PERL & SCRIPTING

Scripts are similar to programs. They tell the computer to perform a series of tasks. The main difference between scripts and programs is that programs are compiled into binaries (machine language) and scripts are interpreted line by line by the shell on the fly. Even though scripts are interpreted line by line, most shells or interpreters will first do several passes through a script checking syntax before actually running the script.

In most other ways, scripts look a lot like programs. They contain many of the same conventions and syntax as programming languages.

Different shells have their own scripting languages with slight syntactical differences. However, there are common scripting languages, such as Perl and Python, which can be utilized on most unix systems. These scripting languages are more powerful than those supported by shells.

WHAT IS PERL?

PERL (Practical Extraction and Reporting Language) was originally designed and used to track system resources across networks. However, the addition of web design modules, originally part of the reporting portion of the language, made PERL a perfect scripting language for dynamically generated web content.

PERL is an interpreted language. This means that PERL scripts are ASCII, human-readable files, and that a special program, the interpreter, actually reads and does what the script tells it to. This is similar to the way a shell interprets scripts.

Being an interpreted language, and having powerful text manipulation capabilities, made PERL a perfect alternative as a scripting language. Nowadays, PERL is installed on most unix systems and is available for other OS such as Windows.

COMMONALITIES OF SCRIPTING LANGUAGES

All scripting languages have a few elements in common:

variables
operators
conditionals
loops

Most everything done in a script involves one or more of these common elements.

However, unlike many other scripting languages, instructions in PERL are always terminated with a semicolon.

SETTING UP A PERL SCRIPT

Basically, all PERL scripts are ASCII text files that begin with a line that tells the shell that this file is a PERL script and should be interpreted by the PERL interpreter. That line is

```
#!<path to PERL interpreter>
```

The location of the PERL interpreter can change from system to system, so this line depends on its location. The PERL interpreter is often in the common bin directory so a typical first line would be

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

PERL scripts can have any extension, like other unix files, but to make them easy for other applications to recognize, they are usually given the extension .pl.

RUNNING A PERL SCRIPT

If you have set up your PERL script with executable permission, you can run it in two ways.

The first is to execute the script as though it were any other program.

```
➢./myscript.pl
```

The other way to run the script is to invoke the PERL interpreter on the file. To do this, type <path of PERL interpreter> <script>.

```
➢/usr/bin/perl myscript.pl
```

VARIABLES

Variables in scripting languages work much the same way as they do in programming languages. They are tokens that are meant to hold a value of some kind (ie integers, floats, strings).

Variables can have any kind of name except for keywords that are used in the PERL language (ie you cannot have a variable called if or for). To get or set the value of a variable, you preface the name with $. (ie the variable color would be referenced as $color). This is similar to most scripting languages.

Unlike most other languages, scripting or programming, in PERL variables are typed by context. In other languages, typically you have to declare the type of a variable and can only use it in contexts that make sense for that type (ie strings cannot be added like integers). However, in PERL, the interpreter considers the context of the variable and determines how it should treat the value.

VARIABLES

For example:

```
$color = 4;
print $color,"\n";
$color = $color + 5;
print $color,"\n";
$color = "$color"."5";
print $color,"\n";
```

```
➢./myscript.pl
# 4
# 9
# 95
```

VARIABLES

In general, PERL will interpret quoted values as strings, and numbers as integers unless the numbers have a decimal (in which case they will be treated as floats).

However, you have to be careful when using variables within quotes in PERL. Double quotes instruct PERL to substitute the value of the variable for the variable name within quotes. Single quotes instruct PERL to treat the variable name as a string.

```
$foo = 5;
print "Give me $foo dollars!\n";
print 'Give me $foo dollars!\n';
```

```
➢/usr/bin/perl myscript.pl
# Give me 5 dollars!
# Give me $foo dollars!\n
```

OPERATORS

Operators allow you to perform actions on variables. The various types are:

ASSIGNMENT	=, .=, +=, -=, =~
MATHEMATICAL	+, -, /, *, %
LOGICAL	<, <=, ==, >=, >, !, !=, and, or, not, eq, ne, =~

CONDITIONALS

Most languages have the notion of conditionals. A conditional is a language structure that allows you to choose a particular outcome or set of instructions based on a certain condition. The most familiar type of conditional is the if-then-else. PERL supports this structure in the following way:

```
if    (<condition>) {   ...instructions...  }
elsif (<condition>) {   ...instructions...  }
else                {   ...instructions...  }
```

The elsif and else blocks are optional and are used to create series of conditions. It is possible to have only the if (<condition>) and following instruction block. There can be as many elsif blocks as you need (or none if you have only two conditions). There is only one else block and it is always the last condition in a series.

