# UNIT-1

# INTRODUCTION TO SCRIPTING LANGUAGES

## 1.1 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 for is qualitatively different from conventional programming languages like C++ and Ada address the problem of developing large applications from the ground up, employing a team of professional programmers, starting from well-defined specifications, and meeting 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.

## 1.2 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 keyword. 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.

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

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 privides a programmable interface:this may be an API,though more commonly the application is construted 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 funcationaliy and ease of use as an alternate to a conventional language for general programming tasks ,particularly system programming and administration.Examples: are UNIX system adminstrators have for a long time used scripting languages for system maintenace tasks,and administrators of WINDOWS NT systems are adopting a scripting language ,PERL for their work.

## 1.4 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 commond to compile the program and then to run the resulting object file,and without the need to link extensive libraries into he 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 strucures 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 effeciency,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.

## 1.5 Users For Scripting Lanuages:

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 arised. 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 it's 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 it's 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.

## 1.6 web scripting:

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

a. processing forms

b. creating pages with enhanced visual effects and user interaction and
c. 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 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.

## 1.7 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 enterprise-wide document management system for a leading aerospace company.

## 1.8 Names and Values in Perl:

### 1.8.1 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 occurance of j on the left denotes a storage location, whilw the right-hand occurance 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 occurances 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.

**Scalar data:**

**Strings and numbers:**

In common with many scripting languages, Perl recognizes just two kinds of scalar data: strings and numbers. There is no distinction between interger and real numbers as different types.Perl is a dynamically typed language: the system keeps track of whether a variable contains a numeric value or a string value, and the user doesn't have to worry about thedifference between strings and numbers since conversions between the two kinds of data are done automatically as required by the context in which they are used.

**Boolean values:**

All programming languages need some way of representing truth values and Perl is no exception. Since scalar values are either numbers or strings, some covention is needed for representing  Boolean values, and Perl adopts the simple rule that numeric zero, "0" and the empty string (" ") mean false, and anything else means true.

**Numeric constants:**

Numeric constants can be written in a variety of ways, including specific notation, octal and hexadecimal. Although Perl tries to emulate natural human communication, the common practice of using commas or spaces to break up a large integer constant into meaningful digit groups cannot be used, since the comma has a syntactic significance in Perl. Instead, underscores can be included in a number literal to improve legibility.

**String constants:**

String constants can be enclosed in single or double quotes. The string is terminated by the first next occurance of the quote which started it, so a single-quoted string can include double quotes and vice versa. The q (quote) and qq (double quote) operators allow you to use any character as a quoting character. Thus

q / any string/ or q ( any string )

are the same as

'any string' and qq / any string / or qq ( any string )

are the same as

"any string"

**1.8.2 Variables and assignment:**

**Assignment:**

Borrowing from C, Perl uses '=' as the assignment operator. It is important to note that an assignment statement returns a value, the value assigned. This permits statements like

```
$b = 4 + ( $a = 3) ;
```

which assigns the value 3 to $a and the value 7 to $b.If it is required to interpolate a variable value without an intervening space the following syntax, borrowed from UNIX shell scripts , is used:

```
$a = "Java ;

$b = "$ { a } Script" ;       which gives $b the value "JavaScript".
```

**<STDIN> - a special value:**

When the 'variable' <STDIN> appears in a context where a scalar value is required, it evaluates to a string containing the next line from standard input, including the terminating newline. If there is no input queued, Perl will wait until a line is typed and the return key pressed. The empty string is treated as false in a Boolean context. If <STDIN> appears on the right-hand side of an assignment to a scalar variable, the string containing the input line is assigned to the variable named on the ;eft. If it appears in any other scalar context the string is assigned to the anonymous variable: this can be accessed by the name $- : many operations use it as a default.

## 1.8.3 Scalar Expressions:

Scalar data items are combined into expressions using operators. Perl has a lot of operators, which are ranked in 22 precedence levels. These are carefully chosen so that the 'obvious' meaning is what You get , but the old advice still applies: if in doubt ,use brackets to force the order of evaluation . In the following sections we describe the available operators in their natural groupings-arithmetic , strings,logical etc .

