capability. But you can use the **-M** flag, which is equivalent to **use**, and pass **strict** as an argument, giving you this:

OBSERVE: -Mstrict

```
perl -Mstrict ...
```

Now the code passed to **-e** must pass strictness checks as well.

Since **-M** is a flag that means **use**, **-Mwarnings** should be the equivalent of **use warnings**. The **-w** flag dates back to the earliest version of Perl, preceding the introduction of the **use** statement. Its effect is technically slightly different from that produced by **use warnings**, but the difference is not relevant until you get much further along with Perl programming. For now, just know that in this course you're learning the best practices.

The -c Flag

Sometimes you want to find out whether a program *compiles* correctly without actually running it.

We can do that. Open the **reg_capture3.pl** you created in an earlier lesson and modify it as shown:

CODE TO TYPE: reg_capture3.pl

```
#!/usr/bin/perl
use strict;
use warnings;

use Socket;

while ( my $line = <DATA> )
{
  chomp $line;
  if ( /(\S+)\s+is\s+(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/ )
  {
    my ($name, $ip) = ($1, $2);
    my $cname = gethostbyaddr( inet_aton( $ip ), AF_INET );
    print "$name -> $ip -> $cname\n";
  }
                        ( $line =~ /(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\s+is\s+(\S+)/ )
  elsifelseif ( $line =~ /(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\s+is\s+(\S+)/ )
  {
    my ($ip, $name) = ($1, $2);
    my $lookup;
    if ( defined( $lookup = gethostbyname( $name ) ) )
    {
      $lookup = inet_ntoa( $lookup );
    }
    else
    {
      $lookup = "Couldn't resolve!";
    }
    print "$ip -> $name -> $lookup\n";
  }
  else
  {
    print "No match for '$line$_'\n";
  }
}
__END__
www.oreillyschool.com is 63.171.219.89
This line won't match
208.201.239.101 is www.perl.com
157.166.226.25 is cnn.com
```

 and run it. Type the code in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl2
cold:~/perl2$  perl -c ./reg_capture3.pl
elseif should be elsif at ./reg_capture3.pl line 16.
Global symbol "$line" requires explicit package name at ./reg_capture3.pl line 16.
syntax error at ./reg_capture3.pl line 17, near ")
{"
Can't use global $1 in "my" at ./reg_capture3.pl line 18, near "($1"
syntax error at ./reg_capture3.pl line 30, near "else"
./reg_capture3.pl had compilation errors.
cold:~/perl2$
```

This example illustrates how fast one error can cascade into many. Start by tackling the first error from compiling or running a program, fix it, then compile or run again. Later errors may be caused by the first one. In this case, Perl identifies the problem on the first line. Let's fix the code:

CODE TO EDIT: reg_capture3.pl

```
#!/usr/bin/perl
use strict;
use warnings;

use Socket;

while ( <DATA> )
{
  chomp;
  if ( /(\S+)\s+is\s+(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/ )
  {
    my ($name, $ip) = ($1, $2);
    my $cname = gethostbyaddr( inet_aton( $ip ), AF_INET );
    print "$name -> $ip -> $cname\n";
  }
  elseifelsif ( $line = /(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\s+is\s+(\S+)/ )
  {
    my ($ip, $name) = ($1, $2);
    my $lookup;
    if ( defined( $lookup = gethostbyname( $name ) ) )
    {
      $lookup = inet_ntoa( $lookup );
    }
    else
    {
      $lookup = "Couldn't resolve!";
    }
    print "$ip -> $name -> $lookup\n";
  }
  else
  {
    print "No match for '$_'\n";
  }
}
__END__
www.oreillyschool.com is 63.171.219.89
This line won't match
208.201.239.101 is www.perl.com
157.166.226.25 is cnn.com
```

Rerun the **perl -c** command. Now the syntax is okay, but perl doesn't run the program. There might still be errors in the code that are only found at runtime.

If the program includes a **use strict** directive, **-c** will cause the program to be compiled with strictness checking enabled; there is no need to add **-Mstrict**.

The Trinary Operator

Cast your mind back to the Regular Expressions lesson (I know, it seems so long ago with everything that we've learned since then!), and open **reg_literal2.pl** again for this lesson:

OBSERVE: reg_literal2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

foreach my $text ( qw(Matt bats the ball at Atticus) )
{
  print qq{"$text" };
  if ( $text =~ /at/ )
  {
    print "matches";
  }
  else
  {
    print "does not match";
  }
  print " /at/\n";
}
```

I want you to be *dissatisfied* any time you have code that doesn't look beautiful or map cleanly to the solution you've devised for a problem. When we translate the solution to a problem from its inception in our minds, into a computer language, some of its original elegance is always lost. Ideally, we'll use a language and write our programs such that degradation is minimized. One way we'll know that we have not translated our vision cleanly is by the presence of repetitive code.

If we apply that philosophical tangent to the program we're working on now, it would go something like this: The **if...else** statement in our program is unsatisfying for some reason. The repetition of the **print** statement may be a clue. The loop contents are: "Print **$text**, followed by "matches" or "doesn't match" depending on whether the regex matched, followed by the closing text." When we wrote our program, the only syntax we had available to us was the **if...else** statement, so we had to compromise.

But Perl incorporates an operator that can help improve our existing code. It has various names: trinary operator, ternary operator, hook operator, tertiary operator. That's a lot of verbiage for one operator; here's what it looks like:

OBSERVE: Trinary Operator

```
condition ? true_result : false_result
```

The value of this expression is *true_result* if *condition* is true, and *false_result* if *condition* is false. It works in scalar context and also list context:

OBSERVE: Trinary Operator Examples

```
$sign = $number < 0 ? '-' : '+';

@tokens = /(\w+)\s+(\w+)/ ? ($1, $2) : @DEFAULT_TOKENS;
```

In the second line, the regex is matching against **$_**; if it succeeds, the two capturing groups get put into **@tokens**, otherwise **@tokens** is assigned from **@DEFAULT_TOKENS**.

Those examples both involve assignments, but the trinary operator shows up just as often as part of a larger expression. Modify **reg_literal2.pl** as shown:

CODE TO EDIT: reg_literal2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

foreach my $text ( qw(Matt bats the ball at Atticus) )
{
  print qq{"$text" },
  $text =~ /at/ )
  {
    print "matches";
  }
  else
  {
    print "does not match";
  }
  print " /at/\n";
  print qq{"$text" }, $text =~ /at/ ? "matches" : "does not match", " /at/\n";
}
```

Check Syntax and run it. Now *that* is some serious code streamlining!

The trinary operator is often helpful in one-liners. Let's modify one from earlier in this lesson. Type the code below in the Unix window as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ perl -MSocket -wle '$packed_addr = gethostbyname shift; print defined $pa
cked_addr ? inet_ntoa $packed_addr : "unknown host"' www.perl.orc
unknown host
code:~/perl2$
```

Now fix the typo in the host name. You can just use the up arrow key to repeat the code and then edit it as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl2$ perl -MSocket -wle '$packed_addr = gethostbyname shift; print defined $pa
cked_addr ? inet_ntoa $packed_addr : "unknown host"' www.perl.orcg
207.171.7.63
cold:~/perl2$
```

Now it prints an IP address. This example also illustrates how you can shift a parameter to the input list in **@ARGV** in a one-liner; the command line arguments to the **-e** code come after the **-e** code, and we get at them by referencing **@ARGV** or by calling **shift** outside of a subroutine.

Nesting the Trinary Operator

The precedence of the trinary operator is such that it can be nested without parentheses to provide the functional equivalent of an **if...else** block, with an arbitrary number of **elsif** clauses. Format this carefully to make sure it's readable. Let's revisit a familiar example from the first O'Reilly School Perl course. Create **bmi2.pl** as shown:

CODE TO TYPE: bmi2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

die "Usage: $0 height (inches) weight (pounds)\n" unless @ARGV == 2;

my ($height, $weight) = @ARGV;
my $bmi = $weight * 703 / $height ** 2;

print $bmi < 18.5 ? "Underweight"
    : $bmi < 24.9 ? "Normal"
    : $bmi < 29.9 ? "Overweight"
    :               "Obese",
    "\n";
```

The formatting makes all the difference in readability here. If that clause were written in a single line, you'd have a much harder time understanding or verifying its operation:

OBSERVE: Bad formatting

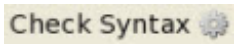
```
print $bmi < 18.5 ? "Underweight" : $bmi < 24.9 ? "Normal" : $bmi < 29.9 ? "Over
weight" : "Obese", "\n";
```

The Trinary Operator as Lvalue

This modest little operator, unlike its counterpart in some other languages, can even be used as an **lvalue** in Perl. To see how, create **lvalue_trinary.pl** as shown:

CODE TO TYPE: lvalue_trinary.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my ($schnauzer, $mackerel) = (7, 12);
my $species = 'fish';
my %current_count = ( mammal => 1001, fish => 7269 );
if ( $species eq 'mammal' )
{
  $schnauzer += $current_count{$species};
}
else
{
  $mackerel += $current_count{$species};
}

print "$schnauzer, $mackerel\n";
```



and run it. Now change it to use the **lvalue** trinary operator:

CODE TO EDIT: lvalue_trinary.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my ($schnauzer, $mackerel) = (7, 12);
my $species = 'fish';
my %current_count = ( mammal => 1001, fish => 7269 );
( $species eq 'mammal' ? $schnauzer : $mackerel ) += $current_count{$species};
if ( $species eq 'mammal' )
+
  $schnauzer += $current_count{$species};
+
else
+
  $mackerel += $current_count{$species};
+
print "$schnauzer, $mackerel\n";
```

Check Syntax and run it again. See how much more readable and natural that is?

The trinary operator may remind you of the **if...(then)...else** block. Perl won't stop you from using it instead of such a block, but it is really bad style to use the trinary operator in void context, that is, where you are not using its result, but for control flow. That could confuse your readers.

