1. Perl Basics - Variables, Flow Control, Basic I/O – AS...

Homework Review

1. Write a script that uses a **loop** to generate a basic project directory tree (using the standard described in the first lecture, or one of your own devising). (5 pts)

```
for i in bin etc lib share tmp ; do mkdir $i ; done
```

1. Write a script that uses a **loop** to copy HTML and CGI files from a project directory to a web server directory. (Assume the project is in `~/project`, with `html` and `cgi` sub-directories, with no sub-directories under those; assume a destination of `~/public_html`.) (5 pts)

```
# set up destination directories
   mkdir -p ~/public_html/html
   mkdir -p ~/public_html/cgi
   # make sure we're in the right place
   cd ~/project
   # copy files from html
   cd html
   for i in * ; do cp $i ~/public_html/html ; done
   # copy files from cgi
   cd ../cgi
   for i in * ; do cp $i ~/public_html/cgi ; done
```

1. Modify the script in the previous problem so that it only copies files if the source file (the original) is newer than the destination file (the copy). (Hint: look at the options for the `test` command.) (10 pts)

```
# set up destination directories
   mkdir -p ~/public_html/html
   mkdir -p ~/public_html/cgi
   # make sure we're in the right place
   cd ~/project
   # copy files from html
   cd html
```

```
    for i in * ; do
      if [ $i -nt ~/public_html/html/$i ]; then
        cp $i ~/public_html/html
      fi
    done
    # copy files from cgi
    cd ../cgi
    for i in * ; do
      if [ $i -nt ~/public_html/cgi/$i ]; then
        cp $i ~/public_html/cgi
      fi
    done
```

1. Extend the script in the previous problems so that an environment variable can be used to control which copying behavior ('update only' versus 'copy all') is used. (That is, `export COPY_ALL=1` will copy all files regardless of file age; `unset COPY_ALL` will result in the "copy only if changed" behavior added in the previous problem.) (10 pts)

```
 #! /bin/sh
    # set up destination directories
    mkdir -p ~/public_html/html
    mkdir -p ~/public_html/cgi
    # make sure we're in the right place
    cd ./project
    # copy files from html
    cd html
    for i in * ; do
        if [ $COPY_ALL ]; then
            cp $i ~/public_html/html
        elif [ $i -nt ~/public_html/html/$i ]; then
            cp $i ~/public_html/html
        fi
    done
    # copy files from cgi
    cd ../cgi
    for i in * ; do
```

```
        if [ $COPY_ALL ]; then
            cp $i ~/public_html/cgi
        elif [ $i -nt ~/public_html/cgi/$i ]; then
            cp $i ~/public_html/cgi
        fi
    done
```

Documentation

Perl has extensive online help available with the `perldoc` command.

`perldoc perlop` will tell you all about Perl's built in operators.

`perldoc -f open` will tell you all about the `open()` function.

The `perldoc` command is the equivalent of the `man` command in the shell -- it's how you access the online documentation for Perl. You're going t to become very familiar with it.

Perl Programming: Variables

Perl variables work basically just like shell variables, with one exception: you have to use a special character -- which is called a "sigil" -- when you set them as well as when you use them:

```
# set a variable
 $var = 1;

 # set it to something different
 $var = 'foo';
```

(Since that script only sets a couple of variables and doesn't do anything else, it doesn't really have any interesting output to speak of.)

Just like shell variable names, valid Perl variable names begin with a letter and contain only letters, numbers, and underscores.

Perl Variables

Perl variables come in three (main) flavors:

- Scalars
- Arrays
- Hashes

We'll discuss each one in turn.

Scalars

A **scalar** is one single thing. It might be a integer, or a string, or a single character, or a floating point number -- or it might be more than one, during its lifespan.

Scalar values are indicated with a '**$**' before their name. (Remember that it looks sort of like an 'S', which is what 'scalar' starts with.)

Here are some scalars:

```
$foo = 1;

 $foo = "bar";

 $bar = "$foo";

 $bar = "${foo}baz";

 $bar = '$bar';
```

Perl does **variable interpolation** in a way very similar to the shell. The third example sets `$bar` equal to the string "bar", because that is the value of `$foo` at that point in time, and the double quotes cause the variable to be interpolated.

The fourth example shows how you can use curly brackets to delimit a variable name if you need to interpolate it right into the middle of a bunch of other text, without surrounding white space. After this line is executed, what do you think the value of `$bar` is?

