

Perl Introduction

- Perl is a programming language which was originally developed for script manipulation. But now Perl is used for a variety of purpose including web development, GUI development, system administration and many more. It is a stable, cross platform programming language.
- For web development, Perl CGI is used. CGI is the gateway which interacts with the web browser and Perl in a system.
- Its typical use is extracting information from a text file and printing out report for converting a text file into another form. This is because it got its name after the expression, "**P**ractical **E**xtraction and **R**eport **L**anguage".
- Programs written in Perl are called **Perl scripts**, whereas system programs to execute Perl scripts are called **Perl program**.
- Perl is an interpreted language. When a Perl program run, it is first compiled into a byte code, then it is converted into machine instructions. So writing something in Perl instead of C saves your time.
- It supports most of the operating systems and is listed in Oxford English dictionary. Its concepts and syntax is taken from many languages like awk, bourne shell, C, sed and even English.

Perl History

- Perl was developed by Larry Wall in 1987 as a scripting language to make report processing easier.
- It was first released with version 1.0 on December 18, 1987.
- Perl 2, released in 1988 adding a much better regular expression engine.
- Perl 3, released in 1989 adding support for binary data streams.
- Perl 4, released in 1991 with a better documentation than earlier.
- Perl 5, released on October 17, 1994. It added many new features to its last version like objects, variables, references and modules.
- The latest version 5.24 is released on May 9, 2016.

Perl Features

- It has a very simple Object-oriented programming syntax.
- It is easily extendible as it supports 25,000 open source modules.
- It supports Unicode.
- It includes powerful tools to process text to make it compatible with mark-up languages like HTML, XML.
- It supports third party database including Oracle, MySQL and many others.
- It is embeddable in other systems such as web servers and database servers.
- It is open source software licensed under GNU.
- Many frameworks are written in Perl.
- It can handle encrypted web data including e-commerce transactions.
- It is a cross platform language.
- It offers a regular expression engine which is able to transform any type of text.

Scripts and programs:

- Scripting is the action of writing scripts using a scripting language, distinguishing neatly between programs, which are written in conventional programming language such as C,C++,java, and scripts, which are written using a different kind of language.
- We could reasonably argue that the use of scripting languages is just another kind of programming.
- Scripting languages are used from the ground up, employing a team of professional programmers, starting from well-defined specifications, and meeting for is qualitatively different from conventional programming languages like C++ and Ada address the problem of developing large applications specified performance constraints.

- Scripting languages, on other hand, address different problems:
 - Building applications from 'off the shelf' components
 - Controlling applications that have a programmable interface
 - Writing programs where speed of development is more important than run-time efficiency.
- The most important difference is that scripting languages incorporate features that enhance the productivity of the user in one way or another, making them accessible to people who would not normally describe themselves as programmers, their primary employment being in some other capacity.
- Scripting languages make programmers of us all, to some extent.

Origin of scripting:

The use of the word 'script' in a computing context dates back to the early 1970s, when the originators of the UNIX operating system create the term 'shell script' for sequence of commands that were to be read from a file and follow in sequence as if they had been typed in at the keyboard. e.g. an 'AWKscript', a 'perl script' etc.. the name 'script' being used for a text file that was intended to be executed directly rather than being compiled to a different form of file prior to execution.

Other early occurrences of the term 'script' can be found. For example, in a DOS-based system, use of a dial-up connection to a remote system required a communication package that used proprietary language to write scripts to automate the sequence of operations required to establish a connection to a remote system.

Note that if we regard a scripts as a sequence of commands to control an application or a device, a configuration file such as a UNIX 'make file' could be regard as a script.

However, scripts only become interesting when they have the added value that comes from using programming concepts such as loops and branches.

Scripting today:

SCRIPTING IS USED WITH 3 DIFFRENT MEANINGS:

1. A new style of programming which allows applications to be developed much faster than traditional methods allow, and makes it possible for applications to evolve rapidly to meet changing user requirements. This style of programming frequently uses a scripting language to interconnect 'off the shelf' components that are themselves written in conventional language. Applications built in this way are called 'glue applications', and the language is called a 'glue language'.

A glue language is a programming language (usually an interpreted scripting language) that is designed or suited for writing glue code – code to connect software components. They are especially useful for writing and maintaining:

Custom commands for a command shell
Smaller programs than those that are better implemented in a compiled language
"Wrapper" programs for executables, like a batch file that moves or manipulates files and does other things with the operating system before or after running an application like a word processor, spreadsheet, data base, assembler, compiler, etc. Scripts that may change Rapid prototypes of a solution eventually implemented in another, usually compiled, language.

Glue language examples:

- | | | |
|--------------------------|---------------|---|
| • AppleScript | • Lua | • Scheme |
| • ColdFusion | • m4 | • Tcl |
| • DCL | • Perl | • Unix Shell |
| • Embeddable Common Lisp | • PHP | • scripts (ksh, csh, bash, sh and others) |
| • ecl | • Pure Python | • VBScript |
| • Erlang | • Rebol | • Work Flow Language |
| • JCL | • Rexx | • Windows PowerShell |
| • JScript and JavaScript | • Ruby | |

2.Using a scripting language to 'manipulate,customize and automate the facilities of an existing system',as the ECMAScript definition puts it.Here the script is used to control an application that provides a programmable interface:this may be an API,though more commonly the application is constructed from a collection of objects whose properties and methods are exposed to the scripting language.Example: use of Visual Basic for applications to control the applications in the Microsoft Office Suite.

3.Using a scripting language with its rich functionality and ease of use as an alternate to a conventional language for general programming tasks ,particularly system programming and administration.Examples: are UNIX system administrators have for a long time used scripting languages for system maintenance tasks,and administrators of WINDOWS NT systems are adopting a scripting language ,PERL for their work.

Characteristics of scripting languages:

These are some properties of scripting languages which differentiate SL from programming languages.

- **Integrated compile and run:**

SL's are usually characterized as interpreted languages,but this is just an oversimplification.They operate on an immediate execution,without need to issue separate command to compile the program and then to run the resulting object file,and without the need to link extensive libraries into the object code.This is done automatically.A few SL'S are indeed implemented as strict interpreters.

- **Low overheads and ease of use:**

- 1.variables can be declared by use
- 2.the number of different data types is usually limited
- 3.everything is string by context it will be converted as number(vice versa)
- 4.number of data structures is limited(arrays)

- **Enhanced functionality:**

SL's usually have enhanced functionality in some areas.For example ,most languages provide string manipulation based on the use of regular expressions,while other languages provide easy access to low-level operating system facilities,or to the API,or object exported by an application.

- **Efficiency is not an issue:**

Ease of use is achieved at the expense of efficiency,because efficiency is not an issue in the applications for which SL'S are designed.

- A scripting language is usually interpreted from source code or bytecode. By contrast, the software environment the scripts are written for is typically written in a compiled language and distributed in machine code form.
- Scripting languages may be designed for use by end users of a program – end-user development – or may be only for internal use by developers, so they can write portions of the program in the scripting language.
- Scripting languages typically use abstraction, a form of information hiding, to spare users the details of internal variable types, data storage, and memory management.
- Scripts are often created or modified by the person executing them, but they are also often distributed, such as when large portions of games are written in a scripting language.
- The characteristics of ease of use,particularly the lack of an explicit compile-link-load sequence,are sometimes taken as the sole definition of a scripting language.