The Scalar Range Operator

Remember the humble range operator **..** that produces a list of consecutive increasing numbers?:

Range Operator

```
@small_integers = 1 .. 99;   # Numbers 1 thru 99 inclusive
```

When I introduced it, I only showed its effect in *list* context. But in *scalar* context, the range operator has positively magical behavior. It acquires a memory of its *state*; in this case, it has two states and is therefore a *bistable flip-flop*. Specifically, every time it is evaluated (and by "it" I mean "this instance in the code of the scalar range operator, not one somewhere else, even if it has the same arguments; this operator is only meaningful inside a loop) in the initial state, Perl evaluates just the expression on the left side, until that expression returns true; after that, the range operator evaluates only its right hand side and returns true until the right hand side becomes true, after which it returns false again.

That's a lot to take in—let's try an example. Create **scalar_range.pl** as shown:

CODE TO TYPE: scalar_range.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  if ( /\A\Z/ .. /\A\.\Z/ )
  {
    print "Body line: $_";
  }
  else
  {
    print "Other line: $_";
  }
}

__END__
From: Peter Scott <peter@psdt.com>
To: Scott Gray <scott@oreilly.com>
Subject: Perl 2, Lesson 11

Hi Scott.  I just uploaded Lesson 11 of the
Intermediate Perl series.  The students are
really going to love the new one-liner capabilities!
.
Junk that's not part of the message...
```

Check Syntax and run it. The specification for email messages is that the body text comes after the first blank line; here we'll also pretend that a line containing just a period ends the email message and anything following that is not part of the message.

You can see the effect of the flip-flop here. The expressions left and right of the operator are both regular expressions operating by default on **$_**. Until the expression on the left becomes true, the operator will return false; the left-side regex (**/\A\Z/**) is true when we reach a line that contains nothing between the beginning of the string and the end of the string, which may be preceded by a newline character (that's our blank line).

Thereafter, the scalar range operator will return true until its right side becomes true; the right-side regex (**/\A\.\Z/**) is true when we reach a line that consists of precisely a period, optionally followed by a newline.

You can have any expression you want on either side. The scalar range operator still has one more trick. If either the left or the right side is an integer, it will be compared to the line number of the current input stream. Create **numbering.pl** as shown:

CODE TO TYPE: numbering.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  print if ( 3 .. 5 ) || ( 10 .. 11 );
}

__END__
Dear Sir,

I received your request for a reference for Mr. Ronald Smith,
who is interviewing with your company and hoping to obtain
employment.  I am quite happy to report that
he performed his duties with exemplary conduct during his
entire tenure at our firm and we were eminently satisfied
with his product.
Whenever we observed him creating computer programs,
it was a pleasure to see him go; I cannot find the
words to describe how disgusted we were by his output
of a letter of resignation.
```

Check Syntax and run it. Oh my—we *do* need to check our output! We've actually got two scalar range operators at work there, which resulted in printing only lines 3 through 5 and 10 through 11 of the DATA stream.

You're doing great so far, and adding lots of handy new tools to your toolbox! Keep it up!

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

More on Numbers and Strings; More on Regular Expressions; More on Hashes

Lesson Objectives

When you complete this lesson, you will be able to:

- use string and Number Literals.
- use character Functions.
- use number Functions.
- use split().
- use hashes: each().
- use read().
- use >

String and Number Literals

We've talked before about some of the escape sequences that can be used within strings; we'll go through them in detail in a minute, but first, a refresher on number literals. Suppose you are defining the speed of light in meters per second:

OBSERVE: c

```
$c = 299792458;
```

That's difficult to verify visually. It would be all too easy to omit a digit or include an unnecessary one. If we were writing —rather than coding—a large number like the speed of light, or the national debt, or O'Reilly's student base, we would put **commas** at every third digit, like this:

OBSERVE: c

```
c = 299,792,458 m/s
```

But that won't work in Perl because the comma is already taken. (In a list context, the comma separates list arguments, and in a scalar context—like this would be—it evaluates the argument on the left and then returns the argument on the right. If you're in a hurry, you can use a comma in a one-liner to run multiple statements after an **and** or an **or** conditional, instead of resorting to an **if** block.)

But Perl does have a character you can use instead: the **underscore**:

OBSERVE: c

```
$c = 299_792_458;
```

This makes it much less difficult to verify that the number has the right magnitude.

Perl does not check to make sure you inserted underscores every third character; it is quite happy with numbers like 299_79245_8, or 2_99_79_24_58, or even 2_9_9_7_9_2_4_5_8—this even works in floating point numbers:

OBSERVE: inappropriate use of underscores

```
$pi = 3.1_415_9;
```

Perl lets you insert the underscore anywhere except at the beginning, and issues a warning if you have two underscores in a row or one at the end. Underscore helps you to verify large numbers by emulating the comma for grouping digits in threes. You might think that it would be useful when partitioning numbers in a social security number or a telephone number to reflect their customary grouping, but there's a problem with that. The grouping would be lost as the value was passed around (Perl does not remember the location of underscores) and both kinds of numbers are really strings, not numbers; you don't do arithmetic on them, so why store them as numbers?

You can also specify numbers in scientific notation with the exponent:

OBSERVE: Avagadro's Number

```
$a = 6.0221415E23;
```

Don't put any spaces around the "E." You can use an upper-case E or a lower-case e.

In addition to integers and floating point numbers, there are other types of numeric literals in the decimal base. Perl allows the specification of numbers in other bases as well:

OBSERVE: alternative bases

```
$signature = 0xfc45;      # Hexadecimal
$mode      = 0755;        # Octal
$episode   = 0b11001001;  # Binary
```

Hexadecimal numbers are introduced by **0x**, followed by one or more hexadecimal digits in either upper or lower case. The prefix may *not* be capitalized to **0X**.

Octal numbers are introduced by a leading zero followed by one or more octal digits (**0** through **7**). *No letter 'O' follows the zero.* The reason for this apparent inconsistency is actually to be consistent with other languages, like C; the decision dates back many years to when Perl needed to look familiar to programmers. This has caused confusion for many novice Perl programmers; let's make sure it doesn't happen to you. Run this one-liner. Type the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le '$x = "0454"; print $x+1; $x = 0454; print $x+1'
455
301
cold:~$
```

The first **$x** is a *string*. Perl's rule for interpreting numbers in a string is based solely on the specification for a decimal number. Leading zeroes are permitted because we might want to format reports with leading zeroes, and people would freak out if Perl suddenly interpreted some of those numbers as octal. The second **$x** is a literal number (hard-coded in the program); there is no reason to put a leading zero in front of it, therefore Perl uses the leading zero to denote that it's an octal number.

String Escapes: There are a number of escape sequences you can put inside of double-quoted strings. You're already familiar with **\n** (newline) and **\t** (tab); there are several others:

\r	Carriage return (more on this in the next lesson)
\f	Form feed (hardly useful any more)
\a	Alarm (ring the bell)
\e	The escape character (ESC)
\b	Backspace
nnn	Octal character *nnn* (each *n* is a digit from 0 through 7)
\x*nn*	Hexadecimal character *nn* (each *n* is a digit from 0 through F, uppercase or lowercase)
\c*c*	Control character *c* (e.g., \cG is Control-G)

This list is incomplete; it does not contain Unicode characters, but they are outside the scope of this course. The complete list can be found in **perldoc perlop**.

These escape sequences only count when they are inside "double-quoted" strings, not 'single-quoted' strings.

Some escape sequences don't get turned into control characters, but instead cause Perl to change what follows them:

- **\l**: Lowercase next character.
- **\L**: Lowercase all characters until end of string or the sequence **\E**.
- **\u**: Uppercase next character.
- **\U**: Uppercase all characters until end of string or the sequence **\E**.
- **\Q**: Quote all regex metacharacters so they match only literally until end of string or the sequence **\E**.

Escape sequences that don't have another meaning inside a regular expression can be used in a regex. So, while **\b** can't be used to mean backspace in a regex (it's an anchor), **\0*nn*** and others can. Let's see how that works. Type the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'print "Ding" if "\a" =~ /\cG/'
Ding
cold:~$
```

Character Functions

There are a couple of character functions that you should know. They aren't often used, but when the time comes, you'll be glad you learned them.

chr(*number*) returns the character with a numerical value that is *number*. For example, **chr(65)** is the letter **'A'** (unless you're running Perl on an EBCDIC system, in which case you have bigger problems, like remembering where you left your time machine :-)). You can use **chr(*number*)** as yet another mechanism for generating Unicode characters; it's also handy for getting at accented characters in the ISO Latin-1 extended character set. Try it out in a one-liner by typing the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'printf "%o: %s\n", $_, chr foreach 0xA0 .. 0xFE'
```

It doesn't work right; nothing prints for every character. Welcome to the world of character encodings and fonts. It's an unfriendly place. The command above will work most places, but not in the special environment that we have for students. The assembly of shells, SSH connections, Java applets, and browser windows that combine to make the environment work for you, means that we have to lie to Perl a little to get the output we want; we tell it that the output stream is in UTF-8 (this is covered in **perldoc perlunitut**, but be warned, this is a confusing and complicated topic). Modify the code in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -CO -le 'printf "%o: %s\n", $_, chr foreach 0xA0 .. 0xFE'
```

(FYI, that's the letter O, not a zero, after the big C.)

You can see here that the default variable for **chr** is, naturally enough, **$_**. **printf** does not get a newline appended to it by **-l**, unlike **print**, so we have to include it explicitly. Remember, we use **printf** to output a number in an alternative base, like octal, hexadecimal, or binary.

Having found the character code you want, you can then use one of the escape sequences from the table above to print it out. Type the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -CO -le 'print "Welcome to our Internet caf\351"'
```

Once again, if you try that on your own system, leave out the `-CO`.

ord(*x*) is the inverse of **chr**: it returns the numeric value of the character ***x***. (If you supply more than one character in the string, Perl only looks at the first one.) Try it in a one-liner by typing the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'print ord "A"'
65
cold:~$
```

Number Functions

If you want to *print* a number in an alternative base, you use **printf** and the appropriate format descriptor. But what about the inverse? For example, suppose you read a hexadecimal string in from a core dump, how would you turn it into the equivalent number?

