LSU INFORMATION TECHNOLOGY SERVICES

LONI

Introduction to Perl

Wei Feinstein

HPC User Services
LSU HPC/LONI

Louisiana State University

XSEDE
Extreme Science and Engineering Discovery Environment

Outline

- Basic Syntax, Scripts
- Perl Variables
- Control Structures
- File I/O
- Functions
- Error Handling
- Advanced Perl – Modules, CPAN

Perl Overview

- High-level, general-purpose programming language
- Similar to C language in syntax
- Incorporate popular UNIX commands, such as sed, awk and etc
- Practical Extraction and Report Language (PERL)

Appropriate Use

- Scan system logs and report failed log-in attempts.
- Process data for web page displays.
- Read and summarize simulation edit data.
- Rapid Application Development/Prototyping

Inappropriate Use

- Multi-day production runs.
- Applications requiring numerical methods.

Get help within Perl

```
$ man perl
$ perldoc perl
```

The **`perldoc`** command, in particular, gives detailed access to language specifics and even user tutorials.

```
perldoc -f PerlFunc
perldoc -q FAQKeywords
```

Listing Perl Document Subsections

```
$ perldoc
```

```
   perlfaq7          Perl Language Issues
   perlfaq8          System Interaction
   perlfaq9          Networking


perlsyn              Perl syntax
perldata             Perl data structures
perlop               Perl operators and precedence
perlsub              Perl subroutines
perlfunc             Perl built-in functions
```

```
. . . output truncated . . .

$ perldoc perlsyn
```

Perl Syntax

Warning: Perl syntax can be hazardous to your sanity.

It can be approached as any other programming language, but may appear capricious with multiple legal ways to write things.

This tutorial assumes some programming background, so it takes a programmer approach with Perl'isms highlighted as alternatives.

Basic Perl Syntax

If you are comfortable programming in C, then you're in luck. The Perl language syntax is almost (but not quite) identical to C. Some basics elements:

- Comments begin with "#" or (=anyword {code block;} =cut)
- Statements end with ";" (like C)
- Statement blocks are grouped with "{}" (like C, or **`gawk`**)
- First character of variable names is a symbol determining the variable type (unlike C, sort of like Fortran)
- Execute statements from command line (unlike C)

Data::Dumper

- Writes out varialbe contents in perl syntax
  - Very useful in debugging and learning

```
use Data::Dumper;
my $contacts = { 'Frank' => {'email' => 'frank@lsu.edu',
                             'phone' => '578-5655'},
                 'Amy'   => {'email' => 'amy@lsu.edu',
                             'phone' => '578-1420'}
               };
print Dumper($contacts);

#Output:
$VAR1 = {
          'Amy' => {
                     'email' => 'amy@lsu.edu',
                     'phone' => '578-1420'
                   },
          'Frank' => {
                       'email' => 'frank@lsu.edu',
                       'phone' => '578-5655'
                     }
        };
```

A Perl Script – Hello World (hw.pl)

```
#!/usr/bin/perl
use strict;
use warnings;
print "Hello World!\n";
```

How to compile/run perl code:

```
$ perl -c hw.pl    compile for syntax errors only
$ perl -w hw.pl    compile for syntax and warnings
$ perl hw.pl       compile and run

   Hello World!
```

Command Line Run

Command line options let you run arbitrary Perl statements. e.g., -e, indicates the command line argument is a Perl statement:

```
$ perl -e '... statement ...  ;'
$ perl -e 'print "Hello World!\n";'; -e …
```

Print statement

```
print "hello World!\n";
```

`""` ..... Indicates a string of characters.
`\n` ..... String literal that inserts a newline character.
`;` ..... Marks the end of the Perl statement.

String Literals

\n is just one of several string literal sequences that represent special characters, usually control and other non-printable characters. There are many more. Here's a few:

Literal	Use		Literal	Use
\t	Tab character		\xnn	Hexadecimal char code
\n	Newline		\c[	Control char (i.e. ^[)
\r	Carriage return		\l	Next char lower case
\f	Form feed		\u	Next char upper case
\b	Backspace		\L	Start all lower case
\a	Alarm or Bell		\U	Start all upper case
\e	Escape		\E	End all upper/lower case
\nnn	Octal char code		\\	Single backslash

- Basic Syntax, Scripts
- **Perl Variables**
- Control Structures
- File I/O
- Functions
- Error Handling
- Advanced Perl – Modules, CPAN

Perl Variable Concepts

Perl does ***lazy*** typing: it figures out what you mean when the data is used (i.e., is a number or is it a character string).

