UNIT-V:

Introduction to PERL, Operators and if statements, Program design and control structures, Arrays,Hashes and File handling, Regular expressions, Subroutines, Retrieving documents from the web withPerl.

PERL Programming

Introduction

⇒ Perl was created by Larry Wall in 1987, based on his earlier UNIX system administrative tool called awk.

⇒ Perl, stands for **Practical Extraction and Report Language**, originally meant for text formatting and processing, has grown over times to cover system administration, network, web and database programming, glue between systems and languages (system integration and rapid prototyping), bioinformatics, data mining, and even application development.

⇒ Perl is good for mission-critical large-scale projects as well as rapid prototyping.

⇒ Perl is a mixture of C, UNIX's shell script, awk, sed, and more. Perl is much more expressive than these languages ("maximum expressivity", "There is more than one way to do it").

⇒ Perl is an interpreted language, and therefore platform-independent. You can run Perl scripts in any platform (UNIX, Windows, and Mac) where Perl interpreter is available.

⇒ Far and away the most popular use of Perl is for CGI programming – that is, dynamically generating web pages. Most popular sites on the web: Slashdot (http://www.slashdot.org/), Amazon (http://www.amazon.com/), and Deja (http://www.deja.com/), and many others.

⇒ Its major features are supportboth procedural and object-oriented (OO) programming, and has one of the world's most impressive collections of third-party modules. Many Perl utilities and add-ons are available at CPAN (Comprehensive Perl Archive Network @ www.cpan.org).

⇒ Perl is Open Source software, licensed under the GNU General Public License (GPL).

⇒ The Perl versions include:

- 1.0 (1987)
- 2.0 (1988)
- 3.0 (1989)
- 4.0 (1991)
- 5.0 (1994),..., 5.5 (1998),..., 5.10 (2007), 5.11 (2009)
- Perl 6 also known as Rakudo Perl 6 isgradually typed language, multi-paradigmatic (supports Procedural, Object Oriented, and Functional programming), based on NQP (Not Quite Perl) and can useMoarVM or the Java Virtual Machine as a runtime environment (Jan 2016).

Installing Perl:

⇒ There are many ways to get the Perl Interpreter:

- For Windows systems: Can be installed as part of CYGWIN or ActiveState Perl or Strawberry Perl (can be downloaded from https://www.perl.org/get.html) or Padre, the Perl IDE based on Strawnerry Perl (http://padre.perlide.org/).

- For UNIX systems: Pre-installed.

First Program:

⇒ Use a programming text editor (such as NotePad++, PSPad, TextPad) to enter the following source codes and save as "Hello.pl":

```
#!/usr/bin/envperl
use strict;                          # Terminate when error occurs
use warnings;                        # Display all warning messages
print "Hello world!\n";              # Print a message
print 'Hello world, ', 'Again!', "\n"; # Print another message
```

How It Works?

⇒ Line 1, called shebang, is meant for UNIX, which specifies the location of the Perl Interpreter. This line is ignored under Windows.

⇒ Line 2 and 3 are directive (or pragma) to instruct Perl on how to handle errors. "use strict" instruct Perl to terminate the program immediately when an error occurs. "use warning" instruct Perl to display all the warning messages.

⇒ Lines 1 to 3 are optional, but recommended for writing robust program.

⇒ A comment begins with '#' and lasts until the end of line. Comments are used to explain the codes; but are ignored by the interpreter.

⇒ A Perl's statement ends with a semi-colon (;).

⇒ The print function prints the given string to the console. \n denotes a new-line. A function can take zero or more arguments (separated by commas). In Perl, you can enclose the function's arguments with parentheses () or omit them.

⇒ Strings can be enclosed in double-quotes or single-quotes.

⇒ Extra white-spaces (blanks, tabs, new-lines) are ignored.

⇒ The file extension of ".pl" is not mandatory but recommended.

Running In Windows

⇒ To run the program under Windows, start a cmd shell and issue the command (change directory to the directory containing Hello.pl).

```
>perl Hello.pl
```

Output:

```
Hello world!
Hello world, Again!
```

Note: The path for Perl Interpreter "perl.exe" must be in included in the PATH environment variable.

⇒ To display the Interpreter's help menu, issue:

```
>perl –h
```

⇒ To find the version of the Perl Interpreter, issue:

```
>perl –v
```

This is a perl, v5.x built for …

Running In UNIX

⇒ To run the program:
- Change directory to the directory containing Hello.pl ...
- Make the file executable (via change-mode command) and then execute the file
  $ chmodu+x Hello.pl
  $ ./Hello.pl

Basic Syntax:

Comments:

⇒ In Perl, a comment begins with a hash (#) character.
⇒ Perl interpreter ignores comments at both compile time and runtime.
⇒ You use the comment to document the logic of your code. The code tells you what it does however comment provides information on why the code does so.
⇒ Comment is very important and useful to you as a programmer in order to understand the code later on, as well as other developers who will maintain your code in the future.
⇒ There is no multi-line comment other than putting# at the beginning of each line.

Variables, Literals & Data Types:

⇒ A variable is a named storage location that holds a value, of a certain data type.
⇒ A literal is a fixed value, e.g., 5566, 3.14, 'Hello', that can be assigned to a variable or form part of an expression.
⇒ Perl supports the following data types. It uses different initial symbols to denote and differentiate the various data types.
- Scalar: begins with symbol $.
- Array: begins with symbol @.
- Hash or Associative Array: begins with symbol %.

⇒ An expression is a combination of variables, literals, operators, and sub-expressions that can be evaluated to produce a single value.

Note: Perl is case-sensitive. A $rose is not a $ROSE, and is not a $Rose.

⇒ Unlike strong-type languages like C/C++/C#/Java, but like JavaScript/UNIX Shell Script:
- Perl's variables name need not be declared before use, which often leads to poor programs. It is strongly recommended to declare a variable before use!!
- The actual type of a scalar (e.g., integer, floating-point number or string) need NOT be specified. Perl's scalar is simply a single item, which could take on context of number (integer or floating-point number), string, or boolean automatically.

⇒ You could assign a value (called literal) to a scalar variable using the assignment operators (=). The scalar variable takes on the context of the literal assigned. For example, a variable takes on a string context if a string literal is assigned; takes on a numeric context if a numeric literal is assigned.

Note: You can declare a local variable via the keyword **my.**

Example:

```
my $num = 123;     # numeric context
my $str = "Hello";  # string context
```

Note: The context of the scalar is important because many operations are confined to a certain context, e.g., arithmetic operations (+, -, *, /) can be applied to numbers but not strings; strings can concatenate using "." operator; logical operations (and, or, not) are applied to boolean. Perl automatically converts between the different contexts as needed to perform an operation. In other word, the context of a scalar is determined by the operation.

Example:

```
# ScalarContextTest.pl
use strict;
use warnings;
my $num1 = 11;                  # Numeric context
my $num2 = 22;                  # Numeric context
my $str1 = 'Hello';             # String context
my $str2 = 'world';             # String context
my $str3 = '33';                # String context
my $str4 = '44';                # String context
print $num1 + $num2 , "\n";     # + takes numbers
print 12 * 3.4 , "\n";          # * takes numbers
print $str1 + $str2 , "\n";     # + takes numbers, not strings. Invalid output
print $str1 . " " . $str2 , "\n"; # . takes strings
print $str3 + $str4 , "\n";     # + takes numbers - String converted to numeric context
print "5.5" - 5 , "\n";         # - takes numbers - String converted to number
print $num1 . $num2 , "\n";     # . takes strings - Numbers converted to string
```

Output:

```
33
40.8
Argument "world" isn't numeric in addition (+) at ScalarContextTest.pl line 13.
Argument "Hello" isn't numeric in addition (+) at ScalarContextTest.pl line 13.
0
Hello world
77
0.5
1122
```

How does Perl know that a variable is a number or a string?

⇒ In fact, Perl does not know. Whenever a variable or string literal is used as an argument to an arithmetic operation (+, -, *, /), Perl tries to convert it to a number. If the variable does not contain a valid number, Perl simply sets it to 0; and you will not be warned unless you specify "use warning" or turn on the -w (warning) flag!

⇒ A variable takes a value called UNDEF, if no value is assigned to it.

⇒ In Perl, numbers are stored as double-precision floating-point. All the arithmetic operations are carried out in floating-point. There is no distinct integer type in Perl!

⇒ Numeric literals include:
- Point-point literals: e.g., 3.1416, -0.8e18, 1.2E-0.5.
- Integer literals: e.g., 5566, -128. You can delimit a long integer with underscore, e.g., 12_111_222_333.
- Octal literals: begin with a leading 0 (zero), e.g., 0127.
- Hexadecimal literals: begin with 0x, e.g., 0xABCD.
- Binary literals: begin with 0b, e.g., 0b10110011.

Expressions

⇒ In Perl, an expression is anything that returns a value.

⇒ The expression can be used in a larger expression or a statement.

⇒ The expression can be a literal number, complex expression with operators, or a function call.

For example:

3 is an expression that returns value of 3.

The $a + $b is an expression that returns the sum of two variables: $a and $b.