**Arithmetic operators:**

Following the principles of 'no surprises' Perl provides the usual

Arithmetic operators, including auto-increment and auto-decrement operators after the manner of C: note that in

```
$c= 17 ; $d= ++$c;
```

The sequence is increment and the assign, whereas in

```
$c= 17 ; $d =  $c++;
```

The sequence is assign then increment . As C, binary arithmetic operations can be combined with assignment, e.g.

```
$a += 3;
```

This adds 3 to $a, being equivalent to

```
$a =$a + 3;
```

As in most other languages, unary minus is used to negate a numeric value; an almost never-used unary plus operator is provided for completeness.

**String Operators**

Perl provides very basic operators on strings: most string processing is one using built-in functions expressions, as described later.

Unlike many languages use + as a concatenation operator for strings, Perl uses a period for this purpose: this lack of overloading means that an operator uniquely determines the context for its operands. The other string operator is x, which is used to replicate strings, e.g.

```
$a =”Hello” x 3;
```

Sets $a to “HelloHelloHello”.

The capability of combining an operator with assignment is extended to string operations. E.g.

```
$foo  .= “ “ ;
```

Appends a space to $foo.

So far, things have been boringly conventional for the most part. However, we begin to get a taste of the real flavor of perl when we see how it adds a little magic when some operators, normally used in arithmetic context, are used in a string context.

Two examples illustrate this.

1.**Auto increment** :

If a variable has only ever been used in a string context, the auto increment operator can be applied to it. If the value consists of a sequence of letters, or a sequence of letters followed by a sequence of digits, the auto increment takes place in string mode starting with the right most character, with 'carry' along the string. For example, the sequence

```
        $a = 'a0' ; $b = 'Az9' ;
Print  ++$a, ' ', ++$b; "/n";
```

Prints a1 Ba0.

2.**Unaryminus** :

This has an unusual effect on non numeric values. Unary minus applied to a string which starts with a plus or minus character returns the same string, but starting with the opposite sign. Unary minus applie to an identifier returns a string consists of minus prefixed to the characters of the identifiers. Thus if we have a variable named $config with the value " foo", then –config evaluates the string "-foo". This is useful, for example, in constructing command arguments are introduced by –

**Comparison operators:**

The value of comparisons is returned as 1 if true, and an empty string (" ") if false, in accordance with the convention described earlier.

Two families of comparison operators provide, one for numbers and one for strings. The operator used determines the context , and perl converts the operands as required to match the operator.

This duality is necessary because a comparison between strings made up entirely numerical digits should apply the usual rules for sorting strings ASCII as a collating sequence, and this may not give the same result as the numerical comparison('5' <'10') returns the value true as a numerical comparison having been converted into (5<10) where as the string comparison ('5' lt '10') returns false, since 10 comes before 5 in the canonical sort order for ASCII strings.

The comparison operator (⇔ for numbers, cmp for strings), performs a three way test, returning -1 for less-than, 0 for equal an +1 for greater-than.

Note that the comparison operators are non associative, so an expression like

```
$a > $b > $c
```

Is erroneous.

**logical operators:**

The logical operators allows to combine conditions using the usual logical operations 'not'(!, not), 'and'(&&,and) and 'or'(||,or). Perl implements the 'and' and 'or' operators in 'shortcut'mode, i.e evaluation stops as soon as the final result is certain using the rules false &&b=false, and true||b=true.

Before Perl 5,only the !, && and || operators were provided.The new set, not, and,or, are provided partly to increase readability, and partly because their extra-low precedence makes it possible to omit brackets in most circumstances-the precedence ordering is choosen so that numerical expressions can be compared without having to enclose them in brackets, e.g.

```
Print"OK\n" if $a<10 and $b<12;
```

**Bitwise operators:**

The unary tilde(~) applied to a numerical argument performs bitwise negation on its operand, generating the one's compliment. If applied to a string operand it complements all the bits in the string – effective way of inverting a lot of bits. The remaining bitwise operators - & (and), | (or) and ^(exclusive or)- have a rather complicated definition. If either operand is a number or a variable that has previously been used as a number, both operands are converted to integers if need be, and the bitwise operation takes place between the integers. If the both operands are strings, and if variables have never been used as numbers, Perl performs the bitwise operation between corresponding bits in the two strings, padding the shorter strings with zeros as required.

