Perl 101 (part 6) – The Perl Toolbox

By Vikram Vaswani and Harish Kamath

Table of Contents

Stringing Things Along

With the arcane mysteries of Perl subroutines behind us, it's now time to move on to the many practical uses of Perl. Over the next few pages, we're going to spend some time investigating Perl's many built−in functions, functions that come in very handy when you need to do a little string manipulation.

Among the items on today's agenda: pattern matching and replacement, a close look at some useful string functions, and a quick trip back in time to math class with Perl's math functions. Pay attention, now!

Expressing Yourself

One of Perl's most powerful features is the ability to do weird and wonderful things with "regular expressions". To the uninitiated, regular expressions, or "regex", are patterns which are built using a set of special characters; these patterns can then be compared with text in a file, or data entered into a Web form. A pattern match can trigger some kind of action...or not, as the case may be.

Though regular expressions can get a little confusing when you're first starting out with them, a little patience will reap rich rewards, as they can save you a tremendous amount of work. Our very first example illustrates a simple pattern−matching operation, and introduces you to Perl's match operator:

```
#!/usr/bin/perl # get a line of input print "Gimme a line!\n";
$line = ; chomp ($line); # get a search term print "Gimme the
string to match!\n"; $term = ; chomp ($term); # check for
match if ($line =~ /$term/) { print "Match found!\n"; } else {
print "No match found\n"; }
```

A quick explanation is in order here. In Perl, the "pattern" is the sequence of characters to be matched − this pattern is usually enclosed within a pair of slashes. For example,

/xyz/ represents the pattern "xyz".

The example above asks for a line of text and a search term − this search pattern is then used with the match operator =~ to test for a match. The result of the =~ operation is true if the pattern is found in the string, and false if not.

Here's the output of the example above:

```
Gimme a line! I'll be back Gimme the string to match! b Match
found!
```

Perl also has the !~ operator, which does the reverse of the =~ operator − it returns true if a match is NOT found.

Engry Young Men

In addition to simple matching, Perl also allows you to perform substitutions.

Take a look at our next example, which prompts you to enter a line of text, and then replaces all occurrences of the letter "a" with the letter "e".

```
#!/usr/bin/perl # get a line of input print "Gimme a line!\n";
$line = ; chomp ($line); # substitute with the substitution
operator $line =~ s/a/e/; # and print print $line;
```

In this case, we've used Perl's substitution operator − it looks like this:

```
s/search−pattern/replacement−pattern/
```

In the example above, the line

```
$line =~ s/a/e/;
```

simply means "substitute a with e in the scalar variable $line".

And here's the output:

```
Gimme a line! angry young man engry young man
```

Ummm...didn't work quite as advertised, did it? All it did was replace the first occurrence of the letter "a". How about adding the "g" operator, which does a global search−and−replace?

```
#!/usr/bin/perl # get a line of input print "Gimme a line!\n";
$line = ; chomp ($line); # substitute with the substitution
operator $line =~ s/a/e/g; # and print print $line;
```

And this time,

```
angry young man
```

is replaced with

```
engry young men
```

Much better! But what if your sentence contains an upper−case "a", also known as "A". Well, that's why Perl also has the case−insensitivity operator "i", which takes care of that last niggling problem:

```
#!/usr/bin/perl # get a line of input print "Gimme a line!\n";
$line = ; chomp ($line); # substitute with the substitution
operator $line =~ s/a/e/gi; # and print print $line;
```

And the output:

```
Gimme a line! Angry young man Engry young men
```

Of course, we're just scratching the tip of the regex iceberg here. Things get even more interesting when you start using patterns and metacharacters instead of actual words...

Aardvark, Anyone?

The example you just saw was pretty cut−and−dried − decide search term, decide replacement term, shove both into Perl script, bang bang and Bob's your uncle. But what happens if you need to replace a class of words, rather than a single word? If, for example, you need to replace not just the letter "a", but words beginning with one or more "a" and ending with a "k"...like "aardvark", for example?