Statements

⇒ A statement is made up of expressions. Statement is executed by Perl at run-time. Each Perl statement must end with a semicolon (;). The following example demonstrates the statements in Perl:

```
$c = $a + $b;
print($c);
```

Blocks

⇒ A block is made up of statements wrapped in curly braces. You use blocks to organize statements in program. The following example demonstrates a block in Perl:

```
{
   $a = 1;
   $a = $a + 1;
print($a);
}
```

⇒ Any variable declared inside a block has its own scope. It means the variables declared inside a block only last as long as the block is executed.

Keywords

⇒ Perl has a set of keywords that have special meanings to its language. Perl keywords fall into some categories such as built-in function and control keywords. You should always avoid using keywords to name variables, functions, modules and other objects.

Calling Perl's Built-in Functions

⇒ Perl has many built-in functions, which takes a comma-separated list of arguments.

⇒ You can enclose the arguments in parentheses or omit them, depending on your programming style.

⇒ For example:

```
print 'Hello, world', "\n";    # Function arguments are separated by commas
print('Hello, world', "\n");   # Parentheses are optional
say 'Hello, world';          # Function say (Perl 5.10) always prints a newline
say('Hello, world');
```

Perl Operators:

⇒ Perl provides numeric operators to help you operate on numbers including arithmetic, Boolean and bitwise operations.

Arithmetic operators: Perl provides the following arithmetic operators for numbers.

Operator	Description
+	Addition
-	Subtraction (or Unary Negation)
*	Multiplication
/	Division
%	Modulus (Division Remainder)
**	Exponentiation
++	Unary Pre- or Post-Increment
--	Unary Pre- or Post-Decrement

⇒ Arithmetic operations are carried out in floating-point (double precision). In other words, 1/2 give 0.5 (whereas in C/Java, 1/2 gives 0). You can truncate a floating point number to integer via built-in function int().

Arithmetic cum Assignment Operators: These are short-hand operators to combine two operations.

OPERATOR	DESCRIPTION
+=	Addition cum Assignment
-=	Subtraction cum Assignment
*=	Multiplication cum Assignment
/=	Division cum Assignment
%=	Modulus cum Assignment
**=	Exponentiation cum Assignment

Comparison Operators: Perl provides the following operators for comparing numbers:

OPERATOR	DESCRIPTION
==	Equal To
!=	Not Equal To
>	Greater Than to
>=	Greater Than or Equal To
<	Less Than
<=	Less Than or Equal To

String Context and Operations:

⇒ Strings are sequence of zero or more characters. String literals can be enquoted with single quotes or double quotes. However, the type of quotes is significant: double quotes interpret (or interpolate) variables and special characters (e.g., \n for new-line, \t for tab, \\ for back-slash), whereas single quotes don't.

⇒ Perl looks for the longest possible variable name in interpolation (i.e., greedy). For example,

```
my $msg = 'Hello';
print "$msg world\n";      # print Hello world followed by newline
print '$msg world\n';      # print $msg world\n literally (no interpretation)
```

String Operators: Perl provides the following string operators:

OPERATOR	DESCRIPTION
.	String Concatenation
x	Duplicate
.=	Concatenation cum Assignment
x=	Duplicate cum Assignment

String Comparison Operators: Perl provides the following operators for comparing strings:

OPERATOR	DESCRIPTION
eq	String Equal To
ne	String Not Equal To
gt	String Greater Than to
ge	String Greater Than or Equal To
lt	String Less Than
le	String Less Than or Equal To
cmp	String Compare To(returns 1 if the fist string is greater than the second string; 0 if equal; -1otherwise)

String Functions: Perl provides many built-in functions for manipulating strings:

⇒ substr(var, index, length): returns the substring from string var, starting from position index, of length. String index begins at 0. You can also use substr to modify the original string.
⇒ index(string, substring): return the index of the substring in string, or -1 if not found.
⇒ rindex(string, substring): return the index but searching from the right.
⇒ length(string): returns the number of characters in string.
⇒ lc(string): returns a lowercase string.
⇒ uc(string): returns an uppercase string.
⇒ lcfirst(string): returns a first-letter lowercase string.
⇒ ucfirst(string): returns a first-letter uppercase string.

Boolean Context and Operations:

⇒ A scalar can take a boolean context of either true or false. "False" includes:
- The number 0.
- An empty string '' or "".
- A string containing a zero (i.e., '0' or "0").
- A variable that has not been assigned a value (i.e. UNDEF).
- An empty array or hash (to be discussed later).

Anything else is considered as true.

Functions defined and undef: defined(var) returns true if the variable var is defined. undef(var) un-defines the variable var.

Boolean Operators: Perl provides the following boolean (or logical) operators:

OPERATOR	DESCRIPTION
&&	C-style's Logical AND
||	C-style's Logical OR
!	C-style's Logical NOT
and	Perl's Logical AND
or	Perl's Logical OR
not	Perl's Logical NOT

Note:

⇒ Perl's not, and, or carry out the same operations as C-style's !, &&, ||, but these logical operators have very low precedence (lower than assignment operator =) and can be useful in certain situations (but you can also use the parentheses to change the precedence). They are also easier to read than the C-style logical operators.
⇒ Logical operations are always short-circuited. That is, the operation is terminated as soon as the result is certain, e.g., false && ... is short-circuited to give false, true || ... gives true.

Input from Keyboard & Formatted Output:

⇒ You can use the operator <> or <STDIN> (called file-handle) to read input from keyboard.

⇒ The input, however, contains the newline character (corresponding to the enter key), which can be stripped away via function chomp.

Functions chomp and chop:

⇒ chop removes the last character of a string.

⇒ chomp removes the last character only if that is a newline character.

⇒ Both chop and chomp returns the number of character removed.

Example:

```
# UserInputTest.pl
use strict;
use warnings;
print 'Enter your message: ';
my $msg = <>;                          # <> to read user's input
print "Your message is $msg";          # $msg include a newline
print 'Enter your last name: ';
my $lastName = <>;
chomp $lastName;                       # Strip ending newline
print 'Enter your first name: ';
my $firstName = <>;
chomp $firstName;                      # Strip ending newline
my $fullName = $firstName . ' ' . $lastName;          # Concatenate
print "Your full name is $fullName\n";                # $fullname does not have newline
```

Function printf and sprintf:

⇒ C-style's printf and sprintf (string printf) are supported in Perl for formatted output.

Example:

```
my $str = 'Hello';
my $float = 1.2;
my $num = 33;
# %s for string, %f for floating-point number, %d for integer
printf "%10s %6.2f and %3d\n", $str, $float, $num;
my $pstr = sprintf "%10s %6.2f and %3d\n", $str, $float, $num;
say $pstr;
```

Conditional Flow Control:

⇒ Perl provides many variations of flow control constructs:

Sntax	Example
if (condition) { trueBlock; }	if ($day eq 'sat' || $day eq 'sun') { print 'Super weekend!'; }

trueSingleStatement if condition;	print 'Super weekend!' if ($day eq 'sat' || $day eq 'sun');
unless (condition) { falseBlock; } same as: if (!condition) { falseBlock; }	unless ($day eq 'sat' || $day eq 'sun') { print 'It is a weekday'; } unless ($day ne 'sat' || $day ne 'sun') { print 'Super weekend!'; } unless $error { print 'Yes, Hello'; }
falseSingleStatement unless condition;	print 'It's a weekday' unless ($day eq 'sat' || $day eq 'sun');

if (condition)
{
trueBlock;
}
else
{
falseBlock;
}
	if ($day eq 'sat' || $day eq 'sun')
	{
	print 'Super weekend!';
	}
	else
	{
	print 'It is a weekday...';
	}

if (condition1)
{
trueBlock1;
}
elsif (condition2)
{
trueBlock2;
}
elsif
{
...
}
else
{
elseBlock;
}
	if ($day eq 'sat' || $day eq 'sun')
	{
	print 'Super weekend!';
	}
	else if ($day eq 'fri')
	{
	print "Thank God, it's friday!";
	}
	else
	{
	print 'It is a weekday...';
	}

condition ?trueStatement : falseStatement;	max = (a > b) ? a : b; abs = (a >= 0) ? a : -a;

# Perl 5.10 switch-case: 	
given (variable)
{
when (value1) { ... }
when (value2) { ... }
.....
}
	given ($day)
	{
	when ('sat', 'sun'){ print 'Super weekend!'; }
	when ('mon', 'tue', 'wed', 'thu') { print 'It is a weekday...'; }
	when ('fri') { print "Thank God, it's friday"; }
	}

Note:

⇒ The curly braces are mandatory even if there is only one statement in the block.

⇒ A negate version of if called unless is provided. It could be hard to read and should be used only for negative logic, e.g., unless $error { ... }, could be better than if not $error { ... }.

⇒ The statement block can be placed before or after if or unless clause.

⇒ The keyword for else-if is elsif.

Flow Control – Loops:

⇒ Perl provides many types of loop constructs:

SYNTAX	EXAMPLE

while (condition)
{
trueBlock;
}
	my $i = 0;
	while ($i < 10)
	{
	  print "$i\n";
	  $i++;
	}

do
{
trueBlock;
}
while (condition);
	my $i = 0;
	do
	{
	  print "$i\n";
	  $i++;
	} while ($i < 10);

until (condition)
{
falseBlock;
}
same as:
while (!condition)
{
falseBlock;
}
	my $i = 0;
	until ($i >= 10)
	{
	  print "$i\n";
	  $i++;
	}

foreach $scalarName (@arrayName)
{
statementBlock;
}
or
for $scalarName (@arrayName)
{
statementBlock;
}
	my @months = ('jan', 'feb', 'mar', 'apr');
	foreach my $month (@months)
	{
	  print $month, "\n";
	}
	for my $i (5, 4, 3, 2, 1)
	{
	  print "$i ";
	}

for (initialization; expression; postIncrement)
{
statementBlock;
}
	my @months = ('jan', 'feb', 'mar', 'apr');
	for (my $i = 0; $i < @months; $i++)
	{
	  print $months[$i], "\n";
	}

Note:

⇒ Again, the curly braces are mandatory, even if there is only one statement in the block.

