UNIT –III: PERL

Objective:

- To be familiar with the development of Perl scripting language.

Syllabus:

Basic Syntax, Perl Language Elements: Variables,Operators, Control Flow Statements, Arrays, Hashes, Subroutines, Packages and Modules, File Handling and Operations on Files, Retrieving Document from the Web using Perl LWP.

Outcomes:

Students will be able to

- Understand the basics of Perl.
- Interpret Perl variables and data.
- Execute different statements and control structures of Perl.
- Understand the importance of subroutines in program execution.

Introduction to PERL

- Perl stands for "Practical Extraction and Reporting Language".
- It is a high-level programming language, written by Larry Wall.
- The Perl language semantics are largely based on the C programming language, while also inheriting many of the best features of **sed**, **awk**, the Unix shell and at least a dozen other tools and languages.
- Perl is particularly strong at process, file and text manipulation.
- This makes it especially useful for system utilities, software tools, systems management tasks, database access, graphical programming, networking and web programming.
- This makes it particularly attractive to CGI script authors, system administrators, mathematicians, journalists and anybody who needs to write applications and utilities very quickly.
- Perl has its roots firmly planted in the Unix environment, but it has since become a cross-platform development tool.
- Perl runs on IBM mainframes, AS/400s, Windows NT, 95 and 98.
- Perl, including the source code, the standard Perl library, the optional modules and all of the documentation, is provided free and is supported entirely by its user community.

Naming Conventions of Perl:

- The convention "Perl" signifies the language to be proper whereas "perl" signifies the language implementation.
- The current stable version of Perl is 5.26.1, released on September 22, 2017.
- However, Version 5 is standard for most of the demanding applications.

Perl Features:

Some of the most basic features are described as follows:

- **Perl is Free:**
- Perl's source code is open and free— anybody can download the C source that constitutes a Perl interpreter.
- We can easily extend the core functionality of Perl both within the realms of the interpreted language and by modifying the Perl source code.

Perl Is Simple to Learn, Concise and Easy to Read:

- Because of its history and roots, most people with any programming experience will be able to program with Perl.
- It has a syntax similar to C and shell script, among others, but with a less restrictive format.
- Most programs are quicker to write in Perl because of its use of built-in functions and a huge standard and contributed library.
- Most programs are also quicker to execute than other languages because of Perl's internal architecture.
- Perl can be easy to read, because the code can be written in a clear and concise format that almost reads like an English sentence.

Perl is Fast:

- A Perl program is compiled into a highly optimized language before it is executed, rather than compilation.
- Compared to most scripting languages, this makes execution almost as fast as compiled C code.

Perl is Extensible:

- We can write Perl-based packages and modules that extend the functionality of the language.
- The external C code can be called directly from Perl to extend the functionality.
- The Perl interpreter can be incorporated into many languages, including C.
- This allows your C programs to use the functionality of the Perl interpreter without calling an external program.

Perl Has Flexible Data Types:

- We can create simple variables that contain text or numbers and Perl will treat the variable data accordingly at the time it is used.
- This means we can embed and merge strings without requiring external functions to concatenate or combine the results.
- We can also handle arrays of values as simple lists, as typical indexed arrays and even as stacks of information.
- We can also create associative arrays which allow you to refer to the items in the array by a unique string, rather than a simple number.
- Finally, Perl also supports references and through references objects, allows you to create complex data structures made up of a combination of hashes, lists and scalars.

Perl Is Object Oriented:

- Perl supports all of the object-oriented features—inheritance, polymorphism and encapsulation.

Perl Is Collaborative:

- There is a huge network of Perl programmers worldwide.
- Most programmers supply and use the modules and scripts available via CPAN (Comprehensive Perl Archive Network).
- Using an existing prewritten module, we can save hundreds or even thousands of hours of development time.

Perl Overview:

Installing and Using Perl:

- Perl is implementable in Unix, Windows and Mac OS, almost the same way on every platform.
- We can create a text file with the help of editors like vi, emacs, kedit, Notepad, WordPad, BBEdit, Pepper etc and then use Perl to execute the statements within that file.
- Perl executes the statements directly from the raw text.

Writing a Perl Script in Windows OS:

- In order to "write" the script, we need to create a text file using any text editor.

- Once you've written the script, we can tell Perl to execute the created

text file. Syntax: C:\>perl myscript.pl

Perl under Windows:

- Originally, Perl development concentrated on providing a Windows-compatible version from the core libraries and then the development split as it became apparent that providing a lot of the built-in support for certain functions was unattainable.
- This lead to a "core" port and a separate development handled by a company called ActiveWare, which later became ActiveState and their changes were rolled back into the core release.
- Now there is only one version of the Perl interpreter that is valid on both platforms, but there are now two distributions.
- The "core" distribution is identical to that under UNIX, so it comes with the standard Perl library but not the extension set originally developed under the original ActiveWare development.

- ActiveState still provides a pre-packaged version of Perl for Windows that includes the core Perl interpreter and an extended set of modules that include the Perl Package Manager, a number of Win32-specific modules as shown below.

Module	Description
Archive::Tar	A toolkit for opening and using Unix tar files.
Compress::Zlib	An interface for decompressing information entirely within Perl.
LWP	GisleAas's Lib WWW Perl (LWP) toolkit. This includes modules for processing HTML, URLs, and MIME- encoded information, and the necessary code for downloading files by HTTP and FTP.
Win32::ChangeNotify	Interface to the NT Change/Notify system for monitoring the status of files and directories transparently.
Win32::Clipboard	Access to the global system clipboard. You can add and remove objects from the clipboard directory.
Win32::Console	Terminal control of an MSDOS or Windows NT command console.
Win32::Event	Interface to the Win32 event system for IPC.
Win32::EventLog	Interface to reading from and writing to the Windows NT event log system.
Win32::File	Allows you to access and set the attributes of a file.

Win32::FileSecurity	Interface to the extended file security options under Windows NT.
Win32::Internet	Interface to Win32's built-in Internet access system for downloading files.
Win32::IPC	Base methods for the different IPC techniques supported under Win32.
Win32::Mutex	Interface to the Mutex (Mutual/Exclusive) locking and access mechanism.
Win32::NetAdmin	Network administration functions for individual machines and entire domains.
Win32::ODBC	ODBC interface for accessing databases. See also the DBI and DBD toolkits.
Win32::Pipe	Named pipes and assorted functions.
Win32::Process	Allows you to create manageable Win32 processes within Perl.
Win32::Semaphore	Interface to the Win32 semaphores.
Win32::Service	Allows the control and access of Windows NT services.
Win32::Shortcut	Access (and modification) of Win32 shortcuts.
Win32::Sound	Allows you to play .WAV and other file formats within a Perl script.