CONDITIONALS

Conditions always have either the value of true or the value of false. In PERL, anything that evaluates to 0 is false. Everything else is considered true. The logical operators yield values that are either true or false, and so you will usually create conditions by using the logical operators. However, by noting how other values are evaluated as true or false, it is possible to have just a variable be the condition. Here are a couple examples:

```
$day = 4;
if ($day > 5)  # this evaluates false
if ($day)      # this evaluates true
```

LOOPS

Most languages also have the notion of looping, or performing some action over and over. In some cases, the purpose of looping is to perform a given action over a set of items. This is achieved in PERL by using the foreach loop. The syntax of the foreach loop is

```
foreach <variable> (<array or list>) {  ...instructions...  }
```

The foreach loops cycles over all the elements of the array or list one at a time, assigning the element to <variable> and performing the instructions given in the instruction block. For example, this will print statements about each day in the array of days:

```
foreach $date (@days)  {  print “Today is $date\n”;  }
```

LOOPS

In other cases, looping is used to perform an action over and over until some condition becomes false. This is done by using the while loop. The syntax for the while loop is

```
while (<condition>) { ...instructions... }
```

Be careful constructing conditions for while loops. If the condition can never be changed by the loop, the while loop will never stop running and the program will hang.

LOOPS

Here are two examples. One works, while the other does not.

```
$i = 0;
while ($i < 5)
{
  $i += 1;
  print $i;    # this will print 1 2 3 4 5 and exit the loop
}

$i = 0;
while ($i < 5)
{
  $i -= 1;
  print $i;    # this will print -1 -2 -3 -4 ... and never stop printing
}
```

ARRAYS

Sometimes, as with other programming languages, we would like to work with collections of data. One way to organize and work with a collection of data is to put it into an array. Then we can use the indices of the array to get at the data. Indices for arrays are always whole numbers.

In PERL, array variables are referenced with @ instead of $. Individual elements of the array are referenced with $<array>[index]. Elements of an array can be any type. Unlike other languages, they can even be different types within the same array.

The advantage to using arrays is that a single variable name can represent a data set and that the data set can be easily looped over using foreach, or a variable for the array index.

ARRAYS

There are many ways to create arrays, but the two most common are assignment. The first way is to declare all the elements of the array at once.

```
@<array> = (<list>);
```

In this form, <array> is the variable name and <list> is the data in comma delimited format (ie `@myArray = (1, 2, 3, 4);`). Another way to create an array is simply to add an element to the array.

```
$<array>[index] = <value>;
```

This will set the array element at index equal to <value> and create the array if it does not already exist. Coincidentally, $<array>[index] is also the way you reference that particular element. It acts exactly the same as a regular variable.

ARRAYS

Besides the foreach and while loops mentioned previously, which provide a convenient way to work with arrays, there are four commands that also help you to get at elements of an array; push, pop, shift, and unshift.

Push will add an element to the end of an array. Pop will take a value off the end of an array. The syntax for push and pop are

```
push(@<array>, $value);
$value = pop(@<array>);
```

Unshift and shift work just like push and pop but instead of adding/removing elements to the end of the array, they work with the beginning of the array. Here's the syntax

```
unshift(@<array>, $value);
$value = $shift(@<array>);
```

ARRAYS

Here is an example:

```
# Create an array by setting a value in it
$myArray[0] = "/home";

foreach $file (`ls $myArray[0]`)
{
  # Put each file listed from /home onto myArray
  push(@myArray, $file);
}
foreach $file (@myArray)
{
  print "$file\n";
}
```

HASHES

Another type of data set represented in PERL is the hash. A hash is an array where the indices can be any type or value. % is used to reference a hash variable (just as @ is used with arrays). Hash elements can also be referenced just as array elements although the index (referred to as a key) is enclosed in {} brackets instead of [] brackets.

```
%<hash> = ();                # creates an empty hash with
                             # name <hash>

$<hash>{<key>} = <value>;    # adds the <value> to <key> in
                             # <hash>.
```

It is important to note that each key can only have one value. It is also important to note that PERL does not necessarily store hashes in key order, so when looping over hashes they may appear in a different order than created.

HASHES

Like arrays there are two common ways to create a hash, by adding values or declaring the hash at once. Both ways are similar to creating arrays.