⇒ foreach loop is handy for reading each item of the array. It cannot modify the array.
⇒ The negation version until should be used only for negative logic, e.g., until ($done) { ... }.

Loop Control Statements:

⇒ last: exit the for loop (similar to break statement in C/Java).
⇒ next: aborts the current iteration and continues to the next iteration of the loop (similar to continue statement in C/Java)
⇒ redo: redo the current iteration (from the begin brace).
⇒ last, next and redo work with a labeled block in the form of labelName: ...

Example:

```
# LoopTest.pl
use strict;
use warnings;
my $num = 1;
while (1)                       # Always true
{
        $num++;
        next if ($num % 3) == 0;  # Continue to next num if num is divisible by 3
        last if $num == 17;     # Break the loop if num is 17
        if (($num % 2) == 0)
        {
                $num += 3;      # Add 3 for even number
        }
        else
        {
        $num -= 3;          # Subtract 3 for odd number
        }
        print "$num ";
}
```

Output:

```
>perl LoopTest.pl
5 4 2 7 11 10 8 13 17 16
```

Special Scalar Variable: The Default Scalar Variable $_

⇒ Perl introduces a feature called the default variable, which is not found in other languages. The default scalar variable is named $_.
⇒ Many constructs and functions, such as foreach loop and print, takes $_ as the default argument.

Example:

```
while (<>)              # while ($_ = <>) to read input from keyboard
{
print;                  # print $_
```

```
chomp;                  # chomp $_ to remove ending newline from $_
last if ($_ eq 'done');  # break the loop if input is 'done'
}
```

Arrays:

⇒ An array contains a list of zero or more scalars.
⇒ An array variable begins with @, whereas a scalar variable begins with $.
⇒ A @name is nothing to do with a $name.
⇒ An array can be assigned to and from a list of commas-separated scalars enclosed in parentheses.

Example:

```
my @months = ('jan', 'feb', 'mar', 'apr');
my @days = qw(montue wed thufri sat sun);              # single-quoted words
(my $first, my $second, my $third, my $fourth) = @months;
print @months, "\n";                                   # janfebmarapr
print $first, "\n";                                    # jan
print $fourth, "\n";                                   # apr
```

⇒ Numbers and strings (and undef) can be mixed inside an array. e.g.,

```
my @mixmonths = ('jan', 2, 'mar', 4);
print @mixmonths, "\n";     # jan2mar4
```

⇒ You can use array index in the form of $arrayName[index] to reference individual element of an array.
⇒ The array index starts at 0.
⇒ Note that scalar context $ is used for referencing individual element instead of array context @. Accessing an array past its bound gives UNDEF.
⇒ You can also refer to a portion (or slice) of an array (i.e., sub-array) using an index range in the form of @arrayName[beginIndex..endIndex] or @arrayName[index1,index2,...].
⇒ For example,

```
my @months = ('jan', 'feb', 'mar', 'apr');
print $months[2], "\n";          # Scalar 'feb'
print @months[1..3], "\n";       # Array slice ('feb', 'mar')
print @months[3,1], "\n";        # Array slice ('apr', 'jan')
print @months[2], "\n";          # Array slice ('feb')
my @emptyArray = ();             # Empty array
```

⇒ In Perl, array is not bounded. Its size will be dynamically expanded when new elements are added.

Example:

```
my @months = ('jan', 'feb', 'mar', 'apr');
@months[4..5]= ('may', 'jun');     # @months is ('jan', 'feb', 'mar', 'apr', 'may', 'jun')
$months[7] = 'aug';                # $month[6] gets UNDEF
```

⇒ The scalar variable $#arrayName maintains the last index of the array @arrayName. You might be tempted to use $#arrayName+1 as the length of the array. This is not necessary, as Perl will return

the length of the array if @arrayName is used in a scalar context (e.g., assign to a scalar, arithmetic and comparison operations). In other words, to reference the length of an array, you can simply assign @arrayName to a scalar context. For example:

```
my @months = ('jan', 'feb', 'mar', 'apr');
print $#months, "\n";                    # Gives 3
print $months[$#months], "\n";           # Gives 'apr'
$months[$#months + 1] = 'may'
my $size = @months;                      # Get the length of the array
print $size, "\n";
for (my $i = 0; $i < @months; $i++)      # @months in scalar context
{
print $months[$i], "\n";
}
```

⇒ Negative array index *n* can be used to reference the *n*th-to-last element of the array, e.g.,

```
my @months = ('jan', 'feb', 'mar', 'apr');
print $months[-1], "\n";                 # Gives 'apr'
print $months[-2], "\n";                 # Gives 'mar'
```

Array Functions: Perl provides many functions to manipulate arrays:

⇒ push(array, list): appends the list of elements to the end of the array.

⇒ pop(array): removes and returns the last element of the array.

⇒ shift(array): removes and returns the first element of the array.

⇒ unshift(array, list): add the list of the elements in front of the array.

⇒ splice(array, offset, legnth, list): removes and returns length elements from array, starting from offset, and optionally, replace them with list.

Example:

```
my @months = ('jan', 'feb', 'mar', 'apr');
push @months, 'may';        # @months = ('jan', 'feb', 'mar', 'apr', 'may')
print @months, "\n";
print pop @months, "\n";                 # @months = ('jan', 'feb', 'mar', 'apr')
print pop @months, "\n";                 # @months = ('jan', 'feb', 'mar')
push (@months, shift @months);     # Move the first element to last
print @months, "\n";               # @months = ('feb', 'mar', 'jan')
```

Special Array Variable: The Command-Line Argument Array @ARGV:

⇒ The command-line arguments (excluding the program name) are packed in an array, and passed into the Perl's program as an array named @ARGV, The function shift, which takes @ARGV as the default argument, is often used to process the command-line argument.

Hash or Associative Array

⇒ Hash or Associative Array, begins with a %. Take note that %name is not a @name is not a $name.
⇒ Hash stores key-value (or name-value) pairs.
⇒ Hash is similar to regular array, except that regular arrays are indexed by numbers; but hashes are indexed by key-strings.
⇒ Hash lets you associate one scalar to another, hence, it is also called associative array.
⇒ To initialize a hash, you could provide a list of key-value pairs in the form of (key1 => value1, key2 =>value2, ...) or (key1, value1, key2, value2, ...). Key must be unique.
⇒ You can retrieve the value associated to a key, in the scalar-context form of $hashName{keyName}. Recall that array uses square bracket with numerical index, $arrayName[index], whereas hash uses curly bracket and key-string index.

Example:

```
# HashTest.pl
use strict;
use warnings;
# Declare and initialize a hash with key-value pairs.
my %countryCodes = ('us' => 'United States', 'sg' => 'Singapore');
# Use $hashName{keyName} (scalar context) to reference the value of an item.
print $countryCodes{'us'}, "\n";   # prints 'United States'
print $countryCodes{'sg'}, "\n";   # prints 'Singapore'
# Add in more key-value pairs
$countryCodes{'fr'} = 'France';
$countryCodes{'cn'} = 'China';
print %countryCodes, "\n";   # prints all items
my %emtpyHash = ();   # an initially empty hash
```

⇒ You can convert a hash to an array and vice versa. The array stores the key-value pairs as sequential entries but in no particular order, e.g.,

```
# Assign Hash to Array
my %countryCodes = ('us' => 'United States', 'sg' => 'Singapore');  # Hash
my @countryArray = %countryCodes;       # Assign a Hash to an array
print $countryArray[0], "\n";           # Referencing array
print $countryArray[1], "\n";
# Assign an Array (a list of items) to a Hash
my %countryHash = ('us', 'United States', 'sg', 'Singapore');  # Hash
print $countryHash{'us'}, "\n";   # Referencing hash
print $countryHash{'sg'}, "\n";
```

Hash Functions:

⇒ keys(hashName): returns an array containing all the keys in hashName.
⇒ values(hashName): returns an array containing all the values in hashName.

⇒ each(hashName): returns a 2-element array (key, value) containing the next key-value pair from hashName.
⇒ delete($hashName{keyName}): removes the key-value pair of keyName from hashName, and returns the deleted value.
⇒ exists($hashName{keyName}): returns true if keyName exists in hashName.
⇒ defined($hashName{keyName}): check if value of keyName is defined in hashName.

Example:

```
my %countryCodes = ('IN' => 'India', 'SG' => 'Singapore');
while ((my $key, my $value) = each %countryCodes)
{
print "$key is associated with $value.\n";
}
```

Special Hash Variable: The Environment Variables Hash %ENV

⇒ A program can access an operating environment which contains information such as the current directory, the username, and etc. Perl stores the environment variables in a special hash named %ENV.

