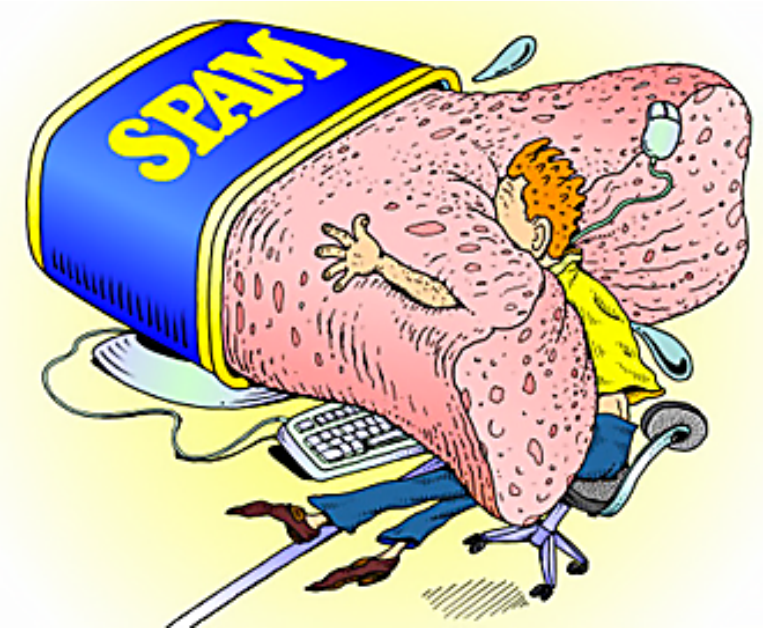


Writing A Spam Filter in Perl

By Chad Jordan – February 21st, 2009

In this guide, you will learn:

1) The syntax and practical purposes of using the Perl programming language
2) How to identify the elements of spam for bypassing unwanted emails
3) Calculating probability using scoring filters, and reducing false positives
4) Implementing commands, and function calls for client-side, and server-side purposes
5) Creating and merging hash tables, sorting data in arrays, scalars, CGI architecture, marking email addresses, and splitting whitespace using regular expressions

Introduction

Spam has been around for about as long as email has existed since the late 1970s. Over time there have been many attempts at keeping spam out of people's inboxes, and most of these are successful. The truth is phishing scams, fee frauds, and other malicious links have still managed to find a way around the filters when scammers have beat the system. Creating spam filters is not an easy endeavor and requires the developer to be mindful of all of the sly actions that a scammer would perform. This level of detail can be difficult to match when implementing counteractive measures against it. In this guide, I will be covering the methods that I took when creating my own spam filter using the Perl programming language, the practicalities of using Perl for a program like this, and essentially an introductive overview of what can be expected when writing a spam filter in the Perl language. While writing the spam filter portions of this guide, my code examples will be from a simple code editor that I used in Linux called Vim. A little more overview of Perl, it was developed by Larry Wall, Perl is a high-level, interpreted, C-based, dynamic programming language with Unix shell scripting features designed for text manipulation. It's used in many system administration, networking, report-processing, user interface applications, and mission-critical projects in the public and private sectors. Though Perl is not officially an acronym some have described it as **Practical Extraction and Report Language**. While my spam filter program will not have much code, Perl is nothing like my previous guide on coding a basic website with HTML and CSS. Perl is an actual programming language for the web with a bigger learning curve than other web programming technologies. This means that while this guide is not designed as a beginner-level programming guide, it could still be very useful as a guide for current web developers interested in using Perl.

The Practicalities of Perl

Perl is cross-platform supporting both procedural and object-oriented programming. Perl's DBI *(Database Integration)* interface supports third-party databases including Oracle, Sybase, Postgres, and MySQL. Perl also works well with markup languages such as HTML, XML, SGML, and more. Among other developers online, Perl has been widely known as "the duct-tape of the Internet and can handle encrypted Web data, including e-commerce transactions. Since Perl is an interpreted language, it means that your code can be run as-is, without a compilation stage that creates a non-portable executable program. Traditional compilers convert programs into machine language. When you run a Perl program, it's first compiled into a byte code, which is then converted (as the program runs) into machine instructions. It is not quite the same as shells, or Tcl, which are strictly interpreted without an intermediate representation. Perl is a free-form language which means you can format and indent it however you like.