Win32::WinError	Access to the Win32 error system.
Win32API::Net	Provides a complete interface to the underlying C++ functions for managing accounts with the NT LanManager.

- The main ActiveState Perl distribution is called ActivePerl, but they also supply a number of extras, such as the Perl Development Kit, which provides a visual package installer and debugger.

Installation:

- There are two ways of installing Perl—
- Download the ActivePerl installer from **www.activestate.com**, run the installer, and then reboot your machine:

- This will do everything required to get Perl working on the system, including installing the Perl binary, its libraries and modules and modifying the **PATH** so that we can find Perl in a DOS window or at the command prompt.
- If you are running Perl under Windows NT or Windows 2000 or using Microsoft's Personal Web Server for Windows 95/98/Me, then the installer will also set up the web server to support Perl as a scripting host for web development.
- Finally, under Windows NT and Windows 2000, the ActivePerl installer will also modify the configuration of the machine to allow Perl scripts ending in .pl to be executed directly.

- The alternative method is to compile Perl from the core distribution:
- The core distribution does not come with any of the Win32-specific modules.
- We need to download and install those modules separately.

- If we want to install a version of the Perl binary based on the latest source code, we need to find a C compiler capable of compiling the application.
- The best platforms for building from the core source code are Windows NT or Windows 2000 using the standard **cmd** shell.

Executing Scripts:

- Once installed correctly, there are two basic ways of executing a Perl script.
- You can either type C:\>perl hello.pl

in a command window, or you can double-click on a script in Windows

Explorer.

- The former method allows you to specify command line arguments.
- The latter method will require that you ask the user for any required information.

Perl Parsing Rules:

The Execution Process:

- Perl works in a similar fashion to many other script-based languages.
- It takes raw input, parses each statement and converts it into a series of opcodes, builds a suitable opcode tree and then executes the opcodes within a Perl "virtual machine."
- This technique is used by Perl, Python and Java.
- It is one of the most significant reasons why Perl is as quick as it is—the code is optimized into very small executable chunks, just like a normal program compiled for a specific processor.

Perl actually classifies only two stages—the parsing stage and the execution or run-time stage.

- Errors reported at the parsing stage relate to problems with the layout or rules of the Perl language

- Run-time errors relate to the execution of a statement.

Syntax and Parsing Rules

The Perl parser thinks about all of the following when it looks at a source line:

- **Basic syntax -** The core layout, line termination and so on.
- **Comments -** If a comment is included, ignore it.
- **Component identity -** Individual terms (variables, functions and numerical and textual constants) are identified.
- **Bare words -** Character strings that are not identified as valid terms.
- **Precedence-**Once the individual items are identified, the parser processes the statements according to precedence rules, which apply to all operators and terms.
- **Context –** The context of the statement is checked whether we expecting a list or scalar a number or a string, and so on. This actually happens during the evaluation of individual elements of a line, which is why we can nest functions such as **sort**,

  reverse, and **keys** into a single statement line.
- **Logic Syntax -** For logic operations, the parser must treat different values, whether constant- or variable-based, as true or false values.

Basic Syntax:

The basic rules are

- Lines must start with a token that does not expect a left

operand. For example,= 99; is invalid

- Lines must be terminated with a semicolon, except when it's the last line of

a block, where the semicolon can be omitted.

For example:

print "Hello\n"

is perfectly legal as a single-line script. Since it's the last line of a block, it doesn't require any semicolon, but

print "Hello "print"World\n" will cause a fault.

- Lines may be split at any point, providing the split is logically between two tokens. The following is perfectly legal:

print "hello"

,

"world";

Comments:

- Comments are treated by Perl as white space—the moment Perl sees a hash on a line outside of a quoted block, the remainder of the line is ignored.
- This is the case even within multi line statements and regular expressions.

matched = /(\S+)	#Host
\s+	#(space separator)
(\S+)	#Identifier
\s+	#(space separator)
(\S+)	#Username
\s+	#(space separator)
\[(.*)\]	#Time
\s+	#(space separator)
"(.*)"	#Request
\s+	#(space separator)
(\S+)	#Result
\s+	#(space separator)
(\S+)	#Bytes sent
/x;	

- Comments end when Perl sees a normal line termination. The following is completely invalid:

print("Hello world");

Greet the user and let them know we're here

• Comments and line directives can be a useful way of debugging and documenting your scripts and programs.

Component Identity:

• When Perl fails to identify an item as one of the predefined operators, it treats the character sequence as a "term."

• Terms are core parts of the Perl language and include variables, functions, and quotes.

• The term-recognition system uses these rules:

• Variables can start with a letter, number, or underscore, providing they follow a suitable variable character, such as **$**, **@**, or **%**.

• Variables that start with a letter or underscore can contain any further combination of letters, numbers and underscore characters.

• Variables that start with a number can only consist of further numbers—be wary of using variable names starting with digits. The variables such as **$0** through to **$9** are used for group matches in regular expressions.

• Subroutines can only start with an underscore or letter, but can then contain any combination of letters, numbers and underscore characters.

• Case is significant—**$VAR**, **$Var** and **$var** are all different variables.

• Each of the three main variable types have their own name space—**$var**, **@var**, and **%var** are all separate variables.

• File handles should use all uppercase characters—this is only a convention, not a rule, but it is useful for identification purposes.

