Perl 101 (Part 5) − Sub−Zero Code

By Vikram Vaswani and Harish Kamath

Table of Contents

A Little Knowledge

If you've been paying attention these last few weeks, you should know enough about Perl to begin writing your own programs in the language. As a matter of fact, you might even be entertaining thoughts of cutting down your weekly visits to this Web site and doing away with this tutorial altogether.

Well, a very wise man once said that a little knowledge was a dangerous thing...and so, as your Perl scripts become more and more complex, you're going to bump your head against the principles of software design, and begin looking for a more efficient way of structuring your Perl programs.

Enter this week's tutorial, which attempts to address that very problem by teaching you all you need to know about a programming construct called a "subroutine". And − since all work and no play is no way to live your life − we'll also be tossing in a few pop culture references that should help make the lesson an interesting [maybe even entertaining?] one.

Great Movies...

Ask a geek to define the term "subroutine", and he'll probably tell you that a subroutine is "a block of statements that can be grouped together as a named entity." Since this definition raises more questions than answers [the primary one being, what on earth are you doing hanging around with geeks in the first place], we'll simplify things by informing you that a subroutine is simply a set of program statements which perform a specific task, and which can be called, or executed, from anywhere in your Perl program.

There are two important reasons why subroutines are a "good thing". First, a subroutine allows you to separate your code into easily identifiable subsections, thereby making it easier to understand and debug. And second, a subroutine makes your program modular by allowing you to write a piece of code once and then re−use it multiple times within the same program.

Let's take a simple example, which demonstrates how to define a sub−routine and call it from different places within your Perl script:

```
#!/usr/bin/perl # define a subroutine sub greatest_movie {
print "Star Wars\n"; } # main program begins here print
"Question: which is the greatest movie of all time?\n"; # call
the subroutine  # ask another question print "Question: which
movie introduced the world to Luke Skywalker, Yoda and Darth
Vader?\n"; # call the subroutine
```

Now run it − you should see something like this:

```
Question: which is the greatest movie of all time? Star Wars
Question: which movie introduced the world to Luke Skywalker,
Yoda and Darth Vader? Star Wars
```

Let's take this line by line. The first thing we've done in our Perl program is define a new subroutine with the "sub" keyword; this keyword is followed by the name of the subroutine. All the program code attached to that subroutine is then placed within a pair of curly braces − this program code could contain loops, conditional statements, calls to other subroutines, or calls to other Perl functions. In the example above, our subroutine has been named "greatest_movie", and only contains a call to Perl's print() function.

Here's the typical format for a subroutine:

```
sub subroutine_name { statement 1... statement 2... . . .
statement n... }
```

...And Memorable Friends

Of course, defining a subroutine is only half of the puzzle − for it to be of any use at all, you need to invoke it. In Perl, this is accomplished by calling the subroutine by its name, as we've done in the last line of the example above. When invoking a subroutine in this manner, it's usually a good idea to precede the name with an ampersand [this is not essential, though, and the following would also work:

```
#!/usr/bin/perl # define a subroutine sub greatest_movie { print "Star Wars\n"; } # main program begins here print "Question: which is the greatest movie of all time?\n"; # call the subroutine greatest_movie; # ask another question print "Question: which movie introduced the world to Luke Skywalker, Yoda and Darth Vader?\n"; # call the subroutine greatest_movie;
```

However, it's good programming practice to precede the name of a Perl subroutine with an ampersand when invoking it − this helps to differentiate the Perl subroutine from array or scalar variables that may have the same name, and from pre−defined Perl functions. For example, consider this piece of code, in which we've defined a subroutine called which also happens to be the name of the in−built Perl print() function:

```
#!/usr/bin/perl # define a subroutine sub print { print "Ross\n"; } # main program begins here print "Question: which Friend once had a pet monkey?\n"; # call the subroutine print;
```

In this case, when you run the program, you'll get the following:

```
Question: which Friend once had a pet monkey?
```

Here, Perl assumes that the line containing the print statement is a call to the in−built Perl print() function, rather than the user−defined Perl subroutine. To avoid this kind of error, you should either avoid naming your subroutines after reserved words, or precede the subroutine call with an ampersand. In the example above, if you changed the last line from

```
print;
```

to

```

```

Perl would understand the subroutine call correctly, and display the desired output.

```
Question: which Friend once had a pet monkey? Ross
```

Popping The Question

Usually, when a subroutine is invoked in Perl, it generates a "return value". This return value is either the value of the last expression evaluated within the subroutine, or a value explicitly returned via the "return" statement. We'll examine both these a little further down − but first, here's a quick example of how a return value works.

```
#!/usr/bin/perl # define a subroutine sub change_temp { $celsius = 35; $fahrenheit = ($celsius * 1.8) + 32; } # assign return value to variable $result =  print "35 Celsius is $result Fahrenheit\n";
```

In this case, the value of the last expression evaluated within the subroutine serves as its return value − this value is then assigned to the variable $result when the subroutine is invoked from within the program.