Example:

```
print $ENV{'PATH'};   # print environment variable PATH
while ((my $key, my $value) = (each %ENV))
{
# prints all environment variables
print "$key=$value\n";
}
```

⇒ %ENV hash is useful in writing server-side CGI Perl scripts.

Sorting the Hash:

```
foreach my $key (sort keys %ENV)     # returns array of sorted keys.
{
print "$key=$ENV{$key}\n";           # get the value with the sorted keys
}
```

File Input/Output:

⇒ File input and output is an integral part of every programming language.
⇒ File operations are performed with Perl is done using FILE HANDLE.
⇒ A filehandle is a variable that associates with a file. Through a filehandle variable, you can read from the file or write to the file depending on how you open the file.
⇒ Once a filehandle is created and connected to a file (or a directory, or a program), you can read or write to the underlying file through the filehandle using angle brackets, e.g., <FILEHANDLE>

Open a file:

⇒ You use open() function to open files. The open() function has three arguments:
- Filehandle that associates with the file
- Mode: you can open a file for reading, writing or appending.
- Filename: the path to the file that is being opened.

Syntax: open(file handle, mode, filename)

⇒ Allowed file open modes are **<** for read but cannot change contents, **>** for write(If the file does not exist, a new file is created. If the file already exists, the content of the file is wipe out),**>>** for append (for appending new content to the existing content of the file, but, cannot change the existing content in the file), **+<** for read and write but does not create, **+>** for creating, clear, read and write, **+>>** for create, read and append.

Closing the files:

⇒ After processing the file such as reading or writing, you should always close it explicitly by using the close() function.

Syntax: close(file handle)

Write to a file:

⇒ Data can be written into an opened file using the function print().

Syntax: print(file handle, string)

Read a File:

⇒ In order to read from a file in read mode, you put the filehandle variable inside angle brackets as follows:

<file handle>

⇒ To read the next line of the file with newline included, you use the following syntax:

$line = <file handles>;

Example: Write to a file

```
use strict;
use warnings;
my $filename = shift;                    # Get the filename from command line.
open(FILE, ">$filename") or die "Can't write to $filename: $!";
print FILE "This is line 1\n";           # no comma after FILE.
print FILE "This is line 2\n";
print FILE "This is line 3\n";
```

Example: Read a file

```
use strict;
use warnings;
my $filename = shift;                    # Get the filename from command line.
```

```
open(FILEIN, "test.txt") or die "Can't open file: $!";
while (<FILEIN>)                    # set $_ to each line of the file.
{
print;                               # print $_
}
```

Example: Appending to a file

```
use strict;
use warnings;
my $filename = shift;          # Get the file from command line.
open(FILE, ">>$filename") or die "Can't append to $filename: $!";
print FILE "This is line 4\n";     # no comma after FILE.
print FILE "This is line 5\n";
```

Functions seek, tell, and truncate:

⇒ seek(filehandle, position, whence): moves the file pointer of the filehandle to position, as measured from whence. seek() returns 1 upon success and 0 otherwise. File position is measured in bytes. whence of 0 measured from the beginning of the file; 1 measured from the current position; and 2 measured from the end. For example:

```
seek(FILE, 0, 2);    # 0 byte from end-of-file, give file size.
seek(FILE, -2, 2);   # 2 bytes before end-of-file.
seek(FILE, -10, 1);  # Move file pointer 10 byte backward.
seek(FILE, 20, 0);   # 20 bytes from the begin-of-file.
```

⇒ tell(file handle): returns the current file position of file handle. To find the length of a file, you could:

```
seek(FILE, 0, 2);   # Move file point to end of file.
print tell(FILE);   # Print the file size.
```

⇒ truncate(FILE, length): truncates FILE to length bytes. FILE can be either a filehandle or a file name.

Example: Truncate the last 2 bytes if they begin with \x0D,

```
use strict;
use warnings;
my $filename = shift;                # Get the file from command line.
open(FILE, "+<$filename") or die "Can't open $file: $!";
seek(FILE, -2, 2);                   # 2 byte before end-of-file.
my $pos = tell FILE;
my $data = <FILE>;                   # read moves the file pointer.
if ($data =~ /^\x0D/)                # begin with 0D
{
truncate FILE, $pos;                 # truncate last 2 bytes.
}
```

Function eof()

⇒ eof(filehandle) returns 1 if the file pointer is positioned at the end of the file or if the filehandle is not opened.

Reading Bytes Instead of Lines

⇒ The function read(filehandle, var, length, offset) reads length bytes from filehandle starting from the current file pointer, and saves into variable var starting from offset (if omitted, default is 0). The bytes includes \x0A, \x0D etc.

Example

```
use strict;
use warnings;
(my $numbytes, my $filename) = @ARGV;
open(FILE, $filename) or die "Can't open $filename: $!";
my $data;
read(FILE, $data, $numbytes);
print $data, "\n----\n";
read(FILE, $data, $numbytes);        # continue from current file ptr
print $data;
print $data, "\n----\n";
read(FILE, $data, $numbytes, 2);     # save in $data offset 2
print $data, "\n----\n";
```

Function stat and lsstat

⇒ The function stat(FILE) returns a 13-element array giving the vital statistics of FILE. lsstat(SYMLINK) returns the same thing for the symbolic link SYMLINK.

⇒ The elements are:

Index	Value
0	The device
1	The file's inode
2	The file's mode
3	The number of hard links to the file
4	The user ID of the file's owner
5	The group ID of the file
6	The raw device
7	The size of the file
8	The last accessed time
9	The last modified time

10	The last time the file's status changed
11	The block size of the system
12	The number of blocks used by the file

⇒ For example: The command

```
perl -e "$size= (stat('test.txt'))[7]; print $size"
```

prints the file size of "test.txt".

Standard Filehandles

⇒ Perl defines the following standard filehandles:

- STDIN – Standard Input, usually refers to the keyboard.
- STDOUT – Standard Output, usually refers to the console.
- STDERR – Standard Error, usually refers to the console.
- ARGV – Command-line arguments.

Example:

```
my $line = <STDIN>              # Set $line to the next line of user input
my $item = <ARGV>    # Set $item to the next command-line argument
my @items = <ARGV># Put all command-line arguments into the array
```

When you use an empty angle brackets <> to get inputs from user, it uses the STDIN filehandle; when you get the inputs from the command-line, it uses ARGV filehandle.

Perl fills in STDIN or ARGV for you automatically. Whenever you use print() function, it uses the STDOUT filehandler.

<> behaves like <ARGV> when there is still data to be read from the command-line files, and behave like <STDIN> otherwise.

Text formatting: write()

⇒ write(filehandle): printed formatted text to filehandle, using the format associated with filehandle. If filehandle is omitted, STDOUT would be used.

Declaring format

```
format name =
text1
text2
.
```

Picture Field @<, @|, @>

⇒ @<: left-flushes the string on the next line of formatting texts.
⇒ @>: right-flushes the string on the next line of formatting texts.
⇒ @|: centers the string on the next line of the formatting texts.

⇒ @<, @>, @| can be repeated to control the number of characters to be formatted. The number of characters to be formatted is same as the length of the picture field. @###.## formats numbers by lining up the decimal points under ".".

Printing Formatting String printf

⇒ printf(filehandle, template, array): prints a formatted string to filehandle (similar to C's fprintf()). For example,

```
printf(FILE "The number is %d", 15);
```

⇒ The available formatting fields are:

Field	Expected Value
%s	String
%c	Character
%d	Decimal number
%ld	Long decimal Number
%u	Unsigned decimal number
%x	Hexadecimal number
%lx	Long hexadecimal number
%o	Octal number
%lo	Long octal number
%f	Fixed-point floating-point number
%e	Exponential floating-point number
%g	Compact floating-point number

Inspecting Files

⇒ You can inspect a file using (-testFILE) condition. The condition returns true if FILE satisfies test. FILE can be a filehandle or filename. The available test are:
- -e: exists.
- -f: plain file.
- -d: directory.
- -T: seems to be a text file (data from 0 to 127).
- -B: seems to be a binary file (data from 0 to 255).
- -r: readable.
- -w: writable.
- -x: executable.
- -s: returns the size of the file in bytes.
- -z: empty (zero byte).

Example

```
use strict;
use warnings;
my $dir = shift;
opendir(DIR, $dir) or die "Can't open directory: $!";
my @files = readdir(DIR);
closedir(DIR);
foreach my $file (@files)
{
if (-f "$dir/$file")
{
print "$file is a file\n";
print "$file seems to be a text file\n" if (-T "$dir/$file");
print "$file seems to be a binary file\n" if (-B "$dir/$file");
my $size = -s "$dir/$file";
print "$file size is $size\n";
print "$file is a empty\n" if (-z "$dir/$file");
}
elsif (-d "$dir/$file")
{
print "$file is a directory\n";
}
print "$file is a readable\n" if (-r "$dir/$file");
print "$file is a writable\n" if (-w "$dir/$file");
print "$file is a executable\n" if (-x "$dir/$file");
}
```

Accessing the Directories

⇒ opendir(DIRHANDLE, dirname) opens the directory dirname.

⇒ closedir(DIRHANDLE) closes the directory handle.

⇒ readdir(DIRHANDLE) returns the next file from DIRHANDLE in a scalar context, or the rest of the files in the array context.

⇒ glob(string) returns an array of filenames matching the wildcard in string, e.g., glob('*.dat') and glob('test??.txt').