Operators and precedence:

• Relational and Equality Operators:

The relational and equality operators enable you to test the equality of numbers and strings, respectively. The full list of relational and equality operators is given bellow **Operator** **Action**

< Returns true if the left statement is numerically less than the right statement

>	Returns true if the left statement is numerically greater than the right statement
<=	Returns true if the left statement is numerically less than or equal to the right statement
>=	Returns true if the left statement is numerically greater than or equal to the right statement
==	Returns true if the left statement is numerically equal to the right statement

	!= Returns true if the left statement is numerically not equal to the rightstatement
<=>	Returns –1, 0, or 1 depending on whether the left statement is numerically
	less than, equal to or greater than the right statement respectively.
lt	Returns true if the left statement is string-wise less than the right statement
gt	Returns true if the left statement is string-wise greater than the rightstatement
le	Returns true if the left statement is string-wise less than or equal to the right statement
ge	Returns true if the left statement is string-wise greater than or equal to theright statement
eq	Returns true if the left statement is string-wise equal to the right statement
ne	Returns true if the left statement is string-wise not equal to theright statement

cmp	Returns –1, 0, or 1 depending on whether the left statement is stringwise
	less than, equal to, or greater than the right statement, respectively

- To logically compare numerical values, you use the symbolic equality and relational operators, for example:

  if ($a > 0)

- For string comparisons, you must use the text operators: if ($a gt 'a')

- A common mistake is to use the wrong operator on the wrong type of value but fail to notice it, because for 99 percent of situations it would resolve true anyway. The statement

  if ($a == $b)

  will work fine if both values are numerical.

- If they are textual, then Perl compares the logical value of the two strings, which is always true.

- Bitwise AND, OR, and Exclusive OR:

  - The bitwise AND(**&**) returns the value of two numbers ANDed together on a bit-by- bit basis.
  - If you supply two strings, then each character of the string is ANDed together individually, and the new string is returned.
  - If you supply only one integer and one string, then the string is converted to an integer and ANDed together as for integers.

    For example:

print '123' & '456';				# Outputs 001	
print	123	&	456;	# Outputs	72
print	123	&	'456';	# Outputs	72

  - Bitwise OR(|), and Exclusive OR(^) work in the same fashion.

➔ Symbolic Logical AND:

- The Perl logical AND, **&&**, works on a short-circuit basis.
- Consider the statement

  a&& b;

- If **a** returns a false value, then **b** is never evaluated.
- The return result will be the right operand in both scalar and list context,

  such that @list = ('a','b');

  @array = ('1','2'); print(@list && @array);

$a = 'a' && 'b'; print $a; produces “12b”.

➔ Symbolic Logical OR

- The Perl logical OR, ||, works on a short-circuit basis. Consider the statement a | | b;
- If **a** returns a true value, then **b** is never evaluated. However, be wary of using it with functions that only return a true value. For example:

  select DATA || die;

  will never call **die**, even if **DATA** has not been defined.

- Conditional Operator:
  - The conditional operator is like an embedded **if...else** statement.
  - The format is

    EXPR ? IFTRUE : IFFALSE
  - If **EXPR** is true, then the **IFTRUE** expression is evaluated and returned, otherwise **IFFALSE** is evaluated and returned.
  - Scalar and list context is propagated according to the expression selected.
  - At a basic level, it means that the following expressions do what we want,

    $value = $expr ? $true :

    $false; @list = $expr ?

    @lista : @listb; while

    $count = $expr ?@lista : @listb;

    populates**$count** with the number of elements in each array.

- Assignment Operators:
  - The assignment operators assign a value to a valid lvalue expression—usually a variable, but it can be any valid lvalue.
  - An lvalue or left-hand value, is an expression to which you can assign a new value.
  - Assignment happens with the = and associated operators.
  - Valid lvalues include

- Any recognizable variable, including object properties
- **vec** function (for setting integer values)
- **substr** function (for replacement strings)
- **keys** function (for setting bucket sizes)
- **pos** function (for setting the offset within a search)
- Any lvalue-defined function (Perl 5.6 only)
- **? :** conditional operator
- Any assignment expression

The full list of assignment operators includes the following:

=	**=	+=	*=	&=	<<=	&&=
		-=	/=	|=	>>=	||=
		.=	%=	^=		
			x=			

- Note that assigning a value to an assignment expression should be read from left to right, such that

  ($a += 10) *= 5;

  reads as

  $a +=10;

  $a *= 5;

- <u>Auto-Increment and Auto-Decrement Operators:</u>
  - The auto-increment and auto-decrement operators allow you to immediately increment or decrement a value without having to perform the calculation within a separate expression.
  - This operates in the same fashion as the C equivalent and can be placed before or after a variable for the increment or decrement to take place before or after the variable has been evaluated.

  - For example:

```
$a = 1;
print ++$a; # incremented before,
outputs 2 print $a++;
# incremented after, outputs 2, $a now equals 3
print --$a; # decremented before, now outputs 2
print $a--; # decremented after, outputs 2, $a now equals 1
```

- Shift Operators:
  - The shift operators shift the bits of an expression right or left, according to the number of bits supplied.
  - For example:

```
2 << 8;
```

    is 512.

  - If a floating point value is supplied to a shift operator, it is converted to an integer without rounding, that is, it is always rounded down, such that:

```
2.9 << 7.9;
```

    produces 256, 2 shifted to the left 7 times.

Variables and Data:

Variables, a core part of any language, allow you to store dynamic values into named locations.

- Perl supports one basic variable type—the scalar.
- Scalars are used to contain a single value, although the value itself can be either a numerical or string constant or a reference to another variable.
- Two other variable types are basically variations on the scalar theme.
- The array, for example, is essentially a sequence (list) of scalar values accessible through a numerical index.
- The hash is a list of key/value pairs that allow you to access a value by name rather than the numerical index offered by arrays.
- Both the key and the value use scalars to hold their contents.

Basic Naming Rules:

- Variable names can start with a letter, a number, or an underscore, although they normally

  begin with a letter and can then be composed of any combination of letters, numbers and the underscore character.
- Variables can start with a number, but they must be entirely composed of that number; for example, **$123** is valid, but **$1var** is not.
- Variable names that start with anything other than a letter, digit or underscore are generally reserved for special use by Perl.
- Variable names are case sensitive; **$foo**, **$FOO**, and **$fOo** are all separate variables as far as Perl is concerned.
- As an unwritten rule, names all in uppercase are constants.
- All scalar values start with **$**, including those accessed from an array of hash, for example

  $array[0] or **$hash{key}**.
- All array values start with **@**, including arrays or hashes accessed in slices, for example

  @array[3..5,7,9] or **@hash{'bob', 'alice'}**.
- All hashes start with **%**.
- Namespaces are separate for each variable type—the variables **$var**, **@var**, and **%var** are all different variables in their own right.
- In situations where a variable name might get confused with other data (such as when embedded within a string), you can use braces to quote the name.
  For example, **${name}** or **%{hash}**.

Scalar Variables:

- A scalar always contains a single value, either a number, a string or a reference to another variable.
- If the variable has no value, then it is said to be "undefined" or to contain the "undefined" value.

