

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 on the fly by either the shell or an interpreter. 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.

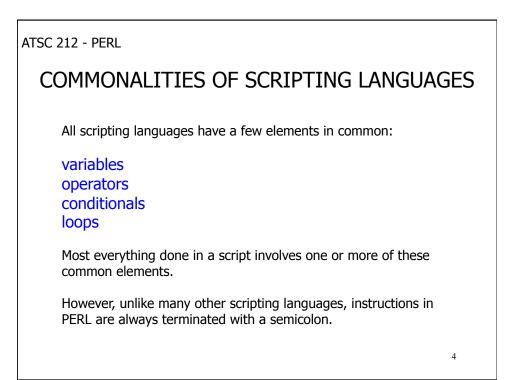
Different shells have their own scripting languages with slight syntactical differences. However, there are common scripting languages, such as Perl and Python, which can be utilized on most unix systems. These scripting languages are more powerful than those supported by shells. 2

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 perl 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 to shell scripts. Nowadays, PERL is installed on most unix systems and is available for other OS such as Windows.



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.

ATSC 212 - PERL

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

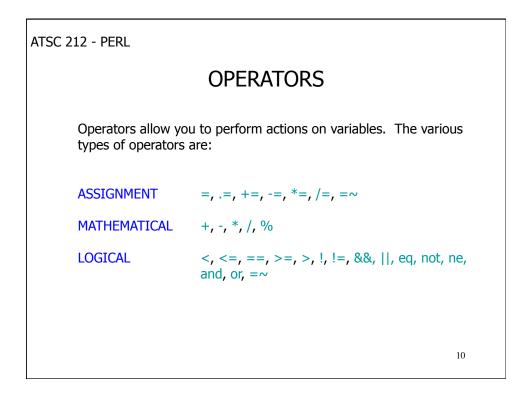
Variables in scripting languages work much the same way as they do in programming languages. They are tokens that are meant to hold a value of some kind (ie integers, floats, strings).

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.

ATSC 212 - PERL	
	VARIABLES
For example:	
<pre>\$color = 4; print \$color,"\n"; \$color = \$color + 5; print \$color,"\n"; \$color = "\$color"."5"; print \$color,"\n";</pre>	
<pre>>./myscript.pl # 4 # 9 # 95</pre>	
	8

ATSC 212 - PERL 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 guotes 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 9



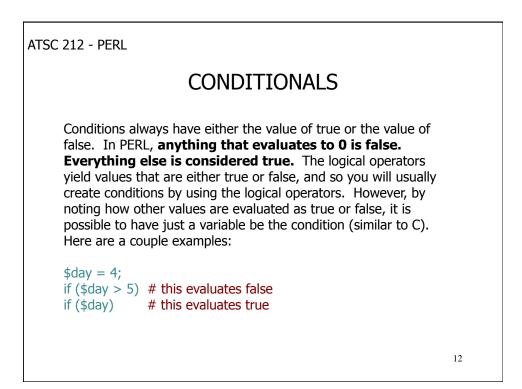
CONDITIONALS

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:

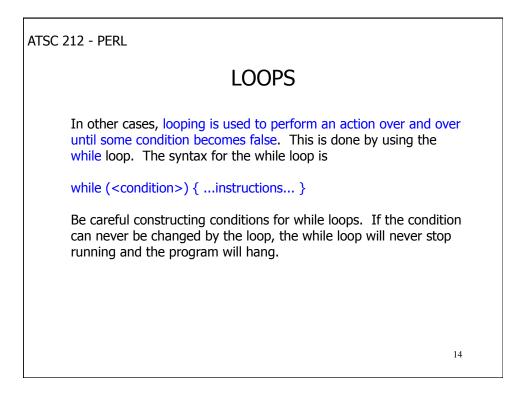
if (<condition>) { ...instructions... }
elsif (<condition>) { ...instructions... }
else { ...instructions... }

else { ...instructions... } The elsif and else blocks are optional and are used to create a

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 one condition). There is only at most one else block and it is always the last condition in a series (acts as a default condition).



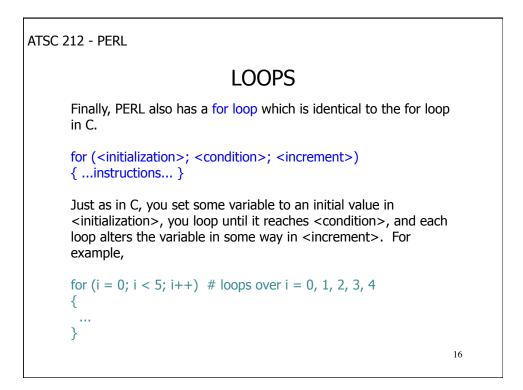
<section-header><text><text><text>



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, 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 (1) 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 for loop with a variable for the array index.