**Conditional expressions:**

A conditional expression is one whose values is choosen from two alternatives at run-time depending on the outcome of a test. The syntax is borrowed from C:

```
Test ? true_exp: false_exp
```

The first expression is evaluated as Boolean value : if it returns true the whole expression is replaced by true_exp,otherwise it is replaced by false_exp, e.g.

```
$a= ($a<0)? 0 : $a;
```

### 1.8.4 Control structures:

The Control Structures for conditional execution and repetition all the control mechanisms is similar to C.

1. **BLOCKS:**

A *block* is a sequence i\of one or more statements enclosed in curly braces.

```
Eg: { $positive =1;
```

```
        $negative=-1;}
```

The last statement is the block terminated by the closing brace.

In, *Perl* they use *conditions* to control the evaluation of one or more blocks. Blocks can appear almost anywhere that a statement can appear such a block called **bare block.**

## 2. CONDITIONS:

A condition is a Perl expression which is evaluated in a Boolean context: if it evaluates to zero or the empty string the condition is false, otherwise it is true.

Conditions usually make use of relational operators.

```
Eg: $total>50
     $total>50 and $total<100
```

Simple Conditions can be combined into a complex condition using the logical operators. A condition can be negated using the ! operator.

```
Eg: !($total>50 and $total<100)
```

## 3. CONDITIONAL EXECUTION:

**If-then-else statements**

```
  if ($total>0){
     print "$total\n"}
 if ($total>0){
     print "$total\n"
} else {
```

```
print “bad total!\n”}
```

A single statement is a block and requires braces round it. The if statement requires that the expression forming the condition is enclosed in brackets. The construct extends to multiple selections

```
Eg: if ($total>70) {
        $grade=”A”;
     } elsif ($total >50) {
       $grade=”B”;
     } elsif ($total>40) {
      $grade=”C”;
     } else {
       $grade=”F”;
       $total=0;
     }
```

**Alternatives to if-then-else**

To use a conditional expression in place of an if-then-else construct.

```
if ($a<0)
   ($b=0)
 else ($b=1)
```

can be written as

```
$b= ($a<0)? 0:1;
```

To use the ‘or’ operator between statements

```
Eg:  open (IN, $ARGV[0] or die
          “Can’t open $ARGV[0]\n”;
```

**Statement qualifiers**

A single statement(not a block) can be followed by a conditional modifier.

```
Eg: print "OK\n"          if    $volts>=1.5;
    print "Weak\n"        if    $volts>=1.2 and
                                $volts<1.5;
    print "Replace\n"     if    $volts<1.2;
```

Code using Conditional expressions,

```
Eg: print (($volts>=1.5)? "Ok\n";

        (($volts>=1.2)? "Weak\n";

        "Replace\n"));
```

**4. REPETITION:**

Repetition mechanisms include both

- Testing Loops
- Counting Loops

**TESTING LOOPS**

```
While ($a! = $b)
        if ($a > $b) {
            $a=$a-$b
     } else {
          $b=$b-$a
     }
}
```

With the if statement, the expression that forms the condition must be enclosed in brackets. But now, *while* can be replaced by *until* to give the same effect. Single statement can use while and until as statement modifiers to improve readability.

```
Eg: $a += 2 while $a <$b;
    $a += 2 until $a > $b;
```

Here, although the condition is written after the statement, it is evaluated *before* the statement is executed, if the condition is initially false the statement will be never executed.

When the condition is attached to a do loop is same as a statement modifier, so the block is executed at least once.

```
      do {
          ……….
    } while $a! = $b;
```

**Counting Loops:**

In C,

```
          for ($i= 1;$i<=10;$i++) {
             $i_square=$i*$i;
             $i_cube=$i**3;
             print “$i\t$i_square\t$i_cube\n”;
}
```

In Perl,

```
     foreach $i (1…10){
             $i_square=$i* $i;
             $i_cube=$i**3;
             print  “$i\t$i_square\t$i_cube\n”;
}
```

## 1.9 LIST,ARRAYS AND HASHES:

### 1.9.1 LISTS:

- A list is a collection of scalar data items which can be treated as a whole, and has a temporary existence on the run-time stack.
- It is a collection of variables, constants (numbers or strings) or expressions, which is to be treated as a whole.
- It is written as a comma-separated sequence of values, eg: “red” , “green” , “blue”.
- A list often appears in a script enclosed in round brackets. For eg:

  ( “red” , “green”, “blue” )
- Shorthand notation is acceptable in lists, for eg:

  (1..8)

  (“A”..”H” , “O”..”Z”)
- qw(the quick brown fox) is a shorthand for (“the” , ”quick” , ”brown” , ”fox”).

