Perl is the Swiss Army chainsaw of scripting languages: powerful and adaptable. It was first developed by Larry Wall, a linguist working as a systems administrator for NASA in the late 1980s, as a way to make report processing easier. Since then, it has moved into a large number of roles: automating system administration, acting as glue between different computer systems; and, of course, being one of the most popular languages for CGI programming on the Web.

Why did Perl become so popular when the Web came along? Two reasons: First, most of what is being done on the Web happens with text, and is best done with a language that's designed for text processing. More importantly, Perl was appreciably better than the alternatives at the time when people needed something to use. C is complex and can produce security problems (especially with untrusted data), Tcl can be awkward and Python didn't really have a foothold.

It also didn't hurt that Perl is a friendly language. It plays well with your personal programming style. The Perl slogan is "There's more than one way to do it," and that lends itself well to large and small problems alike.

In this first part of our series, you'll learn a few basics about Perl and see a small sample program.

A Word About Operating Systems

In this series, I'm going to assume that you're using a Unix system and that your Perl interpreter is located at /usr/local/bin/perl. It's OK if you're running Windows; most Perl code is platform-independent.

Your First Perl Program

Take the following text and put it into a file called first.pl:

```bash
#!/usr/local/bin/perl
print "Hi there!\n";
```

(Traditionally, first programs are supposed to say Hello world!, but I'm an iconoclast.)

Now, run it with your Perl interpreter. From a command line, go to the directory with this file and type perl first.pl. You should see:

```
Hi there!
```

The \n indicates the `"newline" character; without it, Perl doesn't skip to a new line of text on its own.

Functions and Statements

Perl has a rich library of functions. They're the verbs of Perl, the commands that the interpreter runs. You can see a list of all the built-in functions on the
perlfunc main page. Almost all functions can be given a list of parameters, which are separated by commas.

The print function is one of the most frequently used parts of Perl. You use it to display things on the screen or to send information to a file (which we'll discuss in the next article). It takes a list of things to output as its parameters.

    print "This is a single statement."
    print "Look, ", "a ", "list!"

A Perl program consists of statements, each of which ends with a semicolon. Statements don't need to be on separate lines; there may be multiple statements on one line or a single statement can be split across multiple lines.

    print "This is "; print "two statements.\n"; print "But this ", "is only one statement.\n"

Numbers, Strings and Quotes

There are two basic data types in Perl: numbers and strings.

Numbers are easy; we've all dealt with them. The only thing you need to know is that you never insert commas or spaces into numbers in Perl. always write 10000, not 10,000 or 10 000.

Strings are a bit more complex. A string is a collection of characters in either single or double quotes:

    'This is a test.'
    "Hi there!"

The difference between single quotes and double quotes is that single quotes mean that their contents should be taken literally, while double quotes mean that their contents should be interpreted. For example, the character sequence \n is a newline character when it appears in a string with double quotes, but is literally the two characters, backslash and n, when it appears in single quotes.

    print "This string\nshows up on two lines."
    print 'This string \n shows up on only one.'

(Two other useful backslash sequences are \t to insert a tab character, and \ \ to insert a backslash into a double-quoted string.)

Variables

If functions are Perl's verbs, then variables are its nouns. Perl has three types of variables: scalars, arrays and hashes. Think of them as "things," "lists," and "dictionaries." In Perl, all variable names are a punctuation character, a letter or underscore, and one or more alphanumeric characters or underscores.

Scalars are single things. This might be a number or a string. The name of a scalar begins with a dollar sign, such as $i or $abacus. You assign a value to a scalar by telling Perl what it equals, like so:

    $i = 5;
    $pie_flavor = 'apple';
    $constitution1776 = "We the People, etc."

You don't need to specify whether a scalar is a number or a string. It doesn't matter, because when Perl needs to treat a scalar as a string, it does; when it needs to treat it as a number, it does. The conversion happens automatically. (This is different from many other languages, where strings and numbers are two separate data types.)