Perl types have more to do with organization than type of values:

- Scalar
- Array
- Associative Array

The type of a variable is indicated by the first character.

Scalar Variable $var

The simplest type of Perl variable holds a single value, so it is called a scalar variable. Must have a "$" as first letter of name:

```
$a = "Sum of x +y is ";
$x = 15;
$y = 42.137;
$z = $x + $y + $w;

print $a, $z;  #Sum of x + y is 57.137
```

Try scripting and running this. Are you surprised by the answer?

Array Variables @array

Array variables, also called ***lists***, let you collect multiple items so they can be referred to by one name. Use a "@" as the first letter of the name:

```
@month = (Jan, Feb, Mar);
print @month,"\n";
print "@month\n";
print "$month[2]\n";
```

Note how a single element is indexed: **`$month[i]`**

Basically stores elements as strings, so this is legal:

```
@demo = (27, Monday, -38, football)
```

Associative Array Variables %aa

One of the most powerful features of Perl (and other scripting languages) is the ***associative array.*** Recall that we used numbers to index into an array variable. An associative array is said to store *key:value* pairs, and uses strings as indices. They must have a "%" as the first letter of their name:

Consider an array to hold month names and number of days:

```
%dim=(Jan, 31, Feb, 28);
```

Indexing an associative element is different than an array:

```
print "$dim{Jan}\n";
```

Special Variables

The Perl language includes some 60+ special variable names that we'll cover as the come up. The general form is a single letter scalar variable, i.e.

`@_` : List of arguments passed to a subroutine
`$.` : Current line number (file)
`$_` : Default variable
`$0` : Name of the Perl script from the command line
`@ARGV`: command-line args

Perl special vars quick reference

Variable Scope

The scope of a variable means where in the script it is recognized.

Perl variables have GLOBAL scope. A variable name is recognized anywhere in a script, allowing its value to be used or accessed at will.

This is very different from most languages.

Quoting

Syntax of 3 Little Quotes

As with shell scripting, string handling behavior is controlled by one of 3 different quote characters:

' ' . . **Single quotes – suppresses variable expansion.**

` ` . . **Back quotes (back tics) – command substitutions.**

" " . . **Double quotes – allows variable and string literals.**

The type of quote used changes the way the quote contents are interpreted.

Single Quotes

Perl uses the single quote character (`) to indicate a *string*. That is, a sequence of characters that are to be interpreted exactly as presented.

```
$n = 5; print '$n\n';
```

Will display exactly:

```
$n\n
```

Back Quotes

Back quotes take the string, pass it to the shell as a system command, and returns the execution result as a string.

```
$dir = `pwd`;
```

pwd is the shell "print working directory" command and outputs the directory it is run in.

Double Quotes

The use of double quotes (“) causes Perl to scan the string before using it and perform the following actions:

1. If it finds any variable names, it will insert the value of the variable in place of the name.
2. If it sees any string literals, it will insert the value in place of the literal.
3. If it sees back-quotes, the command result is inserted in place of the back-quoted string.

variable expansion and ***command substitution***.

Example

```
$dir = "My current dir";
$files = `ls`;
print "$dir contains:\n$files\n";
```

What is output?

Alternative Quotes

Just in case you decide you don't like fussing with actual characters, Perl provides alternate quoting mechanisms:

`q(stuff)`	`...`	Same as single quotes (`' '`)
`qq(stuff)`	`..`	Same as double quotes (`" "`)
`qx(stuff)`	`..`	Same as back quotes (`` ` ` ``)

And, if you don't want to use `( )` to bracket your quote, you could use `/ /`:

```
$dir = qx/pwd/;
$dir = qx(pwd);      <= all the same
$dir = `pwd`;
```

Operators and Expressions

Operators

Operator	Description	Associativity
`() [] {}`	Function call, array indices	Left to right
`++ --`	Increment, decrement	None
`! ~ -`	logical not, bitwise not, negation	Right to left
`**`	exponentiation	Right to left
`=~ !~`	match, not match	Left to right
`* / % x`	multiply, divide, modules, times	Left to right
`+ - .`	add, subtract, string concat.	Left to right
`<< >>`	bit shift left, bit shift right	Left to right
`-r -w -x -o -f -e -z -s -d -l -p -S -b -c -u -g -k -t -T -B -M -A -C`	file test operators (e.g. (-r $fname) is true if file fname is readable.)	None
`< <= > >= lt le gt ge`	numeric/string relative comparison	None
`== != <=> eq ne cmp`	numeric/string comparison	None
`& | ^`	bitwise and, or, exclusive or	Left to right
`&& ||`	logical and, or	Left to right
`..`	range	None
`x?y:z`	ternary test	Right to left
`= += -= /= %= .= *= **= x= <<= >>= &= |= ^=`	assignment (e.g. $num **= 2 means square value in $num, assign to $num)	Right to left
`, (comma)`	Evaluate left, discard, evaluate right.	Left to right