Of course, it's also possible to explicitly specify a return value − use the "return" statement, as we've done in the next example:

```
#!/usr/bin/perl # define a subroutine sub do_you { if ($tall == 1 &$dark == 1 &$handsome == 1) { return "Yes!\n"; } else { return "Nope, afraid I don't feel the same way about you!\n"; } } $tall = 1; $dark = 1; $handsome = 1; # pop the question print "Will you marry me?\n"; # assign return value to variable $answer =  # print the answer print $answer;
```

Jumping Cows And Extra−Large Pumpkins

Return values from a subroutine can even be substituted for variables anywhere in a program. For example, you could modify the last two lines of the example above to read:

```
#!/usr/bin/perl # define a subroutine sub do_you { if ($tall == 1 &$dark == 1 &$handsome == 1) { return "Yes!\n"; } else { return "Nope, afraid I don't feel the same way about you!\n"; } } $tall = 1; $dark = 1; $handsome = 1; # pop the question print "Will you marry me?\n"; # assign return value to variable and print print(
```

And, of course, return values need not be scalar variables alone − a subroutine can just as easily return an array variable, as we've demonstrated in the following example:

```
#!/usr/bin/perl # define a subroutine sub split_me { split(" ", $string); } # define string $string = "The cow jumped over the moon and turned into a gigantic pumpkin"; # invoke function and assign result to array @words =  # loop for each element of array foreach $word (@words) { print "Word: $word\n"; $count++; } # print total print "The number of words in the given string is $count\n";
```

The output is

```
Word: The Word: cow Word: jumped Word: over Word: the Word: moon Word: and Word: turned Word: into Word: a Word: gigantic Word: pumpkin The number of words in the given string is 12
```

Turning Up The Heat

Now, if you've been paying attention, you've seen how subroutines can help you segregate blocks of code, and use the same piece of code over and over again, thereby eliminating unnecessary duplication. But this is just the tip of the iceberg...

The subroutines you've seen thus far are largely static, in that the variables they use are already defined. But it's also possible to pass variables to a subroutine from the main program − these variables are called "arguments", and they add a whole new level of power and flexibility to your code.

Consider the following simple example:

```
#!/usr/bin/perl # define a subroutine sub add_two_numbers { $sum = $_[0] + $_[1]; return $sum; } $total = 3,5); print "The sum of the numbers is $total\n";
```

A few words of explanation here:

You're already familiar with the special Perl array @ARGV, which contains parameters passed to the Perl program on the command line. Well, Perl also has a variable named @_, which contains arguments passed to a subroutine, and which is available to the subroutine when it is invoked. The value of each element of the array can be accessed using standard scalar notation − $_[0] for the first element, $_[1] for the second element, and so on.

In the example above, once the subroutine is invoked with the numbers 3 and 5, the numbers are transferred to the @_ variable, and are then accessed using standard scalar notation within the subroutine. Once the addition has been performed, the result is returned to the main program, and displayed on the screen via the print() statement.

Note the manner in which arguments are passed to the subroutine, enclosed within a pair of parentheses.

How about something a little more useful? Let's go back a couple of pages, and consider the subroutine we've defined:

```
#!/usr/bin/perl # define a subroutine sub change_temp { $celsius = 35; $fahrenheit = ($celsius * 1.8) + 32; } # assign return value to variable $result =  print "35 Celsius is $result Fahrenheit\n";
```

Now, suppose we alter this to accept the temperature in Celsius from the main program, and return the temperature in Fahrenheit.

```
#!/usr/bin/perl # define a subroutine sub change_temp { $fahrenheit = ($_[0] * 1.8) + 32; } print "Enter temperature in Celsius\n"; $temperature = ; chomp ($temperature); $result = perature); print "$temperature Celsius is $result Fahrenheit\n";
```

And here's what it would look like:

```
Enter temperature in Celsius 45 45 Celsius is 113 Fahrenheit
```

Take it one step further − how about allowing the user to specify the temperature to be converted on the command line itself?

```
#!/usr/bin/perl # define a subroutine sub change_temp { $fahrenheit = ($_[0] * 1.8) + 32; } # get the command−line parameters # and pass them to the subroutine # and assign the result $result = V); # print the result print "$ARGV[0] Celsius is $result Fahrenheit\n";
```

If you saved this program as "convert_temp.pl", and ran it like this

```
$ convert_temp.pl 35
```

you'd see

```
35 Celsius is 95 Fahrenheit
```

The above example also neatly demonstrates the relationship between @ARGV and @_ − the temperature entered on the command line first goes into the @ARGV variable, and is then passed to the subroutine via the @_ variable. Remember that the @_ variable is only available within the scope of a specific subroutine.