The Syntax of Perl

You can use the Perl interpreter with -e option at the command line, which lets you execute Perl statements from the command line. Here is a demonstration at $ prompt.

Interactive Mode Programming:

A Hello World! Program

```
$perl -e 'print "Hello World!\n"'
```

Output: Hello World!

Script Mode Programming:

Create a text file 'hello.pl' with some code

```
#!/usr/bin/perl

#This will print "Hello World!"
print "Hello World!";
```

Here, **/usr/bin/perl** is actual the Perl interpreter binary. Before you execute your script, be sure to change the mode of the script file and give execution privilege, generally a setting of 0755 works perfectly and finally, you execute the above script as follows at the command line:

```
$chmod 0755 hello.pl
$./hello.pl
```

and then the following result will execute:
Hello World!

Perl Comments & Whitespace

Comments in any programming language are friends of developers. Comments can be used to make program user friendly and they are simply skipped by the interpreter without impacting the code functionality. For example, in the above program, where I typed, #This will print "Hello World!" with a hash **#** symbol at the beginning is a comment. Example:

 # This is a comment in Perl

Whitespace serves mostly to separate tokens, unlike languages like Python where it is an important part of the syntax or Fortran where it is immaterial. Block comments in Perl are enclosed within the = and =cut symbols. Everything written after the = symbol is considered part of the comment until =cut is encountered. There should be no whitespace following = in the multi-line comment. Example:

```
=This is
a
block comment
=cut
```

```
=This program will
display 'Hello World!' on the
screen
=cut

$string = "Hello World!";  # A simple assignment statement
print $string;
```

Output: Hello World!

Perl Scalar Variables

A scalar is a single unit of data. That data might be an integer number, floating point, a character, a string, a paragraph, or an entire web page.

```
#!/usr/bin/perl

$age = 26;                  # An integer assignment
$name = "Chad Jordan";      # A string
$salary = 1619.83;          # A floating point

print "Age = $age\n";
print "Name = $name\n";
print "Salary = $salary\n";
```

Output:
Age = 26
Name = Chad Jordan
Salary = 1619.83

Perl Numeric Scalars

A scalar is most often either a number or a string. This example provides how you might use various types of numeric scalars.

```
#!/usr/bin/perl

$integer = 100;
$negative = -200;
$floating = 302.640;
$bigfloat = -5.2E-18;

# 377 octal, same as 255 decimal
$octal = 0377;

# FF hex, also 255 decimal
$hexa = 0xff;

print "integer = $integer\n";
print "negative = $negative\n";
print "floating = $floating\n";
print "bigfloat = $bigfloat\n";
print "octal = $octal\n";
print "hexa = $hexa\n";
```

Output:
integer = 100
negative = -200
floating = 302.640
bigfloat = -5.2E-18
octal = 255
hexa = 255

Scalar Operations

The following are string and numeric operations.

```
#!/usr/bin/perl

$str = "hello" . "world";      # String concatenation
$num = 6 + 20;                 # Adds two numbers
$mul = 4 * 8;                  # Multiplies two numbers
$mix = $str . $num;            # Concatenates string and number

print "str = $str\n";
print "num = $num\n";
print "mul = $mul\n";
print "mix = $mix\n";
```

Output:

```
str = helloworld
num = 26
mul = 32
mix = helloworld26
```

Special Literals

The special literals __FILE__, __LINE__, and __PACKAGE__ represent the current filename, line number, and package name at that point in your program. __SUB__ gives a reference to the current subroutine. They may be used only as separate tokens; they will not be interpolated into strings. If there is no current package (due to an empty package; directive), __PACKAGE__ is the undefined value.

```
#!/usr/bin/perl

print "File name = " . __FILE__ . "\n";
print "Line Number = " . __LINE__ . "\n";
print "Package = " . __PACKAGE__ . "\n";

# they cannot be interpolated
print "__FILE__ __LINE__ __PACKAGE__\n";
```

Output:

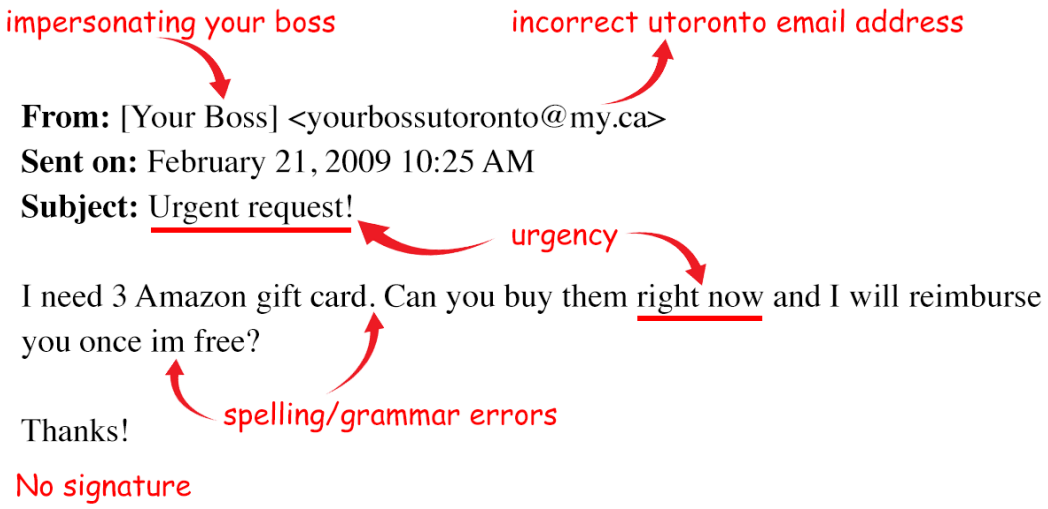
```
File name hello.pl
Line Number 4
Package main
__FILE__ __LINE__ __PACKAGE__
```

How to Identify Spam

Identifying spam is usually pretty obvious, but sometimes scammers still figure out a way to get into your inbox. When it comes to identifying spam, we know to consider the following:

1) The email is from a legitimate source, and not a public domain like 'gmail.com'
2) Emails Requesting Login Credentials, Payment Information or Sensitive Data
3) The message creates a sense of urgency or requests immediate action
4) Inconsistencies in domain names and email addresses
5) The domain name of the email address is misspelled
6) The email itself has misspelled words or bad grammar
7) Suspicious attachments or links in the email

Some common elements of phishing emails:



Calculating Probability in Perl

The problem with spam filters is that in a way they will always be in a state of vulnerability because spammers are always targeting new ways to bypass the filter. A solution to this is comparing the spam messages against legitimate messages and view the incoming message. Words in your incoming message are looked at individually and given a score based on whether it is a spam word or a good word. Total up the score for the incoming message and you have a very good filter. Here is an example using standard deviation for scoring filters in Perl.

```
 1 sub min {
 2     my $min = shift;
 3     for (@_) {
 4         $min = $_ if $_ < $min;
 5     }
 6     return $min;
 7 }
 8
 9 sub max {
10     my $max = shift;
11     for (@_) {
12         $max = $_ if $_ > $max;
13     }
14     return $max;
15 }
16
17 my %good;
18 my %bad;
19 my $ngood;
20 my $nbad;
```

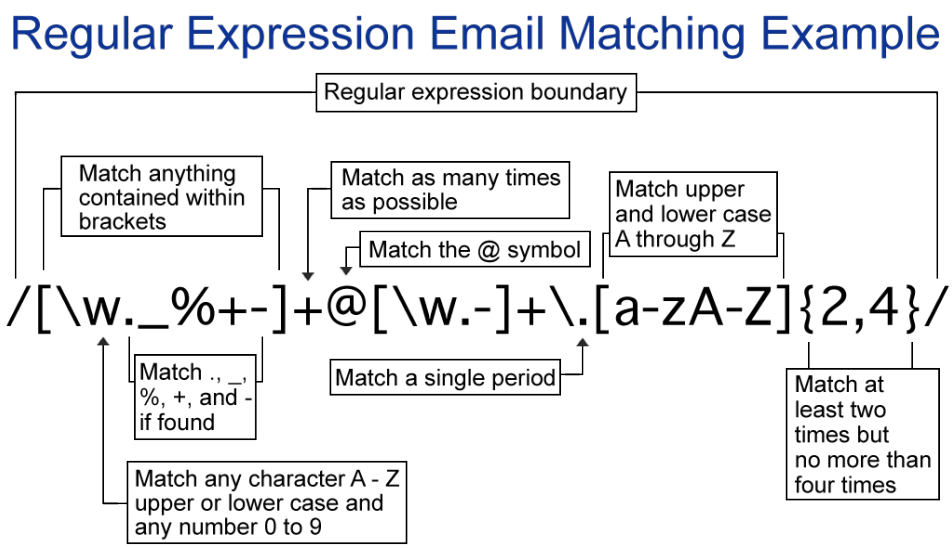
Starting on **line 1** this function returns the smallest argument given, and the function beginning on **line 9** returns the largest argument given.