Expressions

An expression is any Perl construct that produces a value. If an assignment is made, the value assigned is the result of the expression:

```
4 + 9 ..... An arithmetic expression, value 13.
$x = 4 + 9; ..... Also arithmetic, also value 13.
'a' ..... Single letter, value is 'a'.
$s = 'a' . 'b' .. String concatenation, value is 'ab'
$x > 12 ..... TRUE (non-null)
$x == 14 ..... FALSE (null)
-x "/bin/ls" .... TRUE if file /bin/ls is executable.
```

- Basic Syntax, Scripts
- Perl Variables
- **Control Structures**
- File I/O
- Functions
- Error Handling
- Advanced Perl – Modules, CPAN

If…elsif…else (conditional construct)

The **if** statements can take one of 3 forms:

```
if ( expression ) { statement block }

if ( expression ) { statement block 1 }
   else { statement block 2 }

if ( expression 1 ) { stmt block 1 }
  elsif ( expression 2 ) { stmt block 2 }
  . . .
  elsif ( expression N )  { stmt block N }
  else { statement block  }
```

Comparison Operators

	numbers	strings
Equal	==	eq
Not equal	!=	ne
Greater than	>	gt
Greater than or equal	>=	ge
Less than	<	lt
Less than or equal	<=	le

Unless Control Structure (if not)

```
unless ( expression ) { statement block }

unless ( expression ) { statement   block 1 }
    else { statement block 2 }
```

Loop Control Structures

```
while ( expression ) { statement block }

until ( expression ) { statement block }

for ( init expr; test expr; incr expr )
    { statement block }

foreach $varnam ( @array ) { statement block }
```

Loop Control

- `next` – jump to the next iteration
- `redo` – jump to the start of the same iteration
- `last` – jump out of the loop

`for` Loop

The **for** statement implements a classical do-loop.

```
for ( $i = 0; $i < 10; $i++ )
   { $j += $i + 3; print "$j\n"; }
```

`foreach` in Arrays

The foreach statement is quite powerful, as it steps through an array one element at a time. You need not know how many elements it contains:

```
#! /usr/bin/perl
@months = (Mar, Apr, May, 17);
foreach $m ( @months ) { print "$m\n"; }
```

The elements print out in the same order they were stored.

`foreach in` Associative Arrays

Scan through associative arrays require the help of two functions:

`keys` – to extract a list (scalar array) of all keys in the array,
`values` – to extract a list (scalar array) of all values in the array:

Example:

```
#! /usr/bin/perl
%dim = (Jan, 31,  Feb, 28,  Mar, 31,  Apr,  30);
foreach $m (keys  %dim)
   { print "$m has $dim{$m}  days.\n"; }
```

What do you think will be output?

Random Order

Associative arrays are implemented as hash tables, so the order in which elements are added is NOT maintained. Here is the actual result of running the previous script:

```
Mar has 31 days.
Feb has 28 days.
Apr has 30 days.
Jan has 31 days.
```

- Basic Syntax, Scripts
- Perl Variables
- Control Structures
- **File I/O**
- Functions
- Error Handling
- Advanced Perl – Modules, CPAN

Standard File I/O

In general, Perl follows the Unix/Linux model by supporting 3 I/O streams. A particular stream is referenced via a ***file handle***.

`STDIN` `...` The standard input stream (i.e. keyboard)
`STDOUT` `..` The standard output stream (i.e. terminal)
`STDERR` `..` The standard error stream (i.e. terminal)

Having these 3 streams implies than running any Perl script can make full use of shell I/O redirection. User defined file handles are also allowed.

Note: these are string literals and do not need a "$" in front of name.