If you use a double-quoted string, Perl will insert the value of any scalar variables you name in the string. This is often used to fill in strings on the fly:

    $apple_count = 5;
    $count_report = "There are $apple_count apples."
    print "$count_report\n";

The final output from this code is The report is: There are 5 apples..

Numbers in Perl can be manipulated with the usual mathematical operations: addition, multiplication, division and subtraction. (Multiplication and division are indicated in Perl with the * and / symbols, by the way.)

    $a = 5;
    $b = $a + 10;       # $b is now equal to 15.
    $c = $b * 10;       # $c is now equal to 150.
    $a = $a - 1;        # $a is now 4, and algebra teachers are cringing.

You can also use special operators like ++, --, +=, -=, /= and *=. These manipulate a scalar's value without needing two elements in an equation. Some people like them, some don't. I like the fact that they can make code clearer.

    $a = 5;
    $a++;        # $a is now 6; we added 1 to it.
    $a = 10;      # Now it's 16; we added 10.
    $a /= 2;      # And divided it by 2, so it's 8.

Strings in Perl don't have quite as much flexibility. About the only basic operator that you can use on strings is concatenation, which is a $10 way of saying "put together." The concatenation operator is the period. Concatenation and addition are two different things:

    $a = "a";        # Note the quotes. $a is a string.
    $b = $a + "1";   # "1" is a string too.
    $c = $a . "1";   # But $b and $c have different values!
Perl has several different functions that are useful for flow control, the most basic of which is for. When you use the for function, you specify a variable that will be used for the loop index, and a list of values to loop over. Inside a pair of curly brackets, you put any code you want to run during the loop:

```perl
for $i (1, 2, 3, 4, 5) {
    print "$i
";
}
```

This loop prints the numbers 1 through 5, each on a separate line.

A handy shortcut for defining loops is using .. to specify a range of numbers. You can write (1, 2, 3, 4, 5) as (1 .. 5). You can also use arrays and scalars in your loop list. Try this code and see what happens:

```perl
@one_to_ten = (1 .. 10);
```
$top_limit = 25;
for $i (@one_to_ten, 15, 20 .. $top_limit) {
    print "$i\n";
}

The items in your loop list don't have to be numbers; you can use strings just as easily. If the hash %month has contains names of months and the number of days in each month, you can use the keys function to step through them.

for $i (keys %month) {
    print "$i has $month{$i} days.\n";
}

for $marx ('Groucho', 'Harpo', 'Zeppo', 'Karl') {
    print "$marx is my favorite Marx brother.\n";
}

The Miracle of Compound Interest

You now know enough about Perl - variables, print, and for() - to write a small, useful program. Everyone loves money, so the first sample program is a compound-interest calculator. It will print a (somewhat) nicely formatted table showing the value of an investment over a number of years. (You can see the program at compound_interest.pl)

The single most complex line in the program is this one:

$interest = int (($apr / 100) * $nest_egg * 100) / 100;

$apr / 100 is the interest rate, and ($apr / 100) * $nest_egg is the amount of interest earned in one year. This line uses the int() function, which returns the integer value of a scalar (its value after any fractional part has been stripped off). We use int() here because when you multiply, for example, 10925 by 9.25%, the result is 1010.5625, which we must round off to 1010.56. To do this, we multiply by 100, yielding 101056.25, use int() to throw away the leftover fraction, yielding 101056, and then divide by 100 again, so that the final result is 1010.56. Try stepping through this statement yourself to see just how we end up with the correct result, rounded to cents.

Play Around!

At this point you have some basic knowledge of Perl syntax and a few simple toys to play with - print, for(), keys(), and int(). Try writing some simple programs with them. Here are two suggestions, one simple and the other a little more complex:

- A word frequency counter. How often does each word show up in an array of words? Print out a report. (Hint: Use a hash to count the number of appearances of each word.)

- Given a month and the day of the week that's the first of that month, print a calendar for the month. (Remember, you need \n to go to a new line.)