Perl contains some handy functions for converting strings representing numbers in different bases. **hex(*string*)** converts a hexadecimal string to its number. Try it in a one-liner by typing the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'print hex "af"'
175
cold:~$
```

Try some other hexadecimal strings in place of **"af"**, like **"0xdd"** and **"DeadBeef"**.

oct(*string*) converts an octal string to its corresponding number. Try it in a one-liner by typing the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'print oct "43"'
35
cold:~$
```

The **oct** function can also convert strings with other radixes. Try it in a one-liner by typing the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'print oct for qw(0b101001 0755 0xFAA)'
41
493
4010
cold:~$
```

split()

We've talked about **split()** before, but always suggesting that its argument determining *how* to split was a string. In fact, it's a regular expression; we just happened to use examples that only contained literal atoms.

split() is capable of far more than we've seen. This table describes the different uses of the **split()** function:

Form	Explanation
LIST = **split(** *PATTERN, EXPR, LIMIT* **)**	Split *EXPR* on *PATTERN* at most *LIMIT* times, leaving the remainder in the last element of *LIST*.
LIST = **split(** *PATTERN, EXPR* **)**	Split *EXPR* on *PATTERN*, putting the results into *LIST*.
LIST = **split(** *PATTERN* **)**	Split **$_** on *PATTERN*.
LIST = **split()**	Split **$_** on whitespace.

Let's see some examples. Create **split.pl** as shown:

CODE TO TYPE: split.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  chomp;
  my @tokens;

  @tokens = split /[;:=]+\s*/, $_, 3;
  print "split P, E, 3\t", markup( @tokens ), "\n";

  @tokens = split /[;:=]+\s*/;
  print "split P\t\t", markup( @tokens ), "\n";

  @tokens = split;
  print "split\t\t", markup( @tokens ), "\n"
}

sub markup
{
   '|' . join( '|', @_ ) . '|';
}

__DATA__
Part: Cowling==; Cost: $71; Size: 48"
:peter:100:101:Peter Scott:/home/peter:/bin/tcsh:
The three virtues of a Perl programmer are: Impatience, Hubris, and Laziness - Larry Wa
ll
```

Check Syntax and run it. The output shows the list elements that **split()** created for each input, delimited with vertical bars:

INTERACTIVE SESSION:

```
cold:~$ cd perl2
cold:~/perl2$ ./split.pl
split P, E, 3   |Part|Cowling|Cost: $71; Size: 48"|
split P         |Part|Cowling|Cost|$71|Size|48"|
split           |Part:|Cowling==;|Cost:|$71;|Size:|48"|
split P, E, 3   ||peter|100:101:Peter Scott:/home/peter:/bin/tcsh:|
split P         ||peter|100|101|Peter Scott|/home/peter|/bin/tcsh|
split           |:peter:100:101:Peter|Scott:/home/peter:/bin/tcsh:|
split P, E, 3   |The three virtues of a Perl programmer are|Impatience, Hubris, and Laz
iness - Larry Wall|
split P         |The three virtues of a Perl programmer are|Impatience, Hubris, and Laz
iness - Larry Wall|
split           |The|three|virtues|of|a|Perl|programmer|are:|Impatience,|Hubris,|and|La
ziness|-|Larry|Wall|
split P, E, 3   ||
split P         ||
split           ||
cold:~/perl2$
```

The output is explained in detail here:

split P, E, 3	Part	Cowling	Cost: $71; Size: 48"											
split P	Part	Cowling	Cost	$71	Size	48"								
split	Part:	Cowling==;	Cost:	$71;	Size:	48"								
split P, E, 3		peter	100:101:Peter Scott:/home/peter:/bin/tcsh:											
split P		peter	100	101	Peter Scott	/home/peter	/bin/tcsh							
split	:peter:100:101:Peter	Scott:/home/peter:/bin/tcsh:												
split P, E, 3	The three virtues of a Perl programmer are	Impatience, Hubris, and Laziness - Larry Wall												
split P	The three virtues of a Perl programmer are	Impatience, Hubris, and Laziness - Larry Wall												
split	The	three	virtues	of	a	Perl	programmer	are:	Impatience,	Hubris,	and	Laziness	-	Larry	Wall

The first column above uses **P** and **E** as abbreviations for the pattern and expression, respectively. (The pattern is **/[;:=]+\s*/** and the expression is **$_**.)

Did you notice that **__DATA__** can be used interchangeably with **__END__**?

Study that program, its output (as well as the list below) carefully until you understand how each line of output was produced. You'll use the **split** function often, so make sure you're comfortable with it.

There are many special cases in the operation of **split**, so many that when we go through them, it may seem overwhelming. Don't worry—you'll get the hang of them. These special cases serve to make the operation of **split** more intuitive, so in the end, you'll actually have less to remember. Let's look at the details of the operation of **split**.

- Empty leading fields are preserved; if I split **"::abc:def"** on **/:/**, I'll get four elements in the list, the first two being empty.
- Empty trailing fields are deleted; if I split **"abc:def::"** on **/:/**, I'll get two elements in the list, **"abc"** and **"def"**.
- If all the fields end up being empty, they're all deleted. Type this into the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~/perl2$ perl -le 'print "-$_-" foreach split /[=;:%&]/, ";::=&%%"'
```

- Trailing empty fields are *not* deleted if a *LIMIT* is specified.
- If you want to make sure that you get all trailing empty fields and don't want to have to guess at a large enough *LIMIT*, use any negative *LIMIT*, such as **-1**. That splits into as many fields as it would if there were no *LIMIT*, but leaves in trailing empty fields.
- The empty regex (**//**) has a special meaning to **split()**: split the string into its characters. Type this into the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~/perl2$ perl -le 'print for split //, "Intermediate Perl"'
```

- If you surround the pattern with grouping parentheses, Perl puts whatever matched the regex in the output list, interwoven with the usual fields.

Let's take a look at that last rule in closer detail, because it describes something very useful: the ability to save all the parts in between the output fields. Try this one-liner by typing this code into the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~/perl2$ perl -le 'print foreach split /(\W+)/, "Fred,Bill&Ted-Ten%Each...minimum\
$100"'
Fred
,
Bill
&
Ted
-
Ten
%
Each
...
minimum
$
100
cold:~/perl2$
```

Sometimes what comes between the fields needs to be saved as well. This feature of **split** lets you do it.

Hashes: each()

Now we'll look at a function that can make iterating through a hash a bit more convenient. The **each()** function acts as an *iterator* that returns a two-element list of a key and its corresponding value. Each time you call it, you'll get a new (key,value) until you've gone all the way through the hash, at which point it returns the empty list. (In scalar context, **each()** returns a key each time until it reaches the end, at which point it returns **undef**. That won't be very useful.)

This can save us an extra step when we want to put the value in its own variable to save on typing. Create **each.pl** as shown:

CODE TO TYPE: each.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my (%description, %retail_price, %appearance);
my $MARKUP = 1.45;

while ( <DATA> )
{
  next if /\AItem ID/;
  my ($id, $desc, $cost, $color) = split /\s*\|\s*\$?/;
  $description{$id}  = $desc;
  $retail_price{$id} = $cost * $MARKUP;
  $appearance{$id}   = $color;
}

while ( my ($id, $desc) = each %description )
{
  printf "$desc : \$%.2f\n", $retail_price{$id};
}

__DATA__
Item ID|Description|Cost|Color
63784|Futon|$125.00|White
374895 | Tatami mat | $70.00 | Straw
273643 | Stone fountain | $210.00 | Gray
349875|Kimono|$743.00|Varies
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl2$ ./each.pl


Kimono : $1077.35
Tatami mat : $101.50
Futon : $181.25
Stone fountain : $304.50
cold:~/perl2$
```

We used the **split**'s powerful regular expression to deal with the inconsistent separator in the input data (sometimes it has space around it and sometimes it doesn't), and to remove the dollar sign from the cost before multiplying it by **$MARKUP**.

The impact of **each()** here is minimal (it saves us from typing **$description{$id}**), but in certain programs, the **each** function will be essential.

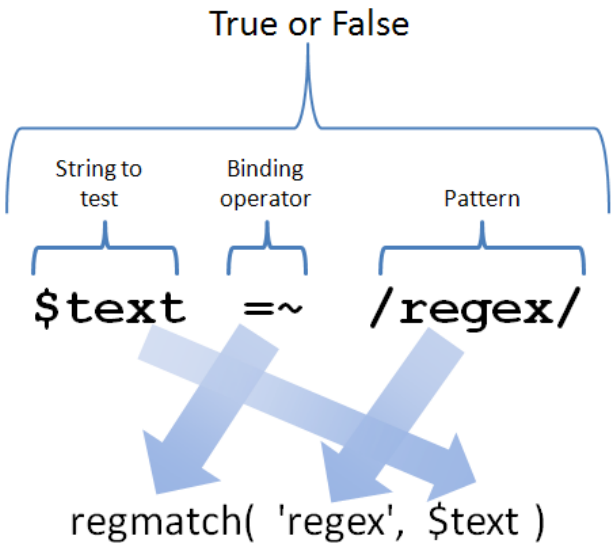
Alright then. Good job! We're really picking up steam! See you in the next lesson...

read()

Our final topic for this lesson is the **read()** function. This is an input function, but it is more primitive than the **readline** operator (**<>**). Instead of reading until it gets to a newline, it reads a specified number of characters. Can you think why that might be useful? Right—when reading binary data, the term we use for data that's not line-oriented. Usually it doesn't look like text, either, since text usually has line breaks in it so that people can read it. Binary files, like images, music, or proprietary document format files, usually contain characters that are unprintable and make a mess on your screen if you accidentally try to list them.

It's rare that you'd want to read such a file into Perl (without using a module that's specialized to the task of parsing whatever format the file is in), but you need to know about **read()** for such occasions. Because if you read a giant binary file in using **<>**, you may use huge amounts of memory unnecessarily since that binary file may not contain any newline characters.

To do an interesting example of this requires that we read something from a binary file. This image from earlier in the course will do.