⇒ mkdir(dirname, mode) creates the directory dirname with the protection specified by mode.

⇒ rmdir(dirname) deletes the directory dirname, only if it is empty.

⇒ chdir(dirname) changes the working directory to dirname.

⇒ chroot(dirname) makes dirname the root directory "/" for the current process, used by superuser only.

Example: Print the contents of a given directory.

```
use strict;
use warnings;
my $dirname = shift;                    # first command-line argument.
opendir(DIR, $dirname) or die "can't open $dirname: $!\n";
@files = readdir(DIR);
closedir(DIR);
foreach my $file (@files)
{
print "$file\n";
}
```

Example: Removing empty files in a given directory

```
use strict;
use warnings;
my $dirname = shift;
opendir(DIR, $dirname) or die "Can't open directory: $!";
my @files = readdir(DIR);
foreach my $file (@files)
{
        if ((-f "$dir/$file") && (-z "$dir/$file"))
        {
        print "deleting $dir/$file\n";
        unlink "$dir/$file";
        }
}
closedir(DIR);
```

Example: Display files matches "*.txt"

```
my @files = glob('*.txt');
foreach (@files)
{
print; print "\n"
}
```

Example: Display files match the command-line pattern.

```
$file = shift;
@files = glob($file);
foreach (@files)
{
        print;
        print "\n"
}
```

Deleting file: Function unlink

⇒ unlink(FILES) deletes the FILES, returning the number of files deleted. Do not use unlink() to delete a directory, use rmdir() instead. For example,

```
unlink $filename;
unlink "/var/adm/message";
unlink "message";
```

Regular Expressions

⇒ A Regular Expression (or Regex) is a pattern (or filter) that describes a set of strings that matches the pattern. In other words, a regex accepts a certain set of strings and rejects the rest.

⇒ Regex is supported in all the scripting languages (such as Perl, Python, PHP, and JavaScript); as well as general purpose programming languages such as Java; and even word processors such as Word for searching texts.

⇒ Perl makes extensive use of regular expressions with many built-in syntaxes and operators. In Perl (and JavaScript), a regex is delimited by a pair of forward slashes (default), in the form of /*regex*/. You can use built-in operators:

m/ regex / modifier:	Match against the regex.
s/ regex/ replacement/modifier:	Substitute matched substring(s) by the replacement.

Matching Operator m//

⇒ You can use matching operator m// to check if a regex pattern exists in a string. The syntax is:

```
m/regex/
m/regex/modifiers          # Optional modifiers
/regex/                    # Operator m can be omitted if forward-slashes are used as delimiter
/regex/modifiers
```

Delimiter

⇒ Instead of using forward-slashes (/) as delimiter, you could use other non-alphanumeric characters such as !, @ and % in the form of m!regex!modifiersm@regex@modifiers or m%regex%modifiers. However, if forward-slash (/) is used as the delimiter, the operator m can be omitted in the form of /*regex*/*modifiers*. Changing the default delimiter is confusing, and not recommended.

⇒ m//, by default, operates on the default variable $_. It returns true if $_ matches regex; and false otherwise.

Example: Regex [0-9]+

```
# try_m_1.pl
use strict;
use warnings;
while (<>)
{  # Read input from command-line into default variable $_
print m/[0-9]+/ ? "Accept\n" : "Reject\n";   # one or more digits?
}
```

Output:

```
123
Accept
00000
Accept
abc
Reject
abc123
Accept
```

Example: Extracting the Matched Substrings

⇒ The built-in array variables @- and @+ keep the start and end positions of the matched substring, where $-[0] and $+[0] for the full match, and $-[n] and $+[n] for back references $1, $2, ..., $n, ...

```
# try_m_2.pl
use strict;
use warnings;
while (<>)              # Read input from command-line into default variable $_
{
if (m/[0-9]+/)
{
print 'Accept substring: ' . substr($_, $-[0], $+[0] - $-[0]) . "\n";
}
else
{
print "Reject\n";
 }
}
```

Output:

```
123
Accept substring: 123
00000
Accept substring: 00000
abc
Reject
abc123xyz
Accept substring: 123
abc123xyz456
Accept substring: 123
```

Example: Modifier 'g' (global)

⇒ By default, m// finds only the first match. To find all matches, include 'g' (global) modifier.

```
# try_m_3.pl
```

```
use strict;
use warnings;
my $regex = '[0-9]+';            # Define regex pattern in non-interpolating string
while (<>)                       # Read input from command-line into default variable $_
{
# Do m//g and save matched substring into an array
my @matches = /$regex/g;
print "Matched substrings (in array): @matches\n";       # print array
  # Do m//g in a loop
print 'Matched substrings (in loop) : ';
while (/$regex/g)
{
printsubstr($_, $-[0], $+[0] - $-[0]), ',';
}
print "\n";
}
```

Output:

```
abc123xyz456_0_789
Matched substrings (in array): 123 456 0 789
Matched substrings (in loop) : 123,456,0,789,
abc
Matched substrings (in array):
Matched substrings (in loop) :
123
Matched substrings (in array): 123
Matched substrings (in loop) : 123,
```

Operators =~ and !~

⇒ By default, the matching operators operate on the default variable $_.

⇒ To operate on other variable instead of $_, you could use the =~ and !~ operators as follows:

```
str =~ m/regex/modifiers    # Return true if str matches regex.

str !~ m/regex/modifiers    # Return true if str does NOT match regex.
```

When used with m//, =~ behaves like comparison (== or eq).

Example: =~ Operator

```
# try_m_4.pl
use strict;
use warnings;
print 'yes or no? ';
my $reply;
chomp($reply = <>);     # Remove newline
print $reply =~ /^y/i ? "positive!\n" : "negative!\n";
```

```
# Begins with 'y', case-insensitive
```

Substitution Operator s///

⇒ You can substitute a string (or a portion of a string) with another string using s/// substitution operator. The syntax is:

s/*regex***/***replacement***/**

s/*regex***/***replacement***/***modifiers* # Optional modifiers

⇒ Similar to m//, s/// operates on the default variable $_ by default. To operate on other variable, you could use the =~and !~ operators. When used with s///, =~ behaves like assignment (=).

Example: s///

```
# try_s_1.pl
use strict;
use warnings;
while (<>)                  # Read input from command-line into default variable $_
{
s/w+/\*\*\*\*/g;            # Match each word
print "$_";
}
```

Output:

```
this is an apple.
*** *** *** ***.
```

Modifiers

⇒ Modifiers (such as /g, /i, /e, /o, /s and /x) can be used to control the behavior of m// and s///.

g (global): By default, only the first occurrence of the matching string of each line is processed. You can use modifier /g to specify *global* operation.

i (case-insensitive): By default, matching is case-sensitive. You can use the modifier /i to enable case in-sensitive matching.

m (multiline): multiline string, affecting position anchor ^, $, \A, \Z.

s: permits metacharacter. (dot) to match the newline.

Parenthesized Back-References & Matched Variables $1, ...,$9

⇒ Parentheses ( ) serve two purposes in regex:

1. Firstly, parentheses ( ) can be used to group sub-expressions for overriding the precedence or applying a repetition operator. For example, /(a|e|i|o|u){3,5}/ is the same as /a{3,5}|e{3,5}|i{3,5}|o{3,5}|u{3,5}/.
2. Secondly, parentheses are used to provide the so called *back-references*. A back-reference contains the matched sub-string. For examples, the regex /(\S+)/ creates one back-reference (\S+), which contains the first word (consecutive non-spaces) in the input string; the regex

/(\S+)\s+(\S+)/ creates two back-references: (\S+) and another (\S+), containing the first two words, separated by one or more spaces \s+.

⇒ The back-references are stored in special variables $1, $2, …, $9, where $1 contains the substring matched the first pair of parentheses, and so on. For example, /(\S+)\s+(\S+)/ creates two back-references which matched with the first two words. The matched words are stored in $1 and $2, respectively.

⇒ For example, the following expression swaps the first and second words:

```
s/(\S+) (\S+)/$2 $1/;   # Swap the first and second words separated by a single space
```

⇒ Back-references can also be referenced in your program.

⇒ For example,

```
(my $word) = ($str =~ /(\S+)/);
```

⇒ The parentheses creates one back-reference, which matches the first word of the $str if there is one, and is placed inside the scalar variable $word. If there is no match, $word is UNDEF.

⇒ Another example,

```
(my $word1, my $word2) = ($str =~ /(\S+)\s+(\S+)/);
```

⇒ The 2 pairs of parentheses place the first two words (separated by one or more white-spaces) of the $str into variables $word1 and $word2 if there are more than two words; otherwise, both $word1 and $word2 are UNDEF. Note that regular expression matching must be complete and there is no partial matching.

⇒ \1, \2, \3 has the same meaning as $1, $2, $3, but are valid only inside the s/// or m//. For example, /(\S+)\s\1/ matches a pair of repeated words, separated by a white-space.

Character Translation Operator tr///

⇒ You can use translator operator to translate a character into another character. The syntax is:

tr/*fromchars***/***tochars***/***modifiers*

replaces or translates *fromchars* to *tochars* in $_, and returns the number of characters replaced.

