Perl 101 (Part 1) – The Basics

By Vikram Vaswani and Harish Kamath

Table of Contents

The Big Picture...

If you're a Web programmer, you're probably already well−versed with the intricacies of client−side scripting. But where there's a client, there must be a server − and so, this week, DevShed is kicking off a series of tutorials on server−side scripting. With power such as this, young Jedi, there is no limit to the evil you will be capable of...

First, though, let's start with the basics.

Server−side scripting is not new. It's been around for quite a while on the Web, and almost every major Web site uses some amount of server−side scripting. Amazon.com uses it to find the book you're looking for, Yahoo! uses it to store your personal preferences, and GeoCities uses it to generate page statistics.

Despite this, you're probably wondering why server−side scripting is such a big deal − after all, you've probably seen what a few simple JavaScripts can do. The reason for its popularity is very simple − JavaScript runs within a client application, usually the browser, and as such can only access resources, such as the current date and time, on the client machine. Since server−side scripts run on the Web server, they can be used to access server resources such as databases, system variables and other useful thingamajigs.

Just as there are different flavours of client−side scripting, there are different languages which can be used on the server as well. Here's a quick list of some of the more popular ones:

Active Server Pages [http://msdn.microsoft.com/workshop/server/asp/ASPover.asp] was introduced by Microsoft in its IIS Web server, ostensibly for the purpose of "web application programming". While ASP currently runs only on the Windows server platform, plans are afoot to port it to the UNIX platform as well.

Next up, ColdFusion, developed by Allaire [http://www.allaire.com]. ColdFusion syntax bears a remarkable resemblance to HTML, making it very easy for a budding web programmer to migrate to it. At the moment, it's available for both Windows NT and Linux. The only drawback: it ain't free, McGee!

Python [http://www.python.org] is an interpreted, object−oriented high−level scripting language for UNIX, often compared to Tcl, Perl or Java. It has modules, classes and interfaces to system calls, and is also extensible. It has been ported to Windows, DOS, OS/2 and the Macintosh, and has a devout following in the UNIX community.

And then there's the current flavour of the month, PHP [http://www.php.net]. Very easy to use, it's free, widely available for UNIX systems, and particularly strong in the areas of database access. The latest version is PHP4, and a final release is expected shortly.

And finally, Perl, one of the most popular languages around [and the language used throughout this tutorial − such is fame!]. Here's how its creator, Larry Wall, describes it: "PERL, an acronym for Practical Extraction and Report Language, is an interpreted language optimized for scanning arbitrary text files, extracting information from these files, and printing reports based on that information. It is also a good language for many system management tasks. The language is intended to be practical − easy to use, efficient, and complete − rather than beautiful − tiny, elegant, and minimal.

...And The Little Language That Could!

Perl is a very popular language for server−side scripting primarily because of its close relationship with the UNIX platform. Most Web servers run UNIX or one of its variants, and Perl is available on most or all of these systems. The language is so powerful that many routine administration tasks on such systems can be carried out in it, and its superior pattern−matching techniques come in very useful for scanning large amounts of data quickly.

Geeks will be happy to hear Perl is an interpreted language. Why is this good? Well, one advantage of an interpreted language is that it allows you to perform incremental, iterative development and testing without going through a create/modify−compile−test−debug cycle each time you change your code. This can speed the development cycle drastically. And programming in Perl is relatively easy [famous last words!], especially if you have experience in C or its clones. Perl can even access C libraries and take advantage of program code written for this language, and the language is renowned for the tremendous flexibility it allows programmers in accomplishing specific tasks.

And then of course, there's cost and availability − Perl is available free of charge on the Internet, for the UNIX, Windows and Macintosh platforms. Source code and pre−compiled binaries can be obtained from http://www.perl.com/, together with installation instructions. The examples in this series of tutorials will assume Perl 5.x on Linux, although you're free to use it on the platform of your choice.

If you're on a UNIX system, a quick way to check whether Perl is already present on the system is with the UNIX "which" command. Try typing this in your UNIX shell:

```
$ which perl
```

If Perl is available, the program should return the full path to the Perl binary, usually

```
/usr/bin/perl
```

or

```
/usr/local/bin/perl
```

Or if you're really lazy, you could just ask your system administrator.

On any other platform, try checking your PATH environment variable for a directory containing the Perl executable.

Your First Perl Program

And now that you've got Perl installed and configured, how about actually doing something with it? Use your favourite text editor to type the following lines of code:

```
#!/usr/bin/perl # Perl 101 print ("Groovy, baby!\n");
```

Save this file as "groovy.pl".

