# Getting Started With Perl

**Jonathan Worthington**
Scarborough Linux User Group

## What is Perl?

- A programming language
- Created by Larry Wall
- Perl 1 released in 1987
- Current version is Perl 5.8.x
- Perl 6 under construction, looks very exciting.
- Probably shipped as part of every Linux distribution.

## Where is Perl used?

- Server side web programming (CGI scripts, mod_perl Apache module)
- System administration
- Bioinformatics
- Natural language processing
- Lots of situations where you want to glue together things that don’t naturally fit.

## What does Perl mean?

**P**ractical

**E**xtraction (and)

**R**eporting

**L**anguage

## What does Perl mean?

**P**ractical

- Focus on getting the job done without getting in the programmer’s way
- Language features map well to real world tasks.

**E**xtraction (and)

**R**eporting

**L**anguage

## What does Perl mean?

**P**ractical

**E**xtraction (and)

- Makes it easy to get hold of the data we want to do stuff with.
- Perl’s regexes are extremely powerful and fast – engine widely used.

**R**eporting

**L**anguage

## What does Perl mean?

**P**ractical

**E**xtraction (and)

**R**eporting

- Easy to produce output from data in the format we need to.
- High performance file I/O, string iterpolation, more later…

**L**anguage

## What does Perl mean?

**P**ractical

**E**xtraction (and)

**R**eporting

**L**anguage

- Designed by a linguist, borrowing from natural languages so Perl feels natural to write.
- *Possible* to write hard to read code.

## What does a Perl program look like?

“Shebang” gives path to Perl interpreter

```
#!/usr/bin/perl
# This is a comment
print “Hello world!\n”;
```

Semicolon ends each statement

“print” will send output to stdout by default.

Strings go in double quotes, \n means “new line”.

## Scalars

- A scalar is a container that can hold one thing.

42 Jonathan 3.14159265

- When talking about a container holding a single item of data, we use “$”.
- The “S” in “$” is to remind you of the “S” in “scalar”.

## Assignment

- Assignment is about putting an item in a container with a name.
- Written with the “=“ sign.

```
$answer = 42;
$name = “Jonathan”;
$pi = 3.14159265;
```

- Do not think of this like equality in maths – you’ll get horribly confused!

## Using scalars in output

- Our previous “Hello world!” program could be re-written as follows.

**Set the scalar $message to contain “Hello world!\n”**

```
#!/usr/bin/perl
$message = “Hello world!\n”;
print $message;
```

**Placing a scalar where a string or number could be written will cause the value held in the scalar to be used in that place.**

## Interpolation

- Putting a scalar in the middle of some output should not be this painful:

```
$name = “Jonathan”;
print “My name is “;
print $name;
print “\n”;
```

- So strings inside doubles quotes will “interpolate” scalars, so we can do this:

```
$name = “Jonathan”;
print “My name is $name\n”;
```

## Interpolation

- Interpolation can sometimes get in the way.
- Single quotes will not interpolate.
- Other characters that become special with interpolation are “@”, “%” and “\”.
- Be careful to use single quotes for email addresses!

```
$email = ‘jonathan@jwcs.net’;
```

## Arithmetic

- Works just as you would expect.

```
$a = 10;
$b = 5;

$c = $a + $b;
$d = $a - $b;
$e = $a * $b;
$f = $a / $b;
$a++;
$b--;
```

**Addition; $c will hold 15**

**Multiplication; $e will hold 50**

**Short for $a = $a + 1;**

**Subtraction; $d will hold 5**

**Division; $f will hold 2**

**Short for $b = $b – 1;**

## Getting input

- We can read data in a line at a time using the diamond operator.
- For example, <STDIN> will read lines from standard input.
- Using it in an assignment to a scalar will read a line and store the contents of the line in the scalar.

```
$line = <STDIN>;
```

## The greeting program

We didn’t put a \n on this line so the cursor for input is left after this prompt. Looks nicer somehow.

```
#!/usr/bin/perl
print "Enter your name: ";
$name = <STDIN>;
chomp $name;
print "Hola $name!\n";
```

A line ends with a new line character and the diamond operator doesn’t strip it. chomp removes a new line character from the end if there is one.

We read the users name into $name.