For example:

```
tr/a-z/A-Z/             # converts $_ to uppercase.
tr/dog/cat/             # translates d to c, o to a, g to t.
$str =~ tr/0-9/a-j/     # replace 0 by a, etc in $str.
tr/A-CG/KX-Z/           # replace A by K, B by X, C by Y, G by Z.
```

Instead of forward slash (/), you can use parentheses (), brackets [], curly bracket {} as delimiter, e.g.,

```
tr[0-9][##########]         # replace numbers by #.
tr{!.}(.!)                  # swap !and ., one pass.
```

If tochars is shorter than fromchars, the last character of tochars is used repeatedly.

```
tr/a-z/A-E/     # f to z is replaced by E.
```

tr/// returns the number of replaced characters. You can use it to count the occurrence of certain characters. For examples,

```
my $numLetters = ($string =~ tr/a-zA-Z/a-zA-Z/);
my $numDigits  = ($string =~ tr/0-9/0-9/);
my $numSpaces  = ($string =~ tr/ / /);
```

Modifiers /c, /d and /s for tr///

/c: complements (inverses) *fromchars*.
/d: deletes any matched but un-replaced characters.
/s: squashes duplicate characters into just one.

For example:

```
tr/A-Za-z/ /c        # replaces all non-alphabets with space
tr/A-Z//d            # deletes all uppercase (matched with no replacement).
tr/A-Za-z//dc        # deletes all non-alphabets
tr/!//s              # squashes duplicate !
```

String Functions: split and join

⇒ split(regex, str, [numItems]): Splits the given str using the regex, and return the items in an array. The optional third parameter specifies the maximum items to be processed.

⇒ join(joinStr, strList): Joins the items in strList with the given joinStr (possibly empty).

For example:

```
use strict;
use warnings;
my $msg = 'Hello, world again!';
my @words = split(/ /, $msg);        # ('Hello,', 'world', 'again!')
for (@words) { say; }                # Use default scalar variable
say join('--', @words);              # 'Hello,--world--again!'
my $newMsg = join '', @words;        # 'Hello,worldagain!'
say $newMsg;
```

Functions grep, map

⇒ grep(regex, array): selects those elements of the array, that matches regex.

⇒ map(regex, array): returns a new array constructed by applying regex to each element of the array.

Sub-routines

⇒ A subroutine is a block of code that can be reusable across programs.
⇒ You can define a subroutine anywhere in your program. If you have subroutines defined in another file, you can load them in your program by using the **use**, **do** or **require** statement.
⇒ A Perl subroutine can be generated at run-time by using the **eval()** function.

⇒ You can define your own subroutine (or functions) by using the keyword sub with a processing block:

```
subsubroutineName
{
statementBlock;
returnaReturnValue;
}
```

⇒ In Perl, subroutine returns a single piece of data or nothing, via statement **return**aReturnValue (or the last expression evaluated if there is no return statement).

⇒ You can invoke a subroutine by referencing it with an ampersand & before the subroutine name. (Recall that $ identifies a scalar; @ identifies an array, and % identifies a hash.)

Example:

```
# Define subroutine
sub hello
{
return 'Hello, world';
}
# Invoke subroutine
print&hello, "\n";
```

Passing arguments into subroutines:

⇒ Perl places the arguments into a special array variable named @_.

⇒ You can access the first element using $_[0], the second with $_[1], and so on. (Recall that $_ is the default scalar variable.)

⇒ You can use keyword **local** to define local variables or **my** to define lexical variables (available inside a block) for the subroutine, which hides the global version temporarily if there is one.

Example:

```
# Define a subroutine add which takes zero or more arguments
sub add
{
        my $sum = 0;
        foreach (@_)
        {
        $sum += $_;
        }
return $sum;
}
# Invoke subroutine add with various number of arguments
print&add(1), "\n";
print&add(2, 3), "\n";
print&add(4, 5, 6), "\n";
```

Perl's Built-in Functions of interest

Mathematical Functions:

⇒ sqrt(number): returns the square root of number.
⇒ abs(number): returns the absolute value of number.
⇒ sin(number): returns the sine of number, in radian.
⇒ cos(number): returns the cosine of number, in radian.
⇒ atan(y, x): returns the arc-tangent of y/x in the range of -π to π radians.
⇒ exp(number): returns the exponent of number.
⇒ log(number): returns the natural logarithm of number.

Converting between Number Bases:

⇒ ord(character) returns the ASCII value of character.
⇒ chr(number) returns the character given its ASCII number.
⇒ oct(number) returns the decimal value of the octal number.
⇒ hex(number) returns the decimal value of the hexadecimal number.

Error Reporting Functions - exit, die, warn

⇒ exit(number): exits the program with the status number. Normal termination of program exits with number 0.
⇒ die(string): exits the program with the current value of the special variable $! and prints string.
⇒ warn(string): prints the string but does not terminates the program.

Example:

```
exit unless open(HANDLE, $file)
open (HADNLE, $file) or die 'cannot open $file!\n';
```

Special Scalar Variable: Error Number $!

⇒ $! (or $ERRNO or $OS_ERROR) contains the system error. In numeric context, it contains the error number; in string context, it contains the error string.

Backquotes `*command*` and Function System

⇒ `*command*` executes command in a sub-shell and returns the command's output.

Example:

```
my $today = `date`;
print $today, "\n";
my @dirlines = `dir`;       # Use `ls -l` for Unix
foreach (@dirlines)
{
print;
}
```

⇒ system(program, args) executes the program with argument args and waits for it to return. system is similar to backquotes. However, backquotes return the output of the program; whereas system returns the exit code of the program (where 0 indicates normal termination). system lets the command go ahead and prints to the console. For example:

```
print system('date'), "\n";
print system('dir'), "\n";
```

Function sort()

⇒ sort(subroutine, array) sorts the array using the comparison function subroutine and returns the sorted array. Inside the subroutine, scalar variables $a and $b are automatically set to the two elements to be compared. If sort is used without the subroutine, it sorts according to string order. (Caution: By default, numbers are sorted as string, that is, the number 10 is less than 2 in string order).

Example:

```
use strict;
use warnings;
my @color = ('black', 'white', 'blue', 'green');
my @sorted = sort @color;
foreach (@sorted)
{
print "$_ ";
}
```

Example:

```
use strict;
use warnings;
# Define sorting subroutine
sub numerically { if ($a > $b) {1} elsif ($a < $b) {-1} else {0} }    # Compare numbers
my @price = (77, 100, 99, 55, 1);
my @sorted = sort numerically @price;
foreach (@sorted)
{
print "$_ ";
}
```

Example:

```
# A "spaceship" operator as the shorthand for the above because it is used very often
@sorted = sort { $a <=> $b } @price;
@sorted = sort @price;
foreach (@sorted)
{
print "$_ ";
```

```
}
```

Example:

```
use strict;
use warnings;
# Define sorting subroutine
sub alphabetically { lc($a) cmplc($b); } # Compare lowercase string
my @color = ('red', 'YELLOW', 'Blue', 'green');
my @sorted = sort alphabetically @color;
foreach (@sorted)
{
print "$_ ";
}
```

Random Number Functions srand() and rand()

⇒ srand(seed): initializes the random number generator with the seed. Use it once at the beginning of the program. If seed is omitted, the current time is used.

⇒ rand(number) returns a random floating-point number between 0 and number.

```
srand;
print rand(1), "\n";            # Generate a random number between 0.0 and 1.0
printint(rand(100)), "\n";      # Generate a random integer between 0 and 99
```

Time Functions time, localtime(), gmttime()

⇒ time: returns the number of second since January 1, 1970, GMT (Greenwich Mean Time).

⇒ localtime(time): converts the numeric time to time/day/date fields in the local time zone.

⇒ gmttime(time): converts the numeric time to time/day/date fields in GMT.

Function sleep()

⇒ sleep(number) makes the program wait for number of seconds before resuming execution.

Encryption Function crypt

⇒ crypt(password, salt) encrypts password with salt, and returns the encrypted password. crypt takes only the first 8 characters of the password for encryption. salt is up to 12 bits (or 16 bits?). The first 2 characters in the encrypted password are the salt. That is needed to verify the password.

Retrieving documents from the web with Perl

Solving a real-world problem with the LWP (Library of WWW modules in Perl)

⇒ Way back in 2004, a popular beverage company had a contest that involved collecting a number of points to earn prizes. These prizes were made available online, but limited quantities of specific prizes were available.

⇒ For the more popular items, these quantities were quickly depleted. In order to ensure that I was one of the lucky people to get the item i wanted - a gaming console - i needed a method to monitor the web page to see when the item became available. Perl to the rescue!

⇒ Using the LWP I was able to quickly create a script to look for certain text ("Now Available," for example) to appear on the page, and then send an e-mail alert when the text was found. With this script set to check every five minutes, I got the gaming console. Of course, this is just one example of how the LWP can be used to solve a real-world problem, albeit a simple one.

Introduction

⇒ Three things made the Web possible: HTML for encoding documents, HTTP for transferring them,and URLs for identifying them.

⇒ To fetch and extract information from web pages, you must know allthree - you construct a URL for the page you wish to fetch, make an HTTP request for it and decodethe HTTP response, then parse the HTML to extract information.

⇒ One can automate the most basic web tasks with the LWP module/library of Perl.

⇒ LWP is an abbreviation for library of WWW modules in Perl which provides a simple and consistent object oriented application programming interface (API) to the World-Wide-Web.The interface is easy to extend and customize for your own needs.