We'll extract the width of the image from it. The PNG specification tells us it's near the beginning, contained in the first four bytes after the string "IHDR." We can tell from the specification that 50 bytes is more than enough to read to make sure we've read in that part of the file. Create **read_png.pl**:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

open my $fh, '<', '/software/Perl2/regmatch.png' or die "Couldn't read file: $!\n";
read $fh, my $buffer, 50;
$buffer =~ /IHDR(....)/ and print "Width = $1\n";
```

The **read()** function takes three arguments, the first being the filehandle to read from, the second a scalar to put the data into, and the third the number of characters to read. (Because we're not opening this file with a Unicode encoding, each character will be one byte long.) Check Syntax and run it. The output should be:

INTERACTIVE SESSION:

```
cold:~/perl2$ ./read_png.pl
Width = ù
```

What happened? Because we're reading binary data, the width is also in binary; it's a 32-bit number. So it came out to the terminal as junk. There's a way we can look at it, using the Unix **od** utility, which will turn binary data into something we can read:

INTERACTIVE SESSION:

```
cold:~/perl2$ ./read_png.pl | od -c
0000000   W   i   d   t   h       =      \0  \0 001 371  \n
```

There you can see that the width is the four characters with octal values 0, 0, 1, and 371. We still need to turn it into a number we can read. Perl's **unpack** function will do that. It allows all kinds of conversions via *conversion specifiers* in a format string rather like **printf** See the voluminous documentation for those specifiers in **perldoc -f pack**, which is the inverse function (that could be used for turning numbers into binary). The one we want is **%N**: "An unsigned long (32-bit) in "network" (big-endian) order." Edit **read_png.pl**:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

open my $fh, '<', '/software/Perl2/regmatch.png' or die "Couldn't read file: $!\n";
read $fh, my $buffer, 50;
$buffer =~ /IHDR(....)/ and print "Width = ", unpack( "%N", $1 ), "\n";
```

Check Syntax and run it. The output should be:

INTERACTIVE SESSION:

```
cold:~/perl2$ ./read_png.pl
Width = 505
```

And we're done! (You can view that image locally to verify that it is indeed 505 pixels wide). (If you really wanted to read PNG files, the proper approach would be to use the module **Image::PNG**; this was just a simple example to illustrate **read()**.) Most people deal with binary data rarely if at all, so we are not going to explore the **pack** function further, but feel free to read up on it if you might be one of those people who needs to read or write binary data.

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

For Loops and More Miscellaneous Topics

Lesson Objectives

When you complete this lesson, you will be able to:

- use For Loops.
- use Loop Labels.
- use Unbuffered Output.

For Loops

If you're familiar with other computer languages, you've probably used a loop. In most other languages, there is a **for** loop, which has three parts to its condition. Perl has a **for** loop as well, and it works much the same way as in those other languages. But the **foreach** loop is *much* more useful in Perl. A common mistake people make when learning Perl is to use the **for** loop where the **foreach** loop would give their programs greater clarity.

With that in mind, let's look at the **for** loop:

OBSERVE: for loop syntax

```
for ( Initialization; condition; iteration )
{
  # ... code ...
}
```

The **for** loop is characterized by the three clauses separated by semicolons inside parentheses; we call it the "directive." Each clause is a single Perl expression. Here's what the clauses do:

- **Initialization:** This is executed right before the first iteration of the loop. It's typically used to set an index variable to its initial value.
- **Condition:** This expression is evaluated in Boolean context (to see if it is true or false) before each execution of the loop. As soon as it evaluates as false, the **for** statement terminates, just like a **while** statement would.
- **Iteration:** This clause is executed at the end of each iteration of the loop before the ***condition*** is tested again.

And though we love the **for** statement, it could be replaced by a **while** statement that has this structure:

OBSERVE: while equivalent

```
Initialization
while ( condition )
{
  # ... code ...
  iteration
}
```

...except the **for** loop imposes its own scoping level on the directive, so that you can declare variables in the Initialization with **my** and restrict their scope to the **for** statement. So strictly speaking, the for loop is equivalent to this:

OBSERVE: while equivalent

```
{
  Initialization
  while ( condition )
  {
    # ... code ...
    iteration
  }
}
```

Now that you understand the operation of **for**, let's look at an example:

OBSERVE: for loop

```
for ( my $i = 0; $i <= $#array; $i++ )
{
  print "Element $i is $array[$i]\n";
}
```

Although a **for** loop isn't a very good way to iterate through an array, it makes for a tidy example.

The expression **$#array** evaluates to the index of the last element of the array **@array**. So, if **@fruits** contains the four elements **apple, orange, banana, cherry**, then **$#fruits** is **3**, and the value of the last element **$fruits[$#fruits]** is **cherry**. Because it's fairly common to want to access the last element of an array, and since that expression is cumbersome and repetitive, Perl also allows you to use *negative* indices to count backwards from the *end* of an array. So you get **$fruits[-1] eq 'cherry'** and **$fruits[-2] eq 'banana'**.

Try the **for** loop in a one-liner by typing the code below in the Unix shell as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'push @sq, $_ ** 2 foreach 1 .. 9; for ( $i = 0; $i <= $#sq; $i++ ) {
print "Element $i is $sq[$i]" }'
Element 0 is 1
Element 1 is 4
Element 2 is 9
Element 3 is 16
Element 4 is 25
Element 5 is 36
Element 6 is 49
Element 7 is 64
Element 8 is 81
cold:~$
```

The **for** loop is a bad way to iterate through an array because there's a good chance of making a *fencepost error*. *Fencepost error* gets its catchy name from this question: "If you have 100 feet of fencing to be used on a fence with posts spaced 10 feet apart, how many posts do you need?" (Answer: 11. Think about it.) Does the operator in the condition need to be **<**, or **<=**? Does the right side need to be **$#sq - 1**, **$#sq**, or **$#sq + 1**? Any time it takes you to come up with the correct answer is too much—if you use the **foreach** loop to iterate through array elements, you don't need to think about that at all.

But, if you also need to know the *index* of an array element in the loop (as in the example above), the **for** loop is at least concise. An alternative to using the **for** loop would be:

OBSERVE: array iteration

```
{
  my $i = 0;
  foreach my $element ( @array )
  {
    print "Element $i is $array[$i]\n";
    $i++;
  }
}
```

But you still have to think about the initial value of **$i** and where to put the postincrement operation in that example.

Another *for* example is suggested by the **perldoc perlsyn** entry for the **for** loop. Let's create a simplified version of it in **prompt.pl** as shown:

CODE TO TYPE: prompt.pl

```
#!/usr/bin/perl
use strict;
use warnings;

for ( prompt(); <STDIN>; prompt() )
{
  print "Input received: ";
  print;
  last if /Quit/i;
}

sub prompt { print "Input: " }
```

Check Syntax and run it. Now type in lines of text at the prompt until you get bored, then type a line containing the word "quit." (You can also end the program with end-of-file on standard input, by typing a line beginning with Control-D, on Unix/Linux systems.)

This example is a prototype for many applications where you need to prompt the user for input in a loop. By abstracting the prompting code into a subroutine, we minimize duplication. It also demonstrates a few new topics. **STDIN** is the predetermined filehandle for input from the terminal (or wherever it has been redirected from), just like **STDOUT** and **STDERR**.

And just like **while** statements, if a readline (**<>**) operator is the only thing in the condition of a **for** statement, Perl assumes the code **defined $_ =** is in front of it.

Finally, since the code for the **prompt** subroutine was short, I chose to put it all on one line and to omit the semicolon at the end of the statement (layout rules don't have to be followed strictly if clarity can be achieved by breaking them). Remember, semicolons are statement *separators* in Perl, not statement *terminators*; you can omit them at the end of a block or the end of a file, but only where the closing curly brace follows immediately on the same line.

I'll share one last thought about **for** and **foreach** here. Compare the forms of the two statements:

OBSERVE: for vs. foreach

```
for ( initialization; condition; iteration )
foreach [ loop_variable ] ( list )
```

Perl can tell these two statements apart not just by the different keywords, but also by the presence of semicolons within the parentheses. Because **foreach** is so much more useful than **for**, you'll type it more often. It would make sense, then, to use a shorter keyword than **foreach** (this is a principle known as *Huffman* coding). Therefore, Perl allows you to type **for** even when you mean **foreach**, and it will know what you mean:

OBSERVE: foreach, spelled 'for'

```
for [ loop_variable ] ( list )
```

From now on, we'll use that form of the **foreach** loop (or postfixed modifier) to save on typing and to illustrate the most common best practice.

Loop Labels

This relatively small topic becomes essential when you have nested loops and need to break out of an outer one. Suppose you have some Minesweeper game-like code that scans through a grid looking for a certain element; create **sweep.pl** as shown:

CODE TO TYPE: sweep.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @row0 = qw(O O O O O);
my @row1 = qw(O O O O O);
my @row2 = qw(O O O O O);
my @row3 = qw(O X O X O);
my @row4 = qw(O O X O O);

for my $row_index ( 0 .. 4 )
{
  my @row = $row_index == 0 ? @row0
          : $row_index == 1 ? @row1
          : $row_index == 2 ? @row2
          : $row_index == 3 ? @row3
          :                   @row4;
  for ( my $column_index = 0; $column_index <= $#row; $column_index++ )
  {
    if ( $row[$column_index] eq 'X' )
    {
      print "Found at [ $row_index, $column_index ]\n";
      last;
    }
  }
}
```

Check Syntax and run it. Our program finds and reports the coordinates of Xs. But which Xs is it reporting? The first X in each row that contains any Xs, because the **last** statement causes it to stop processing the inner loop when it finds an X. That's a little confusing, because the output suggests to us that it's actually reporting on all of them. Now suppose we want to find only the *first* X. And suppose the matrix is really large and we don't want to spend time scanning it after finding the first X. When we find one X, we want to stop processing *both* loops.

