Perl: Introduction

Perl can be viewed as a programming or scripting language or both depending on its use. Larry Wall wrote the language when he couldn't get things done with just shell, awk, sed, grep, cut, sort, etc. Retronyms of Perl include “Practical Extraction and Report Language” and “Pathogically Eclectic Rubbish Lister”

It is quite easy to learn, especially if one already knows C programming language. It is a natural progression to Perl if you already know Shell Scripting. Like Vim, there are different ways of accomplishing the same task. Perl's easy to use and flexible syntax is both a boon and a curse. There's a joke on Perl that its programs look the same before and after encryption. A style guide can be found here

Hello User

```
#!/usr/bin/perl

# Clear the terminal screen - example of shell interaction
# Statements in Perl end with ; (similar to C)
system("clear");
# Assign output of shell command to scalar variable
# \ is used to escape $ symbol, avoids Perl variable inference
$usrname = `echo \$USER`;
# Print greeting message
print "Hello $usrname";

# Forever loop to get user input
# Exit program if the string entered is a palindrome
for(;;)
{
  print "Enter a palindrome to quit: ";
  # Get user input, similar to scanf in C
  $usrinput = <STDIN>;
  # chomp function is used to remove trailing newline
  chomp($usrinput);
  # reverse the input string and compare if it is same
  if($usrinput eq (reverse $usrinput))
  {
    # exit the forever loop
    # last is similar to break in C
    last;
  }
}
```

Introductory perl script - hello_user.pl

The first line is a special type of comment – it instructs the shell to execute this file as a Perl program. The rest of the program shows example of how Perl can seemlessly interact with shell as well as uses elements common in programming languages. To run the script, give execute permission using **chmod** command first.

```
>which perl
/usr/bin/perl
>chmod +x hello_user.pl
>./hello_user.pl
```

Executing hello_user.pl

Debugging Errors and Strict usage

Perl provides ways to reduce errors and warning messages

- use warnings; → enables Perl's inbuilt warning feature
- use strict; → add this statement just after the interpreter line to force variable declaration, etc. Helpful to avoid errors due to variable name typos, etc. An execellent forum on this topic here

```
#!/usr/bin/perl
use strict;
# #!/usr/bin/perl -w is another way of activating warnings
use warnings;

# strict forces variable declaration before usage
# my keyword makes the variable local to the scope it is declared
my $usrname;

system("clear");
$usrname = `echo \$USER`;
print "Hello $usrname";

for(;;)
{
    print "Enter a palindrome to quit: ";
    # usrinput is local to this for-loop
    chomp(my $usrinput = <STDIN>);
    # Process input string
    # Convert UPPERCASE alphabets to lowercase
    # tr is similar to the Linux shell command tr
    $usrinput =~ tr/A-Z/a-z/;
    # Remove characters other than alphabets and numbers
    $usrinput =~ s/[^a-z0-9]//g;

    # check if input string is atleast 3 characters long
    if(length($usrinput) < 3)
    {
        print "Input string should atleast be 3 characters long\n";
        # start loop again, next is similar to continue in C
        next;
    }
    # check if input string has atleast two different characters
    elsif($usrinput =~ m/^(.)\1+$/g)
    {
        print "Input string should have atleast two different
characters\n";
        next;
    }
    # quit program if input is valid palindrome
    last if($usrinput eq (reverse $usrinput));
}
# Define exit value, 0 is considered as success
# non-zero values indicate something went wrong
exit 0;
```

A better hello_user.pl

Perl Debugger

There is a command line perl debugger one can use when things go awry. Usually, a cursory glance by your peer can spot easy typos and mistakes. If it something crazy, make use of perl debugger before cluttering the program with print statements (No need to change/add anything in the program to use debugger)

1. **perl** -d hello_user.pl → invoke perl debugger, it will stop at first statement and give a debugger prompt
2. v → print code around the current statement the debugger is at, useful to visualize the progress of debug effort
3. n → execute next statement
4. p $usrname → print value of a variable
5. h → help on debugger features, how to set breakpoints, etc
6. q → quit the debugger

```
>perl -d hello_user.pl

Loading DB routines from perl5db.pl version 1.39_10
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(hello_user.pl:8):        my $usrname;
  DB<1> n
main::(hello_user.pl:10):       system("clear");
  DB<1> p $usrname

  DB<2> q
>
```