Next, you need to tell the system that the file is executable. On a UNIX system, this is accomplished by setting the "executable bit" with the "chmod" command:

```
$ chmod +x groovy.pl
```

And now run the script − in UNIX, try

```
$ ./groovy.pl
```

On a Windows system, you need to pass the name of the script to the Perl executable as a parameter, like this:

```
> perl groovy.pl
```

or

```
> groovy.pl
```

In both cases, the script should return Austin Powers' trademark line. Ain't Perl groovy, baby!

In case things don't work as they should, it usually means that the system was unable to locate the Perl binary. This is a good time to holler for the system administrator.

Your First Perl Program Dissected

Let's take a closer look at the script. The first line

```
#!/usr/bin/perl
```

is used to indicate the location of the Perl binary. This line must be included in each and every Perl script, and its omission is a common cause of heartache for novice programmers. Make it a habit to include it, and you'll live a healthier, happier life.

Next up, we have a comment.

```
# Perl 101
```

Comments in Perl are preceded by a hash [#] mark. If you're planning to make your code publicly available on the Internet, a comment is a great way to tell members of the opposite sex all about yourself − try including your phone number for optimum results.

And finally, the meat of the script:

```
print ("Groovy, baby!\n");
```

In Perl, a line of code like the one above is called a "statement". Every Perl program is a collection of statements, and a statement usually contains instructions to be carried out by the Perl interpreter.

In this particular statement, the print() function has been used to send a line of text to the screen. Like all programming languages, Perl comes with a set of built−in functions − the print() function is one you'll be seeing a lot of in the future. The text to be printed is included within double quotes, and the entire thing is then surrounded by parentheses. Note the n character, used to indicate a new line.

C programmers will be familiar with the print() function call above, as also with the fact that in C, it is mandatory to place function arguments within parentheses. Perl is more flexible than C in this regard − in many cases, parentheses can be excluded in function calls, as in this example:

```
#!/usr/bin/perl # Perl 101 print "Groovy, baby!\n";
```

Every Perl statement ends with a semi−colon. Don't ask why − just do it!

And here's an interesting bit of trivia − every Perl statement can be further sub−divided into smaller units called "tokens". If you take a look at the statement above, you'll see three distinct tokens − print, ("Groovy, baby!\n") and the semi−colon. The interesting thing about tokens is that they can be separated with white space and tabs − which, translated, means that you could also write the above line as

```
print ("Groovy, baby!\n") ;
```

or

```
print("Groovy, baby!\n") ;
```

and things would still work as advertised. Needless to say, this comes in very useful when dealing with long and complex scripts.

Two Plus Two

So now you know how to print text − how about a little math? Try this:

```
#!/usr/bin/perl

# Some addition

print (10+2);
```

And this should give you the answer of that particular mathematical operation.

You can even try subtraction, multiplication and division:

```
#!/usr/bin/perl

# Some more math

print (10−2);
print (10 * 2);
print (10/2);
```

Note how the various results are concatenated together in the absence of the newline character.

And you can even combine text and mathematical operations − this will be covered in greater detail over the next few weeks, but for the moment, you can play with this simple example:

```
#!/usr/bin/perl

# Synthesis

print ("Hello there! And what are you doing ",
1+1, "night, baby?\n");
```

To Err Is Human...To Debug, Divine!

Perl also comes with a very powerful debugger, which alerts you to mistakes in your code. As an example, try omitting the last double quote from the example above and run the script again. You should see something like this:

```
Can't find string terminator '"' anywhere before EOF at
./test.pl line 5.
```

As you can see, the error message tells you pretty much all you need to know to find and squash the bug. Of course, all Perl's error messages aren't quite that informative − some of them are liable to sound a lot like Greek spoken backwards, especially if you're a novice. And so it's a good idea to use the "−w" flag when running your program − this flag tells the debugger to print extra warnings when it encounters an error.

Take a look at the following example, which runs the Perl script above with the "−w" flag:

```
#!/usr/bin/perl −w # Synthesis print ("Hello there! And what
are you doing ", 1+1, "night, baby?n);
```

If you run this program, here's what you'll see:

```
syntax error at TEST line 5, near "print" Unquoted string
"doing" may clash with future reserved word at TEST line 5.
String found where operator expected at TEST line 5, near
"doing ", 1+1, "" (Do you need to predeclare doing?) Bare word
found where operator expected at TEST line 5, near "", 1+1,
"night" (Missing operator before night?) Unquoted string
"night" may clash with future reserved word at TEST line 5.
Unquoted string "baby" may clash with future reserved word at
TEST line 5. Unquoted string "n" may clash with future
reserved word at TEST line 5. Execution of TEST aborted due to
compilation errors.
```

Of course, there is sometimes such a thing as too much data...

What's Next?

Well, that's about it for this introductory lesson − thus far, you've learned a little bit about server−side scripting, and the basic components of a Perl program. The next lesson will cover Perl's variables and operators, and you'll also be taken on a quick tour of Perl's conditional expressions. See you then!