But **last** only bails out of the innermost loop; that's going to cause problems—unless we use a *loop label*. Modify the program as follows:

CODE TO EDIT: sweep.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @row0 = qw(O O O O O);
my @row1 = qw(O O O O O);
my @row2 = qw(O O O O O);
my @row3 = qw(O X O X O);
my @row4 = qw(O O X O O);

OUTER: for my $row_index ( 0 .. 4 )
{
  my @row = $row_index == 0 ? @row0
          : $row_index == 1 ? @row1
          : $row_index == 2 ? @row2
          : $row_index == 3 ? @row3
          :                   @row4;
  for ( my $column_index = 0; $column_index <= $#row; $column_index++ )
  {
    if ( $row[$column_index] eq 'X' )
    {
      print "Found at [ $row_index, $column_index ]\n";
      last OUTER;
    }
  }
}
```

Check Syntax and run it. There you have it!

Here's how labels work. A label is a bareword followed by a colon; by "bareword" I mean any string of characters that would be legal after a **$** or **@** or **%** for a variable name that you declare with **my**. By universal convention, we restrict labels to uppercase letters, underscores, and digits.

Labels can only come before loop statements: **for**, **foreach**, **while**, and **until**.

Any loop control statement—**next**, **last**, **redo**—can take a target that is the name of the label of an enclosing loop. Any loops within the one that the control statement happens to be inside of, will be ignored for the purpose of executing the control statement.

Unbuffered Output

Suppose you want to display a running count of operations performed so you can keep track of how far you've gotten. Maybe each operation takes a while to perform, so you want to see the count so you can be sure the program hasn't gotten stuck. Let's see a basic example of that; create **progress.pl** as shown:

CODE TO TYPE: progress.pl

```
#!/usr/bin/perl
use strict;
use warnings;

for ( 1 .. 20 )
{
  long_operation();
  print "$_\n";
}

sub long_operation
{
  sleep 1;
}
```

Check Syntax and run it. I introduced a new function here, **sleep()**, which takes as argument an integer number of seconds to pause, to simulate a lengthy operation.

So far, so good. But now suppose that **long_operation()** needs to print information of its own to the terminal occasionally. We'll simulate that with this change:

CODE TO EDIT: progress.pl

```
#!/usr/bin/perl
use strict;
use warnings;

for ( 1 .. 20 )
{
  long_operation( $_ );
  print "$_\n";
}

sub long_operation
{
  my $arg = shift;

  print "Processing element $arg\n" unless $arg % 5;
  sleep 1;
}
```

Check Syntax and run it. The **%** operator is the *modulus* operator that returns the remainder when the left side is divided by the right side. We use it here to print a line every five iterations.

Now the tracking numbers are mixed up in the output with the other information. We'd really like to see only the tracking numbers while the program is running, but when the program is done, then see only the other information. That sounds magical! And in fact we recently learned something that can make that happen. Remember the **\r** escape sequence for carriage return? It rewinds the cursor to the beginning of the line. Make this change:

CODE TO EDIT: progress.pl

```
#!/usr/bin/perl
use strict;
use warnings;

for ( 1 .. 20 )
{
  long_operation( $_ );
  print "$_\n";
  print "$_\r";
}

sub long_operation
{
  my $arg = shift;

  print "Processing element $arg\n" unless $arg % 5;
  sleep 1;
}
```

Check Syntax and run it. Oops. We got the output we wanted, but we didn't see the tracking numbers at all! Why is that? In fact, they were being printed, but we didn't see them in time as a result of *buffering*. The standard output stream is *line buffered* by Perl, which means that nothing goes to the screen until Perl sees a newline (**\n**). So as Perl gets each output string—**"1\r"**, **"2\r"**, **"3\r"**, and **"4\r"**—it saves them up, and then when it sees **"Processing element 5\n"**, it prints everything it's saved up to that **\n**. This means that the cursor returned to the left side of the screen after printing each **\r**, too fast for you to see it. And yet you had to wait five seconds for it to happen.

We *really* don't want Perl to buffer the output. We can tell Perl what we want by setting the magic variable **$|** (that's a vertical bar) to a true value. Make this change:

CODE TO EDIT: progress.pl

```
#!/usr/bin/perl
use strict;
use warnings;

$| = 1;

for ( 1 .. 20 )
{
  long_operation( $_ );
  print "$_\r";
}

sub long_operation
{
  my $arg = shift;

  print "Processing element $arg\n" unless $arg % 5;
  sleep 1;
}
```

Check Syntax and run it. There is something very satisfying about this basic, but effective mechanism.

Note Perl will also flush its standard output buffer when you read a line from STDIN, which is why **prompt.pl** was able to print a prompt that didn't include a newline. Perl does this precisely to make it unnecessary to know about **$|** in order to do prompting.

Another time unbuffering can be helpful is when you have redirected the standard output of your Perl program to a file, and from another process you are monitoring that file with **tail -f** while your program runs. When output is to a file, it is normally *fully* buffered, which means that output won't be flushed until a large buffer fills up, usually about 8 kbytes big. A newline isn't sufficient to force a flush. But setting **$|** will unbuffer standard output again so that you can keep up with your program's output with **tail -f**.

Can you believe we're through 13 lessons already? Great work.

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

Directory Reading Functions

Lesson Objectives

When you complete this lesson, you will be able to:

- use directory reading functions.

Hello! In this lesson we'll look at functions that return the names of files on the filesystem. Perl provides two ways of doing that: the **readdir** function that's borrowed from the C language binding to the Unix system call of the same name, and the **glob** operator that's borrowed from the Unix shell capability.

opendir, readdir, closedir

In the Unix filesystem, a directory is just another file, albeit one that is treated differently and is it is hard for you to read it directly like a file. The functions **opendir**, **readdir**, and **closedir** are used for reading directory files; they have a lot in common with the normal file access functions of **open**, **readline**, and **close**. Perl provides the directory functions even on non-Unix systems, so you don't need to worry about cross-platform compatibility. Here's how they work:

- **opendir** creates a directory handle, like an input filehandle, except that you can only use it for calling the next two functions. If the directory cannot be read for any reason, **opendir** returns false and sets **$!** (similar to **open**).
- **readdir** takes a directory handle as argument and, in a scalar context, acts as an iterator, returning the next filename from the directory until there are none left, at which point it returns **undef** (just like **readline**). In a list context, it returns all the remaining filenames in the directory.
- **closedir** tells Perl that you are done using the directory handle that you pass to this function. If you follow the good practice of creating the directory handle in a scope that ends when you are done with the directory anyway, you can omit this call because Perl will call it for you implicitly when it destroys the storage associated with the directory handle as it goes out of scope.

Let's see these functions in action. Create **dir_read.pl** as shown:

CODE TO TYPE: dir_read.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $dir = '.';
opendir my $dh, $dir or die "Couldn't open $dir: $!\n";
while ( my $file_name = readdir $dh )
{
  print "$dir: $file_name\n";
}
closedir $dh;

$dir = '/etc';
opendir $dh, $dir or die "Couldn't open $dir: $!\n";
for my $file_name ( readdir $dh )
{
  print "$dir: $file_name\n";
}
```

Check Syntax and run it. Let's take a closer look.

OBSERVE: dir_read.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $dir = '.';
opendir my $dh, $dir or die "Couldn't open $dir: $!\n";
while ( my $file_name = readdir $dh )
{
  print "$dir: $file_name\n";
}
closedir $dh;

$dir = '/etc';
opendir $dh, $dir or die "Couldn't open $dir: $!\n";
for my $file_name ( readdir $dh )
{
  print "$dir: $file_name\n";
}
```

In the **while** loop, we call **readdir** in scalar context, so it acts as an iterator. We call the **foreach** loop in list context, so we get back all the filenames. Get in the habit of processing filenames one at a time (in scalar context) to save time and memory.

'.' is the current directory. **'/etc'** is a system directory we know will exist on any Unix or Linux system.

The file names are returned in random order (okay, the order depends on esoteric and idiosyncratic properties of your filesystem but is, for all intents and purposes, random).

Also, consider the scoping of **$dh**, the directory handle: we declared **$dh** with **my** at the moment of first use, which is in the outermost scope level, so it still exists after the loop and should not be declared again.

Finally, each directory contains files with the names **'.'** and **'..'**. They are in fact directories (subdirectories look the same as normal files to **readdir**), and are placed in every directory by the operating system for purposes of navigation. **'.'** is the current directory itself (this is rarely useful), and **'..'** is the parent directory. Even the top-most directory (**'/'** on a Unix or Linux system) still contains **'..'**; it just points to itself. Usually, you'll ignore those files. Now exclude **.** and **..** by modifying the code as shown:

CODE TO EDIT: dir_read.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $dir = '.';
opendir my $dh, $dir or die "Couldn't open $dir: $!\n";
while ( my $file_name = readdir $dh )
{
  next if $file_name =~ /\A\.\.?\z/;
  print "$dir: $file_name\n";
}
closedir $dh;

$dir = '/etc';
opendir $dh, $dir or die "Couldn't open $dir: $!\n";
for my $file_name ( readdir $dh )
{
  next if $file_name =~ /\A\.\.?\z/;
  print "$dir: $file_name\n";
}
```

Check Syntax and run it, and verify that **'.'** and **'..'** are no longer in the output.

glob

The other directory-reading function is called **glob**. It takes as an argument a glob *pattern*. In list context, it returns a list of all the matching filenames. In scalar context, it acts as an iterator just like **readdir**. The big difference between **glob** and **readdir** is that **glob** leaves in the output any directory path, absolute or relative, that you included in the pattern.

So what is the glob pattern? It's *like* a regular expression but it isn't one. It's the same kind of pattern that you can type at a shell after a command like **ls** that gets expanded by the shell into a list of filenames. For example, to delete all the files with names that end in **.o**, you'd type **rm *.o** (or in Windows, **erase *.o**). The shell has its own glob function that it calls to return a list of matching filenames to pass to the **rm** program. In fact, early versions of Perl just called out to the shell to do the globbing for them. But the shell's glob facility wasn't as powerful as the Perl developers wanted it to be, so they wrote their own. You gotta love that ingenuity!

If you're used to typing glob patterns at shell command lines, you're used to the glob metacharacters already. But just in case, here's a list of the meanings of the most common ones:

- *****: Zero or more characters.
- **?**: Exactly one character.
- **[abc]**: Any one of the characters **a**, **b**, or **c**.
- **[a-z]**: Any character in the inclusive range **a** through **z** (same rules as Perl regexes).
- **pattern1 pattern2**: The union of the result of **pattern1** and **pattern2**.

This may seem a lot like regular expression syntax, but it isn't—for example, the period has no special meaning, ***** is the same as the regex **/.*/**, and **?** is the same as the regex **/./**. Here are some examples of glob patterns and their meanings:

- ***.p[lm]**: All files with names ending in **.pl** or **.pm**.
- ***/*~**: All files with names ending in **~** in subdirectories of the current directory.
- **../*/spam?/in.mbx**: All files named **in.mbx** in all subdirectories named **spam** followed by a single character in all sibling directories of the current directory.
- **/etc/rc[1-4].d/K1***: All files with names starting with **K1** in subdirectories **rc1.d**, **rc2.d**, **rc3.d**, and **rc4.d** of **/etc**.