- Although scalar values appear to contain a value of a specific type, Perl doesn't distinguish between numerical values and strings, nor does it care whether a numerical value is an integer or a floating point value.
- In fact, internally, Perl stores numbers as signed integers, or as double precision floating point values if the value contains a decimal component.
- Strings are held internally as a sequence of characters.
- There is no limit on the length of a string and there are no terminators or other characters used to "delimit" the content of the string.

- To create a scalar variable, just select a name and assign it a value:

```
$int = 123;
$float = 123.45;
$string = 'Hello world!';
```

- We can also assign a scalar an empty (undefined) value:

```
$nothing = undef;
```

- The **undef** is actually the name of a built-in function that returns the undefined value.

Literals:

- *Literals* are the raw values that you insert into your source code.
- They can be made upof the normal numerical values and strings.
- Perl also supports a number of advanced literals that enable you to store specific types of data such as version strings.

- Numeric Literals:
  - Perl supports a number of fairly basic methods for specifying a numeric literal in decimal:

$num = 123;		# integer	
$num	= 123.45;	#	floating point
$num	= 1.23e45;	#	scientific notation

- You can also specify literals in hexadecimal, octal and binary by specifying a preceding character to highlight the number types:

$num = 0xff;		# Hexadecimal	
$num	= 0377;	#	Octal
$num	= 0b0010_0000;	#	Binary

- String Literals:
  - Strings are generally surrounded by either single or double quotes.
  - The effects of the quotes are different, however, they follow the same rules as the Unix shell.
  - When using single quotes, the value of the string is exactly as it appears—no form of interpretation or evaluation is performed on the contents of the string (except for\' and \\).
  - When double quotes are used, the effects are quite different. For a start, double-quoted

    strings are subject to backslash and variable interpolation, just as they are under Unix.
  - For example, we can use double-quoted strings and the **\n** backslash sequence to add newline characters to a string.
  - Other backslash (or escape) sequences supported by Perl are listed in Table 4-1.

Code	**Meaning**
\n	Newline
\r	Carriage return
\t	Horizontal tab
\f	Form feed
\b	Backspace
\a	Alert (bell)
\e	ESC (escape) character
\XXX	Character specified in octal, where **XXX** is the character's
	ASCII code.
\xXX	Character specified in hexadecimal, where **XX** is the

	character's ASCII code.
\x{XXXX}	Character specified in hexadecimal, where **XXXX** is the
	character's Unicode number.

Backslash (Escaped) Character Sequences

- The backslash sequence is often called an escape sequence because you "escape" the normal interpretation.

For example:

$string = 'hello world';		# hello world	
$string = 'hello	world\n';	# hello world\n	
$string = "hello	world\n";	# hello world with trailing newline	
$string = "\tHello World\a\a\n"; # hello world			with preceding tab and
		# double bell,	with trailing newline

Statements and Control Structures:

- **Conditional Statements:**
  - The conditional statements are **if** and **unless** and they allow you to control the execution of your script.
  - The **if** statement operates in an identical fashion, syntactically and logically, to the English equivalent.
  - It is designed to ask a question and execute the statement or code block, if the result of the evaluated expression returns true.
  - There are five different formats for the **if**

    statement: if (EXPR)

    if (EXPR) BLOCK

    if (EXPR) BLOCK else BLOCK

    if (EXPR) BLOCK elsif (EXPR) BLOCK ...

    if (EXPR) BLOCK elsif (EXPR) BLOCK ...

    else BLOCK STATEMENT if (EXPR)

- The first format is classed as a simple statement, since it can be used at the end of another statement without requiring a block, as in

```
print "Happy Birthday!\n" if ($date == $today);
```

- The second format is the more familiar conditional statement that you may have come across in other languages:

```
if ($date == $today)
{
 print "Happy Birthday!\n";
}
```

- The third format allows for exceptions. If the expression evaluates to true, then the first block is executed; otherwise (**else**), the second block is executed:

```
if ($date == $today)
{
print "Happy Birthday!\n";
}
else{
print "Happy Unbirthday!\n";
}
```

- The fourth form allows for additional tests if the first expression does not return true. The **elsif** can be repeated infinite number of times to test as many different alternatives as are required:

```
if ($date == $today)
{
print "Happy Birthday!\n";
}
elsif ($date == $christmas)
{
print "Happy Christmas!\n";
}
```

- The fifth form allows for both additional tests and a final exception if all the other tests fail:

```
if ($date == $today)
{
print "Happy Birthday!\n";
}
elsif ($date == $christmas)
```

```
{
print "Happy Christmas!\n";
}
                                    else
{
print "Happy Unbirthday!\n";
}
```

- The **unless** statement automatically implies the logical opposite of **if**, so **unless** the**EXPR** is true, execute the block. This means that the

```
statement print "Happy Unbirthday!\n" unless ($date ==
$today);
                    is equivalent to
print "Happy Unbirthday!\n" if ($date != $today);
```

- However, if you want to make multiple tests, there is no **elsunless**, only **elsif**.
- It is more sensible to use **unless** only in situations where there is a single statement or code block; using **unless** and **else** or **elsif** only confuses the process.
- For example, the following is a less elegant solution to the preceding **if…else**example,

```
unless ($date != $today)
{
  print "Happy Unbirthday!\n";
}
else
{
  print "Happy Birthday!\n";
}
```

- The final conditional statement is actually an operator—the conditional operator.
- It is synonymous with the **if…else** conditional statement but is shorter and more compact.
- The format for the operator is

```
(expression) ? (statement if true) : (statement if false)
```

Ex: ($date == $today) ? print "Happy Birthday!\n" : print "Happy Unbirthday!\n";

- **Loops:**
  - Perl supports four main loop types and all of them should be familiar to most programmers.
  - Perl supports **while**, **until**, **for**, and **foreach** loop statements.
  - In each case, the execution of the loop continues until the evaluation of the supplied expression changes.
  - In the case of a **while** (and **for**) loop, for example, execution continues while the expression evaluates to true.
  - The **until** loop executes while the loop expression is false and only stops when the expression evaluates to a true value.
  - The list forms of **for** and **foreach** loop are special cases—they continue until the end of the supplied list is reached.
    - **While Loops:**

      The **while** loop has three forms:

      while EXPR LABEL

      while (EXPR) BLOCK LABEL

      while (EXPR) BLOCK continue BLOCK

  - The first format follows the same simple statement rule as the simple **if** statement and enables you to apply the loop control to a single line of code.
  - The expression is evaluated first and then the statement to which it applies is evaluated.
  - For example, the following line increases the value of **$linecount** as long as we continue to read lines from a given file:

    $linecount++ while (<FILE>);

Example:

```
do
{
$calc += ($fact*$ivalue);

} while $calc<100;
```

In this case, the code block is executed first, and the conditional expression is only evaluated at the end of each loop iteration.