Users For Scripting Languages:

Users are classified into two types

1. Modern applications
2. Traditional users

Modern applications of scripting languages are:

- **1. Visual scripting:**

A collection of visual objects is used to construct a graphical interface.This process of constructing a graphical interface is known as visual scripting.the properties of visual objects include text on button,background and foreground colors.These properties of objects

can be changed by writing program in a suitable language. The outstanding visual scripting system is visual basic. It is used to develop new applications. Visual scripting is also used to create enhanced web pages.

2. Scripting components:

In scripting languages we use the idea to control the scriptable objects belonging to scripting architecture. Microsoft's visual basic and excel are the first applications that used the concept of scriptable objects. To support all the applications of microsoft the concept of scriptable objects was developed. 3. Web scripting: web scripting is classified into three forms. They are processing forms, dynamic web pages, dynamically generating HTML.

Applications of traditional scripting languages are:

1. System administration,
2. Experimental programming,
3. Controlling applications.

Application areas:

Four main usage areas for scripting languages:

1. Command scripting languages
2. Application scripting languages
3. Markup language
4. Universal scripting languages

1. Command scripting languages are the oldest class of scripting languages. They appeared in 1960, when a need for programs and tasks control arose. The most known language from the first generation of such languages is JCL (Job Control Language), created for IBM OS/360 operating system. Modern examples of such languages include shell language, described above, and also text-processing languages, such as sed and awk. These languages were one of the first to directly include support for regular expression matching - a feature that later was included into more general-purpose languages, such as Perl.

2. Application scripting languages: Application scripting languages were developed in 1980s, in the era of personal computers, when such important applications as spreadsheets and database clients were introduced, and interactive session in front of the PC became the norm. One of the prime examples of these languages is Microsoft-created Visual Basic language, and especially its subset named Visual Basic for Applications, designed explicitly for office applications programming

3. Markup languages are a special case in the sense that they are not a real programming languages, but rather a set of special command words called 'tags' used to mark up parts of text documents, that are later used by special programs called processors, to do all kinds of transformations to the text, such as displaying it in a browser, or converting it to some other data format. The basic idea of markup languages is the separation of contents and structure, and also including formatting commands and interactive objects into the documents. The first markup language named GML (Generic Markup Language) was created in 1969 by IBM. In 1986, ISO created a standard called SGML, based on GML ideas

4. Universal scripting languages: The languages that belong to this class are perhaps the most well-known. The very term "scripting languages" is associated with them. Most of these languages were originally created for the Unix environment. The goals however were different. The Perl programming language was made for report generation, which is even reflected in its name (Practical Extraction and Report Language). It is commonly said that the primary reason for its enormous popularity is the ability to write simple and efficient CGI scripts for forming dynamic web pages with this language. Perl was there in the right place at the right time. The Python language was originally made as a tool for accessing system services of the experimental operating system Amoeba. Later it became a universal object-oriented scripting language. Implementations exist for the Java Virtual Machine and also for Microsoft Intermediate Language used on Microsoft .NET platform. Unlike Perl and Python, which make it easy to write completely standalone programs, Tcl relies heavily on C and C++ extension modules.

Web scripting:

Web is the most fertile areas for the application of scripting languages. Web scripting divides into three areas

- Processing forms
- Creating pages with enhanced visual effects and user interaction and
- Generating pages 'on the fly' from material held in database.

Processing Web forms:

- In the original implementation of the web, when the form is submitted for processing, the information entered by the user is encoded and sent to the server for processing by a CGI script that generates an HTML page to be sent back to the Web browser.
- This processing requires string manipulation to construct the HTML page that constitutes the replay, and may also require system access, to run other processes and to establish network connections. Perl is also a language that uses CGI scripting.
- Alternatively for processing the form with script running on the server it possible to do some client -side is processing within the browser to validate form data before sending it to the server by using JavaScript, VBScript etc.

Dynamic Web pages:

'Dynamic HTML' makes every component of a Web page (headings, anchors, tables etc.) a scriptable object. This makes it possible to provide simple interaction with the user using scripts written in JavaScript/Jscript or VBScript, which are interpreted by the browser.

Microsoft's ActiveX technology allows the creation of pages with more elaborate user interaction by using embedded visual objects called ActiveX controls. These controls are scriptable objects, and can in fact be scripted in a variety languages. This can be scripted by using Perl scripting engine.

Dynamically generated HTML:

Another form of dynamic Web page is one in which some or all of the HTML is generated by scripts executed on the server. A common application of the technique is to construct pages whose content is retrieved from a database. For example, Microsoft's IIS web server implements Active Server Pages (ASP), which incorporate scripts in Jscript or VBScript.

The universe of scripting languages:

Scripting can be traditional or modern scripting, and Web scripting forms an important part of modern scripting. Scripting universe contains multiple overlapping worlds:

- the original UNIX world of traditional scripting using Perl and Tcl
- the Microsoft world of Visual Basic and Active controls
- the world of VBA for scripting compound documents
- The world of client-side and server-side Web scripting.

The overlap is complex, for example web scripting can be done in VBScript, JavaScript/Jscript, Perl or Tcl. This universe has been enlarged as Perl and Tcl are used to implement complex applications for large organizations e.g Tcl has been used to develop a major banking system, and Perl has been used to implement an enterprisewide document management system for a leading aerospace company.

=====

Names and Values in Perl:

Names:

Like any other programming language, Perl manipulates variables which have a name (or identifier) and a value: a value is assigned to a variable by an assignment statement of the form

name=value;

Variable names resemble nouns in English, and like English, Perl distinguishes between singular and plural nouns. A singular name is associated with a variable that holds a single item of data (a scalar value), a plural name is associated with a variable that holds a

collection of data items (an array or hash). A notable characteristic of Perl is that variable names start with a special character that denotes the kind of thing that the name stands for - scalar data (\$), array (@), hash (%), subroutine (&) etc. The syntax also allows a name that consists of a single non-alphanumeric character after the initial special character, eg. \$\$, \$?; such names are usually reserved for the Perl system.

If we write an assignment, eg. `j=j+1`, the occurrence of `j` on the left denotes a storage location, while the right-hand occurrence denotes the contents of the storage location. We sometimes refer to these as the lvalue and rvalue of the variable: more precisely we are determining the meaning of the identifier in a left-context or a right-context. In the assignment `a[j] = a[j] + 1`, both occurrences of `j` are determined in a right-context, even though one of them appears on the left of the assignment.

In conventional programming languages, new variables are introduced by a declaration, which specifies the name of the new variable and also its type, which determines the kind of value that can be stored in the variable and, by implication, the operations that can be carried out on that variable.

Perl Variables

A variable is a place to store values. They can be manipulated throughout the program. When variables are created they reserve some memory space.

There are three types of variables:

- Scalar defined by \$
- Arrays defined by @
- Hashes defined by %

The same variable can be used for all these three types of variables in a program. For example, `$name`, `@name` and `%name`, all three variables will be considered different in a program.