We have to keep track of the number of occurrences for each word in a normal email, the number of occurrences for each word in a spam message, and then separate the number of good messages and the number of bad messages. I set up the following declarations from **lines 17 – 20** to use in my functions.

Beginning on **line 22** this returns the score for the words given, then on **line 23** the words in the email, followed on **line 24** by the probabilities for each word not defined yet. After this, I iterate through the good and bad words in the messages, calculate the minimum from the maximum, and return the values for **$prod**.

```
21
22 sub score {
23     my @words = @_;
24     my @probs;
25
26     for my $word (@words) {
27         my $g = 2 * ($good{$word} || 0);
28         my $b = $bad{$word} || 0;
29         unless ($g + $b < 5) {
30             push @probs,
31                 max( .01,
32                     min( .99, 1 / min($b / $nbad) ),
33                     min( 1, $g / $ngood ) + min( 1, $b / $nbad) );
34         }
35     }
36
37     my $prod = 1;
38     $prod *= $_  for (@probs);
39     my $prod2 = 1;
40     $prod2 *= $_ for (map {1 - $_ } @probs);
41     return $prod / ($prod + $prod2);
42 }
```

After splitting up the message into separate words, you call 'score' with the words to retrieve the probability that the message is spam. This is a good start, but there's a lot more that can be done for better accuracy than what I've provided in this example. One method is to throw more filters at our program, but an even more efficient way is to reduce what is known as **false positives** using **regular expressions**. To begin, let's first go over what regular expressions are, and how you implement them using Perl. Short for *regular expression*, a **regex** is a string of text that allows you to create patterns that help match, locate, and manage sequences of characters in the text. We can use this method when matching email addresses.



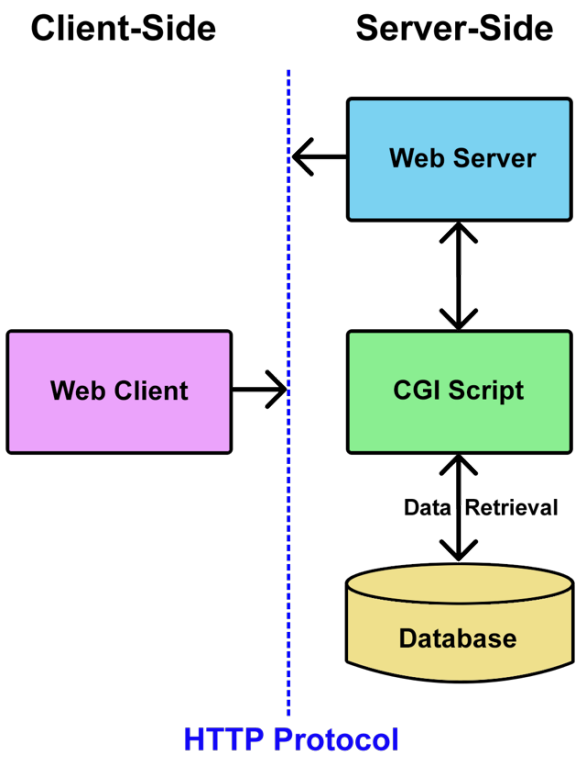
The power of regular expressions comes from its use of metacharacters, which are special characters (or sequences of characters) used to represent something else. For instance, in a regular expression, the metacharacter ^ means **"not"**. So, while "a" means "match lowercase a", **"^a"** means **"do not match lowercase a"**. Regex does have slightly different *versions* if you will, with different sets of rules for different languages. E.g Java, Perl, Python, etc. With this abstract method, we can perform split, trim, search, and replace operations for characters. With this in mind, it's time to consider how to reduce false positives. We know that we need to prevent false positives from inspecting the results of the match to ensure they're relevant to your search. One common way to do this is to relax your regular expression. For example, replace a single space with **/\s*/** to allow for any amount of whitespace. The second way is to make another pass through the document with a separate regular expression or processing technique, to catch the data you missed the first time around. For example, extract into an array all the things that look like news headlines, then remove the first element from the array if you know it's always going to be an advertisement instead of an actual headline. I will provide the code for my spam filter shortly with similar examples.