- The second two forms of the **while** loop repeatedly execute the code block as long as the result from the conditional expression is true. For example, you could rewrite the preceding example as:

```
while($calc< 100)
{
$calc += ($fact*$ivalue);
}
```

- **until Loops:**
- The inverse of the **while** loop is the **until** loop, which evaluates the conditional expression and reiterates over the loop only when the expression returns false.
- Once the expression returns true, the loop ends.
- In the case of a **do…until** loop,the conditional expression is only evaluated at the end of the code block.
- In an **until(EXPR) BLOCK** loop, the expression is evaluated before the block executes. Using an **until** loop, you could rewrite the previous example as

```
do
{
$calc += ($fact*$ivalue);
} until $calc>= 100;
```

- **for Loops:**

- A **for** loop is basically a **while** loop with an additional expression used to revaluate the original conditional expression.
- The basic format is

```
LABEL for (EXPR; EXPR; EXPR) BLOCK
```

- The first **EXPR** is the initialization—the value of the variables before the loop starts iterating.
- The second is the expression to be executed for each iteration of the loop as a test. The third expression is executed for each iteration and should be a modifier for the loop variables.

- We can write a loop to iterate 100 times

  like this:

```
for ($i=0;$i<100;$i++)
{
----------
}
```

- We can place multiple variables into the expressions using the standard list operator (the comma):

```
for ($i=0, $j=0;$i<100;$i++,$j++)
```

- **foreach** Loops

- The last loop type is the **foreach** loop, which has a format

  like this: LABEL foreach VAR (LIST)

  BLOCK

  LABEL foreach VAR (LIST) BLOCK continue BLOCK

- This is identical to the **for** loop available within the shell.
- Imagine that you want to iterate through a list of values stored in an array, printing each value (we'll use the month list from our earlier variables example).
- Using a **for** loop, you can iterate through the list using

```
for ($index=0;$index<=@months;$index++)
{
print "$months[$index]\n";
}
```

- This is messy, because we're manually selecting the individual elements from the array and using an additional variable, **$index**, to extract the information.
- Using a **foreach** loop, we can simplify the process:

```
foreach (@months)
{
print "$_\n";
}
```

- The **foreach** loop can even be used to iterate through a hash, providing you return the list of values or keys from the hash as the list:

```
foreach $key (keys %monthstonum)
{
print "Month $monthstonum{$key} is $key\n";
}
```

Arrays: An array is a variable that stores an ordered list of scalar values. Array variables are preceded by an "at" (@) sign. To refer to a single element of an array, you will use the dollar sign ($) with the variable name followed by the index of the element in square brackets.

```
@array = (1, 2, 'Hello');
@array = qw/This is an array/;
```

When accessing individual elements from an array, you must prefix the variable with a dollar sign ($) and then append the element index within the square brackets after the name of the variable.

```
@days = qw/Mon Tue Wed Thu Fri Sat Sun/;
print "$days[0]\n";print "$days[1]\n";
```

The size of an array can be determined using the scalar context on the array - the returned value will be the number of elements in the array.

```
@array = (1,2,3);
print "Size: ",scalar @array,"\n";
```

push @ARRAY, LIST

pop @ARRAY

shift @ARRAY

unshift @ARRAY, LIST

Hashes:

A hash is a set of key/value pairs. Hash variables are preceded by a percent (%) sign. To refer to a single element of a hash, you will use the hash variable name preceded by a "$" sign and followed by the "key" associated with the value in curly brackets.

Creating Hashes

Hashes are created in one of the two following ways. In the first method, you assign a value to a named key on a one-by-one basis.

```
$data{'John Paul'} = 45;
$data{'Lisa'} = 30;
$data{'Kumar'} = 40
```

OR

```
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
```

OR

```
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
```

Extracting Keys and Values

```
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@names = keys %data;
print "$names[0]\n";print "$names[1]\n";print "$names[2]\n";

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@ages = values %data;
print "$ages[0]\n";print "$ages[1]\n";print "$ages[2]\n";
```

Getting Hash Size

```
@keys = keys %data;
$size = @keys;
print "Hash size:  is $size\n";
```

Add and Remove Elements in Hashes

```
$data{'Ali'} = 55;      // to add
delete $data{'Ali'};    // to remove
```

Subroutines:

Function:

- A function is a named code block that is generally intended to process specified input values into an output value.
- For example, the **print** function takes variables and static text and prints the values on the screen.
- We can define functions anywhere within a program, including importing them from external files or having them generated on the fly using an **eval** statement.
- Furthermore, you can generate *anonymous subroutines*, which are functions that are attached, by reference to a variable.
- This enables you to treat a subroutine as any other entity within Perl, even though you may consider it to be a fundamental part of the blocks that make up the Perl language.

Function or Subroutine?

- The two terms *function* and *subroutine* are used interchangeably in Perl.
- If you want to be strict on the semantics, small pieces of named blocks of code that accept arguments and return values are called “subroutines”.
- The built-in subroutines in Perl are usually referred to as Perl’s functions, because they provide additional functionality.
- A subroutine created as part of a class definition is called a method.
- Subroutines, like variables, can be declared (without defining what they do) or declared and defined.
- To simply declare a subroutine, you use one of the following forms:

sub NAME

sub NAME PROTO

sub NAME ATTRS

sub NAME PROTO ATTRS

- where **NAME** is the name of the subroutine you are creating, **PROTO** is the prototype for the arguments the subroutine should expect when called and **ATTRS** is a list of attributes that the subroutine exhibits.
- The **PROTO** and **ATTRS** arguments are optional.