Perl Variable Declaration

The equal sign (=) is used to assign values to variables. At the left of (=) is the variable name and on the right it is the value of the variable.

- `$name = "Anastasia";`
- `$rank = "9th";`
- `$marks = 756.5;`

Here we have created three variables `$name`, `$rank` and `$marks`.

Perl use strict

If you are using **use strict** statement in a program, then you have to declare your variable before using it. It is mandatory. Otherwise you'll get an error.

The `$a` and `$b` are special variables used in perl sort function. There is no need to declare these two variables. So it is recommended not to use these two variables except in connection to **sort**.

Variables can be declared using **my**, **our**, **use vars**, **state** and **\$person::name** (explicit package name). Although, they all have different meanings.

```

1. use 5.010;
2. use strict;
3. my $x = 23;
4. say $x;
5. state $name = "Anastasia";
6. say $name;
7. our $rank = "9th";
8. say $rank;
9. use vars qw($marks);
10. $marks = 756.5;
11. say $marks;
12. $Person::name = 'John';
13. say $Person::name;
14. $a = 1224365;
15. say $a;
16. $b = "Welcome at CSE";
17. say $b;

```

Output:

```

23
Anastasia
9th
756.5
John
1224365
Welcome at CSE

```

Look at the above output, the last two variable are \$a and \$b. so we have not defined them. Yet their output is displayed.

Perl - Scalars

A scalar is a single unit of data. That data might be an integer number, floating point, a character, a string, a paragraph, or an entire web page.

Here is a simple example of using scalar variables –

```

#!/usr/bin/perl

$age = 25;           # An integer assignment
$name = "John Paul"; # A string
$salary = 1445.50;   # A floating point

print "Age = $age\n";
print "Name = $name\n";
print "Salary = $salary\n";

```

This will produce the following result –

```

Age = 25
Name = John Paul
Salary = 1445.5

```

Numeric Scalars

A scalar is most often either a number or a string. Following example demonstrates the usage of various types of numeric scalars –

```

#!/usr/bin/perl

$integer = 200;
$negative = -300;

```

```

$floating = 200.340;
$bigfloat = -1.2E-23;

# 377 octal, same as 255 decimal
$octal = 0377;

# FF hex, also 255 decimal
$hexa = 0xff;

print "integer = $integer\n";
print "negative = $negative\n";
print "floating = $floating\n";
print "bigfloat = $bigfloat\n";
print "octal = $octal\n";
print "hexa = $hexa\n";

```

This will produce the following result –

```

integer = 200
negative = -300
floating = 200.34
bigfloat = -1.2e-23
octal = 255
hexa = 255

```

String Scalars

Following example demonstrates the usage of various types of string scalars. Notice the difference between single quoted strings and double quoted strings –

```

#!/usr/bin/perl

$var = "This is string scalar!";
$quote = 'I m inside single quote - $var';
$double = "This is inside single quote - $var";

$escape = "This example of escape -\tHello, World!";

print "var = $var\n";
print "quote = $quote\n";
print "double = $double\n";
print "escape = $escape\n";

```

This will produce the following result –

```

var = This is string scalar!
quote = I m inside single quote - $var
double = This is inside single quote - This is string scalar!
escape = This example of escape -      Hello, World

```

Scalar Operations

You will see a detail of various operators available in Perl in a separate chapter, but here we are going to list down few numeric and string operations.

```

#!/usr/bin/perl
$str = "hello" . "world";      # Concatenates strings.
$num = 5 + 10;                 # adds two numbers.
$mul = 4 * 5;                  # multiplies two numbers.
$mix = $str . $num;            # concatenates string and number.

print "str = $str\n";
print "num = $num\n";
print "mul = $mul\n";

```

```
print "mix = $mix\n";
```

This will produce the following result –

```
str = helloworld  
num = 15  
mul = 20  
mix = helloworld15
```

Multiline Strings

If you want to introduce multiline strings into your programs, you can use the standard single quotes as below –

```
#!/usr/bin/perl  
  
$string = 'This is  
a multiline  
string';  
  
print "$string\n";
```

This will produce the following result –

```
This is  
a multiline  
string
```

You can use "here" document syntax as well to store or print multilines as below –

```
#!/usr/bin/perl  
  
print <<EOF;  
This is  
a multiline  
string  
EOF
```

This will also produce the same result –

```
This is  
a multiline  
string
```

V-Strings

A literal of the form v1.20.300.4000 is parsed as a string composed of characters with the specified ordinals. This form is known as v-strings.

A v-string provides an alternative and more readable way to construct strings, rather than use the somewhat less readable interpolation form "\x{1}\x{14}\x{12c}\x{fa0}".

They are any literal that begins with a v and is followed by one or more dot-separated elements. For example –

```
#!/usr/bin/perl  
  
$smile = v9786;  
$foo = v102.111.111;  
$martin = v77.97.114.116.105.110;  
  
print "smile = $smile\n";  
print "foo = $foo\n";  
print "martin = $martin\n";
```

This will also produce the same result –

```
smile = ☺  
foo = foo  
martin = Martin  
Wide character in print at main.pl line 7.
```

Special Literals

So far you must have a feeling about string scalars and its concatenation and interpolation operation. So let me tell you about three special literals `__FILE__`, `__LINE__`, and `__PACKAGE__` represent the current filename, line number, and package name at that point in your program.

They may be used only as separate tokens and will not be interpolated into strings. Check the below example —

```
#!/usr/bin/perl  
  
print "File name " . __FILE__ . "\n";  
print "Line Number " . __LINE__ . "\n";  
print "Package " . __PACKAGE__ . "\n";  
  
# they can not be interpolated  
print "__FILE__ __LINE__ __PACKAGE__\n";
```

This will produce the following result –

```
File name hello.pl  
Line Number 4  
Package main  
__FILE__ __LINE__ __PACKAGE__
```

PERL CONTROL STRUCTURE/STATEMENTS

Perl if-else Statement

The if statement in Perl language is used to perform operation on the basis of condition. By using if-else statement, you can perform operation either condition is true or false. Perl supports various types of if statements:

- If
- If-else
- If else-if

Perl If Example

The Perl single if statement is used to execute the code if condition is true. The syntax of if statement is given below:

- `if(expression){`
- `//code to be executed`
- `}`

Let's see a simple example of Perl language if statement.

- `$a = 10;`
- `if( $a %2==0 ){`
- `printf "Even Number\n";`
- `}`
- `}`

Output:

- Even Number

Here, output is even number as we have given input as 10.

Perl If-else Example

The Perl if-else statement is used to execute a code if condition is true or false. The syntax of if-else statement is given below:

- `if(expression){`
- `//code to be executed if condition is true`
- `}else{`
- `//code to be executed if condition is false`
- `}`

Let's see the simple example of even and odd number using if-else statement in Perl language.

- `$a = 10;`
- `if( $a %2==0 ){`
- `printf "Even Number\n";`
- `}else{`
- `printf "Odd Number\n";`
- `}`

Output:

```
Even Number
```

Here, input is an even number and hence output is even.