There is one giant exception to all of the rules and examples listed above though. *They do not match filenames beginning with a period unless you explicitly include the period.* (This restriction does not apply to **readdir**.) This is the Unix way of trying to hide files. It's not particularly effective, since you can get at the "hidden" files with the glob pattern **".*"**, and then match all files with the glob pattern **"* .*"**.

It might seem as though you need to qualify every glob search to match both kinds of files, but in practice, you only use **glob** to match files that couldn't possibly start with a period. In other cases, you'll generally use **readdir**. **glob** is handy when you want to find a list of files matching a pattern you can turn into a glob pattern; with **readdir** you would need to take the additional step of selecting candidates based on a regular expression.

Let's try an example using **glob** now; create **glob_read.pl** as shown:

CODE TO TYPE: glob_read.pl

```
#!/usr/bin/perl
use strict;
use warnings;

print ".: $_\n"    for glob "*";

print "/etc: $_\n" for glob "/etc/*";
```

Check Syntax and run it. It's the **glob** equivalent of **dir_read.pl**, with the exception that it doesn't list files with names beginning with periods, and so it omits **'.'** and **'..'** (among others).

Let's try another short example using **glob** in scalar context, with one of the patterns that was used as an example above. Create **glob_while.pl** as shown:

CODE TO TYPE: glob_while.pl

```
#!/usr/bin/perl
use strict;
use warnings;

while ( my $file = glob "/etc/rc[1-4].d/K1*" )
{
  print "Matching file: $file\n";
}
```

Check Syntax and run it.

There's another part of the **glob** pattern syntax that behaves a bit differently. If you don't use any of the metacharacters listed above, Perl does no filename matching. So, for instance, glob "nonesuch.pl" will return **"nonesuch.pl"**, regardless of whether such a file exists in the current directory. Don't take my word for it though, try it yourself! Type a one-liner in the Unix shell as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ ls -l nonesuch.pl
ls: nonesuch.pl: No such file or directory
cold:~$ perl -le 'print glob "nonesuch.pl"'
nonesuch.pl
cold:~$
```

It doesn't seem very likely that you would write a **glob** pattern without any expansion metacharacters, though, does it?

Well, there are some metacharacters that do not trigger filename matching. Curly brackets surrounding strings, separated by commas, expand to the list of elements in brackets; so, for example, **"{foo,bar}.pl"** is treated as **"foo.pl bar.pl"**, so that's the result you will get regardless of whether those files exist.

You can even use this facility as a quick-and-dirty way of generating permutations! Run this one-liner:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -le 'print for glob "{a,b,c}{d,e}{f,g,h,i}"'
```

If you want to use brackets in a **glob** pattern and make sure that you only get results that match filenames, insert any of the metacharacters *****, **?**, or **[]** outside of the braces to force a name check. For example, run these two one-liners and observe the different results:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ perl -le 'print for glob "{foo,bar}.pl"'
foo.pl
bar.pl
cold:~$ perl -le 'print for glob "{foo,bar}.p[l]"'
cold:~$
```

The second pattern is exactly the same as the first one because there is only one character inside the character class.

To get details on the syntax of **glob** patterns, run **perldoc File::Glob**.

Note In older Perl programs, it was common to use angle brackets (**<>**) as a synonym for **glob**. Perl is (usually) able to tell the difference between a filehandle and a glob pattern and call **readline** or **glob** as necessary. This practice can be confusing though, and isn't preferred now.

The range of useful programs you can write in Perl is dramatically broader than it was at the beginning of this course. You are now able to write Perl programs that can solve lots of the problems you encounter in real life. Amazing! In the next and final lesson we'll create the biggest example program we've tackled yet!

Once you finish this lesson, go back to the syllabus page by clicking on the page tab above and do the assignments.

More Math Functions and Course Wrap-Up

Lesson Objectives

When you complete this lesson, you will be able to:

- use additional built-in Perl math functions.

Welcome to the final lesson of Intermediate Perl! Congratulations on making it to the end of this course. This lesson will introduce a few new topics, and then we'll develop a relatively long program to demonstrate what you have learned in this course.

More Math Functions

The O'Reilly School of Technology founder, Scott Gray, is a mathematician by vocation and this section is dedicated to him. (Don't panic if you aren't the same sort of math whiz!) Here are some built-in Perl functions I haven't mentioned before:

- **sin** **$arg**: sine of **$arg**, where **$arg** is in radians.
- **cos** **$arg**: cosine of **$arg**, where **$arg** is in radians.
- **atan2** **$y**, **$x**: arctangent of **$y**/**$x** in radians.

  (Using the above three functions, you can easily derive the familiar functions **tan**, **acos**, and **asin**.)
- **sqrt** **$x**: square root of **$x**.
- **abs** : absolute value of **$x**.
- **exp** **$x**: $e^{\$x}$.
- **log** **$x**: Natural logarithm (base *e*) of **$x**.

If you want to see additional math functions, there are modules on the Comprehensive Perl Archive Network for just about everything, although they are outside the scope of this course.

crypt()

The **crypt** function computes what is most often called a *hash*, but to avoid confusion we will call it a *digest*. Despite its name, the **crypt** function does not perform encryption in the sense that you are most likely to think of it. The properties of a digest are:

- It converts a string to another string.
- It can take any length input, but provides an output of small size, fixed or closely bounded in length.
- The chances of two different inputs, no matter how big, being converted to the same digest by the **crypt** function are astronomically small.
- It is impossible to reconstruct any part of the input from a digest.
- You can't figure out what input the **crypt** function used to create a digest by looking at the digest.

The **crypt** function is handy for performing authentication; **crypt** is the function generally used on Unix and Linux systems for logging in a user, and it is the function used for authenticating users to most web pages. The properties of a digest are such that when a password is set, you can compute the digest for it and then store the digest. The password cannot be extracted from the digest, but when a user signs in and provides a password, you can run it through **crypt** again and, if you get the same digest, you know that user has the correct password.

The **crypt** function also takes as argument a *salt*, which is a random string usually of two characters, that makes it harder for the digest to be used on a different system. Here's the way to pick a salt, recommended according to **perldoc**:

OBSERVE:

```
sub create_salt
{
  my @chars = ('.', '/', 0..9, 'A'..'Z', 'a'..'z');
  return join '', $chars[rand @chars], $chars[rand @chars];
}
```

Then you would call **crypt**, like this:

OBSERVE: Tales from the Crypt

```
my $password = <STDIN>
chomp $password;
my $digest = crypt $password, create_salt();
```

$digest always starts with the salt; that way you know which salt to use when verifying a password later. Perl takes only the necessary characters from the salt argument, so the verification step looks like this:

OBSERVE: Password Verification

```
if ( crypt( $input_password, $stored_digest ) eq $stored_digest )
{
  # Password is correct
}
```

Final Example Program

For the final example in this course, we will develop a (relatively) long program from scratch. We will follow agile programming methods, so there will be many iterations. To give us a real-world type of task, this program will manage a **.htpasswd** file, which is what the Apache and other web servers use to authenticate users to password-protected web pages. You don't need to have used a .htpasswd file before to work through this example.

WARNING The **htpasswd** program that comes with Apache has many more options and abilities than what we will develop here; our program should *not* be used as a substitute for **htpasswd**.

The **.htpasswd** file contains one line per user, which contains the username, followed by a colon, followed by the digest of the password. (If you want to store additional information in the file for use in other programs, like the user's email address, you can append it after another colon on each line.)

Let's start! Create **htmanage.pl** as shown:

CODE TO TYPE: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Input_File = '.htpasswd';

init( @ARGV );
run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  if ( open my $fh, '<', $Input_File )
  {
  }
  else
  {
    print "$Input_File does not exist (yet)\n";
  }
}
```

Check Syntax and run it. Don't give it any arguments just yet, just verify that it prints **".htpasswd does not exist (yet)"**. Our program already allows for one input argument: **-f** followed by a filename allows us to act on a file other than the default of **.htpasswd**.

Now let's add a built-in test. Modify **htmanage.pl**:

CODE TO EDIT: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';

init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-f' )
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  if ( open my $fh, '<', $Input_File )
  {
  }
  else
  {
    print "$Input_File does not exist (yet)\n";
  }
}

sub test
{
  print "No tests yet!\n";
}
```

Check Syntax and run it with the **-t** flag to perform testing—only there aren't any tests yet. You get the message **"No tests yet!"**. That's okay, we can change that. Modify **htmanage.pl** as shown:

CODE TO EDIT: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';
my $TEST_FILE = 'test.htpasswd';

init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  if ( open my $fh, '<', $Input_File )
  my %digest = parse_file( $Input_File );
}

sub parse_file
{
  my $filename = shift;
  if ( open my $fh, '<', $filename )
  {
    return ( foo => 'bar' );
  }
  else
  {
    print "$Input_File does not exist (yet)\n";
  }
}

sub test
{
  print "No tests yet!\n";
  my %test_data = ( scott => '123', peter => '456', steve => '789' );
  write_file( $TEST_FILE, %test_data );
  my %test_data_read = parse_file( $TEST_FILE );
  for my $user ( keys %test_data )
  {
    delete $test_data_read{$user} or die "User $user not found";
  }
  keys %test_data_read and die "Spurious user(s) found in input";
}
```

```
sub write_file
{
}
```

We started by writing the **test** function, then realized that we need to be able to read and write **.htpasswd** files from there, so we moved the **open()** call to its own subroutine that could be called for file reading. The **test** function verifies that the file reading routine turns a file into a hash whose keys are the first and second colon-separated fields, respectively, and verifies that the writing routine turns such a hash into the equivalent file. Study that testing code. The **delete()** function removes an element from a hash and returns the value that it had, or returns **undef** if it had no value. In a scalar context, **keys()** returns the number of elements in a hash; we are using this to determine whether there is anything left in the comparison hash.