The final example shows how you can disable variable interpolation -- by using single quoted strings. The value of `$bar` after this line executes is *literally* the string '$bar', because the single quotes disable string interpolation.

Arrays

An **array** holds a list of things -- of scalars. Each thing in the list can be a different type -- one can be an integer, another a floating point number, a third a string -- but the array variable contains the entire list.

Array values are indicated with a '**@**' before their name. (Remember that it's like an 'A', which is what 'array' starts with.)

In order to access an individual item in the list, you use its **index value** -- its position in the list. The first item in the list has an index of **0**. (*I.e.*, Perl's arrays are **zero-indexed**. Some other languages are **one-indexed**, but we're not going to worry about those. Perl starts counting at zero.)

When accessing a single thing from an array, you're accessing a **scalar** -- and therefore, you use a '$' in front of the variable name. You also have to specify the index value, which you do in square brackets ([]) following the variable name.

Some examples:

```
# setting an array to contain a list:
 @array = ( 1 , 2.3 , "foo" );

 # accessing a particular value in the array:
 $item = $array[1];

 # special variable $#array contains the index of
 # the last item in the list
 $last_item = $array[$#array]

 # can also use negative indexes to access items
 # from the end of the list inward
 $item = $array[-2];
```

Notes on these examples:

- In the first example, note that in Perl, a list is a comma-separated set of scalars. Here the parentheses are grouping the list together to clearly indicate the intent to assign all three items to `@array`.
- In the second example, note that when we're accessing a single element from the array, we use a `$` sigil in front of the variable name, not a `@` sigil. If you use the latter sigil, you're going to get a cryptic error message.

Hashes

A **hash** is similar to an array, in that it too is a list of things. But instead of being indexed by numbers, hashes are indexed by **unique keys**.

Every **value** in a hash is associated with a unique **key**, which is used to access and set its associated value.

Hash variables are indicated with a '**%**' before their name. (Remember that hashes involve pairs of keys and values, like the pair of circles in the percent sign.)

Just like an array, when you access an individual item in a hash, you're accessing a **scalar**, so you use a '$' in front of the variable name. You also have to specify the **key value**, which you do in a pair of curly brackets ({}) following the variable name, as in the following examples.

Important note: unlike arrays, the set of items in a hash **have no particular ordering**. Don't expect to get things back out in the same order that you put them into the hash in.

```
# set up a hash variable:
%hash = ( 'one' , '2' ,
          '3'   , 'four' );

# clearer way of setting up a hash:
%hash = ( 1     => 'two' ,
          three => 4     );

# read out a value
$item = $hash{'three'};

# easier way to read out a value
$item = $hash{three};

# get a list of all keys in the hash
@hashkeys = keys %hash;

# get a list of the values
@hashvals = values %hash;
```

Notes on these examples:

- In the first example, we're again assigning a list -- a series of comma-separated values, enclosed in paratheses. This time, because we're assigning the list to a hash, the odd-numbered list elements (first, third, etc.) become hash keys, and the even-numbered list elements (second, fourth, etc.) become the values associated with the preceeding key. What do you think would happen if you tried to assign a list with an odd number of elements to a hash?

- In the second example, we're seeing a demonstration of the so-called "fat comma" or "fat arrow" operator. The "fat comma" (`=%gt;`) acts just like a comma except that it also has the effect of causing the item on its left to be treated as if it is quoted. (Note how the elements corresponding to keys in the first example are quoted, while those in the second example are not.)
- Similarly, the third and fourth examples demonstrate that the inside of the curly brackets in a hash value lookup also cause what is inside them to be treated like it is surrounded by quotes.
- The utility of this quoting behavior -- both to the left of the fat comma and inside the curly brackets -- will become more apparent shortly, when we talk about `strict` mode. For the moment, remember that it happens, and trust that it is a good thing.

Don't Do This

Another important note: because the different variable types exist in different **namespaces**, you can have a scalar, a hash, and an array, all with the same name.

```
# scalar
$foo = 'bar';

# array
@foo = ( 'bar' , $foo );

# hash
%foo = ( $foo => 'bar' );
```

It's generally a very confusing thing to do. **Don't do this!**

Variable Scope

Variable scope refers to the issue of when a variable is **visible** as a program executes. You don't always want all variables to be accessible all the time, because it makes things harder to debug. (In fact, you generally want the smallest possible set of variables available at any given time, to cut down on the chance of bugs.)