The Perl CGI Architecture

A CGI *(Common Gateway Interface)*, is a set of standards that defines how information is exchanged between the web server and a custom script. In Perl, CGI is a protocol for executing dynamic scripts via web requests. It is a set of rules and standards that define how the information is exchanged between the web server and custom scripts.

The CGI specs are currently maintained by the NCSA *(National Center for Supercomputing Applications)* and NCSA defines CGI is as follows: The **Common Gateway Interface**, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.

The diagram to the right, demonstrates a simple layout of how the CGI architecture works. Your browser contacts web server using HTTP protocol and demands for the URL, i.e., web page filename.



Web Server will check the URL and will look for the filename requested. If web server finds that file then it sends the file back to the browser without any further execution otherwise sends an error message indicating that you have requested a wrong file. The Web Client takes the response from the web server and displays either the received file content or an error message in case the file is not found.

Writing the Spam Filter

For the process of implementing my spam filter, I begin on **line 4** by creating hash tables for storing the counts of good, bad, and odd emails.

```
1  #!/usr/bin/perl
2  use strict;
3
4  my %good_msg=();
5  my %bad_msg=();
6  my %odds=();
7  my $num_bad=0;
8  my $num_good=0;
9  my $g=0;
10 my $b=0;
11 my $freq_g=0;
12 my $freq_b=0;
13 my %both_msg=();
14
15 my $spamfile;
16 open($spamfile, $ARGV[0]);
17 my @spamlines=<$spamfile>;
18 close $spamfile;
19 foreach my $line(@spamlines){
20         if ($line=~/^From:/){
21                 $num_bad+=1;
22         }
23         my @words=split(/[^\w\-'\$]/,$line);
24         foreach my $word(@words){
25                 $bad_msg {$word}+=1;
26         }
27 }
```

Then, beginning on **line 16** I check the spam file to see if it has a reference to the file name that I'm getting the spam from, then dumping the file into the array, *spamlines*.

Next, on **line 23** I use a simple regular expression to split the string where anything is not a word, apostrophe, hyphen, or dollar sign and create temporary storage for one line of email text.

Line 25 searches through the hash table for bad words and adds 1 until there are no bad words.

```
29 my $goodfile;
30 open($goodfile, $ARGV[1]);
31 my @goodlines=<$goodfile>;
32 close $goodfile;
33 foreach my $line(@goodlines){
34         if ($line=~/^From:/){
35                 $num_good+=1;
36         }
37         my @words=split(/[^\w\-'\$]/,$line);
38         foreach my $word(@words){
39                 $good_msg {$word}+=1;
40         }
41 }
42
43 %both_msg =(%good_msg, %bad_msg);
44 foreach my $word(keys %both_msg){
45         $g=$good_msg{$word}*2;
46         $b=$bad_msg{$word};
47         if ($g+$b>=5){
48                 $freq_g=$g/$num_good;
49                 if($freq_g > 1) {
50                         $freq_g=1;
51                 }
52                 $freq_b=$b/$num_bad;
53                 if($freq_b > 1) {
54                         $freq_b=1;
55                 }
56                 $odds{$word}=$freq_b/($freq_g + $freq_b);
57                 if($odds{$word} > .99){
58                         $odds{$word} = .99;
59                 }
60                 elsif($odds{$word} < .01){
61                         $odds{$word} = .01;
62                 }
63         }
64 }
```

Note:
Between lines **32** and **33**, if you want to check the status of *spamlines* you can run the command, *print @spamlines;* and this will provide you with any spam found within *$goodfile*. Between lines **38** and **39** we can *print $word, "\n";*

Starting on **line 43**, I merge the two hash tables together, and then allowing to loop over all words in all emails.