Check Syntax and run it in test mode—it fails! That's because we haven't written the code that it's testing yet. This is a hallmark of test-driven development. If the test passes before the code is written, there is something wrong with the test. Let's give our program the right code. Modify your code as shown:

CODE TO EDIT: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';
my $TEST_FILE = 'test.htpasswd';


init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  my %digest = parse_file( $Input_File );
}

sub parse_file
{
  my $filename = shift;

  if ( open my $fh, '<', $filename )
  {
    return ( foo => 'bar' );
  }
  else
  {
    print "$Input_File does not exist (yet)\n";
  }
}

sub test
{
  my %test_data = ( scott => '123', peter => '456', steve => '789' );
  write_file( $TEST_FILE, %test_data );
  my %test_data_read = parse_file( $TEST_FILE );

  for my $user ( keys %test_data )
  {
    delete $test_data_read{$user} or die "User $user not found";
    delete $test_data_read{$user} or warn "User $user not found";
  }
  keys %test_data_read and die "Spurious user(s) found in input";
```

```
}

sub write_file
{
  my ($filename, %data) = @_;
  open my $fh, '>', $filename or die "Couldn't open $filename for writing: $!\n";
  print {$fh} "$_:$data{$_}\n" for keys %data;
}
```

Check Syntax and run it. We changed a **die** to a **warn** so we could see more errors caused by the **parse_file()** routine, returning only fixed data. Let's code that routine now (and change the **warn** back to a **die**). Modify your code as shown:

CODE TO EDIT: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';
my $TEST_FILE = 'test.htpasswd';


init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  my %digest = parse_file( $Input_File );
}

sub parse_file
{
  my $filename = shift;
  my %second_field;

  if ( open my $fh, '<', $filename )
  {
    return ( foo => 'bar' );
    while ( <$fh> )
    {
      chomp;
      my ($field1, $field2) = split /:/;
      $second_field{$field1} = $field2;
    }
    return %second_field;
  }
  else
  {
    print "$Input_File does not exist (yet)\n";
  }
}

sub test
{
  my %test_data = ( scott => '123', peter => '456', steve => '789' );
  write_file( $TEST_FILE, %test_data );
```

```
  my %test_data_read = parse_file( $TEST_FILE );

  for my $user ( keys %test_data )
  {
    delete $test_data_read{$user} or warndie "User $user not found";
  }
  keys %test_data_read and die "Spurious user(s) found in input";
  unlink $TEST_FILE;
}

sub write_file
{
  my ($filename, %data) = @_;

  open my $fh, '>', $filename or die "Couldn't open $filename for writing: $!\n";
  print {$fh} "$_:$data{$_}\n" for keys %data;
}
```

Check Syntax and run it with the **-t** flag. It should run without errors now. We also want to keep our test clean too, so we remove our test file using **unlink**.

Let's handle nonexistent input files properly now, so that they return an empty hash. Because we'll want to handle the hash of digests in several routines, we'll make it a global variable. And we'll start filling out a menu-based user interface. Modify your code as shown:

CODE TO EDIT: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';
my $TEST_FILE = 'test.htpasswd';
my %Digest;

init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  my %digest = parse_file( $Input_File );
  %Digest = parse_file( $Input_File );
  for ( menu(); my $command = <STDIN>; menu() )
  {
    chomp $command;
    act_on( $command );
  }
}

sub menu
{
  print "Entries in $Input_File:\n";
  my @options = make_options();
  for my $index ( 0 .. $#options )
  {
    printf "%2d %s\n", $index, $options[$index];
  }
  print "User or Option (0 - $#options)? ";
}

sub make_options
{
  my @options = 'Select to Quit';
  push @options, ( sort keys %Digest ), 'Select to Add New';
  return @options;
}

sub act_on
{
```

```
  my $command = shift;
  my @options = make_options();
  exit if $options[$command] =~ /to Quit/i;
  $options[$command] =~ /to Add/i  and return ask_add();
  do_change( $options[$command] );
}

sub ask_add
{
  print "Add: \n";
}

sub do_change
{
  print "Change: \n";
}

sub parse_file
{
  my $filename = shift;

  my %second_field;
  if ( open my $fh, '<', $filename )
  {
    while ( <$fh> )
    {
      chomp;
      my ($field1, $field2) = split /:/;
      $second_field{$field1} = $field2;
    }
    return %second_field;
  }
  else
    return %second_field;
  {
    print "$Input_File does not exist (yet)\n";
  }
}

sub test
{
  my %test_data = ( scott => '123', peter => '456', steve => '789' );
  write_file( $TEST_FILE, %test_data );
  my %test_data_read = parse_file( $TEST_FILE );

  for my $user ( keys %test_data )
  {
    delete $test_data_read{$user} or die "User $user not found";
  }
  keys %test_data_read and die "Spurious user(s) found in input";

  unlink $TEST_FILE;
}

sub write_file
{
  my ($filename, %data) = @_;

  open my $fh, '>', $filename or die "Couldn't open $filename for writing: $!\n";
  print {$fh} "$_:$data{$_}\n" for keys %data;
}
```

Check Syntax and run it. The tests are unchanged, so run it without the **-t** flag. You can enter an option **0** (zero) to quit, but the 'Add' option just prints "Add:". That's alright, we're making progress. We have the stub of an interface needed to change an existing entry, but since there's no way to add an entry to a file yet, we have no way of seeing it (unless we give it an **.htpasswd** file from somewhere else).

Let's fill in those new routines so we can add entries. Modify your code as shown:

CODE TO EDIT: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';
my $TEST_FILE = 'test.htpasswd';
my %Digest;

init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  %Digest = parse_file( $Input_File );
  for ( menu(); my $command = <STDIN>; menu() )
  {
    chomp $command;
    act_on( $command );
  }
}

sub menu
{
  print "Entries in $Input_File:\n";
  my @options = make_options();

  for my $index ( 0 .. $#options )
  {
    printf "%2d %s\n", $index, $options[$index];
  }
  print "User or Option (0 - $#options)? ";
}

sub make_options
{
  my @options = 'Select to Quit';
  push @options, ( sort keys %Digest ), 'Select to Add New';
  return @options;
}

sub act_on
{
```

```
  my $command = shift;

  my @options = make_options();
  exit if $options[$command] =~ /to Quit/i;
  $options[$command] =~ /to Add/i  and return do_add();
  do_change( $options[$command] );
  if ( $options[$command] =~ /to Add/i )
  {
    ask_add();
  }
  else
  {
    do_change( $options[$command] );
  }
  write_file( $Input_File, %Digest );
}

sub ask_add
{
  print "Add: \n";
  print "Add (username): ";
  chomp (my $username = <STDIN>);
  print "    (password): ";
  chomp (my $password = <STDIN>);
  add( $username, $password );
}

sub add
{
  my ($username, $password) = @_;
  $Digest{$username} = crypt $password, create_salt();
}

sub create_salt
{
  my @chars = ('.', '/', 0..9, 'A'..'Z', 'a'..'z');
  return join '', $chars[rand @chars], $chars[rand @chars];
}

sub do_change
{
  print "Change: \n";
}

sub parse_file
{
  my $filename = shift;

  my %second_field;
  if ( open my $fh, '<', $filename )
  {
    while ( <$fh> )
    {
      chomp;
      my ($field1, $field2) = split /:/;
      $second_field{$field1} = $field2;
    }
  }
  return %second_field;
}

sub test
{
  my %test_data = ( scott => '123', peter => '456', steve => '789' );
  write_file( $TEST_FILE, %test_data );
  my %test_data_read = parse_file( $TEST_FILE );

  for my $user ( keys %test_data )
```

```
  {
    delete $test_data_read{$user} or die "User $user not found";
  }
  keys %test_data_read and die "Spurious user(s) found in input";

  unlink $TEST_FILE;
}

sub write_file
{
  my ($filename, %data) = @_;

  open my $fh, '>', $filename or die "Couldn't open $filename for writing: $!\n";
  print {$fh} "$_:$data{$_}\n" for keys %data;
}
```

Check Syntax and run it. If you select option **1** (Add) and create a user, that user appears the next time the menu displays. And now we're writing out the new file after any addition.

Let's start populating the **do_change** routine by having it delete an entry if the user clicks **Enter**, or else accept a new password. Modify your code as shown:

CODE TO EDIT: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';
my $TEST_FILE = 'test.htpasswd';
my %Digest;

init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  %Digest = parse_file( $Input_File );
  for ( menu(); my $command = <STDIN>; menu() )
  {
    chomp $command;
    act_on( $command );
  }
}

sub menu
{
  print "Entries in $Input_File:\n";
  my @options = make_options();

  for my $index ( 0 .. $#options )
  {
    printf "%2d %s\n", $index, $options[$index];
  }
  print "User or Option (0 - $#options)? ";
}

sub make_options
{
  my @options = 'Select to Quit';
  push @options, ( sort keys %Digest ), 'Select to Add New';
  return @options;
}

sub act_on
{
```

```
  my $command = shift;

  my @options = make_options();
  exit if $options[$command] =~ /to Quit/i;
  if ( $options[$command] =~ /to Add/i )
  {
    ask_add();
  }
  else
  {
    do_change( $options[$command] );
  }
  write_file( $Input_File, %Digest );
}

sub ask_add
{
  print "Add (username): ";
  chomp (my $username = <STDIN>);
  print "    (password): ";
  chomp (my $password = <STDIN>);
  add( $username, $password );
  set( $username, $password );
}

sub add
sub set
{
  my ($username, $password) = @_;

  $Digest{$username} = crypt $password, create_salt();
}

sub create_salt
{
  my @chars = ('.', '/', 0..9, 'A'..'Z', 'a'..'z');
  return join '', $chars[rand @chars], $chars[rand @chars];
}

sub do_change
{
  print "Change: \n";
  my $username = shift;
  print "Change $username: Return to delete, or new password: ";
  chomp( my $input = <STDIN> );
  if ( $input )
  {
    set( $username, $input );
  }
  else
  {
    delete $Digest{$username};
  }
}

sub parse_file
{
  my $filename = shift;

  my %second_field;
  if ( open my $fh, '<', $filename )
  {
    while ( <$fh> )
    {
      chomp;
      my ($field1, $field2) = split /:/;
      $second_field{$field1} = $field2;
    }
```