Read from STDIN

Reading from STDIN is nearly as easy as writing to STDOUT. Try this little snippet in a script:

```
print "Enter first name: ";
$first  = <STDIN>;   #Jone
print "Enter last name: ";
$last = <>;    #Smith
print "$first, $last";
```

Chop()

The output looks funny because the read operation retains the newline character from the input. Unfortunately, the user must take steps to deal with this. Try using the **chop** function and see if there is a difference:

```
$first = <>;
chop($first);
```

and/or

```
chop($last=<>);
```

Create file handles

```
open($handle, $direction, $name);
```

```
open( $fh, "<", "foobar" ) ...  Open for input.
open( $fh, ">", "barfoo" ) ...  Open for output.
open( $fh, ">>", "fiefum" ) ..  Open to append.
```

Other ways

The open command is used to associate file names with I/O streams. Other allowed forms for input include:

```
open( $fh, $name );
$fh = $name;
open( $fh, "> new.lst" );
open( $fh, ">> append.lst" );
close ($fh);
close $fh;
```

Open For Pipe I/O

Input from a command pipe:

```
open( $inpipe,  "date |" );
```

Output to a command pipe:

```
open( $outpipe,  "| wc -l" );
```

Putting Pieces Together

Sample Data File (grade.dat)

```
Jones, Alpha : 90 : 90 : 100 : 50 : 90 : 100 : 95
Doe, Betty : 89 : 91 : 99  : 60 : 70 : 100 : 90
Johnson, Charlie : 88 : 92 : 98  : 70 : 80 : 95  : 95
Miller, Daniel : 87 : 93 : 97  : 80 : 90 : 60  : 80
```

What is the average for each student ?

grade-avg.pl:

```
#!/usr/bin/perl -w
use strict;

my (@Fld, $sum, $name, $n, $i, $fh);
my $string;
$[ = 1;                    # set array base to 1 instead of 0
open( $fh, "<" ,"grade.dat" );
while (<$fh>) {
    chomp();               # strip record separator
    @Fld = split(/:/, $_);      # $_ is default variable
    for ($i = 1; $i <= $#Fld - 1; $i++) {
        # $#Fld is number of elements in Fld
        $sum += $Fld[$i+1];
    }
    $name=$Fld[1];
    $n=scalar @Fld - 1;
    printf "Average for HW%s is %5.2f\n", $name, $sum/$n;
    $sum=0;

}
close $fh;
```

```
[wfeinste@mike5 examples]$ perl grade-avg.pl
Average for HWJones, Alpha is 87.86
Average for HWDoe, Betty is 85.57
Average for HWJohnson, Charlie is 88.29
Average for HWMiller, Daniel is 83.86
```

- Basic Syntax, Scripts
- Perl Variables
- Control Structures
- File I/O
- **Functions**
- Error Handling
- Advanced Perl – Modules, CPAN

Built-in Functions

Much of the power, and steepest barrier to learning, lies in using the appropriate Perl function for the task at hand. We've seen several already:

```
%dim = (Jan, 31, Feb, 28, Mar, 31, Apr, 30);
@months = keys(%dim);  # Remember "()" are optional!
@numdays = values(%dim);
chop($string);
print "string";
open( $handle );
close( $handle );
```

Numeric Functions

- Again the usual suspects

`abs`	Absolute value
`cos, sin, tan`	Trigonometric
`exp`	Exponentiation
`log`	Logarithm
`rand`	Random number generator
`sqrt`	Square root

Examples of Build-in Functions

split: my $s = "The black cat climbed the green tree";
my @array = split " ",$s; print $array[1]; # black

join: my @names = ('Frank', 'John', 'Amy', 'Olivia');
print join ':',@names; # Frank:John:Amy:Olivia
String operations:

concatenation (.): $f="jone"; $l="Smith";
$name=$f." ".$l; #Jone sSith

repeat(x): $A="a"; print "$A" x 3; #aaa
Length: $name="Smith"; print length $name; #6

reverse: @array = (2, 5, 3, 1);
@reversed = reverse @array; #(1, 3, 5, 2)

Array Functions

pop, push, shift, unshift

```
my @names = ('Frank', 'John');

#pop: remove and return last element
print pop @names; # will print 'John'
#push: add a new element to the end
push @names,'Amy'; # @names is now ('Frank','Amy')
#shift: remove and return first element  say
shift @names; # will print 'Frank'
#unshift: add a new element to the start
unshift @names,'Fred'; @names is now ('Fred','Amy')
```

@_

- The default array
  - Within a function, @_ contains the parameters passed to that subroutine
  - Many array operations within the function use its value if none is provided