To declare a hash use %<hash> = (<list>); where <hash> is the name of the hash and <list> is all the key/value pairs in comma delimited format. For example,

```
%newhash = ( "one", "two", "three", "four" );
print $newhash{"one"};      # two
print $newhash{"three"};    # four
```

Because delimiting the key value pairs themselves with commas becomes confusing, programmers often use => which PERL treats as a comma.

```
%newhash = ( "one" => "two", "three" => "four" );
```

HASHES

As mentioned, keys can be any type or value. It is important to note that if you create a key that is a string, you must reference it in the same manner (using the appropriate quotes, etc) otherwise the hash will not return a value.

Although there is no push/pop sorts of functions for hashes, you can add key/value pairs through assignment as already mentioned. You can delete elements from a hash using the delete function.

```
delete($<hash>{<key>});
```

As to looping, you can use foreach and while on a hash. In the case of while, you can use the each function to get key/value pairs.

```
while (($<key>, $<value>) = each(%<hash>)) { ... }
```

This loop structure will return each key/value pair in $<key> and $<value>.

HASHES

Foreach loops are handled a little differently. In a foreach loop you can only get either the keys or the values. To get the keys use

```
foreach $<key> (keys %<hash>) { ... }
```

This will loop over the keys storing them in $<key>.

To get the values use

```
foreach $<value> (values %<hash>) { ... }
```

This will loop over the values storing them in $<value>.

Keep in mind that the order may not be the same as the way elements were added/declared in the hash.

HASHES

Here is an example.

```
%air_temp = ("Vancouver" => 12, Seattle => "11", "Hope" => 5);
# Print the locations on the same line separated by tabs
foreach $location (keys %air_temp)
{
   print "$location\t";
}
# Print a newline for formatting
print "\n";
# Print the values on the next line tabbed under the locations
foreach $temp (values %air_temp)
{
   print "$temp\t";
}
print "\n";
```

FILE HANDLING

PERL has many functions for handling files. We will cover basic file opening, closing, reading, and writing.

To open a file, you use the command

open(<filehandle>, "<ctl><filename>");

<filehandle> is a special name that PERL uses to refer to the file. This is not a normal variable and is not prefaced by $. It can be any string (other than PERL commands). Most programmers who are opening one file at a time use the name FILEHANDLE.

<ctl> is a special character that tells PERL if the file is opened for reading or writing, or both. < specifies the file is read only. > specifies the file is write only. +< specifies the file is both read and write. >> appends data to the file (> deletes the file contents on opening).

FILE HANDLING

To close a file, use

close(<filehandle>);

To write to a file, use the print command.

print <filehandle> <data>;

<data> is whatever you want to write to the file (string, number, etc). If you do not include <filehandle>, PERL will print to the screen.

There are many ways to read from a file, but the easiest to understand is to use a while loop to read lines out of the file one at a time. while (<<filehandle>>) will loop over the lines of the file, storing the data in each line in $_.

FILE HANDLING

This will read the comments out of a C file and write them to a file.

```
$file = “myCfile.c”;
open(FILEHANDLE, “<$file”);
while (<FILEHANDLE>)
{
  if ($_ =~ m/.*\/\*.*\*\//) {  push(@comments, $_);  }
}
close(FILEHANDLE);
$file = “comments”;
open(FILEHANDLE, “>$file”);
foreach $comment (@comments)
{
  print FILEHANDLE “$comment\n”;
}
close(FILEHANDLE);
```

DATA PROCESSING

In addition to all we have covered, PERL has many modules and built in functions that can help you perform tasks. Here are a few of the more commonly used ones.

chomp(X)	This removes a newline from the end of string X.
exp(X)	Returns e to the power of X.
log(X)	Returns the natural logarithm of X.
sqrt(X)	Returns the square root of X.
int(X)	Truncates X to an integer.
sin(X)	Returns the sine of X (X must be in radians).
cos(X)	Returns the cosine of X (X must be in radians).

DATA PROCESSING

defined($X)	Returns true if $X is a variable with a value.
undef($X)	Undefines the variable $X.
length(X)	Returns the number of characters in string X.
join("X", Y)	Form a string from the elements of array Y using X as the delimiter between each element.
split(/X/, Y)	Form an array from parts of the string Y that are separated by the pattern X.
substr(X,Y,Z)	Return a substring of string X. Start the substring at offset Y from the beginning of X (if Y is negative, the offset is counted backwards from the end of X). Z, optional, is the length of the substring.