⇒ LWP modules enable you to incorporate common web tasks such as retrieving web pages, submitting web forms, authenticate, mirroring a web site and so on into your Perl program through a set of functions that canbe imported into your namespace.

⇒ The main features of the library are:

- Contains various reusable components (modules) that can be used separately or together.
- Provides an object oriented model of HTTP-style communication. Within this framework currently supported is access to http, https, gopher, ftp, news, file, and mailto resources.
- Provides a full object oriented interface or a very simple procedural interface.
- Supports the basic and digest authorization schemes.
- Supports transparent redirect handling.
- Supports access through proxy servers.
- Provides parser for robots.txt files and a framework for constructing robots.
- Supports parsing of HTML forms.
- Implements HTTP content negotiation algorithm that can be used both in protocol modules and in server scripts (like CGI scripts).
- Supports HTTP cookies.
- Some simple command line clients, for instance lwp-request and lwp-download.

Note:

⇒ To use the LWP modules, you need to first obtain and install them. The LWP modules are available from your favorite CPAN mirror.

⇒ The command to check whether the modules are already installed prior to going through the job of installing them is executed from the shell:

```
perl -MLWP -le "print(LWP->VERSION)"
```

⇒ You have the LWP installed if you see a version number such as this output:5.803

Keeping It Simple with LWP::Simple

⇒ LWP::Simple is a simple procedural interface to LWP.

⇒ It provides five functions that enable you to use the GET HTTP method very easily: get(), getprint(), getstore(), head(), and mirror(). They don't support cookies or authorization, setting header lines in HTTP request, reading header lines in HTTP response.

get():

⇒ The get function fetches the document/content identified by the given URL and returns it. It returns **undef** if it fails. The syntax is

```
get($url)
```

⇒ The $url argument can be either a string or a reference to a URI object.

Example: Retrieving the HTML to a variable

```
use strict;
use warnings;
use LWP::Simple;
my $content = get('http://www.perlmeme.org') or
                        die 'Unable to get page';
print $content;
exit 0;
```

getprint():

⇒ Sends the page whose URL you provided as argument to STDOUT; otherwise it complains to STDERR. The syntax is

```
getprint($url)
```

Example:

```
use strict;
use warnings;
use LWP::Simple;
getprint('http://www.perlmeme.org') or
                        die 'Unable to get page';
exit 0;
```

getstore():

⇒ This function writes the web page source directly to a file to the given name. The syntax is
getstore($url)

Example:

```
use strict;
use warnings;
use LWP::Simple;
getstore('http://www.perlmeme.org','test.html') or die 'Unable to get page';
exit 0;
```

Note: The getstore() function also returns the status of the GET method and sets is_success() if the status is in the 200 range. It sets is_error() if the status is in the 400 or 500 range. This effectively means that you can test to ensure that the GET request was successful by looking to see if is_success() is true.

head():

⇒ This function is normally used to test hyperlinks for validity and, when implemented by the server, returns the header information, never returns the body of the resource.
⇒ On success, it returns the following five values: the content type, document length, modification time, expiration time, and server. It returns an empty list if it fails. The syntax is
head($url)

Example:

```
use strict;
use warnings;
use LWP::Simple;
my ($content_type, $doc_length,  $mod_time, $expires, $server) =
                                        head('http://www.perlmeme.org');
print "Content type: $content_type\n";
print "Document length: $doc_length\n";
print "Modification time: $mod_time\n";
print "Server: $server\n";
exit 0;
```

Example: Link checking with HEAD

```
use strict;
use LWP::Simple;
foreach my $url ('http://us.a1.yimg.com/us.yimg.com/i/ww/m5v9.gif',
                        'http://hooboy.no-such-host.int/','http://www.yahoo.com',
                        'http://www.guardian.co.uk/',
                        'http://www.pixunlimited.co.uk/siteheaders/Guardian.gif',
                )
```

```
{
        print "\n$url\n";
        my ($type, $length, $mod) = head($url);
        # so we don't even save the expires or server values!
        unless (defined $type) {
        print "Couldn't get $url\n";
        next;
}
print "That $type document is ", $length || "???", " byteslong.\n";
if ($mod)
{
my $ago = time() - $mod;
print "It was modified $ago seconds ago; that's about ",int(.5 + $ago / (24 * 60 * 60)),
                                                    " days ago, at ",scalar(localtime($mod)), "!\n";
}
else
{
print "I don't know when it was last modified.\n";
}
}
```

Note: Unfortunately, the HEAD method is not supported by all web servers and is turned off byothers. This means that the use of the HEAD method is unreliable.

mirror():

⇒ The mirror() function works in much the same as the getstore() function, but also includes a check to compare the modification time of the local file and the modification time of the remote resource, based on the If-Modified-Since response header.

Example:

```
use strict;
use warnings;
use LWP::Simple;
my $url = "http://www.perlmeeme.org/";
my $file = "/tmp/perlmememirror";
my $status = mirror($url,$file);
die "Cannot retrieve $url" unless is_success($status);
```

⇒ The example program won't produce any output to the terminal unless there is an error.

Note: While LWP consists of dozens of classes, the two that you have to understand are LWP::UserAgent and HTTP::Response.

⇒ LWP::UserAgent is a class for "virtual browsers," which you use for performing requests.

⇒ HTTP::Response is a class for the responses (or error messages) that you get back from those requests.

LWP::UserAgent

⇒ The UserAgent plays a central role in web transactions.
⇒ The UserAgentis roughly synonymouswith the browser or client side of an HTTP request and response transaction. The basic syntax is
⇒ The UserAgentis frequently used to create a new **browser** object. This object can havea number of attributes set to define the behavior and operation of the resulting browserobject.

Key	Default
agent	"libwww-perl/#.###"
conn_cache	undef
cookie_jar	undef
from	undef
Keep_alive	No default
Max_redirect	7
max_size	undef
parse_head	1
protocols_allowed	undef
protocols_forbidden	undef
requests_redirectable	['GET', 'HEAD']
timeout	180

Table: Constructor options and default values for LWP::UserAgent

Define your User Agent:

⇒ To define user agent following is the way:

```
my $ua = LWP::UserAgent->new;
```

⇒ This is the object that acts as a browser and makes requests and receives responses.

Define the request

⇒ Next you need to create the request object that will be used to request the url. Since we are using the HTTP::Request::Common module, we can use the exported POST method.
⇒ It accepts a URL as its first parameter, and a list of arguments to be passed to the url (e.g. form arguments).

```
my $req = POST 'http://www.perlmeme.org', [];      OR
```

Passing form arguments:

```
my $req = POST 'http://www.perlmeme.org', [name => 'Bob', age => 24];
```

⇒ The GET method is used in a similar way to the first example:

```
my $req = GET 'http://www.perlmeme.org';
```

⇒ You can also pass header data to the GET and POST methods.

Making the request

⇒ Once you have defined your request object, use the UserAgent to make the request:

```
my $res = $ua->request($req);
```

⇒ The request method returns a HTTP::Response object. This object contains the status code of the response, and the content of the page if the request was successful.

The response

⇒ You can check if the request was successful by using the is_success method:

```
if ($res->is_success)
{
print $res->content;
}
Else
{
print $res->status_line . "\n";
}
```

User Agents

⇒ If you want your program to be represented as a particular agent, for example Mozilla 8.0, you can set this using the agent method:

```
$ua->agent('Mozilla/8.0');
```
OR,

For example, an Internet Explorer example:

```
$ua->agent('Mozilla/4.0 (compatible; MSIE 5.0; Windows 95)');
```

Example:

```
use strict;
use warning;
use LWP;
my $browser = LWP::UserAgent->new(agent=>'Mozilla');
print "the browser agent is ", $browser->agent(), "\n";
```

OR

```
use strict;
use warnings;
use LWP;
my $browser = LWP::UserAgent->new();
$browser->agent("Mozilla");
print "the browser agent is ", $browser->agent(), "\n";
```

Example:Setting a User Agent and retrieving aWeb Page

```
use LWP;
use strict;
my $browser = LWP::UserAgent->new(agent => 'Perly v1');
my $result = $browser->get("http://www.braingia.org/ewfojwefoj");
die "An error occurred: ", $result->status_line( ) unless
```

```
$result->is_success;
#Do something more meaningful with the content than this!
print $result->content;
```

⇒ The output from this program is raw HTML and JavaScript.

Example:

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
my $res = $ua->post('http://localhost/target.php', ['name'  =>  'Jan']);
if ($res->is_success)
{
        print $res->content . "\n";
}
else
{
        print $res->status_line . "\n";
}
```

⇒ The script sends a request with a name key having Jan value.

Hello Jan is the output of the above script.

Proxies

⇒ For whatever reason, you may want your requests to be made through a proxy. You can set different proxies for different protocols. Here is an example of setting a proxy for the ftp protocol:

```
$ua->proxy(ftp => 'http://some.proxy.com');
```

Cookies

⇒ Sometimes you will want your program to store the cookies created by retrieved web pages. The LWP bundle provides a HTTP::Cookies module that will handle cookies for you. You need to use this module:

```
use HTTP::Cookies;
```

And then set up a cookie_jar:

```
$au->cookie_jar(HTTP::Cookies->new(file => 'mycookies.txt',autosave => 1) );
```

⇒ LWP User Agent will now automatically store the cookies in the specified file, and they cookies will be available to future requests.

SSL

⇒ If you are requesting any urls using the SSL protocol (for example, a https page) you will first need to install an appropriate SSL module.

⇒ The two modules currently supported by LWP are Crypt::SSLeay and IO::Socket::SSL.

⇒ The Crypt::SSLeay module is preferred. Once you have installed either of these modules, you can request SSL encrypted urls just like other urls.

Example:

⇒ Below is a working script that requests a url and, if successful, prints the contents to standard out.