**Arrays and Hashes:** These are the collections of scalar data items which have an assigned storage space in memory, and can therefore be accessed using a variable name.

**Arrays:**

- An array is an ordered collection of data whose comparisions are identified by an ordinal index: It is usually  the value of an array variable.
- The name of the variable always starts with an @, eg: @days_of_week.

  NOTE: An array stores a collection, and List is a collection, So it is natural to assign a list to an array.

  ```
      Eg: @rainfall = (1.2 , 0.4 , 0.3 , 0.1 , 0 , 0 , 0 );
  ```
- A list can occur as an element of another list.

  ```
      Eg: @foo = (1 , 2 , 3, "string");
          @foobar  = (4 , 5 , @foo , 6);
  ```

  The foobar result would be (4 , 5 , 1 , 2 , 3 , "string" , 6);

**<u>Hashes:</u>**

- An associative array is one in which each element has two components : a key and a value, the element being 'indexed' by its key.
- Such arrays are usually stored in a hash table to facilitate efficient retrieval, and for this reason Perl uses the term hash for an associative array.
- Names of hashes in Perl start with a % character: such a name establishes a list context.
- The index is a string enclosed in braces(curly brackets).

  ```
  Eg: $somehash{aaa} = 123;
      $somehash{"$a"}  = 0;       //The key is a the current value of $a.
     %anotherhash =%somehash;
  ```

**Working With Arrays And Lists:**

**Array Creation:**

- Array variables are prefixed with the @sign and are populated using either paranthesis or the qw operator.
  Eg: @array = (1 , 2 ,"Heelo");
      @array = qw/This is an array/;
- In C, C++, Java; Array is a collection of homogeneous elements, whereas; In Perl, Array is a collection of heterogeneous elements.

**Accessing Array Elements:**

- When accessing an individual element, we have to use the '$' symbol followed by variable name along with the index in the square brackets.
  Eg: $bar = $foo[2];
      $foo[2] = 7;
- A group of contiguous elements is called a slice , and is accessed using a simple syntax:
  @foo[1..3] is the same as the list ($foo[1], $foo[2], $foo[3])
- A slice can be used as the destination of an assignment,
  Eg: @foo[1..3] = ("hop" , "skip" , "jump");
- Like a slice, a selection can appear on the left of an assignment: this leads to a useful idiom for rearranging the elements in a list.
  Eg: To swap the first two elements of an array, we write as;
      @foo[0 , 1] = @foo[1 , 0];

**<u>Manipulating Lists:</u>**

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 $_

 }
```

**Working With Hashes:**

- A hash is a set of key/value pairs.
- Hash variables are preceded by a "%" sign.
- To refer to a single element of a hash, you will use the hash variable name preceded by a '$' sign and followed by the "key" associated with the value in the curly brackets.
- It is also called as associative array.

**Creating Hashes:**

- We can assign a list of key-value pairs to a hash, as, for example,

```
%foo = (key1, value1, key2, value2, …….);
```

- An alternative syntax is provided using the => operator to associate key-value pairs, thus:

```
%foo =  (banana => ‘yellow’ , apple => ‘red’ , grapes => ‘green’, ...........);
```

**Manipulating Hashes:**

Perl provides a number of built-in functions to facilitate manipulation of hashes. If we have a hash called HASH,

- keys % HASH returns a list of the keys of the elements in the hash, and
- values % HASH returns a list of the values of the elements in the hash.

Eg:     %foo =  (banana => ‘yellow’ , apple => ‘red’ , grapes => ‘green’, ...........);

keys % HASH returns banana, apple ,grapes

values % HASH returns yellow, red, green.

These functions provide a convenient way to iterate over the elements of a hash using foreach:

```
foreach (keys % HASH) {

     process $magic($_)

}
```

Other useful operators for manipulating hashes are delete and exists.

- delete $HASH{$key} removes the element
- exists  $HASH{$key} returns true.

## 1.10 Strings, Pattern Matching & Regular Expressions in Perl:

The most powerful features of Perl are in its vast collection of string manipulation operators and functions. Perl would not be as popular as it is today in bioinformatics applications if it did not contain its flexible and powerful string manipulation capabilities.

**String concatenation:**

To concatenate two strings together, just use the . dot:

- $a . $b;
- $c = $a . $b;
- $a = $a . $b;
- $a .= $b;

The first expression concatenates $a and $b together, but the the result was immediately lost unless it is saved to the third string $c as in case two. If $b is meant to be appended to the end of $a, use the .= operator will be more convenient. As is any other assignments in Perl, if you see an assignment written this way $a = $a op expr, where op stands for any operator and expr stands for the rest of the statement, you can make a shorter version by moving the op to the front of the assignment, e.g., $a op= expr.

**Substring extraction**

The counterpart of string concatenation is substring extraction. To extract the substring at certain location inside a string, use the substr function:

- $second_char = substr($a, 1, 1);
- $last_char = substr($a, -1, 1);
- $last_three_char = substr($a, -3);

The first argument to the substr function is the source string, the second argument is the start position of the substring in the source string, and the third argument is the length of the substring to extract. The second argument can be negative, and if that being the case, the start

position will be counted from the back of the source string. Also, the third argument can be omitted. In that case, it will run to the end of the source string.
A particularly interesting feature in Perl is that the substr function can be *assigned into* as well, meaning that in addition to string extraction, it can be used as string replacement:

- substr($a, 1, 1) = 'b'; # change the second character to b
- substr($a, -1) = 'abc'; # replace the last character as abc (i.e., also add two new letters bc)
- substr($a, 1, 0) = 'abc'; #insert abc in front of the second character

**Substring search**

In order to provide the second argument to substr, usually you need to locate the substring to be extracted or replaced first. The index function does the job:

- $loc1 = index($string, "abc");
- $loc2 = index($string, "abc", $loc+1);
- print "not found" if $loc2<0;

The index function takes two arguments, the source string to search, and the substring to be located inside the source string. It can optionally take a third argument to mean the start position of the search. If the index function finds no substring in the source string anymore, then it returns -1.

**Regular expression**

Regular expression is a way to write a pattern which describes certain substrings. In general, the number of possible strings that can match a pattern is large, thus you need to make use of the regular expression to describe them instead of listing all possibilities. If the possible substring matches are just one, then maybe the index function is more efficient.

The following are some basic syntax rules of regular expression:

- Any character except the following special ones stands for itself. Thus abc matches 'abc', and xyz matches 'xyz'.

- The character . matches any single character. To match it only with the . character itself, put an escape \ in front of it, so \. will match only '.', but . will match anything. To match the escape character itself, type two of them \\ to escape itself.