The special characters you'll use to modify your pattern are called "metacharacters", which is another of those words that sounds impressive but means absolutely nothing useful. We'll explain some of them here, and point you to a resource for more information a little further down.

Two of the more useful metacharacters in Perl are the "+" and "*" characters, which match "one or more" instances and "zero or more" instances of the preceding pattern respectively.

For example,

```
/boo+/
```

would match "book", and "booo" but not "bottle", while

```
/bo*/
```

would match all of "bottle", "bog", "book", and "booo". One common use of Perl pattern−matching is to remove unnecessary blank spaces from a block of text. This is accomplished using the "\s" metacharacter, which matches whitespace, tab stops and newline characters. The simple substitution

```
/\s+/ /
```

would "substitute zero or more occurrences of whitespace with nothing"

Perl also has two important "anchor" characters, which allow you to build patterns which specify which characters come at the beginning or end of a string. The ^ character is used to match the beginning of a string, while the $ character is used to match the end. So,

```
/^h/
```

would match "hello" and "house", but not "shop"

while

```
/g$/
```

would match "dig" and "bog", but not "gold" or "eagle"

And you can even specify more than one pattern to match with the | operator.

```
/(the|a)/
```

would return true if the string contained either "the" or "a".

Obviously, regular expressions is a topic in itself − if you're interested, we've put together a well−written, comprehensive guide to the topic at http://www.devshed.com/Server_Side/Administration/RegExp/

Needles In Haystacks

Perl comes with a wide variety of functions that come in handy when manipulating strings. The first one on our list is the length() function, which returns the length of a specified string.

Here's an example of how the length() function can be used to restrict the length of a login name to between six and ten characters:

```
#!/usr/bin/perl do { # ask for a name print ("Please enter a
username:"); $username = ; chomp($username); # and repeat
until the username is between 6 and 10 characters long } while
((length($username) < 6) || (length($username) > 10)); print
"A new star is born...and its name is $username!\n";
```

In this case, each time a username is entered, we use the length() function to count the number of characters in it. If this number is less than 6, or greater than 10, the loop is repeated until a username of the correct length is entered.

Here's what it looks like:

```
Please enter a username:me Please enter a username:galapaloozy
Please enter a username:godzilla A new star is born...and its
name is godzilla!
```

The next string function that we're going to unravel is the index() function. This function is typically used to find out if a particular pattern exists within a string. Here's what it looks like:

```
$var = index(string, pattern)
```

where "string" is the string to be searched for "pattern". If the pattern is found within the string, the position of the first character of the matched pattern will be assigned to the variable $var; if not, $var will be assigned the value −1.

Here's a quick example:

```
#!/usr/bin/perl # how to find a needle in a haystack #
Perl-style print "THE HAYSTACK\n"; print "------------\n"; #
set up the string $haystack = "211643831 923465971315874643
13729247620352625 9235923595232305232095 8284529 2347392901847
32393482562502925352395327202358"; print $haystack . "\n"; #
ask for a search term print "Gimme a needle: "; $needle = ;
chomp ($needle); # use index() to look for the string
$location = index($haystack, $needle); # print appropriate
message if ($location >= 0) { print "Needle located $location
```

```
characters deep in the haystack\n"; } else { print "Sorry,
this haystack contains no needle\n"; }
```

And here's the output:

```
THE HAYSTACK ------------ 211643831 923465971315874643
13729247620352625 9235923595232305232095 8284529 2347392901847
32393482562502925352395327202358 Gimme a needle: 1267 Sorry,
this haystack contains no needle THE HAYSTACK ------------
211643831 923465971315874643 13729247620352625
9235923595232305232095 8284529 2347392901847
32393482562502925352395327202358 Gimme a needle: 6250 Needle
located 101 characters deep in the haystack
```

In this case, we've set up a string containing a set of random numbers. The user is then asked to enter a number of his own choice, and the index() function is used to scan the string for the number. Depending on the result, an appropriate message is printed.