## `while` loops

- Looping is about doing the same thing multiple times.
- Often also called iteration.
- A while loop basically says “while some condition C is true, do X.
- In Perl, a while loop is written like this:

```
while (condition) {
    # do stuff
}
```

## Looping over all lines of input

- Often we want to do something for each line piped to standard in.
- We can do this with the following loop:

```
while ($line = <STDIN>) {
    # do per-line stuff
}
```

- This works because when there is no more to read, $line will be undefined.
- An undefined value is always false.

## The line counting program (“wc -l”)

**$lines will store the number of lines read so far.**

```
#!/usr/bin/perl
$lines = 0;
while (<STDIN>) {
    $lines++;
}
print “$lines lines\n”;
```

**As we do not care about the contents of the lines, we just want to count them, we don’t assign what is read to a scalar. (Assigned to $_ - more later).**

**Increment the counter for each line.**

## Writing less code

- We can do the same with less code.
- Need a careful trade-off between being brief and being clear.

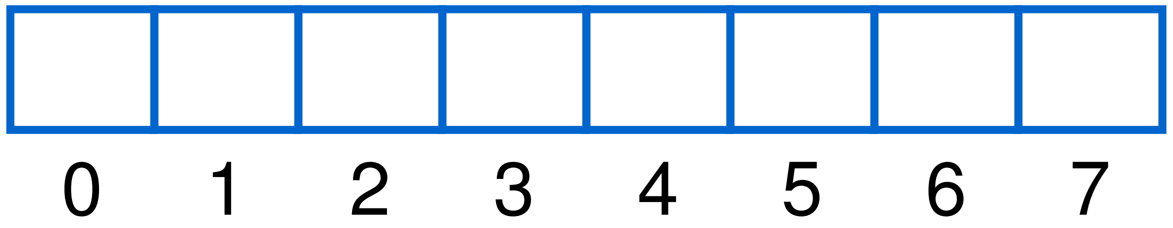
```
#!/usr/bin/perl
$lines++ while <>;
print “$lines lines\n”;
```

**It is possible, when you are doing a single thing per line, to use the post-fix form of while. Neater here – don’t overuse it.**

**When the diamond operator is empty, it uses STDIN by default.**

## Arrays

- The second type of container in Perl.
- Contains many items indexed by the integers starting from 0.
- Think of slots numbered 0, 1, 2, 3…



- Written with the “@”, to remind you of the “a” in “array”.

## Assigning to arrays

- Can assign lots of values to an array at once.

```
@names = (‘Noah’, ‘Abraham’, ‘Sarah’);
```

- The same could be achieved by assigning to each element of the array.

```
$names[0] = ‘Noah’;
$names[1] = ‘Abraham’;
$names[2] = ‘Sarah’;
```

- Note use of $ rather than @ here, as we are talking about a single value.

## Getting values from an array

- We could reference each element by its index, but it’s a lot of work.

```
print “$names[0]\n$names[1]\n$names[2]\n”;
```

- If we do this, there’s no advantage over using 3 scalars.
- What makes arrays powerful is that we can iterate (loop) over the values stored in them – even without knowing how many there are.

## `foreach` loops

- Allow us to take each of the values in an array (in order) and do something with that value.
- A foreach loop looks like this:

```
foreach $element (@array) {
    # Do something
}
```

- The code in the braces is executed once for each element in @array, the current value being stored in $element.

## A simple array example

**Assign some numbers to the array @numbers.**

```
#!/usr/bin/perl
@numbers = (42,15,29,14);
$total = 0;
foreach $n (@numbers) {
    $total += $n;
}
print “Total: $total\n”;
```

**Each element of @numbers is placed in $n**

**We add each element to the total. Note that “$total += $n;” is short for “$total = $total + $n;”.**

## `split`

- Often need to read input separated by a certain character or characters.
- For example, tab delimited files or output from “ps” where we have data in columns separated by spaces.
- The `split` function allows a line of input to be split up into an array each time a certain character or sequence of characters is encountered.

## `split` example

- We'll take the output produced by running "ps –ef" and total the times that processes have been running for.
- The output of "ps –ef" looks like this:

```
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 Jun20 ?        00:00:52 init
root           2     1  0 Jun20 ?        00:01:06 [keventd]
root           3     1  0 Jun20 ?        00:00:02 [kapmd]
...
```

- Need to ignore line 1 and get time from column 7.