- If instead of matching any character, you just want to match a subset of characters, put all of them into brackets [ ], thus [abc] will match 'a', 'b', or 'c'. It is also possible to shorten the listing if characters in a set are consecutive, so [a-z] will match all lowercase alphabets, [0-9] will match all single digits, etc. A character set can be negated by the special ^ character, thus [^0-9] will match anything but numbers, and [^a-f] will match anything but 'a' through 'f'. Again, if you just want to match the special symbols themselves, put an escape in front of them, e.g., \[, \^ and \].
- All the above so far just match single characters. The power of regular expression lies in its ability to match multiple characters with some meta symbols. The * will match 0 or more of the previous symbol, the + will match 1 or more of the previous symbol, and ? will match 0 or 1 of the previous symbol. For example, a* will match 'aaaa...' for any number of a's including none '', a+ will match 1 or more a's, and a? will match zero or one a's. A more complicated example is to match numbers, which can be written this way [0-9]+. To matching real numbers, you need to write [0-9]+\.?[0-9]*. Note that the decimal point and fraction numbers can be omitted, thus we use ?, and * instead of +.
- If you want to combine two regular expressions together, just write them consecutively. If you want to use either one of the two regular expressions, use the | meta symbol. Thus, a|b will match a or b, which is equivalent to [ab], and a+|b+ will match any string of a's or b's. The second case cannot be expressed using character subset because [ab]+ does not mean the same thing as a+|b+.
- Finally, regular expressions can be grouped together with parentheses to change the order of their interpretation. For example, a(b|c)d will match 'abd' or 'acd'. Without the parentheses, it would match 'ab' or 'cd'.

The rules above are simple, but it takes some experience to apply them successfully on the actual substrings you wish to match. There are no better ways to learn this than simply to write some regular expressions and see if they match the substrings you have in mind.

The following are some examples:

- [A-Z][a-z]* will match all words whose first character are capitalized
- [A-Za-z_][A-Za-z0-9_]* will match all legal perl variable names
- [+-]?[0-9]+\.?[0-9]*([eE][+-]?[0-9]+)? will match scientific numbers
- [acgtACGT]+ will match all DNA strings
- ^> will match the > symbol only at the beginning of a string
- a$ will match the a letter only at the end of a string

In the last two examples above, we introduced another two special symbols. The ^ which when not used inside a character set to negate the character set, stands for the beginning of the string. Thus, ^> will match '>' only when it is the first character of the string. Similarly, $ inside a regular expression means the end of the string, so a$ will match 'a' only when it is the last character of the string. These are so called *anchor* symbols.

Another commonly used anchor is \b which stands for the boundary of a word. In addition, Perl introduces predefined character sets for some commonly used patterns, thus \d stands for digits and is equivalent to [0-9], \w stands for word letters or numbers, and \s stands for space characters ' ', \t, \n, \r, etc. The captial letter version of these negates their meaning, thus \D matches non-digit characters, \W matches non-word characters, and \S matches non-whitespaces. The scientific number pattern above can therefore be rewritten as:

- [+-]?\d+\.?\d*([eE][+-]?\d+)?

**Pattern matching:**

Regular expressions are used in a few Perl statements, and their most common use is in pattern matching. To match a regular expression pattern inside a $string, use the string operator =~ combines with the pattern matching operator / /:

- $string =~ /\w+/; # match alphanumeric words in $string
- $string =~ /\d+/; # match numbers in $string

The pattern matching operator / / does not alter the source $string. Instead, it just returns a true or false value to determine if the pattern is found in $string:

-
  ```
  if ($string =~ /\d+/) {
    print "there are numbers in $string\n";
  }
  ```

Sometimes not only you want to know if the pattern exists in a string, but also what it actually matched. In that case, use the parentheses to indicate the matched substring you want to know, and they will be assigned to the special $1, $2, ..., variables if the match is successful:

-
  ```
  if ($string =~ /(\d+)\s+(\d+)\s+(\d+)/) {
    print "first three matched numbers are $1, $2, $3 in $string\n";
  }
  ```

Note that all three numbers above must be found for the whole pattern to match successfully, thus $1, $2 and $3 should be defined when the if statement is true. The same *memory* of matched substrings within the regular expression are \1, \2, \3, etc. So, to check if the same number happened twice in the $string, you can do this:

-
  ```
  if ($string =~ /(\d).+\1/) {
    print "$1 happened at least twice in $string\n";
  }
  ```

You cannot use $1 in the pattern to indicate the previously matched number because $ means the end of the line inside the pattern. Use \1 instead.

**Pattern substitution:**

In addition to matching a pattern, you can replace the matched substring with a new string using the substitution operator. In this case, just write the substitution string after the pattern to match and replace:

- `$string =~ s/\d+/0/; # replace a number with zero`
- `$string =~ s:/:\\:; # replace the forward slash with backward slash`

Unlike the pattern matching operator, the substitution operator **does** change the $string if a match is found. The second example above indicates that you do not always need to use / to break the pattern and substitution parts apart; you can basically use any symbol right after the s operator as the separator. In the second case above, since what we want to replace is the forward slash symbol, using it to indicate the pattern boundary would be very cumbersome and need a lot of escape characters:

- $string =~ s/\//\\/; # this is the same but much harder to read

For pattern matching, you can also use any separator by writing them with m operator, i.e., m:/: will match the forward splash symbol. Natually, the substitution string may (and often does) contain the \1, \2 special memory substrings to mean the just matched substrings. For example, the following will add parentheses around the matched number in the source $string:

- $string =~ s/(\d+)/(\1)/;

The parentheses in the replacement string have no special meanings, thus they were just added to surround the matched number.

**Modifiers to pattern matching and substitution:**

You can add some suffix modifiers to Perl pattern matching or substitution operators to tell them more precisely what you intend to do:

- /g tells Perl to match *all* existing patterns, thus the following prints all numbers in $string
  ```
  while ($string =~ /(\d+)/g) {
    print "$1\n";
  }
  ```
- $string =~ s/\d+/0/g; # replace all numbers in $string with zero
- /i tells Perl to ignore cases, thus
  $string =~ /abc/i; # matches AbC, abC, Abc, etc.
- /m tells perl to ignore newlines, thus
  "a\na\na" =~ /a$/m will match the last a in the $string, not the a before the first newline if /m is not given.

**Perl- Subroutines:**

A Perl subroutine or function is a group of statements that together performs a task. You can divideup your code into separate subroutines. How you divide up your code among different subroutines is up to you, but logically the division usually is so each function performs a specific task.

**Define and Call a Subroutine**

The general form of a subroutine definition in Perl programming language is as follows −

```
sub subroutine_nam e{
body of the subroutine
}
```

The typical way of calling that Perl subroutine is as follows –

```
subroutine_nam e( list of argum ents );
```

Let's have a look into the following example, which defines a simple function and then call it. Because Perl compiles your program before executing it, it doesn't matter where you declare your
subroutine.

```
#!/usr/bin/perl
# Function definition
sub Hello{
print "Hello, World!\n";
}
# Function call
Hello();
```

When above program is executed, it produces the following result −

```
Hello, World!
```

**Passing Arguments to a Subroutine**

You can pass various arguments to a subroutine like you do in any other programming language and they can be acessed inside the function using the special array @_. Thus the first argument to the function is in [ 0], *thesecondisin*_[1], and so on.

You can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities.

**Passing Lists to Subroutines**

Because the @_ variable is an array, it can be used to supply lists to a subroutine. However, because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract the individual elements from @_. If you have to pass a list along with other scalar arguments, then

make list as the last argument as shown below −

```
#!/usr/bin/perl
# Function definition
sub PrintList{
m y @ list = @ _;
print "Given list is @ list\n";
}
$ a = 10;
@ b = (1, 2, 3, 4);
# Function call with list param eter
PrintList($ a, @ b);
```

When above program is executed, it produces the following result −

```
Given list is 10 1 2 3 4
```

**Passing Hashes to Subroutines**

When you supply a hash to a subroutine or operator that accepts a list, then hash is automatically
translated into a list of key/value pairs. For example −

```
#!/usr/bin/perl
# Function definition
sub PrintHash{
m y (%hash) = @ _;
foreach m y $ key ( keys %hash ){
m y $ value = $ hash{$ key};
print "$ key : $ value\n";
}
}
%hash = ('nam e' => 'Tom ', 'age' => 19);
# Function call with hash param eter
PrintHash(%hash);
```

When above program is executed, it produces the following result −

```
nam e : Tom
age : 19
```

**Returning Value from a Subroutine**

You can return a value from subroutine like you do in any other programming language. If you are
not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value.

You can return arrays and hashes from the subroutine like any scalar but returning more than one
array or hash normally causes them to lose their separate identities. So we will use references *explainedinthenextchapter* to return any array or hash from a function.

Let's try the following example, which takes a list of numbers and then returns their average −

```
#!/usr/bin/perl

# Function definition

sub Average{

# get total num ber of argum ents passed.

$ n = scalar(@ _);

$ sum = 0;

foreach $ item (@ _){

$ sum += $ item ;

}

$ average = $ sum / $ n;

return $ average;

}

# Function call

$ num = Average(10, 20, 30);

print "Average for the given num bers : $ num \n";
```

When above program is executed, it produces the following result −

```
Average for the given num bers : 20
```