- An undefined subroutine lets the rest of the script know that such a subroutine exists.
- If you want to declare and define a function, then you need to include the **BLOCK**

that defines its operation:

sub NAME BLOCK

sub NAME PROTO BLOCK

sub NAME ATTRS BLOCK

sub NAME PROTO ATTRS BLOCK

- You can also create *anonymous* subroutines—subroutines without a name—by omitting the **NAME** component:

sub BLOCK

sub PROTO BLOCK sub ATTRS BLOCK

sub PROTO ATTRS BLOCK

Example of subroutine:

```
sub message
{
    print "Hello!\n";
}
```

- To call this function you would use one of the following forms:

NAME

NAME LIST NAME (LIST)

&NAME

Packages:

- The main principle behind packages in Perl is to protect the name space of one section of code from another, therefore helping to prevent functions and variables from over- writing each other's values.
- If no package name is specified, then the default package name is **main**.
- You can change the current package to another by using the **package** keyword.
- The current package determines what symbol table is consulted when a user makes a function call or accesses a variable.
- The current package name is determined at compile and run time because certain operations, such as dereferencing require Perl to know what the "current" package is.
- Any **eval** blocks are also executed at run time, and the current package will directly affect the symbol table to which the **eval** block has access.
- All identifiers (except those declared with **my** or with an absolute package name) are created within the symbol table of the current package.
- The package definition re-mains either until another package definition occurs or until the block in which the package was defined terminates.
- The **package** declaration only changes the default symbol table.
- For example, in the following code, both the **add** and **subtract** functions are part of the **Basemath** package, even though the **square** function has been inserted within a **Multimath** package:

```
package Basemath;
sub add { $_[0]+$_[1] }

package Multimath;
sub square { $_[0] *= $_[0] }


package Basemath;
sub subtract { $_[0]-$_[1] }
```

- You can reference a symbol entry from any package by specifying the full package and symbol name.
- The separator between the package and symbol entry is the “double colon”.
- You could refer to the preceding **add** function as **Basemath::add**.
- If you are refering to a variable, you place the character for the variable type before the package name; for example, **$Basemath::PI**.
- The main package can either be specified directly, as in **$main::var** or you can ignore the name and simply use **$::var**.
- You can also nest package names in order to create a package hierarchy. Using the math module again, you might want to split it into three separate packages.
- The main **Math** package contains the constant definitions, and it has two nested packages - **Math::Base** and **Math::Multi**.

Package Symbol Tables

- The symbol table is the list of active symbols (functions, variables, objects) within a package.
- Each package has its own symbol table and with some exceptions, all the identifiers starting with letters or underscores are stored within the corresponding symbol table for each package.
- The symbol table for a package can be accessed as a “Hash”.
- For example, the **main** package’s symbol table can be accessed as **%main::** or, more simply, as **%::**.
- Likewise, symbol tables for other packages are **%MyMathLib::**.
- The format is hierarchical so that the symbol tables can be traversed using standard Perl code.
- The **main** symbol table includes a reference to all the other top-level symbol tables, so the preceding nested example could be accessed as **%main::Math::Base**.

- The keys of each symbol hash are the identifiers of the symbols for the specified package; the values are the corresponding “typeglob” values.
- This explains the use of a typeglob, which is really just accessing the value in the hash for the corresponding key from the symbol table.

The following code prints out the symbol table for the **main** package:

```
foreach $symname (sort keys %main::)
{
   local *symbol = $main::{$symname};
   print "\$$symname is defined\n" if defined
   $symbol; print "\@$symname is defined\n" if
   defined @symbol; print "\%$symname is
   defined\n" if defined %symbol;
}
```

Module:

- Modules are the loadable libraries of the Perl world.
- A Perl module is generally just another Perl source file that defines a number of functions and/or variables, although it can also be an interface to an external C library.
- Modules are the main way for supporting additional functionality in your Perl scripts and for properly dividing up your module into a reusable format.
- For example, we can import the **CGI** module, which supports a range of web-related functions and tools using the keyword “use”.

  Syntax: use CGI;

Creating Modules:

- At the simplest level, a module is just another name for a package that has been moved to a separate file with the same name as the package, and that has the extension .pm attached.

- Perl doesn't know how to magically import the functions defined within the module file; for that we need to use the **Exporter** module, which supplies the necessary intelligence for us.
- To explain the process, let's consider a simple module called **MyMathLib**, which is contained in the file MyMathLib.pm:

package MyMathLib;		#	Define the package (and module) name
require Exporter;		#	Import the functions required to export
		#	functions from our own module
@ISA	= qw/Exporter/; #		Set the inheritance tree so that Perl can
		#	find the function required
@EXPORT	= qw/add/;	#	Specify the functions we want to export

```
sub add                    # The function we want to export
{
$_[0]+$_[1];
}
1;                         # Modules must return a true value
```

The Exporter Module

- The **Exporter** module supplies the **import** function required by the **use** statement to import functions.
- The absolute minimum required at the top of your module is this: package ModuleName; require Exporter; @ISA = qw(Exporter);
- The package name should reflect the module's file name. Here, the module **MyGroup::MyModule** equates to a file name of MyGroup/MyModule.pm.

- The remaining statements import the **Exporter** module, and the **@ISA** array defines the inheritance—it's inheritance that allows the **import** module in **Exporter** to be inherited by the module you are creating.

Scope:

- When you create a variable, it's created within the scope of the current package.
- In the case of the **main** package, it means that you are creating a "global" variable.
- Although packages allow you to split up the global variables that you create into different sections, many programs would be difficult to work with if we had to keep giving unique names to all the variables we used.
- For that reason, Perl also allows us to create variables that are lexically scoped i.e., they are declared as existing only until the end of the innermost enclosing scope, which is either a block, a file, or an eval statement.
- Perl supports three scoping declarations that enable us to create private variables (using **my**), selectively global variables (using **our**) and temporary copies of selected global variables (using **local**).
- At the simplest level, you just prefix the variable with the declaration

  keyword: my $var;

  our $var; local $var;
- If you want to specify more than one variable, then supply the names in parentheses, my ($var, @var, %var);

- If you want to assign a value,

  my ($var, $string) = (1,'hello');

FILE HANDLING:

File handles:

- A filehandle is a named internal Perl structure that associates a physical file with a name.
- A filehandle can be reused. It is not permanently attached to a single file, nor is it permanently related to a particular file name.
- The name of the filehandle and the name of the file are not related.
- All operating systems support three basic filehandles — **STDIN**, **STDOUT**, and **STDERR**.
- The exact interpretation of the filehandle and the device or file it is associated with depends on the OS and the Perl implementation.
- All filehandles are capable of read/write access, so you can read from and update any file or device associated with a filehandle.
- However, when you associate a filehandle, you can specify the mode in which the filehandle is opened.
- This option prevents you from accidentally updating or overwriting information in a file that you only wanted to read.
- All filehandles are by default buffered on both input and output. In most cases, this helps to improve performance by reading more than is needed from the physical device, or by block writing to a physical device.
- Information is buffered on a block-by-block basis.
- The only exception to this rule is **STDOUT**, which is buffered on a line basis: appending the newline character to a printed string will automatically flush the buffer.