Similar to the index() function is the rindex() function, which also searches for a pattern within the specified string, but starts from the end.

Slice And Dice

Next up, the substr() function. As the name implies, this is the function that allows you to slice and dice strings into smaller strings. Here's what it looks like:

```
substr(string, start, length)
```

where "string" is a string or a scalar variable containing a string, "start" is the position to begin slicing at, and "length" is the number of characters to return from "start".

Here's a Perl script that demonstrates the substr() operator:

```
#!/usr/bin/perl # get a string print "Gimme a line!\n"; $line
= ; chomp ($line); # get a chunk size print "How many
characters per slice?\n"; $num_slices = ; chomp ($num_slices);
$length = length($line); $count = 0; print "Slicing...\n"; #
slice the string into sections while (($num_slices*$count) <
$length) { $temp = substr($line, ($num_slices*$count),
$num_slices); $count++; print $temp . "\n"; }
```

Here, after getting a string and a block size, we've used a "while" loop and a counter to keep slicing off pieces of the string and displaying them on separate lines.

And here's what it looks like:

```
Gimme a line! The cow jumped over the moon, giggling madly as
a purple pumpkin with fat ears exploded into confetti How many
characters per slice? 11 Slicing... The cow jum ped over th e
moon, gig gling madly as a purpl e pumpkin w ith fat ear s
exploded into confet ti
```

You've already used the print() function extensively to sent output to the console. However, the print() function doesn't allow you to format output in any significant manner − for example, you can't write 1000 as 1,000 or 1 as 00001. And so clever Perl programmers came up with the printf() function, which allows you to define the format in which data is printed to the console.

Consider a simple example − printing decimals:

```
#!/usr/bin/perl print (5/3);
```

And here's the output:

```
1.66666666666667
```

As you might imagine, that's not very friendly. Ideally, you'd like to display just the "significant digits" of the result. And so, you'd use the printf() function:

```
#!/usr/bin/perl printf "%1.2f", (5/3);
```

which returns

```
1.67
```

A quick word of explanation here: the Perl printf() function is very similar to the printf() function that C programmers are used to. In order to format the output, you need to use "field templates", templates which represent the format you'd like to display.

Some common field templates are:

```
%s string %c character %d decimal number %x hexadecimal number
%o octal number %f float number
```

You can also combine these field templates with numbers which indicate the number of digits to display − for example, %1.2f implies that Perl should only display two digits after the decimal point.

Here are a few more examples of printf() in action:

```
printf("%05d", 3); # returns 00003 printf("$%2.2f", 25.99); #
returns $25.99 printf("%2d%", 56); # returns 56%
```

And here's a calculator which uses the printf() function to display numbers in various numerical bases like hexadecimal and octal.

```
#!/usr/bin/perl print "Enter a number: "; chomp($number = );
printf("In decimal format: %d\n",$number); printf("In
hexadecimal format: %x\n",$number); printf("In octal format:
%o\n",$number);
```

Perl also comes with a sprintf() function, which is used to send the formatted output to a variable instead of standard output.

Going Backwards

The next few string functions come in very handy when adjusting the case of a text string from lower−to upper−case, or vice−versa:

```
lc($string) − convert $string to lower case uc($string) −
convert $string to upper case lcfirst($string) − convert the
first character of $string to lower case ucfirst($string) −
convert the first character of $string to upper case
```

Here's an example:

```
#!/usr/bin/perl # get a string print "Say something: ";
chomp($string = ); # convert case $output= lc($string); print
"All lower case: $output\n"; $output= uc($string); print "All
upper case: $output\n"; $output= lcfirst($string); print "Look
at the first character: $output\n"; $output =
ucfirst($string); print "Look at the first character:
$output\n";
```

And here's the output:

```
Say something: Something's rotten in the state of Denmark All
lower case: something's rotten in the state of denmark All
upper case: SOMETHING'S ROTTEN IN THE STATE OF DENMARK Look at
the first character: something's rotten in the state of
Denmark Look at the first character: Something's rotten in the
state of Denmark
```