ATSC 212 - PERL		
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.		
<pre>@<array> = (<list>);</list></array></pre>		
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.</list></array>		
<pre>\$<array>[index] = <value>;</value></array></pre>		
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.</array></value>		

ATSC 212 - PERL ARRAYS Besides the for, 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); 19 \$value = \$shift(@<array>);

ATSC 212 - PERL	
ARRAYS	
Here is an example:	
<pre># Create an array by setting a value in it \$myArray[10] = "/home\n";</pre>	
foreach \$file (`ls \$myArray[10]`)	
{ # Put each file listed from /home onto myArray push(@myArray, \$file); }	
foreach \$file (@myArray)	
{ print ``\$file";	
}	
	20

HASHES

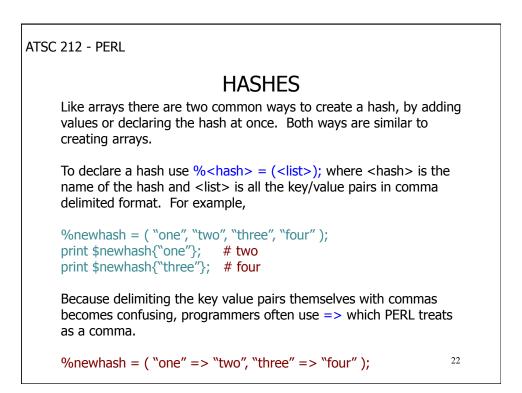
Another type of data set represented in PERL is the hash. A hash is an array where the indices can be any type or value. % is used to reference a hash variable (just as @ is used with arrays). Hash elements can also be referenced just as array elements although the index (referred to as a key) is enclosed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

```
%<hash> = ();
```

creates an empty hash with
name <hash>

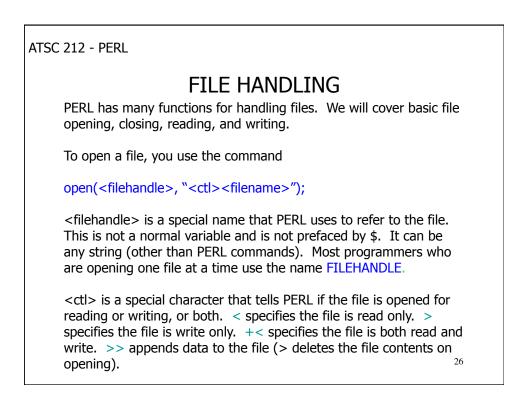
\$<hash>{<key>} = <value>; # adds the <value> to <key> in
<hash>.

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. 21

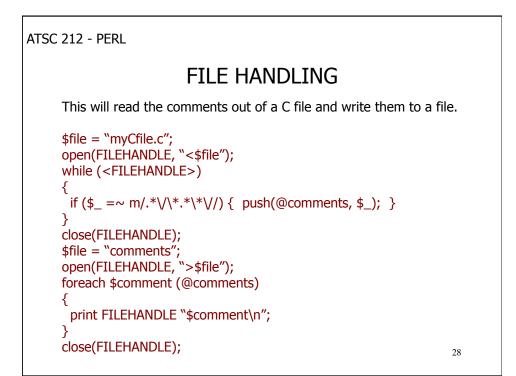


ATSC 212 - PERL HASHES As mentioned, keys can be any type or value. It is important to note that if you create a key that is a string, you must reference it in the same manner (using the appropriate quotes, etc) otherwise the hash will not return a value. Although there is no push/pop sorts of functions 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 23 \$<value>.

ATSC 212 - PERL HASPHES 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. ATSC 212 - PERL 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"; 25



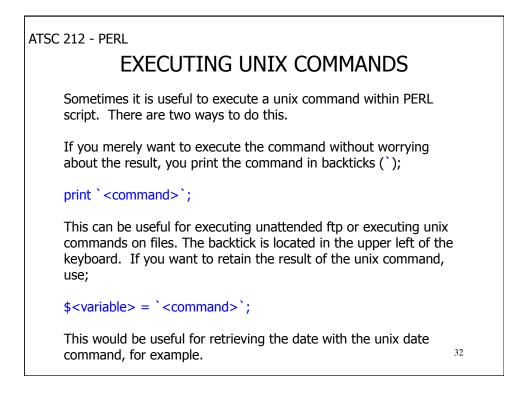
ATSC 212 - PERL FILE HANDLING To close a file, use close(<filehandle>); To write to a file, use the print command. print <filehandle> <data>; <data> is whatever you want to write to the file (string, number, etc). If you do not include <filehandle>, PERL will print to the screen. There are many ways 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 \$_.