Using Perl Debugger

Perl Version

To know the Perl version installed, use the **perl** -v shell command. This tutorial is based on Perl 5. The latest version is Perl 6, which isn't as prevalent as Perl 5 in the industry and not yet production ready according to perl website

Suggested Reading

- Must read about Perl tutorials
- Perl quick guide, also has detailed tutorials
- Offical Perl Documentation
- Modern Perl book
- Learn Perl

Perl: Variables

Unlike C where variables are declared like int, char, etc, Perl has only three type of variables – Scalar, Array and Hashes. There are also some special, default and predefined variables which makes Perl easy to use and reduce coding

Scalar Variable

A scalar variable holds a single value – can be integer, character, string, float, etc. Perl treats them according to context. Scalar variable is prefixed by $

```
# Integer
$year = 2015;
# Random Integer (0,1,2,3,4)
$rand_num = int(rand(5));
# Octal
$octal_number = 012;
# Hex
$hex_number = 0xA12;
# Floating point
$pi = 3.14;
$large_number = 1.983e23;
# String
$sweet = "Kinder Joy";
$number_string = "   100    ";
# String concatenation
$fav = "I like "."$sweet";
# Mix 'n' Match
$num = $number_string + 1;
$year_str = "Year is: "."$year";
$int_value_oct = int($octal_number);
$int_value_hex = int($hex_number);
```

Scalar Variable Examples

Array Variable

Array variables are used to hold multiple values, they can be made up of any number of scalar variables. So, a single array can contain integer, string, character, etc. Size of array need not be declared, it can increased or decreased at any time. Array variable is prefixed by @

```
# Different ways of declaring arrays
@place = ("Bangalore", "Chennai", "Coimbatore");
@prime_numbers = (2, 3, 5, 7);
# Note the use of $ when it is individual element
$book[0] = "Harry Potter";
$book[1] = "Sherlock Holmes";
$book[2] = "To Kill a Mocking Bird";
# prints all elements of array separated by space
print "@place\n";
print "@prime_numbers\n";
print "@book\n";
```

Initializing Array Variable

```
# prints individual element of array
print "$place[1]\n";
# prints last element of array, same as $place[$#place]
print "$place[-1]\n";
# prints second last element of array
print "$place[-2]\n";
# prints 3 5 7 - Note use of @
print "@prime_numbers[1..3]\n";
# Size of array
# No. of elements-1, prints 2
print "$#place\n";
# No. of elements, $arr_size = 3
$arr_size = @place;
print "$arr_size\n";
```

Accessing Array elements and Array size

```
# Add an element, same as $place[$#place + 1]
$place[3] = "Salem";
# Delete first element and shift left
shift(@place);
# Delete last element
pop(@place);
# Add an element at end of array
push(@place, "Mysore");
# Add an element at beginning of array
unshift(@place, "Hyderabad");
# Manipulate array size - just assign value to $#place
# Change no. of elements to 3
$#place = 2;
# Change no. of elements to 0
$#place = -1;
```

Adding/Removing elements of Array and changing size

Hash Variable

Hash, or associative array, is similar to array, but each element has a key associated with it. Hash variable is prefixed with %

```
# Initializing Hash variable
%marks = ('Rahul' => 86, 'Ravi Kumar' => 92, 'Rohit Sharma' => 75);
# Accessing individual element, key is 'Rahul'
print "$marks{'Rahul'}\n";
# Adding element
$marks{'Rajan'} = 79;
print "$marks{'Rajan'}\n";
# Delete element
delete $marks{'Ravi Kumar'};
# Accessing all keys
@marks_keys = keys %marks;
print "@marks_keys\n";
# Accessing all values
@marks_values = values %marks;
print "@marks_values\n";
```