Custom Function

Define a subroutine:
Declare local $m, :
@_ is list of arguments passed
find maximum:
Return value:
Call the function:
Print out the result:

```
#! /usr/bin/perl

sub max {
    my ( $m, $n );
    $m = shift( @_ );
    foreach $n (@_) {
        $m = $n if $m < $n;
    }
    return $m;
}

$m = max( 42, 17, 86, 103 );
print $m, "\n";
```

Return > one values

```
#!/usr/bin/perl -w
    use strict;

    # Subroutine prototypes
    sub get_two_arrays();

    # Get two variables back
    my ($one, $two) = get_two(); #call sub
    print "One: $one\n";
    print "Two: $two\n";

    sub get_two() {     #define sub
        return ("one", "two");
    }
```

Return > 1 arrays

```
#!/usr/bin/perl -w
    use strict;
    # Subroutine prototypes
    sub get_two_arrays();

    # Get two variables back
    my ($one_ref, $two_ref) = get_two_arrays();

    my @one = @$one_ref;     #derefernece returned array
    my @two = @$two_ref;
    print "First: @one\n";
    print "Second: @two\n";

    sub get_two_arrays() {
        my @array1 = ("a", "b", "c", "d");
        my @array2 = (1, 2, 3, 4);
        return (\@array1, \@array2); #return by reference
    }
```

Correct: First: a b c d
Second: 1 2 3 4

Wrong: First: a b c d 1 2 3 4
Second:

- Basic Syntax, Scripts
- Perl Variables
- Control Structures
- File I/O
- Functions
- **Error Handling**
- Advanced Perl – Modules, CPAN

Dealing with Errors: Programmer Way

Programs on a Unix-like system are expected to return a status code when they terminate. A value of 0 is taken to mean success. Any non-zero value is taken to mean there was a problem. The actual value indicates the type of problem:

```
if(...)exit( expression );
 e.g., exit 1;
```

The Perl Way: `die/warn/eval`

Die: a script to exit if a test statement is true.

`die.pl:`

```
#! /usr/bin/perl
$x = -42;
if ( $x < 0 ) { die "x is $x, stopped"; }
```

`The output:`

```
x is -42, stopped at ./d line 3
```

The text in **red** was added by the **die** function.

Dealing With Errors: **warn**

warn.pl

```
#! /usr/bin/perl
$name = "";
if ( $name == "" ) {
   warn "name is empty, but continuing"; }
```

The output:

```
name is empty, but continuing at ./w line 3.
```

Dealing With Errors: eval

The script:

```
#! /usr/bin/perl
$x = 0;
eval{ if ( $x == 0 ) { die "x is 0!"; } }
if ( $@ ) { warn $@; } # just msg
print "Pressing on . . .\n";
```

The output:

```
x is 0! at ./ev line 3.
Pressing on . . .
```

- Basic Syntax, Scripts
- Perl Variables
- Control Structures
- File I/O
- Functions
- Error Handling
- **Advanced Perl**

Advanced Perl

There are features that move Perl closer to a general purpose language, among them:

Regex
Modules and the CPAN.

Regex

- Regex stand for **REG**ular **Ex**pression
  - Powerful tool for text processing
  - Allows users to define a pattern to describe characteristics of a text segment
  - Can be used to match or modify text
  - Perl regex documentation: `perlretut`, `perlreref`, `perlre`

Regular Expressions

You may have run into this concept under the name "wild cards" used in file name searches on DOS or Unix. Perl takes the concept much further. They support ***matching*** and ***substitution.*** First ***m:***

`$_ =~ /alpha/` .. Does a sequence of characters in $_ match with "alpha".

`$_ =~ /^alpha/` .. Match only if "alpha" at start of line.

`m/alpha/` .... "**m**" is optional if "**/**" are used.

`m?alpha?` .... does same thing (any character pair will do). For instance, `m;/alpha;` lets you match a pattern containing "**/**".

Substitution

Substitution takes the same form as matching, but allows the matched text to be replaced with something else. Use **s** instead of **m**, and most of the rules still apply:

s/pat/repl/ . . .	Replace first occurance of **pat** with **repl**.
s/pat/repl/g . .	Replace all occurrances of **pat** with **repl**.
s!pat!repl! . . .	Use **!** instead of **/**

Metacharacters

A subset of the list:

`. ..........`	Match any letter.
`^ ..........`	Match at start of string.
`$ ..........`	Match at end of string.
`[abc] .......`	Match any letter in the set.
`[^abc] ......`	Match any letter NOT in the set.
`x? ..........`	Match 0 or 1 occurence of "x"
`x* ..........`	Match 0 or more occurrences of "x"
`x+ ..........`	Match 1 or more occurrences of "x"
`x{m,n} ......`	Match at least m, but no more than n, "x"
`hat|cat|bat ..`	Match one of the words.
`(str) .......`	Mark a pattern occurrence
`\2 (or $2) ...`	Insert result of second occurrence
`\b ..........`	Match a word boundary.
`\d ..........`	Match a digit.