The reverse() function is used to reverse the contents of a particular string.

```
#!/usr/bin/perl # ask for input print "Say something: ";
$something = ; chomp ($something); # reverse and print
$gnithemos = reverse($something); print "Sorry, you seem to be
talking backwards − what does $gnithemos mean?";
```

Here's the output:

```
Say something: God, I'm good Sorry, you seem to be talking
backwards − what does doog m'I ,doG mean?
```

And the chr() and ord() functions come in handy when converting from ASCII codes to characters and vice−versa. For example,

```
print chr(65); returns A while print ord("a"); returns 97
```

Math Class

And finally, Perl also comes with a set of math functions that allow you to carry out complex mathematical operations. You probably won't need these, but you should at least know of their existence.

Sine of a angle: sin($radians)

Cosine of a angle: cos($radians)

Square root of a number: sqrt($variable)

Exponent of a number: exp($variable)

Natural logarithm of a number: log($variable)

Absolute value of a number: abs($variable)

Decimal value of a number from hexadecimal: hex($variable)

Decimal value of a number from octal: oct($variable)

Integer portion of a number: int($variable)

And here's an example that demonstrates all these:

```
#!/usr/bin/perl # set up the choices print "Pick from the
choices below:\n"; print "Sine of an angle[1]\n"; print
"Cosine of an angle[2]\n"; print "Square root of a
number[3]\n"; print "Exponent of a number[4]\n"; print
"Natural logarithm of a number[5]\n"; print "Absolute value of
a number[6]\n"; print "Decimal value of a number from
hexadecimal[7]\n"; print "Decimal value of a number from
octal[8]\n"; print "Integer value[9]\n"; chomp($choice = ); #
and process them if ($choice == 1 || $choice == 2) { print
"Enter the angle in radians: "; chomp($angle = ); if($choice
== 1) { $value = sin($angle); print("Sine of $angle is
$value\n"); } else { $value = cos($angle); print("Cosine of
$angle is $value\n"); } } elsif($choice == 3) { print "Enter a
positive number: "; chomp($number = ); $value = sqrt($number);
print("The square root of $number is $value\n"); }
elsif($choice == 4) { print "Enter a number: "; chomp($number
= ); $value = exp($number); print("e ** $number = $value\n");
} elsif($choice == 5) { print "Enter a number: ";
chomp($number = ); $value = log($number); print("The natural
log of $number is $value\n"); } elsif($choice == 6) { print
"Enter a number: "; chomp($number = ); $value = abs($number);
print("The absolute value of $number is $value\n"); }
elsif($choice == 7) { print "Enter a number: "; chomp($number
```

```
= ); $value = hex($number); print("The decimal value of
$number is $value\n"); } elsif($choice == 8) { print "Enter a
number: "; chomp($number = ); $value = oct($number);
print("The decimal value of $number is $value\n"); }
elsif($choice == 9) { print "Enter a number: "; chomp($number
= ); $value = int($number); print("The integer value of
$number is $value\n"); } else { print("Invalid choice\n"); }
```

And finally, if you need to use Perl to generate random numbers, you should know about the rand() function. The rand() function takes a number as parameter, and generates a random number between 0 and that number. Here's an example:

```
#!/usr/bin/perl print rand(9);
```

And this could return

```
7.06539493566379
```

If you omit the parameter, you'll get a random number between 0 and 1. And here's a script that asks you for a numerical range, and then returns a random number within that range:

```
#!/usr/bin/perl # get the limits print "Enter the lower limit
of the range: "; $lower = ; chomp ($lower); print "Enter the
upper limit of the range: "; $upper = ; chomp ($upper); # keep
generating until number falls within range while ($random <
$lower) { $random = int(rand($upper)); } # then print print
$random;
```

And that's about all we have time for today. We'll be back with more in a couple of weeks − so keep coming back!