Before we can talk about variable scope, we have to talk about code organization. In Perl, code is organized into **BLOCKS**. A code block may consist of all the code in a particular file, or it may only consist of a very small region bounded by curly brackets.

Perl variables come in two different sorts: **globally-scoped variables**, which are visible everywhere in a program, and **lexically-scoped variables**, which are only visible in the scope in which they're declared.

Different keywords are used for globally- and lexically-scoped variable declarations. Globals are declared with `our`; lexicals are declared with `my`. (There's a third declaration operator, `local`, which produces a different kind of scoping that we're not going to talk about.)

Variable Scoping Examples

The scope of a lexically-scoped variable begins at its declaration and extends to the end of the current code block. Observe:

```
# declare a global:
our $global;

# start a code block:
{
  # $global is visible here

  # declare a lexical
  my $lex;

  # block ends:
}

# we can still see global here, but can we see $lex?
```

use strict; use warnings;

Perl by itself won't force you to declare all your variables -- but doing so is good for you (in a you-must-eat-your-vegetables sort of way). The way you ask Perl to make you declare everything is this:

```
use strict;
```

Perl will also **warn** you about many common programming errors, which can save you lots and lots of time. The way you ask for these warnings is:

```
use warnings;
```

You should *strongly* consider always using both these flags, at the very top of your programs, immediately below the shebang line, like so:

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

`strict` and `warnings` are both examples of what are called **compiler pragmata** -- directives you can put in your code that affect how your code is compiled and run.

While you are not *required* to use these pragamata in your code, it will be graded with both of them in effect, and any warnings or failure to run properly due to errors that would have been caught by them will be greatly detrimental to your grade. (In other words, use them. Always, always use them.)

Many code examples in this class will not explicitly declare these pragamata, however, unless otherwise noted, they should always be assumed to be in effect.

Location of the Perl interpreter

The traditional location of the Perl interpreter -- the program responsible for running all your Perl code -- is `/usr/bin/perl`. The version of Perl at that location, however, is somewhat old and out of date. It's generally considered to be a bad idea to update this so-called **system Perl**, as there may be parts of the operating system and associated code that depend on quirks of this particular version.

I have instead installed a newer version of Perl at `/opt/perl/bin/perl`. Modules that you need to use during this class (such as the Bioperl modules and others) will be installed with this version of Perl, so you should specify this path in the shebang line of the code you write for this class.

Later on in the class, we'll see how you can use the `perlbrew` script and the `local::lib` module to install your own up-to-date copy of Perl and your own modules, even if the sysadmins on the machine won't do it for you.

Truth

Just like the shell, Perl assigns a value to every command that executes, and you can evaluate that command for truth or falsehood.

Unlike the shell, Perl considers **zero to be false**, and **any non-zero value to be true**. Yes, this is 100% opposite of the shell.

But aside from that complete contradiction at the heart of things, pretty much everything else is exactly the same!

Binary Comparison Operators

Perl has a variety of **binary operators** -- commands which evaluate relationships between two variables (or commands).

There are some differences from shell here too. First, there's no `test` or `[` command -- you just do a comparison where you need to do a comparison.

Second, some of the operators do different things. To compare equality in shell, you use '=' -- but in Perl, '=' is for **assignment**. To test equality in Perl, you must use '**==**'.

Perl treats scalar comparisons differently depending on whether the variables contain strings or numbers. How does it know? You tell Perl what to expect by which operators you use:

```
# numeric comparisons:
< , <= , == , != , >= , > , <=>

# string comparisons:
lt , le , eq , ne , ge , gt , cmp
```

Unary File Test Operators

Perl also supports "file test" operators (aka **unary operators**). These work just like they do in the shell:

```
-e $FILE  # check file existence
-x $FILE  # check if file is executable
-d $FILE  # check if file is actually directory
# etc
```

Flow Control

Perl supports (almost) all the same **flow control constructs** as the shell does, and more besides.

All Perl flow control constructs have the same basic form:

```
OPERATOR ( TEST_CONDITION ) {
  BLOCK
}
```

You can also write the code in a "short cut" form:

```
BLOCK OPERATOR ( TEST_CONDITION );
```

(That's only really used when BLOCK is just a single statement.)