Define Patterns

- The qr// operator define patterns that can be used and reused for later match

```
my $http = qr/^http:\/\//;
my $www = qr/www/;
while (<FILE>) {
    print if /$http$www/;
}
```

Examples (pattern)

```
[wfeinste@mike5 examples]$ cat grade.dat
Jones, Alpha:90:90:100:50:90:100:95
Doe, Betty:89:91:99:60:70:100:90
Johnson, Charlie:88:92:98:70:80:95:95
Miller, Daniel:87:93:97:80:90:60:80
[wfeinste@mike5 examples]$
[wfeinste@mike5 examples]$ perl -ne 'print if /^Jo/' grade.dat
Jones, Alpha:90:90:100:50:90:100:95
Johnson, Charlie:88:92:98:70:80:95:95
[wfeinste@mike5 examples]$
[wfeinste@mike5 examples]$ perl -ne 's/Johnson/Bush/g; print ' grade.dat
Jones, Alpha:90:90:100:50:90:100:95
Doe, Betty:89:91:99:60:70:100:90
Bush, Charlie:88:92:98:70:80:95:95
Miller, Daniel:87:93:97:80:90:60:80
[wfeinste@mike5 examples]$
```

Awk in Perl

```
my $line="1 2 3 4 5 6";
my $third_last=`echo "$line"|awk '{print \$(NF-3)}'`;
print $third_last

Result: 4
```

Perl Modules .pm

- Think Perl modules as libraries – reusable codes
- CPAN – tons of modlues developed by the Perl community (www.cpan.org)
  - Before setting out to write something serious, check CPAN first

Installing Modules

- Option 1: manual installation
  - Download the tarball and extract the content
  - Create a Makefile
  - Make, make test and make install
  - No root access, e.g. perl Makefile.PL prefix=…
- Option 2: use the "cpan" module
  - Provides a console to search, download and install modules and their dependencies automatically

Use a Perl Module

Someone has done the heavy lifting and created Text::CSV.

```
  load module:
set file name:
create parser:
 open to read:
 scan by line:
   parse line:
access fields:
      reprint:
```

```
#! /usr/bin/perl

use lib "/home/wfeinste/perl-local/share/perl5";
use Text::CSV;
my $file = 'directory.csv';
my $csv = Text::CSV->new();
open (my $fh, "<", $file) or die $!;
while (<$fh>) {
    if($csv->parse($_)) {
        my @columns = $csv->fields();
        print join("|",@columns) . "\n";
    }
}
close $fh;
```

The output =>

```
12345678|Doe, Joe C.|Engineer|EE Dept
23456789|Jane, Mary S.|Engineer|Manufactoring
34567890|Kilgore, Was H.|Artist|Marketing
```

Module Hiding Places

The use function finds the named module much the same way as the shell finds commands: it searches a list of directories. When Perl is installed, it creates a list, @INC, with the names of all the include directories. The contents can be reviewed from the command line:

```
$ perl -le 'print foreach @INC'
```

Custom Module Use

Normally it requires a system administrator to install Perl and add extension modules. Such actions update @INC automagically.

Users are free to install their own modules and have multiple ways of telling Perl where to search for them:

`PERL5LIB .........` An environment variable with ":" separated directory names. Searched before the contents of @INC.

`-I dirname .......` A command line option used at script runtime, either with Perl, or on the shebang line in the script.

`use lib dirname ..` Perl command in the script to name directory.

Custom Examples

These 4 approaches all accomplish the same thing:

```
$ export PERL5LIB=/home/$USER/Perl
```

or

```
$ perl -I ~/Perl directory.pl
```

or

```
#! /usr/bin/perl -I /home/$USER/Perl
```

or

```
#! /usr/bin/perl
use lib "/home/$USER/Perl"
```

Closing

Perl syntax is flexible (maybe too flexible?).

Do some research before writing something new – may find it available, or at least a good starting point.

Be aware of Perl version differences.

Get comfortable with a good Perl reference or two or three ....

http://perldoc.perl.org

http://learn.perl.org