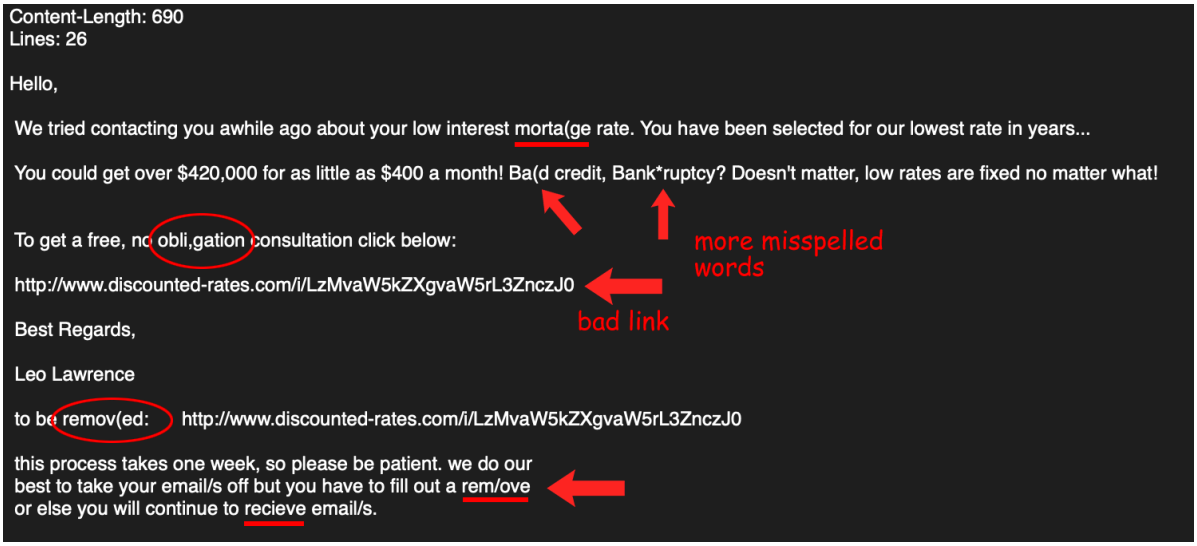
```
65
66 my $unknown_file;
67 my %unknown_msg=();
68 open($unknown_file, $ARGV[2]);
69 my @unknown_lines=<$unknown_file>;
70 close $unknown_file;
71 foreach my $line(@unknown_lines){
72         my @words=split(/[^\w\-'\$]/,$line);
73         foreach my $word(@words){
74                 if(!exists $odds{$word}){
75                         $unknown_msg{$word}=.4;
76                 }
77                 else{
78                         $unknown_msg{$word}=$odds{$word};
79                 }
80         }
81 }
```

This section is really just like the preceding functions. We should really have a contingency check in place for unknown origins of data within the message. This function allows us to check for probable bad data.

Beginning on **line 83**, I'm sorting the words from unknown messages by their distance from .05 giving an array from least interesting to most interesting. On **line 90** I'm multiplying together one minus the probability of all the words. Then, the final function starting on **line 93** calculates the probability of all found data, and then prints it out to the screen.

```
82
83 my @sort_words=sort{ abs ($unknown_msg {$main::a}-.5) <=> abs ($unknown_msg {$main::b}-.5) } keys %unknown_msg;
84 my @top_words=@sort_words[-1,-2,-3,-4,-5,-6,-7,-8,-9,-10,-11,-12,-13,-14,-15];
85
86 my $prob_times=1;
87 my $one_minus_probs=1;
88 foreach my $word(@top_words){
89         $prob_times*=$unknown_msg{$word};
90         $one_minus_probs*=1-$unknown_msg{$word}
91 }
92
93 my $answer=$prob_times/($prob_times - $one_minus_probs);
94 printf("The Probability is: %.3f\n", $answer);
95 if($answer > .9){
96         print "SPAM!\n";
97 }
98 else{
99         print "Not Spam\n";
100 }
```

Here is a partial result of my spam filter catching a bad email provided by the CSE department servers.



Conclusion

This concludes my spam filter program. While there is a lot more involved with how Perl interacts with databases, CGI programming, regular expressions, and other algorithms, this guide is intended to provide a fairly lightweight, but effective spam filter program using the Perl programming language. This Perl program is actually my third assignment for my COS 264 web programming course at Taylor University. My hope is that this guide was helpful in learning more about Perl, and especially about the process of building your own spam filter. All diagrams and code in this guide were created, written, and provided by Chad Jordan. For any possible inquiries such as general questions regarding this guide or other professional inquiries please feel free to email me at cjordan@wondercreationstudios.com

---

Resources Used:

- Schwartz, Randal L., Phoenix, Tom, Foy, Brian d. *Learning Perl – Fifth Edition* – 2008
- LearnPerl.org
- PerlTutorial.org
- W3schools.com