```
  }
  return %second_field;
}

sub test
{
  my %test_data = ( scott => '123', peter => '456', steve => '789' );
  write_file( $TEST_FILE, %test_data );
  my %test_data_read = parse_file( $TEST_FILE );

  for my $user ( keys %test_data )
  {
    delete $test_data_read{$user} or die "User $user not found";
  }
  keys %test_data_read and die "Spurious user(s) found in input";

  unlink $TEST_FILE;
}

sub write_file
{
  my ($filename, %data) = @_;

  open my $fh, '>', $filename or die "Couldn't open $filename for writing: $!\n";
  print {$fh} "$_:$data{$_}\n" for keys %data;
}
```

Check Syntax and run it. Because we can reuse the **add** routine by calling it from the **do_change** routine, it's appropriate to name it **set**. Now we can reuse that **set** routine for testing. Modify your code as shown:

CODE TO EDIT: htmanage,pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';
my $TEST_FILE = 'test.htpasswd';
my %Digest;

init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f')
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  %Digest = parse_file( $Input_File );
  for ( menu(); my $command = <STDIN>; menu() )
  {
    chomp $command;
    act_on( $command );
  }
}

sub menu
{
  print "Entries in $Input_File:\n";
  my @options = make_options();

  for my $index ( 0 .. $#options )
  {
    printf "%2d %s\n", $index, $options[$index];
  }
  print "User or Option (0 - $#options)? ";
}

sub make_options
{
  my @options = 'Select to Quit';
  push @options, ( sort keys %Digest ), 'Select to Add New';
  return @options;
}

sub act_on
{
```

```
  my $command = shift;

  my @options = make_options();
  exit if $options[$command] =~ /to Quit/i;
  if ( $options[$command] =~ /to Add/i )
  {
    ask_add();
  }
  else
  {
    do_change( $options[$command] );
  }
  write_file( $Input_File, %Digest );
}

sub ask_add
{
  print "Add (username): ";
  chomp (my $username = <STDIN>);
  print "    (password): ";
  chomp (my $password = <STDIN>);
  set( $username, $password );
}

sub set
{
  my ($username, $password) = @_;

  $Digest{$username} = crypt $password, create_salt();
}

sub create_salt
{
  my @chars = ('.', '/', 0..9, 'A'..'Z', 'a'..'z');
  return join '', $chars[rand @chars], $chars[rand @chars];
}

sub do_change
{
  my $username = shift;

  print "Change $username: Return to delete, or new password: ";
  chomp( my $input = <STDIN> );
  if ( $input )
  {
    set( $username, $input );
  }
  else
  {
    delete $Digest{$username};
  }
}

sub parse_file
{
  my $filename = shift;

  my %second_field;
  if ( open my $fh, '<', $filename )
  {
    while ( <$fh> )
    {
      chomp;
      my ($field1, $field2) = split /:/;
      $second_field{$field1} = $field2;
    }
  }
  return %second_field;
```

```
}

sub verify
{
  my ($digest, $password) = @_;
  return crypt( $password, $digest ) eq $digest;
}

sub test
{
  my %test_data = ( scott => '123', peter => '456', steve => '789' );
  write_file( $TEST_FILE, %test_data );
  my %test_data_read = parse_file( $TEST_FILE );

  for my $user ( keys %test_data )
  {
    delete $test_data_read{$user} or die "User $user not found";
  }
  keys %test_data_read and die "Spurious user(s) found in input";
  unlink $TEST_FILE;
  $Input_File = $TEST_FILE;
  set( $_, $test_data{$_} ) for keys %test_data;
  write_file( $Input_File, %Digest );
  %test_data_read = parse_file( $Input_File );
  verify( $test_data_read{$_}, $test_data{$_} )
    or die "Incorrect password for $_"
      for keys %test_data_read;

  unlink $TEST_FILE;
}

sub write_file
{
  my ($filename, %data) = @_;

  open my $fh, '>', $filename or die "Couldn't open $filename for writing: $!\n";
  print {$fh} "$_:$data{$_}\n" for keys %data;
}
```

Check Syntax and run it with the **-t** flag. No output means the tests passed! Now run it without the **-t** flag and play with the options. This **test** routine now tests password encryption by verifying that the digest matches what it should.

Hey, that **verify** routine looks like it would provide a valuable user function: the ability to verify a password for an entry in the file. But to make it do that, we'll have to change our user interface. Modify the code below as shown:

CODE TO EDIT: htmanage.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $Testing;
my $Input_File = '.htpasswd';
my $TEST_FILE = 'test.htpasswd';
my %Digest;

init( @ARGV );
$Testing and test() or run();

sub init
{
  my $read_filename;

  while ( @_ )
  {
    my $arg = shift;
    if ( $arg eq '-t' )
    {
      $Testing = 1;
    }
    elsif ( $arg eq '-f' )
    {
      $read_filename = 1;
    }
    elsif ( $read_filename )
    {
      $Input_File = $arg;
      $read_filename = 0;
    }
  }
}

sub run
{
  %Digest = parse_file( $Input_File );
  for ( menu(); my $command = <STDIN>; menu() )
  {
    chomp $command;
    act_on( $command );
  }
}

sub menu
{
  print "Entries in $Input_File:\n";
  my @options = make_options();

  for my $index ( 0 .. $#options )
  {
    printf "%2d %s\n", $index, $options[$index];
  }
  print "User or Option (0 - $#options)? ";
}

sub make_options
{
  my @options = 'Select to Quit';
  push @options, ( sort keys %Digest ), 'Select to Add New';
  return @options;
}

sub act_on
{
```

```
  my $command = shift;

  my @options = make_options();
  die "Invalid option" if $command !~ /\A\d+\z/ || ! $options[$command];
  exit if $options[$command] =~ /to Quit/i;
  if ( $options[$command] =~ /to Add/i )
  {
    ask_add();
  }
  else
  {
    do_change( $options[$command] );
  }
  write_file( $Input_File, %Digest );
}

sub ask_add
{
  print "Add (username): ";
  chomp (my $username = <STDIN>);
  print "    (password): ";
  chomp (my $password = <STDIN>);
  set( $username, $password );
}

sub set
{
  my ($username, $password) = @_;

  $Digest{$username} = crypt $password, create_salt();
}

sub create_salt
{
  my @chars = ('.', '/', 0..9, 'A'..'Z', 'a'..'z');
  return join '', $chars[rand @chars], $chars[rand @chars];
}

sub do_change
{
  my $username = shift;

  print "Change $username: Return to delete, or new password: ";
  print "Change $username: (D)elete, (V)erify, (N)ew: ";
  chomp( my $input = <STDIN> );
  chomp( my $cmd = <STDIN> );
  if ( $input )
  if ( $cmd =~ /\AN/i )
  {
    print "New password for $username: ";
    chomp( my $input = <STDIN> );
    set( $username, $input );
  }
  else
  elsif ( $cmd =~ /\AD/i )
  {
    delete $Digest{$username};
  }
  elsif ( $cmd =~ /\AV/i )
  {
    print "Password to check: ";
    chomp( my $password = <STDIN> );
    print verify( $Digest{$username}, $password ) ? "Correct" : "Wrong", "\n";
  }
}

sub parse_file
{
```

```
  my $filename = shift;

  my %second_field;
  if ( open my $fh, '<', $filename )
  {
    while ( <$fh> )
    {
      chomp;
      my ($field1, $field2) = split /:/;
      $second_field{$field1} = $field2;
    }
  }
  return %second_field;
}

sub verify
{
  my ($digest, $password) = @_;

  return crypt( $password, $digest ) eq $digest;
}

sub test
{
  my %test_data = ( scott => '123', peter => '456', steve => '789' );
  write_file( $TEST_FILE, %test_data );
  my %test_data_read = parse_file( $TEST_FILE );

  for my $user ( keys %test_data )
  {
    delete $test_data_read{$user} or die "User $user not found";
  }
  keys %test_data_read and die "Spurious user(s) found in input";

  unlink $TEST_FILE;

  $Input_File = $TEST_FILE;
  set( $_, $test_data{$_} ) for keys %test_data;
  write_file( $Input_File, %Digest );
  %test_data_read = parse_file( $Input_File );
  verify( $test_data_read{$_}, $test_data{$_} )
    or die "Incorrect password for $_"
      for keys %test_data_read;

  unlink $TEST_FILE;
  print "Tests pass\n";
}

sub write_file
{
  my ($filename, %data) = @_;

  open my $fh, '>', $filename or die "Couldn't open $filename for writing: $!\n";
  print {$fh} "$_:$data{$_}\n" for keys %data;
}
```

Check Syntax and run it with the **-t** flag. Now we have some reassurance that the tests are passing. Play with the new Verify option. Now it checks to make sure that the user doesn't enter a command that would cause an error if used as an array index.

Check out this sample dialog to verify that your program is behaving the same way. Type the code below in the Unix shell as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl2
cold:~/perl2$ ./htmanage.pl
Entries in .htpasswd:
 0 Select to Quit
 1 Select to Add New
User or Option (0 - 1)? 1
Add (username): ralph
    (password): malph
Entries in .htpasswd:
 0 Select to Quit
 1 ralph
 2 Select to Add New
User or Option (0 - 2)? 1
Change ralph: (D)elete, (V)erify, (N)ew: V
Password to check: malph
Correct
Entries in .htpasswd:
 0 Select to Quit
 1 ralph
 2 Select to Add New
User or Option (0 - 2)? 1
Change ralph: (D)elete, (V)erify, (N)ew: N
New password for ralph: secret
Entries in .htpasswd:
 0 Select to Quit
 1 ralph
 2 Select to Add New
User or Option (0 - 2)? 1
Change ralph: (D)elete, (V)erify, (N)ew: D
Entries in .htpasswd:
 0 Select to Quit
 1 Select to Add New
User or Option (0 - 1)? 0
cold:~/perl2$
```

Congratulations! You have just finished *Intermediate Perl*! Remember to go back to the syllabus page by clicking on the page tab above to do the final assignments.

I sincerely hope you'll continue to have fun using Perl. Thank you for taking this course! It's been a real pleasure having you. If you're interested in going further, stay tuned! More OST Perl courses are on the horizon that will help you to become even more effective as a Perl developer.