Hash Variable

List

Lists are sort of array without a variable name, defined within (). We have already seen lists being used to initialize Array and Hash variables

```
# Assigning multiple variables in one statement
($str, $num, $char) = ("Good day", 2, x);
($str1, $str2) = split/\s/,$str;
# Handy way to declare a sequential array of numbers
@digits = (0..9);
# Scalar and Array types can be freely mixed
($digits_0, $digits_1, @digits_rem)=($digits[0], $digits[1], @digits[2..9]);
# @_ is a special variable holding subroutine args
($arg1, $arg2) = @_;
```

Versatile usage of Lists

Reference

Similar to pointers in C, Perl provides references

```
# Scalar Reference
$year = 2015;
$scalar_ref = \$year;
# Use $ before reference variable to get Scalar value
print "$$scalar_ref\n";
# Array Reference
@prime_numbers = (2, 3, 5, 7);
$array_ref = \@prime_numbers;
# Use @ before reference variable to get array
print "@$array_ref\n";
# Use $ before reference variable to get individual element
print "$$array_ref[1]\n";
```

References in Perl

Variable Scope

- my $local_variable → local variable, visible only within code block defined by {} or entire file when declared outside {}
- our $global_variable → global variable

Suggested Reading

- Difference between my and our
- Scope of Variables

Special Variables

Perl provides various predefined variables to ease coding lie $_, @_, @ARGV, $0, etc. They will be explained as and when encountered in this tutorial. Do check out this lesson for detailed explanation

Perl: Regular Expressions

Perl and regular expressions are almost always spoken in same breadth, after all, popular retronym for Perl is "**Practical Extraction and Report Language**". **grep** command even allows -P option to accept Perl regular expressions for those so much used to Perl regex

- regexr site allows to learn, test, share a regular expression. For example, check this to see how the palindrome validation regex used in introductory example works
- There are plenty of other wonderful sites to learn regex – regexcrossword learn and play games, regespresso on iOS, regexgen generates regex from user requirement – app on Android, etc

Metacharacter	Description
^	Match from beginning of string
$	Match end of string
.	Match any character except newline
|	OR operator for matching multiple patterns
()	Enclosed pattern is extracted/reused
[]	Character class – match one character among many
Pattern Qualifier	**Description**
*	Match zero or more times the preceding character
+	Match one or more times the preceding character
?	Match zero or one times the preceding character
{n}	Match exactly n times
{n,}	Match atleast n times
{n,m}	Match atleast n times but not more than m times
Character Patterns	**Description**
\d	Match a digit [0-9]
\D	Match non-digit [^0-9]
\w	Match alphanumeric and underscore character [a-zA-Z_]
\W	Match non-alphanumeric and underscore character [^a-zA-Z_]
\s	Match white-space character – space,tab,etc
\S	Match non white-space character
\t	Match horizontal tab character
Modifier	**Description**
g	Match all patterns in string, not just the first pattern
i	Ignore case
Variable	**Description**
$1, $2, $3...	Matched patterns inside ()
$`	String before matched pattern
$&	Matched pattern
$'	String after matched pattern

Sample of Perl Regular Expression elements

Pattern Matching

- =~ → check if pattern is matching
- !~ → check if pattern is NOT matching
- Use list to assign matched pattern(s) to variables, modifier g is compulsory

```
my $line = "This is a test line";
if($line =~ m/a/)
{
    print "Before match: $`\n";      # This is
    print "Matched Pattern: $&\n";   # a
    print "After match: $'\n";       # test line
}
print "No x in \$line\n" if($line !~ m/x/);
my ($str) = $line =~ m/is a (.*) line/g;      # test
# Create array of words
my (@words) = $line =~ m/([a-z]+)/ig;
# No need to use () if only one pattern is specified
my (@four_letter_words) = $line =~ m/[a-z]{4}/ig;
# Scalar and Array can freely be mixed
my ($first, @rem_words) = $line =~ m/[a-z]+/ig;
```

Pattern matching and usage of Perl Special Variables

Translate

Translate character(s) to corresponding character(s). Regular expressions cannot be used here except for range of characters like a-f, 0-9, A-Z, etc

```
my $str = "SHELL";
my $line = "This is your's";
my $lose = "Don't loose hope";
my $words = "Only23 wo2r4ds: here;";
# Change uppercase to lowercase
$str =~ tr/A-Z/a-z/;          #shell
# Delete characters with d modifier
$line =~ tr/'//d;             #This is yours
# Delete repeated characters with s modifier
$lose =~ tr/o/o/s;            #Don't lose hope
# Complement pattern with c modifier
$words =~ tr/a-zA-Z //cd;     #Only words here
```