`if/elsif/else`

First type of conditional: the `if/elsif/else`:

```
#! /usr/bin/perl

use strict;
use warnings;

my $var = 'foo';

if ( $var eq 'foo' ) {
  print "'$var' equals 'foo'!\n";
}
else {
  print "'$var' doesn't equal 'foo'!\n";
}

# short cut way
print "'$var' equals 'foo' this way too!" if ( $var eq 'foo' );
```

```
   $ ./if.pl
    'foo' equals 'foo'!
    'foo' equals 'foo' this way too!
```

Things to note:

- The use of **pragmas**
- Declaring variable before use
- Initializing it at the same time
- No "test" command; just evaluate the conditional operator
- It's "`elsif`" not "`elif`"

Now with `elsif`!

Here's one with an `elsif`:

```
#! /usr/bin/perl

use strict;
use warnings;

my $var = 'bar';

if ( $var eq 'foo' ) {
  print "'$var' equals 'foo'!\n";
}
elsif ( $var eq 'bar' ) {
  print "'$var' equals 'bar'!\n";
}
else {
  print "'$var' doesn't equal 'foo'!\n";
}

# short cut way
print "'$var' equals 'foo' this way too!" if ( $var eq 'bar' );
```

```
    $ ./if.pl
     'bar' equals 'bar'!
     'bar' equals 'bar' this way too!
```

Unless you want to do it this way...

Perl also offers `unless`, which is an `if` with the *sense* of the test reversed:

```
#! /usr/bin/perl

use strict;
use warnings;

my $var = 'foo';

unless ( $var eq 'foo' ) {
  print "darnit, '$var' doesn't equal 'foo'!\n";
}
else {
  print "'$var' _does_ equal 'foo' -- YAY!\n";
}

# short cut way
print "'$var' doesn't equal 'foo' this way either!" unless ( $var eq 'foo' );
```

```
    $ ./unless.pl
     'foo' _does_ equal 'foo' -- YAY!
```

`for`

The `for` construct in Perl works differently than the shell `for`. Here's the generic form of the Perl construct:

```
for ( INITIALIZATION ; TEST ; MODIFY ) {   BLOCK }
```

As this is basically the same syntax as a for loop in the C programming language, this style of loop is often called a 'C-style' for loop.

`for` example

And here's a more specific example:

```
#! /usr/bin/perl

use strict;
use warnings;

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

```
   $ ./for.pl
    0
    1
    2
    3
    4
    5
    6
    7
    8
    9
```

`foreach`
The `foreach` construct provides something more like a shell-style `for`:

```
#! /usr/bin/perl
```

```
use strict;
use warnings;

my @list = ( 'one' , 'two' , 'three' );

foreach my $var ( @list ) {
  print "$var\n";
}
```

```
   $ ./foreach.pl
    one
    two
    three
```

`for == foreach`
IMPORTANT: The previous two slides lied. `for` and `foreach` are actually **the exact same thing**. (Perl distinguishes between them by looking at the conditional part and deciding whether it's a list-based or an index-based loop.)

`while` and `until`

Perl also provides `while` and `until`, which work exactly as in the shell:

```
#! /usr/bin/perl

use strict;
use warnings;

my $i = 0;
while ( $i < 10 ) {
  print "$i\n";
  $i++;       # same thing as '$i = $i + 1'
}
```

```
$ ./while.pl
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
```

Note that this `while` is **exactly** the same thing as the first `for` loop.

Any indexed `for` loop can be replaced with an equivalent `while` loop.

`do {} while` and `do {} until`

Both `while` and `until` come in an alternate form:

```
do {   BLOCK } while ( TEST );
```

The difference between this form and the other form is that the `do` form insures that the BLOCK will execute at least once.

`next` and `last`

Perl provides ways to **break out** of loops: to skip over parts of a BLOCK or to end the loop prematurely.

```
#! /usr/bin/perl

use strict;
use warnings;

# first, next
for my $i ( 0 .. 9 ) {
  next if $i == 5;
  print "$i\n";
}

# two new lines, to space things out
print "\n\n";

# then, last
for my $j ( 0 .. 9 ) {
  last if $j == 3;
  print "$j\n";
}
```

```
   $ ./nextlast.pl
    0
    1
    2
    3
    4
    6
    7
    8
    9


    0
    1
    2
```

`next` and `last` in nested loops

What if you want to use `next` or `last` in a nested loop? Which loop is affected? Let's test:

```
#! /usr/bin/perl

use warnings;
use strict;

foreach my $out ( 0 .. 2 ) {
  foreach my $in ( 0 .. 2 ) {
    next if $in == 1;
    print "OUT: $out\tIN: $in\n";
  }
}
```