My() Hero!

Let's now talk a little bit about the variables used within a subroutine, and their relationship with variables in the main program. Unless you specify otherwise, the variables used within a subroutine are global − that is, the values assigned to them are available throughout the program, and changes made to them during subroutine execution are not restricted to the subroutine space alone.

For a clearer example of what this means, consider this simple example:

```
#!/usr/bin/perl # define a subroutine sub change_value { $hero = "Wolverine"; } # define a variable $hero = "The Incredible Hulk"; # before invoking subroutine print "Today's superhero is $hero\n"; print "Actually, I've changed my mind...";  # after invoking subroutine print "...gimme $hero instead.\n";
```

And here's what you'll see:

```
Today's superhero is The Incredible Hulk Actually, I've changed my mind.....gimme Wolverine instead.
```

Obviously, this is not always what you want − there are numerous situations where you'd prefer the variables within a subroutine to remain "private", and not disturb the variables within the main program. And this is precisely the reason for Perl's my() construct.

The my() construct allows you to define variables whose influence does not extend outside the scope of the subroutine within which they are enclosed. Take a look:

```
#!/usr/bin/perl # define a subroutine sub change_value { # this statement added my ($hero); $hero = "Wolverine"; } # define a variable $hero = "The Incredible Hulk"; # before invoking subroutine print "Today's superhero is $hero\n"; print "Actually, I've changed my mind...";  # after invoking subroutine print "...gimme $hero instead.\n";
```

And here's what you'll get:

```
Today's superhero is The Incredible Hulk Actually, I've changed my mind.....gimme The Incredible Hulk instead.
```

What happens here? Well, when you define a variable with the "my" keyword, Perl first checks to see if a variable already exists with the same name. If it does [as in the example above], its value is stored and a new variable is created for the duration of the subroutine. Once the subroutine has completed its task, this new variable is destroyed and the previous value of the variable is restored.

The my() operator can be used with both scalar and array variables. And − since Perl is all about efficiency − you can assign a value to the variable at the same that that you declare it, like this:

```
sub change_value { my ($hero) = "Wolverine"; }
```

The Age Gauge

So Perl gives you "public" variables and "private" variables − more than enough for most programmers. But you know what geeks are like...they're never satisfied. And so Perl also provides a useful middle ground − variables which are available between subroutines, but are hidden from the main program.

Why would you want to use something like this? Well, consider the following example, which demonstrates the concept:

```
#!/usr/bin/perl # define some subroutines sub display_value { print "During the subroutine...you are $age years old.\n"; } sub change_age { local ($age) = $age + $increment; ge); } # ask for age print "How old are you?\n"; $age = ; chomp ($age); # ask for increment print "How many years would you like to add?\n"; $increment = ; chomp ($increment); # demonstrate local variable print "Before invoking the subroutine...you are $age years old.\n"; print "After invoking the subroutine...you are $age years old.\n";
```

And here's what it looks like:

```
How old are you? 32 How many years would you like to add? 9 Before invoking the subroutine...you are 32 years old. During the subroutine...you are 41 years old. After invoking the subroutine...you are 32 years old.
```

When making calls between subroutines in this manner, it often becomes necessary to store the value of a variable across subroutines − and that's where local() comes in. In the example above, the variable $age is assigned an initial [global] value on the basis of user input. However, once the subroutine is invoked, this global value is stored and a new value is assigned to $age.

So far so good...we've already seen this with my(). But now, needs to call and pass it the value of the variable $age. By declaring $age to be a "local" variable, Perl makes it possible for the subroutine to access the new value of $age, and display it.

Once the subroutines finish and return control to the main program, the original value of $age is restored, and displayed. Thus, the example demonstrates how the local() keyword can be used to share variable values between subroutines, without affecting the global value of the variable.

And that's about it for this week. Next time, we'll be taking a close look at some of Perl's built−in string, math and pattern−recognition functions...so make sure you come back!