Perl If-else Example with Input from user

In this example, we'll take input from user by using standard input (<STDIN>/<>).

- `print "Enter a Number?\n";`
- `$num = <>;`
- `if( $num %2==0 ){`
- `printf "Even Number\n";`
- `}else{`
- `printf "Odd Number\n";`
- `}`

Output:

```
Enter a Number?
```

```
5
```

```
Odd Number
```

```
Enter a Number?
```

```
4
```

```
Even Number
```

In the first output, user has entered number 5 which is odd. Hence the output is odd.

In the second output, user has entered number 4 which is even. Hence the output is even.

Perl If else-if Example

The Perl if else-if statement executes one code from multiple conditions. The syntax of if else-if statement is given below:

- `if(condition1){`
- `//code to be executed if condition1 is true`
- `}else if(condition2){`
- `//code to be executed if condition2 is true`
- `}`
- `else if(condition3){`
- `//code to be executed if condition3 is true`
- `}`
- `...`

- **else**{
- *//code to be executed if all the conditions are false*
- }

The example of if else-if statement in Perl language is given below.

- print "Enter a Number to check grade\n";
- \$num = <>;
- **if**(\$num < 0 || \$num > 100){
- printf "Wrong Number\n";
- }elseif(\$num >= 0 && \$num < 50){
- printf "Fail\n";
- }elseif(\$num >= 50 && \$num < 60){
- printf "D Grade\n";
- }elseif(\$num >= 60 && \$num < 70){
- printf "C Grade\n";
- }elseif(\$num >= 70 && \$num < 80){
- printf "B Grade\n";
- }elseif(\$num >= 80 && \$num < 90){
- printf "A Grade\n";
- }elseif(\$num >= 90 && \$num <= 100){
- printf "A+ Grade\n";
- }

Output:

```
Enter a Number to check grade
66
C Grade
Enter a Number to check grade
2
Wrong Number
```

Perl Switch Statement

The perl switch statement is used to execute the code from multiple conditions. There is no case or switch statement in perl. Instead we use 'when' in place of case and 'given' in place of case.

The syntax of switch statement in perl language is given below:

- given(expression){
- when (value1)
- *{//code to be executed;}*
- when (value2)
- *{//code to be executed;}*
- when (value3)
- *{//code to be executed;}*
- **default**
- *{//code to be executed if all the cases are not matched.}*
- }

Perl Switch Example

Let's see a simple example of c language switch statement.

- **use** feature qw(**switch** say);
- print "Enter a Number\n ";
- chomp(my \$grade = <>);
- given (\$grade) {
- when ('10') { say 'It is 10' ;}

- `when ('20') { say 'It is 20' ;}`
- `when ('30') { say 'It is 30' ;}`
- `default { say 'Not 10, 20 or 30';}`
- `}`

Output:

```
Enter a Number
10
It is 10
Enter a Number
55
'Not 10, 20 or 30'
```

Perl goto Statement

The Perl goto statement is the jump statement. It is used to transfer control by jumping to other label inside a loop.

There are three goto forms:

goto LABEL:

It jumps to the statement labeled with LABEL and resumes normal execution from there.

goto EXPR:

It is a generalization of goto LABEL. The expression returns a label name and then jumps to that labeled statement.

goto &NAME:

For currently running subroutine it substitutes a call to the named subroutine.

Syntax of Perl goto statement is as follows:

- `goto LABEL`
-
- `or`
-
- `goto EXPR`
-
- `or`
-
- `goto &NAME`

Perl goto Statement Example

Let's see a simple example to use goto statement in Perl language.

- `LOOP:do`
- `print "You are not eligible to vote!\n";`
- `print "Enter your age\n";`
- `$age = <>;`
- `if( $age < 18){`
- `goto LOOP;`
- `}`
- `else{`
- `print "You are eligible to vote\n";`
- `}`

Output:

```
You are not eligible to vote!
Enter your age:
11
You are not eligible to vote!
Enter your age:
5
You are not eligible to vote!
```

Enter your age:

26

You are eligible to vote!

Perl Comments

Comments are a necessity for a developer of any language. It is a good habit to comment in your code. They don't take part in program execution. They just provide us an explanation and hints about our coding. With the help of comments, one can easily understand the code after a long time. Even other people will also be able to understand someone else work with the help of comments. Like other languages, we can also comment in Perl language.

There are two ways to comment in Perl:

- Single line comments
- Multi line comments

Perl Single Line Comment

Perl single line comment starts with a hash sign (#).

- `$x = 10;`
- `print"$x\n"; #Here, x is a variable`

Output:

10

Perl Multi Line Comment

Perl multi line comment starts with (`=begin`) statement and ends with (`=cut`) statement.

- `$x = 20;`
- `print"$x\n";`
- `=begin`
- Let's **declare and**
- print variable in Perl.
- `=cut`

Output:

20

Perl last Statement

The **last** statement in Perl is like **break** statement in C. It is used inside a loop to exit the loop immediately. In other words, last condition iterates the loop.

The Perl syntax for last statement is given below:

- `last;`

Perl last Statement Example

Following is a simple example showing Perl last statement.

- `use warnings;`
- `use strict;`
- `my ($key, $value);`
- `my %rate = ("shoes" => 550,`
- `"boots" => 1200,`
- `"jaggi" => 800,`
- `"jacket" => 1500);`
- `print("Please enter an item to know the price:\n");`
- `$key = <STDIN>;`
- `chomp($key);`
- `$value = 0;`
- `# searching`
- `foreach(keys %rate){`

```

•     if($_ eq $key){
•         $value = $rate{$_};
•         last;    }
•     }
•     # result
•     if($value > 0){
•         print("$key costs is Rs. $value \n");
•     }else{
•         print("$key is not in our list. We apologise!!\n");
•     }

```

Output:

Please enter an item to know the price:

boots

boots costs is Rs. 1200

Please enter an item to know the price:

top

top is not in our list. We apologise!!

In the above program

- When user provides a key, hash elements runs via loop and each hash key is compared with the key provided by the user.
- If match is found, we exit the loop using last statement otherwise search continues.
- On finding a match result is displayed.

Perl last Statement with LABEL

Using the Perl last statement alone, you can exits only innermost loop. If you want to exit a nested loop, put a label in the outer loop and pass label to the last statement.

If LABEL is specified with last statement, execution drops out of the loop encountering LABEL instead of currently enclosing loop.

The Perl syntax for last statement with LABEL is given below:

1. last LABEL;

Perl last Statement with LABEL Example

```

•     use warnings;
•     use strict;
•     my ($key, $value);
•     my %rate = ("shoes" => 550,
•     "boots" => 1200,
•     "jaggi" => 800,
•     "jacket" => 1500);
•     $value = 0;
•     print("Please enter an item to know the price:\n");
•     OUTER: while(<STDIN>){
•         $key = $_;
•         chomp($key);
•         # searching
•         INNER: foreach(keys %rate){
•             if($_ eq $key){
•                 $value = $rate{$_};
•                 last outer;
•             }
•         }
•         print("$key is not in our list. We apologise!!\n") if($value ==0);
•         # result
•     }
•     print("$key costs is Rs. $value \n");

```

Output:

Please enter an item to know the price:

jaggi

boots costs is Rs. 800

Please enter an item to know the price:

jeans

jeans is not in our list. We apologise!!