STANDARD INPUT

In addition to reading data from a file, it is possible in PERL to read data from the command line. The way to do this is very similar to the manner of reading lines from a file.

```
while (<>) { … }
```

If we leave out the filehandle, PERL will read a line from the command line (terminated by a newline) into $_.

EXECUTING UNIX COMMANDS

Sometimes it is useful to execute a unix command within PERL script. There are two ways to do this.

If you merely want to execute the command without worrying about the result, you print the command in backticks (`);

```
print `<command>`;
```

This can be useful for executing unattended ftp or executing unix commands on files. The backtick is located in the upper left of the keyboard. If you want to retain the result of the unix command, use;

```
$<variable> = `<command>`;
```

This would be useful for retrieving the date with the unix date command, for example.

SUBROUTINES

Like other programming languages, you can create functions in PERL that can be called by your script or other scripts. In PERL, these functions are called subroutines.

To create a subroutine, you use the keyword sub, the subroutine name, and encapsulate it in {} braces like this...

```
sub test {
...
}
```

To call a subroutine, you type its name followed by any arguments in a comma delimited list in parentheses.

```
test(arg1, arg2, arg3);
```

SUBROUTINES

Notice that in the declaration of the subroutine we do not declare the number of arguments. The arguments supplied to a subroutine are completely variable. The user can supply as many or as little as she/he wishes. This means that the programmer needs to make the subroutine capable of handling too few arguments (usually extra arguments are just ignored).

Arguments are passed in through the @_ array. You can get at the arguments by accessing the array values directly (ie $_[0]) or by looping over the array with foreach.

This flexibility allows you to easily add more arguments without having to push them into an array or other data structure. However, it also means that you might not get enough arguments. To set default values for arguments, you can use the ||= assignment operator.

SUBROUTINES

For example...

```
($arg1, $arg2) = @_;
$arg1 ||= 1;
$arg2 ||= 2;
```

will attempt to load the first two values in the array into $arg1 and $arg2. If $arg1 isn’t set, then the next line will set it to 1. The following line takes care of $arg2 if it is not set.

Subroutines can return one value. To do this, use the return statement.

```
return $result;
```

If you attempt to return multiple values, they will be flattened into an array.

MODULES

If you want to reuse subroutines in other scripts, you will need a way to get at them. You can place them in a special file called a module and then import them into your script when you want to use them.

Modules are like scripts (typically suffixed with .pm). They begin with the typical #!/usr/bin/perl line (or appropriate location). Then you need a line for the declared module name (this is how you will recognize the module later and is typically the filename minus .pm. The typical line looks like...

```
package <module>;
```

MODULES

After this you will need a bit of boilerplate to let any scripts know how to import your module. It looks like this...

```
BEGIN {
        use Exporter();
        @ISA = qw(Exporter);
        @EXPORT = qw(&<subroutine1> &<subroutine2> ...);
}
```

The list of subroutines in the EXPORT array are the ones you are making available in the module. Each subroutine must be prefaced by a '&' and separated by a space.

After this, you can add the code for each subroutine declared in the usual fashion.

MODULES

To finish the module, you need to add a bit more boilerplate.

```
return 1;
END { }
```

Now to use the module in your script, simply add the ‘use’ line before calling the subroutines (typically at the top of the script).

```
use MyModule;
```

You do not add the .pm to the ‘use’ line. If the module is not in your path or your running directory, you can tell the script where to look for it by adding the following line prior to your module.

```
use lib <location of MyModule>;
use MyModule;
```

REGULAR EXPRESSIONS

Regular expressions are a powerful tool for matching strings. You can use regular expressions to determine if a string fits a particular pattern or to isolate parts of a string to use as variables, or even replace portions of a string.

There are three common operators for regular expressions in PERL, m//, s///, and tr///. m// is used to match strings to patterns (and to isolate parts of a string for use as variables). s/// is used to replace substrings with other substrings (ie replace the word young with old). tr/// is used to translate strings (ie replace all instances of a given character with another character).

The way to invoke a regular expression is

```
$<string> =~ <operator>
```

REGULAR EXPRESSIONS

The m// operator will return a true or false value, so it is often used in conditionals. The s/// and tr/// operate on strings and will change $<string>. Here are a couple examples of using s/// and tr///.

```
$newstring = “Hi my name is Tom”;
$newstring =~ tr/o/i/;
print $newstring   # Hi my name is Tim
$newstring =~ s/name/surname/;
print $newstring   # Hi my surname is Tim
```

If s/// or tr/// cannot find a match in the string for the given substring or character, it will make no changes. This can be a particularly sticky issue with s///, for the substring must match exactly for a change to be made. (ie Young is not the same as young).