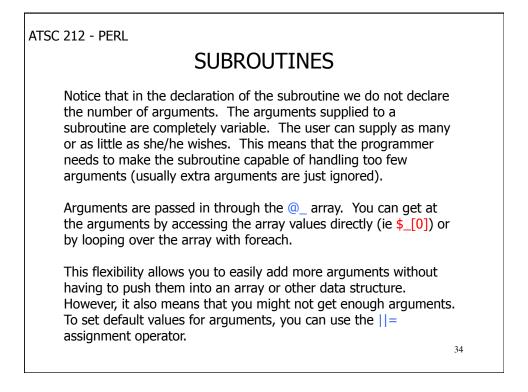
<text><text><text><text><text><text><page-footer>

ATSC 212 - PERL				
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). $_{30}$			

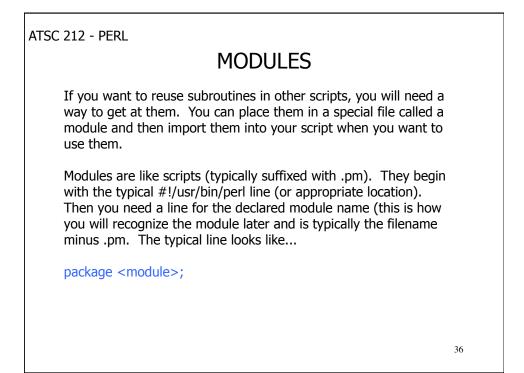
ATSC 212 - PERL	
	DATA PROCESSING
defined(\$X)	Returns true if \$X is a variable with a value.
undef(\$X)	Undefines the variable \$X.
length(X)	Returns the number of characters in string X.
join("X", Y)	Form a string from the elements of array Y using X as the delimiter between each element.
split(/X/, 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. 31



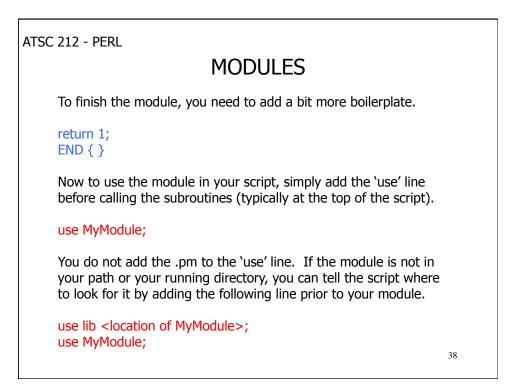
ATSC 21 2 - PERL SUBROUTINES Like other programming languages, you can create functions in PERL that can be called by your script or other scripts. In PERL, tese functions are called subroutines. To create a subroutine, you use the keyword sub, the subroutine and encapsulate it in {} braces like this... sub test { ... for call a subroutine, you type its name followed by any arguments a comma delimited list in parentheses. test(arg1, arg2, arg3);



ATSC 212 - PERL 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.



<text><text><text><code-block><text><text><text></code>



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>

39

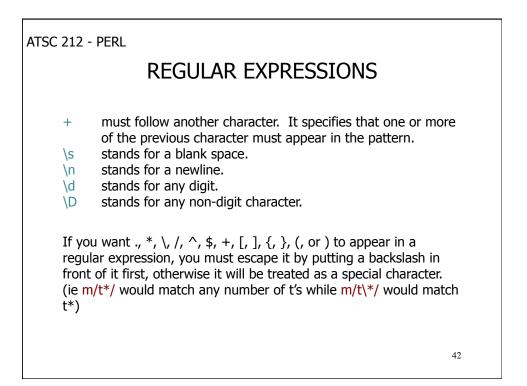
ATSC 212 - PERL **REGULAR EXPRESSIONS** The m// operator will return a true or false value, so it is often used in conditionals. The s/// and tr/// operate on strings and will change the value of a given string variable. Here are a couple examples of using s/// and tr///. \$newstring = "Hi my name is Tom"; $snewstring = \sim tr/o/i/;$ print \$newstring # Hi my name is Tim $snewstring = \sim s/name/surname/;$ 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). 40

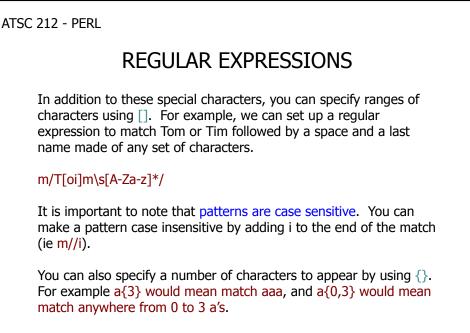
REGULAR EXPRESSIONS

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. 41





ATSC 212 - PERL		
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\$/ m/[a-z]/	matches any line ending with "Tom" 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) 44	

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.

ATSC 212 - PERL		
REGULAR I Consider the following examples:	EXPRESSIONS	
<pre>\$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</pre>		
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	
]*%*([a-zA-Z\s]*)[\d\s]*[a-zA-Z\s]*(\d*)/) `Rel Hum``, \$2 = ``Air Temp" and \$3 = 13	
<pre>if (\$line2 =~ m/\d*\s*([a-zA-Z\s]*)[\d\ # This would return true and set \$1 = `</pre>	s]*%*([a-zA-Z\s]*)[\d\s]*[a-zA-Z\s]*(\d*)/) 'NO DATA'' 46	

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.