The above program works in the same way except that it asks the user to enter a search key again if it couldn't find a match.

Two labels OUTER and INNER are used.

Inside foreach loop, if match is found, we'll exit both the loops because OUTER label is passed to the last statement.

Perl next Statement

The Perl next statement is like continue statement in C. It helps you to move on to the next element of your array or hash skipping all the elements in between.

The syntax of Perl next statement is given below:

1. next;

Perl next statement Example

- `@array = (a..z);`
- `print("@array\n");`
- `for ($index = 0; $index < @array; $index++) {`
- `if ($index == 0 || $index == 4 || $index == 8 || $index == 14 || $index == 20) {`
- `next;`
- `}`
- `$array[$index] = "-";`
- `}`
- `print("@array\n");`

Output:

a b c d e f g h i j k l m n o p q r s t u v w x y z

a - - - e - - - i - - - - o - - - - u - - - -

In the above program, we declared an array, from (a-z).

The alphabetical array is printed as it is.

for loop is used, with next statement for all the vowels.

We have assigned (-) sign to the rest of the array elements other than vowels.

Now due to next statement, only vowel elements will be printed and other elements will be replaced with (-) sign.

Perl next statement with label

The next label allows you to transfer control to an enclosing loop other than the neighboring loop.

If label is specified with next statement, execution skips to the next iteration identified with label.

The syntax of Perl next statement is given below:

- next LABEL;

Perl next statement with label Example

- OUTER: `for ($i = 0; $i < 3; $i++) {`
- `for ($j = 0; $j < 3; $j++) {`
- `print("inner: $i,$j\n");`
- `if ($j == 1) {`
- `next OUTER;`
- `}`
- `}`
- `print("outer: $i,$j\n\n");`
- `}`

Output:

```
Inner: 0,0
Inner: 0,1
Inner: 1,0
Inner: 1,1
Inner: 2,0
Inner: 2,1
```

Perl redo Statement

The Perl redo statement restarts the current loop without evaluation of the control statement. Further statements in the block will not be executed.

The syntax of redo statement is given below:

- redo;

Perl redo statement Example

- \$a = 5;
- while(\$a < 150){
- if(\$a == 40){
- \$a = \$a + 10;
- redo;
- }
- print "a = \$a\n";
- }continue{
- \$a = \$a * 2;
- }

Output:

```
a = 5
a = 10
a = 20
a = 50
a = 100
```

Perl for Loop

Perl for loop is also known as **C-style for loop**. The for loop iterates the statement or a part of the program several times.

It has three parameters:

- **Initialize** : This part is executed first, and only once. It initializes and declares loop variable.
- **Condition** : The for loop executes till the condition is true. When condition is false, loop execution stops and execution terminates out of the loop.
- **Increment/Decrement** : The for loop variable increment or decrement as long as it satisfies the loop condition. When condition is not satisfied loop terminates and output is printed.

The syntax of for loop in Perl language is given below:

- for(initialization;condition;incr/decr){
- //code to be executed
- }

Perl for loop Example:

Let's see the simple program of for loop that prints table of 1.

- for(\$a = 1; \$a <= 10; \$a++){
- print " \$a\n";
- }

Output:

```
1
```

```
2
3
4
5
6
7
8
9
10
```

Perl Nested for Loop

In this example, one for loop is nested inside another for loop. Inner loop is executed completely while outer loop is executed only once. It means if loop is running for 3 times, outer loop will execute 3 times but inner loop will execute 9 times.

Example:

- `for( $i = 1; $i <= 3; $i++ ){`
- `for( $j = 1; $j <= 3; $j++ ){`
- `print " $i $j\n";`
- `}`
- `}`

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Perl Infinite for Loop

If double semicolon (;;) is used in for loop, the loop will execute for infinite times. You can stop the execution using **ctrl + c**.

- `for( ;; )`
- {
- `printf "Infinite For Loop\n";`
- }

Output:

```
Infinite For Loop
Infinite For Loop
Infinite For Loop
Infinite For Loop
Infinite For Loop
Ctrl+c
```

Perl while Loop

The Perl **while loop** is used to iterate the part of program or statements many times.

In while loop, condition is given before the statement. When loop execution starts, it first checks whether the condition is true or false. If condition is true, loop executes. If condition is false, the loop terminates out of the loop.

Syntax of while loop in C language

The syntax of while loop in Perl language is given below:

- `while(condition){`

- `//code to be executed`
- `}`

Perl while loop Example

- `$i = 1;`
- `# while loop execution`
- `while( $i <= 10 ){`
- `printf "$i\n";`
- `$i++;`
- `}`

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Perl Nested while Loop Example

In nested while loop, one while loop is used inside another while loop. The inner while loop is executed completely while outer loop is executed once.

- `$i = 1;`
- `# while loop execution`
- `while( $i <= 3 ){`
- `$j = 1;`
- `while( $j <= 3 ){`
- `printf "$i $j\n";`
- `$j++;`
- `}`
- `$i++;`
- `}`

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Perl Infinite while Loop Example

To create infinite while loop, pass true as the test condition. Press **ctrl + c** to stop the loop execution.

- `while(true)`
- `{`
- `printf"Infinitive While Loop\n";`
- `}`

Output:

```
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
ctrl+c
```

Perl do while Loop

Unlike for and while loop, the do while loop checks its condition at the bottom of the loop. So do while loop will execute at least once.

do while loop syntax

The syntax of Perl language do-while loop is given below:

- `do{`
- `//code to be executed`
- `}while(condition);`

Perl do while Example

This is the simple program of Perl do while loop where we are printing the table of 1.

- `$i = 1;`
- `# do...while loop execution`
- `do{`
- `printf "$i\n";`
- `$i++;`
- `}while( $i <= 10 );`

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Perl Nested do while Loop Example

A nested do while loop is the one in which a do while loop is nested inside another do while loop.

It is executed completely for both inner and outer loops.

Example:

- `$i = 1;`
- `# do...while` loop execution
- `do{`
- `$j = 1;`
- `do{`
- `printf "%i %j\n";`
- `$j++;`
- `}while( $j <= 3 );`
- `$i++;`
- `}while( $i <= 3 );`

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Perl Infinite do while Loop Example

By passing true in while condition, infinite do while loop is executed. Its execution can be stopped by using **ctrl + c**.

- `do{`
- `printf "Infinite do-while Loop\n";`
- `}while( true );`

Output:

```
Infinite While Loop
Infinite While Loop
Infinite While Loop
Infinite While Loop
Infinite While Loop
ctrl+c
```

Perl Array

A Perl array variable stores an ordered list of scalar values.

To refer a single element of Perl array, variable name will be preceded with dollar (\$) sign followed by index of element in the square bracket.

Syntax:

- `@arrayName = (element1, element2, element3..);`

Perl Simple Array Example

This is a simple example to use Perl array.