```
use strict;
use warnings;
use LWP::UserAgent;
use HTTP::Request::Common qw(GET);
use HTTP::Cookies;
my $ua = LWP::UserAgent->new;
# Define user agent type
$ua->agent('Mozilla/8.0');
# Cookies
$ua->cookie_jar(HTTP::Cookies->new(file => 'mycookies.txt', autosave => 1) );
# Request object
my $req = GET 'http://www.perlmeme.org';
# Make the request
my $res = $ua->request($req);
# Check the response
if ($res->is_success)
{
        print $res->content;
}
else
{
        print $res->status_line . "\n";
}
exit 0;
```

Complete List of LWP classes

Module	Description
File::Listing	Module for parsing directory listings. Used by Net::FTP.

HTML::Form	Class for objects representing HTML forms.
HTML::FormatPS	Class for objects that can render HTML::TreeBuilder tree contents as PostScript.
HTML::Formatter	Internal base class for HTML::FormatPS and HTML::FormatText.
*HTML::FormatText	Class for objects that can render HTML::TreeBuilder tree contents as plain text.
*HTML::Entities	Useful module providing functions that &-encode/decode strings (such as C.&E.Brontë to and from C.&E.Brontë).
HTML::Filter	Deprecated class for HTML parsers that reproduce their input by default.
HTML::HeadParser	Parse <HEAD> section of an HTML document.
HTML::LinkExtor	Class for HTML parsers that parse out links.
HTML::PullParser	Semi-internal base class used by HTML::TokeParser.
*HTML::TokeParser	Friendly token-at-a-time HTML pull-parser class.
HTML::Parser	Base class for HTML parsers; used by the friendlier HTML::TokeParser and HTML::TreeBuilder.
HTML::AsSubs	Semi-deprecated module providing functions that each construct an HTML::Element object.
*HTML::Element	Class for objects that each represent an HTML element.
HTML::Parse	Deprecated module that provides functions accessing HTML::TreeBuilder.
HTML::Tree	Module that exists just so you can run perldocHTML-Tree.
*HTML::TreeBuilder	Class for objects representing an HTML tree into which you can parse source.
*HTTP::Cookies	Class for objects representing databases of cookies.
HTTP::Daemon	Base class for writing HTTP server daemons.
HTTP::Date	Module for date conversion routines. Used by various LWP protocol modules.
HTTP::Headers	Class for objects representing the group of headers in an HTTP::Response or HTTP::Request object.
HTTP::Headers::Auth	Experimental/internal for improving HTTP::Headers's authentication support.
HTTP::Headers::ETag	Experimental/internal module adding HTTP ETag support to HTTP::Headers.
HTTP::Headers::Util	Module providing string functions used internally by various other LWP modules.
*HTTP::Message	Base class for methods common to HTTP::Response and HTTP::Request.
HTTP::Negotiate	Module implementing an algorithm for content negotiation. Not widely used.
HTTP::Request	Class for objects representing a request that carried out with an

	LWP::UserAgent object.
HTTP::Request::Common	Module providing functions used for constructing common kinds of HTTP::Request objects.
*HTTP::Response	Class for objects representing the result of an HTTP::Request that was carried out.
*HTTP::Status	Module providing functions and constants involving HTTP status codes.
*LWP	Module that exists merely so you can say "use LWP" and have all the common LWP modules (notably LWP::UserAgent, HTTP::Request, and HTTP::Response). Saying "use LWP5.64" also asserts that the current LWP distribution had better be Version 5.64 or later. The module also contains generous documentation.
LWP::Authen::Basic	Module used internally by LWP::UserAgent for doing common ("Basic") HTTP authentication responses.
LWP::Authen::Digest	Module used internally by LWP::UserAgent for doing less-common HTTP Digest authentication responses.
LWP::ConnCache	Class used internally by some LWP::Protocol::protocol modules to reuse socket connections.
*LWP::Debug	Module for routines useful in tracing how LWP performs requests.
LWP::MediaTypes	Module used mostly internally for guessing the MIME type of a file or URL.
LWP::MemberMixin	Base class used internally for accessing object attributes.
LWP::Protocol	Mostly internal base class for accessing and managing LWP protocols.
LWP::Protocol::data	Internal class that handles the new data: URL scheme (RFC 2397).
LWP::Protocol::file	Internal class that handles the file: URL scheme.
LWP::Protocol::ftp	Internal class that handles the ftp: URL scheme.
LWP::Protocol::GHTTP	Internal class for handling http: URL scheme using the HTTP::GHTTP library.
LWP::Protocol::gopher	Internal class that handles the gopher: URL scheme.
LWP::Protocol::http	Internal class that normally handles the http: URL scheme.
LWP::Protocol::http10	Internal class that handles the http: URL scheme via just HTTP v1.0 (without the 1.1 extensions and features).
LWP::Protocol::https	Internal class that normally handles the https: URL scheme, assuming you have an SSL library installed.
LWP::Protocol::https10	Internal class that handles the https: URL scheme, if you don't want HTTP v1.1 extensions.
LWP::Protocol::mailto	Internal class that handles the mailto: URL scheme; yes, it sends mail!
LWP::Protocol::nntp	Internal class that handles the nntp: and news: URL schemes.
LWP::Protocol::nogo	Internal class used in handling requests to unsupported protocols.

*LWP::RobotUA	Class based on LWP::UserAgent, for objects representing virtual browsers that obey robots.txt files and don't abuse remote servers.
*LWP::Simple	Module providing the get, head, getprint, getstore, and mirror shortcut functions.
*LWP::UserAgent	Class for objects representing "virtual browsers."
Net::HTTP	Internal class used for HTTP socket connections.
Net::HTTP::Methods	Internal class used for HTTP socket connections.
Net::HTTP::NB	Internal class used for HTTP socket connections with nonblocking sockets.
Net::HTTPS	Internal class used for HTTP Secure socket connections.
*URI	Main class for objects representing URIs/URLs, relative or absolute.
URI::_foreign	Internal class for objects representing URLs for schemes for which we don't have a specific class.
URI::_generic	Internal base class for just about all URLs.
URI::_login	Internal base class for connection URLs such as telnet:,rlogin:, and ssh:.
URI::_query	Internal base class providing methods for URL types that can have query strings (such as foo://...?bar).
URI::_segment	Internal class for representing some return values from $url->path_segments( ) calls.
URI::_server	Internal base class for URL types where the first bit represents a server name (most of them except mailto:).
URI::_userpass	Internal class providing methods for URL types with an optional user[:pass] part (such as ftp://itsme:foo@secret.int/).
URI::data	Class for objects representing the new data: URLs (RFC 2397).
*URI::Escape	Module for functions that URL-encode and URL-decode strings (such as potpie to and from pot%20pie).
URI::file	Class for objects representing file: URLs.
URI::file::Base	Internal base class for file: URLs.
URI::file::FAT	Internal base class for file: URLs under legacy MSDOS (with 8.3 filenames).
URI::file::Mac	Internal base class for file: URLs under legacy (before v10) MacOS.
URI::file::OS2	Internal base class for file: URLs under OS/2.
URI::file::QNX	Internal base class for file: URLs under QNX.
URI::file::Unix	Internal base class for file: URLs under Unix.
URI::file::Win32	Internal base class for file: URLs under MS Windows.
URI::ftp	Class for objects representing ftp: URLs.
URI::gopher	Class for objects representing gopher: URLs.
URI::Heuristic	Module for functions that expand abbreviated URLs such as *ora.com*.
URI::http	Class for objects representing http: URLs.

URI::https	Class for objects representing https: URLs.
URI::ldap	Class for objects representing ldap: URLs.
URI::mailto	Class for objects representing mailto: URLs.
URI::news	Class for objects representing news: URLs.
URI::nntp	Class for objects representing nntp: URLs.
URI::pop	Class for objects representing pop: URLs.
URI::rlogin	Class for objects representing rlogin: login URLs.
URI::rsync	Class for objects representing rsync: URLs.
URI::snews	Class for objects representing snews: (Secure News) URLs.
URI::ssh	Class for objects representing ssh: login URLs.
URI::telnet	Class for objects representing telnet: login URLs.
URI::URL	Deprecated class that is like URI; use URI instead.
URI::WithBase	Like the class URI, but objects of this class can "remember" their base URLs.
WWW::RobotsRules	Class for objects representing restrictions parsed from various *robots.txt* files.
WWW::RobotRules::AnyDBM_File	Subclass of WWW::RobotRules that uses a DBM file to cache its contents.

References:

1. https://www.perltutorial.org/
2. https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/Perl1_Basics.html
3. https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/Perl2_Regexe.html
4. http://perlmeme.org/tutorials/lwp.html
5. OReilly Perl & LWP by Sean M Burke: Chapter 2 and Chapter 3
6. Beginning Perl Web Development: Chapter 5 – Internet Interaction with LWP
7. https://resources.oreilly.com/examples/9780596001780/

*******END*******