REGULAR EXPRESSIONS

To use m// properly, you need to understand how to develop patterns. A pattern can be a pure string or a collection of symbols to describe strings. Patterns, even pure strings, in regular expressions do not need to be in quotations (in fact the quotation would be treated as part of the pattern). To match a pure string just put in the string exactly as it should appear between the slashes.

To make more complex patterns, we need some special symbols.

^	is used to represent the beginning of a line. Using this symbol forces a match that begins at the start of a line.
$	is used to represent the end of a line. Again this will force a match to lines that end a certain way.
.	is used to match any character other than a newline.
*	must follow another character. It specifies that any number of the character it follows may appear.

REGULAR EXPRESSIONS

+	must follow another character. It specifies that one or more of the previous character must appear in the pattern.
\s	stands for a blank space.
\n	stands for a newline.
\d	stands for any digit.
\D	stands for any non-digit character.

If you want *, \, /, or + to appear in a regular expression, you must escape it by putting a backslash in front of it first, otherwise it will be treated as a special character. (ie m/t*/ would match any number of t's while m/t*/ would match t*)

REGULAR EXPRESSIONS

In addition to these special characters, you can specify ranges of characters using []. For example, we can set up a regular expression to match Tom or Tim followed by a space and a last name made of any set of characters.

m/T[oi]m\s[A-Za-z]*/

It is important to note that patterns are case sensitive. You can make a pattern case insensitive by adding i to the end of the match (ie m//i).

You can also specify a number of characters to appear by using {}. For example a{3} would mean match aaa, and a{0,3} would mean match anywhere from 0 to 3 a's.

REGULAR EXPRESSIONS

Let's consider some examples:

m/^Hi\smy\sname\sis/	matches any line starting with "Hi my name is"
m/hi\smy\sname\sis/i	matches any line containing the words "hi my name is" in any case
m/Tom$/	matches any line ending with "Tom"
m/[a-z]/	matches any line containing a lower case letter
m/[a-zA-Z]{3,6}/	matches any line containing 3 to 6 letters (either case)
m/Hi\smy.*is\sTom/	matches any line with "Hi my" and "is Tom"
m/^200[5-7]\d{4}/	matches any line beginning with "200" followed by 5, 6, or 7 and 4 digits. (ie 20060708)

REGULAR EXPRESSIONS

There is one last thing you can do with m//, and that is to select portions of a string for use as variables. The way you do this is to enclose portions of a pattern in parentheses. Each set of parentheses is assigned to a number variable corresponding to the occurrence of the parentheses in the pattern. (ie $1 for first set, $2 for second set, etc). For example;

```
m/^([a-z]*)\s+([0-9]*)/
```

would assign any set of lowercase letters at the beginning of a line to $1 and any set of numbers from 0 to 9 to $2.

There are many more ways to make complex patterns in PERL than we have covered, but even with these basic tools you can create a pattern to match any line and pull portions of data from a matching line.

REGULAR EXPRESSIONS

Consider the following examples:

```
$line = "20060718   Rel Hum 86%  Air Temp 25  Wind Spd 13";
$line2 = "NO DATA";
if ($line =~ m/^200[5-7]\d{4}.*(\d{2})%.*(\d{2}).*(\d{2})$/)
# This would return true and would set $1 = 86, $2 = 25, $3 = 13

if ($line2 =~ m/^200[5-7]\d{4}.*(\d{2})%.*(\d{2}).*(\d{2})$/)
# This would return false (not match)

if ($line =~ m/no\sdata/i)              # This would return false

if ($line2 =~ m/no\sdata/i)             # This would return true

if ($line =~ m/\d*\s*([a-zA-Z\s]*)[\d\s]*%*([a-zA-Z\s]*)[\d\s]*[a-zA-Z\s]*(\d*)/)
# This would return true and set $1 = "Rel Hum", $2 = "Air Temp" and $3 = 13

if ($line2 =~ m/\d*\s*([a-zA-Z\s]*)[\d\s]*%*([a-zA-Z\s]*)[\d\s]*[a-zA-Z\s]*(\d*)/)
# This would return true and set $1 = "NO DATA"
```

That is all we will cover in PERL, however, there is considerably more to PERL (CGI to MySQL to compilation scripting). For more information, try the O'Reilly book on PERL or *PERL Core Language* by Steven Holzner.