Perl Filehandle	**C File Descriptor**	**Associated Device**	**Access Mode**
STDIN	0	Keyboard/terminal	Write-only
STDOUT	1	Monitor/terminal	Read-only
STDERR	2	Monitor/terminal	Write-only

Standard Perl FileHandles

- It's possible to set the buffering on other files if you use the **IO::Handle** module with the **autoflush** method.
- For example, the following code turns buffering off for the **DOOR** filehandle:

```
use IO::Handle;
open(DOOR,"<file.in") or die "Couldn't open file"; autoflush
DOOR 1;
```

Note: To switch it back on again:

```
autoflush DOOR 0;
```

- A filehandle can be referred to by either a static token or an expression. If an ex- pression is specified, then the value of the expression is used as the filehandle name.
- The only limitation with a filehandle is that it cannot be supplied directly to a user- defined function.

OPERATIONS ON FILES

Opening and Closing Files

- A fundamental part of the Perl language is its ability to read and process file data very quickly.
- All file data is exchanged through the use of a filehandle, which associates an external data source (a file, network socket, external program, pipe) with an internal data structure (the filehandle).
- For files, you use the **open** function to open ordinary files and the **sysopen** function to

  handle more complex opening procedures.
- The **close** function is used to close any open filehandle, regardless of how it was opened.

open Function:

- The **open** function is used to open any existing file along with the mode of opening it. open FILEHANDLE, EXPR
  open FILEHANDLE

- The first form is the one used most often.
- The **FILEHANDLE** is a token name that allows you to refer to a file with a specific name.
- A **FILEHANDLE** in any function can alternatively be an expression, which is evaluated; the value being used as the filehandle name.
- The **EXPR** is more complex. Perl takes the value supplied, interpolates the string where necessary and then strips any leading or trailing white space. The string is then examined for special characters at the start and end of the string that define the mode and type of file to be opened.
- The basic operators are the greater-than/less-than signs.
- The syntax is taken from the shell, which uses a less-than sign to pass file contents to the standard input of a command. Within Perl, this translates to this:

  open(DATA, "<file.txt");
- The **EXPR** for the function shows that the file is being opened read-only. If you want to write to a file, you use the greater-than sign:

  open(DATA, ">file.txt");
- For example, to open a file for updating without truncating it:

  open(DATA, "+<file.txt");
- To truncate the file first:

  open DATA, "+>file.txt" or die "Couldn't open file file.txt, $!";

sysopen Function:

- The **sysopen** function is similar to the main **open** function, except that it uses the

  system **open()** function, using the parameters supplied to it as the parameters for the system function:

  sysopen FILEHANDLE, FILENAME, MODE, PERMS

  sysopen FILEHANDLE, FILENAME, MODE
- There are some differences between the **sysopen** and **open** functions. The **FILENAME**

  argument is not interpreted by **sysopen**.

- There are some standard values if you want to remain completely portable. A **MODE** of zero opens the file read-only; one, write-only; and two, read/write.
- These correspond to the constants **O_RDONLY**, **O_WRONLY**, and **O_RDWR**, which are defined in the **Fcntl** module.
- Two other standard constants are **O_CREAT**, which creates a file if it does not already exist, and **O_TRUNC**, which truncates a file before it is read or written.
- For example, to open a file for updating, emulating the **"+<filename"** format from **open,**

sysopen(DATA, "file.txt", O_RDWR); or to truncate the file before updating,

sysopen(DATA, "file.txt", O_RDWR|O_TRUNC);

close Function:

- To close a filehandle, and therefore disassociate the filehandle from the corresponding file, you use the **close** function. This flushes the filehandle's buffers and closes the system's file descriptor.

  Syntax: close FILEHANDLE close

- If no **FILEHANDLE** is specified, then it closes the currently selected filehandle. It returns **true** only if it could successfully flush the buffers and close the file.
- If you have been writing to a file, then **close** can be used as an effective method of checking that information has been successfully written.

  For example:

```
open(DATA,"+<data.txt") || die "Can't open
data.txt"; #do some work
close(DATA) || die "Couldn't close file properly";
```

- You do not need to close a filehandle before reassigning the filehandle to a new file.
- The **open** function implicitly closes the previous file before opening the new one, but be warned that there is no way of guaranteeing the file status in this way.

Reading and Writing Filehandles

- Once you have an open filehandle, you need to be able to read and write information
- There are a number of different ways of reading and writing data,

The <FILEHANDLE> Operator

- The main method of reading the information from an open filehandle is the

<FILEHANDLE> operator.

- In a scalar context it returns a single line from the filehandle.
- For example:

```
print "What is your name?\n";
$name = <STDIN>;
print "Hello $name\n";
```

where STDIN used to demonstrate how to read information from the keyboard or terminal.

readline Function:

- The **readline** function is actually the internal function used by Perl to handle the **<FILEHANDLE>** operator function.

  Syntax: readline EXPR

- The only difference is that **readline** accepts an expression directly, instead of the usual filehandle.
- This means you need to pass a typeglob to the **readline** function, instead of the normal filehandle.

  Syntax: while (readline *DATA)

getc Function:

- The **getc** function returns a single character from the specified **FILEHANDLE**, or

STDIN if none is specified:

Syntax: getc FILEHANDLE

getc

- If there was an error, or the filehandle is at end of file, then **undef** is returned instead.

read Function:

- The **<FILEHANDLE>** operator or **readline** function reads data from a filehandle using the input record separator, the **read** function reads a block of information from the buffered filehandle:

  Syntax: read FILEHANDLE, SCALAR, LENGTH, OFFSET

  read FILEHANDLE, SCALAR, LENGTH

- The length of the data read is defined by **LENGTH**, and the data is placed at the start of **SCALAR** if no **OFFSET** is specified.
- Otherwise, data is placed after **OFFSET** bytes in **SCALAR**, allowing you to append information from the filehandle to the existing scalar string.
- The function returns the number of bytes read on success, zero at end of file, or **undef** if there was an error.
- This function can be used to read fixed-length records from files, just like the system **fread()** function on which it is based.

print Function:

- For all the different methods used for reading information from filehandles, the main function for writing information back is the **print** function.
- Unlike in C, in Perl **print** is not just used for outputting information to the screen, it can be used to print information to any open filehandle.
- This is largely due to the way Perl structures its internal data.
- Because scalars are stored precisely, without using the traditional null termination seen in other languages, it's safe to use the **print** function to output both variable and fixed-length information.