```
   $ ./nestloop.pl
    OUT: 0  IN: 0
    OUT: 0  IN: 2
    OUT: 1  IN: 0
    OUT: 1  IN: 2
    OUT: 2  IN: 0
    OUT: 2  IN: 2
```

`next` and `last` apply to the innermost loop.

Loop Labels

But what if you **want** to skip in the outer loop? Use a **loop label** and tell the `next` which loop to skip:

```
#! /usr/bin/perl

use warnings;
use strict;

OUTER: foreach my $out ( 0 .. 2 ) {
 INNER: foreach my $in ( 0 .. 2 ) {
    next OUTER if $in == 1;
    print "OUT: $out\tIN: $in\n";
  }
}
```

```
   $ ./nestloop2.pl
    OUT: 0  IN: 0
    OUT: 1  IN: 0
    OUT: 2  IN: 0
```

`rand`om Things

Perl has a special function you can use to generate a random number: `rand`. `rand` returns a value greater than or equal to 0 and less than 1, or, if you give it an argument, greater than or equal to 0 and less than value of the argument.

To simulate rolling a die, you would do something like this:

```
$result = int( rand 6 ) + 1
```

(What do you think `int` is doing here? What happens if you just use the result of `rand` directly?)

This is also useful for picking a random item from a list that you have stored in an array:

```
my @list = ( 'coffee' , 'tea' , 'juice' );

my $choice = int( rand $#list + 1 );

print "I'll have $list[$choice], thanks.\n";
```

Or, another way:

```
my @list = ( 'coffee' , 'tea' , 'juice' );

my $choice = int( rand @list );

print "I'll have $list[$choice], thanks.\n";
```

`print`ing Stuff Out

We've been using Perl's basic **output** operator all day: `print`. What I haven't told you is that print is actually slightly more complicated:

```
print FILEHANDLE EXPR;
print EXPR;  # same as print STDOUT EXPR
```

All Unix processes get three **filehandles** automatically opened for them when they start up:

- **standard output** (`STDOUT`)
- **standard error** (`STDERR`)
- **standard input** (`STDIN`)

When you don't give `print` a filehandle, it defaults to `STDOUT`.

`print`ing Stuff Out To Files

Writing to a file (instead of standard output) is easy: you just give `print` the name of another filehandle to use. But how do you associate the filehandle with a file? With the `open` function:

```
open( FILEHANDLE , MODE_OPERATOR , FILENAME )
```

`open` is our first example of a **syscall** -- a Perl function that interacts with the underlying operating system. You need to be aware of syscalls, because they might

fail. Your code is responsible for handling the consequences of a failed syscall. Generally, all you can do is issue an error message and exit, which you can do with the `die()` keyword:

```
 open( FILEHANDLE , MODE_OPERATOR , FILENAME ) or die( "Can't open FILENAME ($!)" )
```

Note: we'll talk more about `die()` later. For now, just know that `die()` causes the program to immediately stop executing and to print out whatever message you gave it to the standard error filehandle.

Note 2: That `$!` is a special Perl variable, not unlike the special `$?` variable in the shell. In Perl, if there's an error associated with a syscall, any human readable error message from the operating system will be placed into that `$!` variable -- so it's a good idea to print it out, as that will be the best indication of just what went wrong.

Actually `print`ing Stuff To A File

Assuming that the syscall was successful, writing to the file is easy: just use `print` with your shiny new filehandle:

```
#! /usr/bin/perl

 use warnings;
 use strict;

 my $file = "./output";
 open( my $OUT , '>' , $file ) or die( "Can't open $file ($!)" );
 foreach my $i ( 1 .. 5 ) {
   print $OUT "$i\n";
 }
 close $OUT;
```