- `#!/usr/bin/perl`
-
- `@num = (2015, 2016, 2017);`
- `@string = ("One", "Two", "Three");`
- `print "$num[0]\n";`
- `print "$num[1]\n";`
- `print "$num[2]\n";`
- `print "$string[0]\n";`
- `print "$string[1]\n";`
- `print "$string[2]\n";`

Output:

```
2015
2016
2017
One
Two
Three
```

In the above example, we have defined two arrays, one with number element and other with string element. Both arrays are printed with their index elements.

Perl Array Accessing

To access a single element of a Perl array, use (\$) sign before variable name. You can assume that \$ sign represents singular value and @ sign represents plural values.

Variable name will be followed by square brackets with index number inside it. Indexing will start with 0 from left side and with -1 from right side.

- `@months = qw/Jan Feb Mar Apr May Jun Jul Aug Sept Oct Nov Dec/;`
- `print "$months[0]\n";`
- `print "$months[1]\n";`
- `print "$months[7]\n";`
- `print "$months[9]\n";`
- `print "$months[6]\n";`
- `print "$months[-3]\n";`

Output:

```
Jan
Feb
Aug
Oct
Jul
Oct
```

Perl Array Size or Length

The size of an array is determined with scalar context on the array. The returned value will be always one greater than the largest index. In short the size of an array will be (\$#array + 1). Here, \$#array is the maximum index of the array.

- `@array = (you, me, us);`
- `$array[5] = 4;`
- `$size = @array;`
- `$index_max = $#array;`
- `print "Size: $size\n";`
- `print "Maximum Index: $index_max\n";`

Output:

```
Size: 6
Maximum Index: 5
```

In the output, there are only three elements containing information, but the given array has total 5 elements.

Perl Array Functions

You can add or remove an element from an array using some array functions.

We'll discuss following array Perl functions:

- Push
- Pop
- Shift
- Unshift

1) Push on Array

The push array function appends a new element at the end of the array.

- `@array = ("pink", "red", "blue");`
- `push @array, "orange";`
- `print "@array\n";`

Output:

```
pink red blue orange
```

In the above program, "orange" element is added at the end of the array.

2) Pop on Array

The pop array function removes the last element from the array.

- `@array = ("pink", "red", "blue");`
- `pop @array;`
- `print "@array\n";`

Output:

```
pink red
```

In the above program, "blue" element is removed from the end of the array.

3) Shift on Array

The shift array function removes the left most element of array and thus shortens the array by 1.

- `@array = ("pink", "red", "blue");`
- `shift @array;`
- `print "@array\n";`

Output:

```
red blue
```

In the above program, "pink" is removed from the array.

4) Unshift on Array

The unshift array function adds a new element at the start of the array.

- `@array = ("pink", "red", "blue");`
- `unshift @array, "orange";`
- `print "@array\n";`

Output:

```
orange pink red blue
```

In the above program, "orange" is added at the start of the array.

Perl Replacing Array Elements, splice()

The splice array function removes the elements as defined and replaces them with the given list.

- `@alpha = (A..Z);`

- print "Before - @alpha\n";
- splice(@alpha, 8, 8, U..Z);
- print "After - @alpha\n";

Output:

Before - A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

After - A B C D E F G H U V W X Y Z Q R S T U V W X Y Z

In the above program, the replacement begins counting from 9th position (I) to 8 elements that is P.

Perl Strings to Arrays, split()

With the help of split() function, we can split a string into array of strings and returns it.

- # original string
- \$string = "Where-There-Is-A-Will-There-Is-A-Way";
- # transforming strings into arrays.
- @string = split('-', \$string);
- print "\$string[4]\n";

Output:

Will

In the above program, we have transformed \$string into array at hyphen (-) values. Now from this array, we have printed fourth element of the array.

Perl Arrays to Strings, join()

The join() function is used to combine arrays to make a string. It combines the separate arrays into one string and returns it.

- # original string
- \$string = "Where-There-Is-A-Will-There-Is-A-Way";
- # transforming arrays into strings.
- @string = split('-', \$string);
- \$string_full = join(' ', @string);
- print "\$string_full\n";

Output:

Where-There-Is-A-Will-There-Is-A-Way

In the above program, the string is splitted at hyphens (-). We have used join() in \$string_full and printed it.

Perl Merging Two Arrays, merged()

Two arrays can be merged together using merged() function as a single string removing all the commas in between them.

- #two arrays
- @array1 = ("Girl", "in", "front", "of", "me");
- @array2 = ("is", "very", "beautiful");
- #merging both the arrays
- @merged = (@array1, @array2);
- print "@merged\n";

Output:

Girl in front of me is very beautiful

In the above program, array1 and array2 are merged into one single string and then printed.

Perl Sorting Arrays, sort()

To sort an array, sort() array function is used. The sort() function sorts all the elements of an array according to the ASCII standard.

- # defining array
- @days = ("sun", "mon", "tue", "wed", "thu", "fri", "sat");
- print "Original array: @days\n";
- # sorting array
- @days = sort(@days);
- print "Sorted array: @days\n";

Output:

Original array: sun mon tue wed thu fri sat

Sorted array: fri mon sat sun thu tue wed

Perl List

A Perl list is a sequence of [scalar](#) values. You use parenthesis and comma operators to construct a list. Each value in the list is called list element. List elements are indexed and ordered. You can refer to each element by its position.

Simple Perl list

The following example defines some simple lists:

- `()`;
- `(10,20,30)`;
- `("this", "is", "a", "list")`;

In the example above:

- The first list `()` is an empty list.
- The second list `(10,20,30)` is a list of [integers](#).
- The third list `("this", "is", "a", "list")` is a list of [strings](#).

Each element in the list is separated by a comma (,). In the previous tutorials, we used the `print` operator many times to display a number or a string. The `print` operator is a list operator. So let's display our lists above with the `print` operator to see how it works:

- `#!/usr/bin/perl`
- `use warnings;`
- `use strict;`
- `print();` # display nothing
- `print("\n");`
- `print(10,20,30);` # display 10,20,30
- `print("\n");`
- `print("this", "is", "a", "list");` # display: thisisalist
- `print("\n");`

We passed several lists to the `print` operator to display their elements. All the lists that we have seen so far contain an element with the same data type. These lists are called simple lists.

Complex Perl list

A Perl list may contain elements that have different data types. This kind of list is called a complex list. Let's take a look at the following example:

- `#!/usr/bin/perl`
- `use warnings;`
- `use strict;`
- `my $x = 10;`
- `my $s = "a string";`
- `print("complex list", $x, $s, "\n");`

Using qw function

Perl provides the `qw()` function that allows you to get a list by extracting words out of a string using the space as a delimiter. The `qw` stands for quote word. Two lists below are the same:

- `#!/usr/bin/perl`
- `use warnings;`
- `use strict;`
- `print('red','green','blue');` # redgreenblue

- `print("\n");`
- `print(qw(red green blue)); # redgreenblue`
- `print("\n");`

Similar to the `q/` and `q//` operators, you can use any non-alphanumeric character as a delimiter. The following lists are the same:

- `qw\this is a list\;`
- `qw{this is a list};`
- `qw[this is a list];`

Flattening list

If you put a list, called internal list, inside another list, Perl automatically flattens the internal list. The following lists are the same:

- `(2,3,4,(5,6))`
- `(2,3,4,5,6)`
- `((2,3,4),5,6)`

Accessing list element

You can access elements of a list by using the zero-based index. To access the n^{th} element, you put $(n - 1)$ index inside square brackets.