Syntax: print FILEHANDLE LIST

print LIST

print

- The **print** function prints the evaluated value of **LIST** to **FILEHANDLE**, or to the current output filehandle (**STDOUT** by default). For example:

  print "Hello World!\n";

  or

  print "Hello", $name, "\n How are you today?\n";

  which prints

Printf Function:

Hello Martin

How are you today?

- The Perl parser decides how a particular value is printed which means that floating point numbers are printed as such, when you may wish to restrict the number of places past the decimal point that the number is printed.
- Alternatively, you may wish to left- rather than right-justify strings when you print them.

Syntax: printf FILEHANDLE FORMAT,

LIST printf FORMAT, LIST

- Within C, the only function available is **printf**, which uses a formatting string as the first element and formats the remaining values in the list according to the format specified in the format string.
- For example, the statement

  printf "%d\n", 3.1415126;

  only prints the number 3. The "%d" conversion format determines that an integer should be printed.
- Alternatively, you can define a "currency" format like this, printf "The cost is $%6.2f\n",499;

which would print

The cost is $499.00

Format	**Result**
%%	A percent sign.

%c	A character with the given ASCII code.
%s	A string.
%d	A signed integer (decimal).

%u	An unsigned integer (decimal).
%o	An unsigned integer (octal).
%x	An unsigned integer (hexadecimal).
%X	An unsigned integer (hexadecimal using uppercase characters).
%f	A floating point number (fixed decimal notation).
%g	A floating point number (%e of %f notation according to value size).
%G	A floating point number (as %g, but using “E” in place of “e” when
	appropriate).
%b	An unsigned integer in binary.

sysread Function:

- The **sysread** function reads a fixed number of bytes from a specified

filehandle into a scalar variable:

Syntax:

sysread FILEHANDLE, SCALAR, LENGTH, OFFSET

sysread FILEHANDLE, SCALAR, LENGTH

- If **OFFSET** is specified, then data is written to **SCALAR** from **OFFSET** bytes, effectively appending the information from a specific point.
- If **OFFSET** is negative, data is written from the number of bytes specified counted backward from the end of the string.
- The function is based on the system **read()** function and therefore it avoids the normal buffering supported by standard streams-based **stdio** functions.

Syswrite Function:

- The **syswrite** function is the exact opposite of **sysread**. It writes a fixed-sized block of information from a scalar to a filehandle:

  Syntax:

  syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET

  syswrite FILEHANDLE, SCALAR, LENGTH

- If OFFSET has been specified, then LENGTH bytes are read from the SCALAR and written to FILEHANDLE.

- If the length of the scalar is less than LENGTH, the data is padded with nulls.
- In both cases, you should avoid using the **sysread** and **syswrite** functions with the functions that do use buffered I/O, including **print**, **seek**, **tell**, **write**, and especially **read**.
- If you use these two functions in combination with the **sysseek** function, you can update a database with a fixed record size:

Locating Your Position within a File

- If you are accessing fixed-length information—for example, a database, you are likely to require access to the information in a more random fashion.
- In order to work correctly, you need to be able to discover your current location and set a new location within the file.

tell Function:

- The first requirement is to find your position within a file, which you do using the **tell** function:

  Syntax: tell FILEHANDLE

  tell

- This returns the position of the file pointer, in bytes, within **FILEHANDLE** if specified, or the current default selected filehandle if none is specified.
- The function returns **undef** if there is a problem getting the file position information

seek Function:

- The **seek** function positions the file pointer to the specified number of bytes within a file:

  Syntax: seek FILEHANDLE, POSITION, WHENCE

- The function uses the **fseek** system function, and you have the same ability to position relative to three different points: the start, the end, and the current position.
- You do this by specifying a value for **WHENCE**. The possible values are 0, 1, 2 for positions relative to the start of the file, the current position within the file, end of the file.
- If you import the **IO::Seekable** module, you can use the constants **SEEK_SET**, **SEEK_CUR**, and **SEEK_END**, respectively.
  - Zero sets the positioning relative to the start of the file. For example, the line

    Syntax: seek DATA, 256, 0;

    Sets the file pointer to the 256th byte in the file. Using a value of one sets the position relative to the current position; so the line

    Syntax: seek DATA, 128, 1;

    Moves the file point onto byte 384, while the line.

    Syntax: seek DATA, -128,

    SEEK_CUR; Moves back to byte 256.

Retrieving Documents from the Web using Perl LWP:

Remember, this article is just the most rudimentary introduction to LWP–to learn more about LWP and LWP-related tasks, you really must read from the following:

LWP::Simple: Simple functions for getting, heading, and mirroring URLs.

LWP: Overview of the libwww-perl modules.

LWP::UserAgent: The class for objects that represent “virtual browsers.”

HTTP::Response: The class for objects that represent the response to a LWP response, as in $response = $browser->get(...).

HTTP::Message and HTTP::Headers: Classes that provide more methods to HTTP::Response.

URI: Class for objects that represent absolute or relative URLs.

URI::Escape: Functions for URL-escaping and URL-unescaping strings (like turning “this & that” to and from “this%20%26%20that”).

HTML::Entities: Functions for HTML-escaping and HTML-unescaping strings (like turning “C. & E. Brontë” to and from “C. & E. Brontë”).

HTML::TokeParser and HTML::TreeBuilder: Classes for parsing HTML.

HTML::LinkExtor: Class for finding links in HTML documents.

```
use LWP::Simple;

my $content = get $url;

die "Couldn't get $url" unless defined $content;



# Then go do things with $content, like this:



if($content =~ m/jazz/i) {

  print "They're talking about jazz today on Fresh Air!\n";

} else {

  print "Fresh Air is apparently jazzless today.\n";

}
```