Translate examples

Search and Replace

```
my $str = "Sample string";
my $date = "14/04/2015";
my $line = "i always forget to capitalize i";
# Change Sample to test
$str =~ s/sample/test/i;              #test string
# Inter change date and month
$date =~ s|(\d+)/(\d+)/|$2/$1/|;      #04/14/2015
# Change word i to I
$line =~ s/\bi\b/I/g;                 #I always forget to capitalize I
```

Substitute examples

Perl: Control Structures

Most of control structures in Perl are very similar to those in C. Perl also provides some useful constructs borrowed from shell scripting and other code efficient ones

if-elsif-else

We have already seen if control structure in the introductory example. Below is the general syntax. Always remember that braces {} are required even for a single statement within the control structure – unless the single statement precedes the control structure

```
# if-elsif-else syntax
# Braces {} are necessary even for single statement
if(condition)
{
    statements;
}
elsif(condition)
{
    statements;
}
else
{
    statements;
}
# Multiple statements inside if
if(condition)
{
    statements;
}
# if-elsif syntax
if(condition)
{
    statements;
}
elsif(condition)
{
    statements;
}
# if-else syntax
if(condition)
{
    statements;
}
else
{
    statements;
}
# Single statement if, avoids {}
statement if(condition);
```

if-elsif-else syntax

for-loop

The for-loop control structure is again very similar to construct in C. The keywords to alter the behavior of loop are different though as seen in the introductory example which required a forever loop

- next → skip remaining statements in the loop and start next iteration
- last → skip remaining statements in the loop and exit the loop

```
#!/usr/bin/perl
use strict;
use warnings;

my $user_input;
print "Enter a line and I'll give word count: ";
chomp($user_input = <STDIN>);

# Split the input string based on white-spaces
my @words = split/\s+/,$user_input;
my $count = 0;
print "The words are: \n";
for(my $i = 0; $i <= $#words; $i++)
{
    print "$words[$i]\n";
    $count++;
}
print "$user_input has $count words\n";
```

for-loop example

foreach-loop

Iterating through an array element by element is a common operation. So, Perl provides an easy short-cut – the foreach loop

```
#!/usr/bin/perl
use strict;
use warnings;

my $user_input;
print "Enter a line and I'll give word count: ";
chomp($user_input = <STDIN>);

# Split the input string based on white-spaces
my @words = split/\s+/,$user_input;
my $count = 0;
print "The words are: \n";
foreach my $i (@words)
{
    # $i will automatically get the value of each array element
    print "$i\n";
    $count++;
}
print "$user_input has $count words\n";
```

foreach-loop example

In the preceding example, $i was declared and used just to hold each element of an array. For this, and lot other similar use cases, Perl provides another short-cut – the use of $_ special variable

```
foreach (@words)
{
    # $_ will automatically get the value of each array element
    print "$_\n";
    $count++;
}
```

Using $_ special variable

while-loop

The while loop syntax is given below. It is widely used in handling files. For other types of loop and more detail on loops, check this tutorial

```
while(condition)
{
    statements;
}
```

while-loop syntax

Perl: Arguments, Filehandles and Subroutines

Command line arguments passed to Perl script are stored in the special array variable @ARGV. The name of the script invoked is stored in $0. The same script can be saved under another name (either soft link or hard link) and by testing value of $0, it can be made to behave differently

```
#!/usr/bin/perl
use strict;
use warnings;

my $output;
my $string;

# Remove ./ from script name
$0 =~ s|./||;
# For each file name given as command line argument
foreach (@ARGV)
{
    # Sanity check if File exists or not
    unless(-e $_)
    {
        print "Error! File $_ not found\n";
        next;
    }
    # Count lines
    if($0 eq "count_lines.pl")
    {
        $output = `wc -l < $_`;
        $string = "lines";
    }
    # Count words
    elsif($0 eq "count_words.pl")
    {
        $output = `wc -w < $_`;
        $string = "words";
    }
    # Display output
    print "No. of $string in $_ is: $output";
}
```