Let's take a look at the following example:

- `#!/usr/bin/perl`
- `use warnings;`
- `use strict;`
- `print(`
- `(1,2,3)[0] # 1 first element`
- `);`
- `print "\n"; # new line`
- `print(`
- `(1,2,3)[2] # 3 third element`
- `);`
- `print "\n"; # new line`

To get multiple elements of a list at a time, you can put a list inside square brackets. This feature is called list slice. You can omit the parenthesis of the list inside the square bracket.

- `(1,2,3,4,5)[0,2,3] # (1,3,4)`

The above code returns a list of three elements `(1, 3, 4)`.

Ranges

Perl allows you to build a list based on a range of numbers or characters e.g., list of numbers from 1 to 100, list of characters from a to z. The following example defines two lists:

- `(1..100)`
- `(a..z)`

Manipulating Lists

Perl provides several **built-in functions** for list manipulation. Three useful ones are:

shift LIST : Returns the first item of LIST, and moves the remaining items down, reducing the size of LIST by 1.

unshift ARRAY, LIST : The opposite of shift. Puts the items in LIST at the beginning of ARRAY, moving the original contents up by the required amount.

push ARRAY, LIST : Similar to unshift, but adds the values in LIST to the end of ARRAY.

Iterating over Lists

foreach:

The foreach loop performs a simple iteration over all the elements of a list

Eg:

```
foreach $item (list)
```

```
{
```

```
.....
}
```

The block is executed repeatedly with the variables \$item taking each value from the list in turn. The variable can be omitted, in which case \$_ will be used. The natural Perl idiom for manipulating all items in an array is ;

```
foreach (@array)
{
.....#process $_
}
```

Perl Hashes

The hashes is the most essential and influential part of the perl language. A hash is a group of key-value pairs. The keys are unique strings and values are scalar values.

Hashes are declared using my keyword. The variable name starts with a (%) sign.

Hashes are like arrays but there are two differences between them. First arrays are ordered but hashes are unordered. Second, hash elements are accessed using its value while array elements are accessed using its index value.

No repeating keys are allowed in hashes which makes the key values unique inside a hash. Every key has its single value.

Syntax:

- my %hashName = ("key" => "value";)

Perl Hash Accessing

To access single element of hash, (\$) sign is used before the variable name. And then key element is written inside {} braces.

- my %capitals = (
- "India" => "New Delhi",
- "South Korea" => "Seoul",
- "USA" => "Washington, D.C.",
- "Australia" => "Canberra"
-);
- print"\$capitals{'India'}\n";
- print"\$capitals{'South Korea'}\n";
- print"\$capitals{'USA'}\n";
- print"\$capitals{'Australia'}\n";

Output:

```
New Delhi
Seoul
Washington, D.C.
Canberra
```

Perl Hash Indexing

Hashes are indexed using \$key and \$value variables. All the hash values will be printed using a while loop. As the while loop runs, values of each of these variables will be printed.

- my %capitals = (
- "India" => "New Delhi",
- "South Korea" => "Seoul",
- "USA" => "Washington, D.C.",
- "Australia" => "Canberra"
-);
- # LOOP THROUGH IT
- while ((\$key, \$value) = each(%capitals)){

```

    • print $key.", ".$value.\n";
    • }

```

Output:

```

Australia, Canberra
India, New Delhi
USA, Washington, D.C.
South Korea, Seoul

```

Perl sorting Hash by key

You can sort a hash using either its key element or value element. Perl provides a `sort()` function for this. In this example, we'll sort the hash by its key elements.

```

    • my %capitals = (
    •     "India" => "New Delhi",
    •     "South Korea" => "Seoul",
    •     "USA" => "Washington, D.C.",
    •     "Australia" => "Canberra"
    • );
    • # Foreach loop
    • foreach $key (sort keys %capitals) {
    •     print "$key: $capitals{$key}\n";
    • }

```

Output:

```

Australia: Canberra
India: New Delhi
South Korea: Seoul
USA: Washington: D.C.
=====

```

Perl String

Strings are an essential part of the Perl language. They are scalar variables, so they start with (\$) sign. A string can be defined within a single quote (') or double quote (").

Perl String Operators

The operators make it easy to manipulate a string in different ways. There are two types of string operators:

- Concatenation (.)
- Repetition (x)

Perl Concatenation Operator

Perl strings are concatenated with a (.) sign instead of (+) sign.

```

    • $firstName = "Christian";
    • $lastName = "Grey";
    • $fullName = $firstName . " " . $lastName;
    • print "$fullName\n";

```

Output:

```

Christian Grey

```

Perl Repetition Operator

Perl strings can be repeated a number of times with (x) variable.

```

    • $text = "Thank You ";
    • $output = $text x 3;
    • print "$output\n";

```

Output:

```

Thank You Thank You Thank You

```

Perl Initializing and Declaring a String

In Perl, to declare a string use **my** keyword before variable name.

A string can be initialised and declared with the following syntax:

- `my $variableName = "";`

In this example, we have shown how to initialize and declare a string. We have printed several strings together by the following ways:

- Join strings using a dot (.) operator.
- Supply strings as separate arguments.
- Embed strings in a bigger string.

We have shown all the three methods to print the output.

- `use strict;`
- `use warnings;`
- `# Declaring and initializing a string.`
- `my $msg1 = "Welcome at Java.";`
- `my $msg2 = "This is our Perl Tutorial.";`
- `#printing using . operator.`
- `print $msg1 . " " . $msg2. "\n";`
- `#print as separate arguments.`
- `print $msg1, " ", $msg2, "\n";`
- `#embedd string in a bigger string.`
- `print "$msg1$msg2\n";`

Output:

```
Welcome at Java. This is our Perl Tutorial.
Welcome at Java. This is our Perl Tutorial.
Welcome at Java. This is our Perl Tutorial.
```

Perl Single quote Vs Double quote String

Strings can be placed within a single quote (') or double quote (") but they have little different behavior.

- `my $user = 'Ana';`
- `print 'Hello $user, welcome at our site.\n';`
- `print "\n";`
- `my $user = 'Ana';`
- `my $day = "today";`
- `print "Hello $user, welcome at our site $day.\n";`

Output:

```
Hello $user, welcome at our site.\n
Hello Ana, welcome at our site today.
```

Perl substr Function

This function returns a substring of EXPR, starting at OFFSET within the string. If OFFSET is negative, starts that many characters from the end of the string. If LEN is specified, returns that number of bytes, or all bytes up until end-of-string if not specified. If LEN is negative, leaves that many characters off the end of the string.

If REPLACEMENT is specified, replaces the substring with the REPLACEMENT string.

If you specify a substring that passes beyond the end of the string, it returns only the valid element of the original string.

Syntax

Following is the simple syntax for this function –

```
substr EXPR, OFFSET, LEN, REPLACEMENT
```

```
substr EXPR, OFFSET, LEN
```

```
substr EXPR, OFFSET
```

Return Value

This function returns string.