```
    $ ./print.pl

     $ cat output
     1
    2
    3
    4
    5
```

Note in the example code that we're using a lexically-scoped variable (i.e., a "my" variable) to hold the file handle. If you have to use or maintain legacy code, you may see older versions, so-called "2 argument", forms of this command, where the file mode (i.e., '>' or '<') is concatentated together with the file name. This is considered a bad practice for a variety of reasons, and any new code should be written with the "3 argument" form that the above example demonstrates.

If you currently and may in the future be in the position of having to maintain such legacy code, you're encouraged to carefully read `perldoc -f open` (but note that `perldoc perlopentut`, which is recommended in the former, is rather out of date with regard to current best practices).

Reading Input

Reading input from **standard input** is simple: you use the angle-bracket input operators, which read a line at a time:

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

 use warnings;
 use strict;

 print "Give me a number between 1 and 5: ";
 my $input = <STDIN>;

 if(( $input >= 1 ) and ( $input <= 5 )) {
   print "\nThanks!\n";
 }
 else { print "\nYou don't follow directions very well!\n"; }
```

```
    $ ./input.pl
     Give me a number between 1 and 5: 1

     Thanks!
     $ ./input.pl
     Give me a number between 1 and 5: 9

     You don't follow directions very well!
```

Reading Input From Files

What if you want to read in from a file instead? Than you need to obtain a **filehandle**, and use the angle-bracket input operators on that filehandle instead of `STDIN`:

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

 use warnings;
 use strict;

 my $file = "./output";
 open( my $IN , '<' , $file ) or die ( "Can't open $file ($!)" );
 while ( my $line = <$IN> ) {
   chomp $line;
   print $line;
 }
 close $IN;
```

```
   $ ./input2.pl
    12345
```

Input Shortcuts

Shortcut #1: You don't have to use the '<' when you `open` a file to read; that's the assumed default action.

Shortcut #2: If you don't assign the value to a variable, the angle bracket operator puts its return value in the special `$_` variable.

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

 use warnings;
 use strict;

 my $file = "./output";
 open( my $IN , $file ) or die ( "Can't open $file ($!)" );
 while ( <$IN> ) {
   chomp;
   print;
 }
 close $IN;
```

```
    $ ./input3.pl
     12345
```

Note that we're going to see quite a bit of that `$_` variable -- Perl uses it effectively as a generic pronoun in quite a few places. Actually, since it's the default argument for funtions such as `chomp` and `print` you **won't** be seeing it, but it will be getting used.

Think of it like saying "it", where the antecedent for that pronoun is usually something that was produced by the context at hand.

Shortcut #3: If you assign the return value of the angle bracket operator to an array variable, the whole file gets read into that array, one line per array element. (This is frequently called '**slurping the file**'.)

Shortcut #4: You don't have to use a variable for the file handle; you can use a **bareword**. Again, this is something that's considered not a current best practice, and should not be done when writing new code -- but you may see it if you have to maintain legacy code. These bareword filehandles should be avoided because they're globally scoped, and can lead to obscure difficult to track down bugs in cases where the same filehandle gets reused in a nested fashion. (This is rare but exceedingly hard to figure out; using variables to hold your filehandles avoids this issue.)

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

```
use warnings;
use strict;

my $file = "./output";
open( IN , $file ) or die ( "Can't open $file ($!)" );
my @lines = <IN>;
close (IN);

print foreach @lines;
```

```
   $ ./input4.pl
    1
    2
    3
    4
    5
```

Modern Perl

If you're able to use a more recent version of Perl, there are some newer features that can make your life easier.

`say` : the `say()` function is like a `print()` that automatically adds a newline at the end. In order to use it you need to have `use feature 'say';` at the top of your program.

`autodie` : this is a convenience pragma that will handle throwing exceptions for you when common operations file. If you put `use autodie;` at the top of your program, you don't have to check the return value of `open()` calls -- they will automagically throw exceptions -- that is, exit with an error -- if and when they fail.

File::Slurp : this module will make it easier to read in a whole file at once. You can read the documentation with `perldoc File::Slurp`.

Modern::Perl : this convenient module turns on all the "newer" Perl features (`say`, in addition to others) and is also equivalent to have `use strict;` and `use warnings;` at the top of your program.

Depending on how old the version of Perl is on your system, you may or may not have access to these newer bits of Perl. They will work on the class server as long as you're using `/opt/perl/bin/perl` as your Perl interpreter.

Exercises

Log on to the server and practice writing some simple Perl scripts that set variables, use conditionals, and use looping constructs.

Does a `next` or `last` work inside a `for` loop? How about in a `while` loop?

Write a small script that prints the even numbers between 1 and 10. Are there other ways you could have accomplished the same goal?

Practice working with the basic I/O functions. Write out some files. Read them back in.

There's (at least) one more way the final example in the previous slide can be made shorter. Can you find it?

What happens if you `close` one of the standard file handles and then try to use it? Can you think of any instances where you'd *want* this behavior?

Table of Contents

•