Command line arguments and use of $0 special variable

```
> gvim count_lines.pl
> ln -s count_lines.pl count_words.pl
> ls -ltrh
total 4.0K
-rwxr-xr-x 1 guest guest 387 Apr  7 16:33 count_lines.pl
lrwxrwxrwx 1 guest guest  14 Apr  7 16:34 count_words.pl -> count_lines.pl
> ls -ltrh > test_file.txt
> ./count_lines.pl test_file.txt
No. of lines in test_file.txt is: 4
> ./count_words.pl test_file.txt
No. of words in test_file.txt is: 31
>
```

Command line argument and $0 in action

Filehandle

In C, even the simplest of file operations could be daunting. Many would just skip file handling in first year of engineering. Fret not, Perl is renowned for its ease of text processing. One of the reason being the easy to use Filehandles. Read this article for in-depth explanation. It is a convention to name filehandles in uppercase

```
open(IP,"<input.txt");    # Open file for reading, IP is filehandle
open(IP,"input.txt");     # < symbol is optional to read file
open(OUT,">output.txt"); # Open file for writing, deletes existing
content if any
open(APPEND,">>result.txt");  # Open file to append data
```

Filehandles

```
#!/usr/bin/perl
use strict;
use warnings;

# Displaying help
if( ($#ARGV == -1) || ($ARGV[0] eq "-h") || ($ARGV[0] eq "h") ||
($ARGV[0] eq "help") )
{
    print
"-------------------------------------------------------------\n";
    print "This script counts and displays word count of text file\n";
    print "The words will be displayed in lowercase and alphabetical
order\n";
    print "Provide a filename as command line argument\n";
    print
"-------------------------------------------------------------\n";
    # Otherwise script would proceed with -h/h/help treated as filename
    exit;
}

# Hash variable to store words as keys and word count as value
my %count;
# Avoid \n in error message if you want Perl to display line number of
die statement
open(IP,"$ARGV[0]") || die "Cannot open file $ARGV[0] for reading\n";
# Loop through input file, line by line
while(<IP>)
{
    # Each line is stored automatically in $_
    # $_ is acted upon when variable is not specified
    tr/A-Z/a-z/;
    # remove all characters except alphanumeric and space characters
    s/[^a-zA-Z0-9\s]//g;
    my @arr = split/\s+/,$_;
    # increment value of each word, initial value is zero!
    $count{$_}++ foreach (@arr);
}
# Close Filehandle after reading is done
close(IP);
# print each word alphabetically and its count
print "$_ $count{$_}\n" foreach (sort keys %count);
```

Display count of unique words

```
#!/usr/bin/perl
use strict;
use warnings;
# Script to mimic - sed 's/search/replace/g' -i file_name(s)
print "Enter search pattern: ";
chomp(my $search_pattern = <STDIN>);
print "Enter replace pattern: ";
chomp(my $replace_pattern = <STDIN>);
foreach my $file (@ARGV)
{
    open(IP,"$file") or die "Cannot open $file for reading\n";
    open(TMP,">tmp.txt") or die "Cannot open tmp.txt for writing\n";
    while(<IP>)
    {
        $_ =~ s/$search_pattern/$replace_pattern/g;
        # Note how Filehandle is used to print to file
        print TMP $_;
    }
    # Close Filehandle before performing another action!
    close(IP);
    close(TMP);
    # Perform inplace text replacement as sed does with -i option
    system("mv tmp.txt $file");
}
```

Writing to a file

Subroutine

```
#!/usr/bin/perl
use strict;
use warnings;

# subroutine call with arguments
my $total = add(3,4);
print "3 + 4 = $total\n";
# subroutine without arguments
print_date();

sub add
{
    # @_ --> array of arguments passed to subroutine
    ($a,$b) = @_;
    my $sum = $a + $b;
    # Above two lines can be shortened as my $sum = $_[0] + $_[1];
    return $sum;
}

sub print_date
{
    my $date = `date`;
    print "Today is $date";
}
```

Subroutine examples