Example

Following is the example code showing its basic usage –

```
$temp = substr("okay", 2);
print "Substring valuye is $temp\n";

$temp = substr("okay", 1,2);
print "Substring valuye is $temp\n";
```

OUTPUT:

```
Substring valuye is ay
Substring valuye is ka
```

Perl Regular Expression

A regular expression is a string of characters that defines a specific pattern. The perl regular expression syntax is quite similar with that of awk, grep and sed.

There are three regular expression operators inside perl:

- Matching regular expression operator
- Substitute regular expression operator
- Transliterate regular expression operator

Perl Matching Operators

Perl matching operators have various modifiers. It is mainly used to match a string or statement to a regular expression.

Perl Matching Operator =~

The matching operator =~ is used to match a word in the given string. It is case sensitive, means if string has a lowercase letter and you are searching for an uppercase letter then it will not match.

- `$line = "This is javaTpoint.";`
- `if ($line =~ /java/){`
- `print "Matching\n";`
- `}else{`
- `print "Not Matching\n";`
- `}`
- `if ($line =~ /Java/){`
- `print "Matching\n";`
- `}else{`
- `print "Not Matching\n";`
- `}`

Output:

```
Matching
```

```
Not Matching
```

Perl Matching Operator !~

It is the opposite of the earlier one (=`~`). If the letters match it gives the output as not matched and vice versa.

- `$line = "This is javaTpoint.";`
- `if ($line!~ /java/){`
- `print "Matching\n";`
- `}else{`
- `print "Not Matching\n";`
- `}`
- `if ($line!~ /Java/){`
- `print "Matching\n";`
- `}else{`
- `print "Not Matching\n";`
- `}`

Output:

```
Not Matching
```

```
Matching
```

Perl Substitution Operator

The substitution operator is just an extension of the matched operator. It allows the replacement of text matched with some new text.

Its basic syntax is:

- `s/oldPattern/newPattern /;`

Perl Substitution Operator with s///

Here we are replacing liquid with solid in the first part with s///.

In the second part, 'liquid' is replaced with 'solid' globally with s///g.

- `$line = "liquid will remain liquid until it is evaporated";`
- `$line =~ s/liquid/solid/;`
- `print "$line\n";`
- `print "\n";`
- `$line = "liquid will remain liquid until it is solidified";`
- `$line =~ s/liquid/solid/g;`
- `print "$line\n";`

Output:

```
solid will remain liquid until it is evaporated
solid will remain solid until it is evaporated
```

Perl Functions and Subroutines

Perl functions and subroutines are used to reuse a code in a program. You can use a function at several places in your application with different parameters.

There is only one difference in function and subroutine, subroutine is created with **sub** keyword and it returns a value. You can divide your code into separate subroutines. Logically each function in each division should perform a specific task.

Syntax of subroutine:

- `sub subName{`
- `body`
- `}`

Perl define and call subroutine function

The syntax for Perl define a subroutine function is given below:

- `sub subName{`
- `body`
- `}`
- `OR`
- `subName(list of arguments);`
- `&subName(list of arguments);`

In the following example, we are defining a subroutine function 'myOffice' and call it.

- `#defining function`
- `sub myOffice{`
- `print "java\n";`
- `}`
- `# calling Function`
- `myOffice();`

Output:

```
java
=====
```

Perl split Function

The Perl **split** function splits a string into an array. A string is splitted based on delimiter specified by pattern. By default, it whitespace is assumed as delimiter.

split syntax is:

- `Split /pattern/, variableName`

Perl split Example

In this example, split returns a list of strings. The delimiter is (-). It means string is splitted at (-).

- `my $str = "Cu-K-Na-Hg-Pb-Li";`
- `my @break = split /-/, $str;`
- `print "@break\n";`

Output:

```
Cu K Na Hg Pb Li
```

Perl split Limit number of parts

We can limit the number of parts to be splitted for a string.

In this example, string splits in 3 parts.

- `my $str = "Cu-K-Na-Hg-Pb-Li";`
- `my @break = split(/-/, $str, 3);`
- `print "@break\n";`

Output:

```
Cu K Na-Hg-Pb-Li
```

Perl split on Multiple Characters

We can split a character at more than one delimiter. In the following example, we have split the string at (=) and (,).

- `my $str = 'Vishal=18Sept,Anu=11May,Juhi=5Jul';`
- `my @words = split /[=,]/, $str;`
- `print "@words\n";`

Output:

```
Vishal 18Sept Anu 11May Juhi 5Jul
```

As they are key-value pairs, we can assign the result into a hash instead of an array.

- `use Data::Dumper qw(Dumper);`
- `my $str = 'Vishal=18th Sept,Anu=11th May,Juhi=5th Jul';`
- `my %words = split /[=,]/, $str;`
- `print Dumper \%words;`

Output:

```
$VAR1 = (
    'Anu' => '11th May',
    'Vishal' => '18th Sept',
    'Juhi' => '5th Jul',
);
```

Perl split on Empty String

Split on empty string means the string will split at every place where it will find an empty string. There is an empty string, between every two characters. It means it will return the original string split into individual characters.

- `my $str = "JAVA";`
- `my @break = split //, $str;`
- `print "@break\n";`

Output:

```
J A V A
```

Perl join Function

Perl join character, joins elements into a single string using a delimiter pattern to separate each element. It is opposite of split.

join syntax is:

- `join 'pattern', variableName`

Perl join Example

In the following example, we have joined in three different ways.

First, we are joining (:) in between each elements of array.

Second, we will join (- vowels) with the output of first one.

Third, we added two more characters in the original array.

- `my @vowels = ('A', 'E', 'I');`
- `my $str = join ':', @vowels;`
- `print "$str\n";`
- `my $data = join "-", $str, "vowels";`
- `print "$data\n";`
- `$str = join "", @vowels, 'OU';`
- `print "$str\n";`

Output:

A:E:I

A:E:I -vowels

AEIOU

IMPORTANT QUESTIONS:

UNIT-1: Introduction to PERL and Scripting:

1. Explain the modern applications of scripting languages?
2. Discuss the various characteristics of a scripting language?
3. What are lists concept in PERL?
4. Explain pattern matching modifiers?
5. List out the three different meanings of the term "scripting"?
6. Write briefly about scripts and programs?
7. Illustrate the traditional applications of scripting languages?
8. Explain in detail about hash variables?
9. Discuss available built-in array functions in PERL?
10. Explain different methods of iterating over lists?
11. Discuss about Regular Expressions in Perl?
12. State the functionalities of tr, split and join functions in Perl?
13. What is a subroutine? How to define subroutine and call a subroutine with an example?
14. Discuss about local variables and global variables in Perl?
15. What is the difference between hashes and arrays?
16. Describe the role of whitespaces in PERL. Describe Perl Subroutines?
17. Explain the different types control statements available in Perl. Discuss about them with examples.
18. Explain about accessing, creating and processing of arrays with an example in Perl.
19. Illustrate a brief note on special variable in Perl. Explain about scalar expressions in Perl.
20. Illustrate a brief note on special variable in Perl. Explain about scalar expressions in Perl.