## `split` example, step 1

- Start off with a simple skeleton script that gives us a place to store the total and loops through the lines of input.

```
#!/usr/bin/perl
$total = 0; # In seconds

while ($line = <STDIN>) {
}

print "Total time: $total seconds\n";
```

## `split` example, step 2

- Add logic to skip first line.

```
#!/usr/bin/perl
$total = 0; # In seconds
$line_count = 0;

while ($line = <STDIN>) {
    # Need to skip line 1.
    $line_count++;
    next if $line_count == 1;
}

print "Total time: $total seconds\n";
```

**Skips to next line**

**== means "is equal to"**

## `split` example, step 3

- Using split, extract 7th column...

```
# Get time in HH:MM:SS form.
@info = split(/\s+/, $line);
my $time = $info[6];
```

**"\s" means white space, "+" means one or more character**

- ...and the parts of the time.

```
# Convert time to seconds and add to total.
($hours, $mins, $secs) = split(/:/, $time);
$total += $secs +
          $mins * 60 +
          $hours * 3600;
```

## `split` example in full

```
#!/usr/bin/perl
my $total = 0; # In seconds
my $line_count = 0;

while ($line = <STDIN>) {
    # Need to skip line 1.
    $line_count++;
    next if $line_count == 1;

    # Get time in HH:MM:SS form.
    @info = split(/\s+/, $line);
    my $time = $info[6];

    # Convert time to seconds and add to total.
    ($hours, $mins, $secs) = split(/:/, $time);
    $total += $secs + $mins * 60 + $hours * 3600;
}

print "Total time: $total seconds\n";
```

## Hashes

- The third type of container in Perl.
- Like an array, contains many values.
- Each value is indexed by a key, but this time it can be any scalar.
- Visualize it like a table.

| | |
|---|---|
| **Key 1** | **Value 1** |
| **Key 2** | **Value 2** |

- Written with a “%”.

## Assigning to hashes

- We put entries into a hash specifying the key and value.

```
%ages = (
    Noah    => 950,
    Abraham => 175,
    Sarah   => 127
);
```

- We can add a single value to a hash – note the use of $ for a single entry.

```
$ages{‘Isaac’} = 180;
```

## Getting values from a hash

- Accessing a single value:

```
print “Isaac lived to $ages{‘Isaac’}\n”;
```

- It is possible to iterate over the keys in the hash and print each key/value pair.

```
foreach $name (keys %ages) {
    print “$name lived to $ages{$name}\n”;
}
```

- Can also iterate over just the values.

```
foreach $age (values %ages) {
   …
}
```

## Reading from files

- First, open the file.

```
open $fh, “< path/to/file.txt”;
```

**scalar stores file handle**

**< means “read”**

- Use diamond operator to read lines.

```
while ($line = <$fh>) {
    ...
}
```

- When finished with the file, close it.

```
close $fh;
```

## Writing to files

- First, open the file to write.

```
open $fh, “> path/to/file.txt”;
```

**> means “write, replacing anything in the file”**

- Can use >> for append in place of >.
- Use print to write data to the file.

```
print $fh “some stuff\n”;
```

- When finished with the file, close it.

```
close $fh;
```

## Regexes

- One of the scarier looking things you’ll find in Perl code.
- Immensely powerful for extracting and validating input.
- Think of it in terms of pattern matching.
- A regex describes a pattern.
- We test if some text matches it or not.

## Regexes

- Patterns are (usually) written between slashes.
- =~ operator attempts to match a pattern to a scalar, giving true or false.
- Alphabetic characters and numbers match themselves.

```
if ($text =~ /badger/) {
    print “Text contains a badger\n”;
}
```

## Regexes – quantifiers

- Quantifiers state how many of a certain character is required.
- ? 0 or 1
  + 1 or more
  * 0 or more
- Example: pattern /a?b+/

| | |
|---|---|
| xxabxx | matches |
| xaxbbxx | matches |
| xxaxx | doesn't match |

## Regexes – character classes

- Used when any one of a set of characters will do.
- Written in square brackets
- For example, /[aeiou]/ will match a string containing any vowel.
- Can also provide ranges; to match any lowercase letter use [a-z].
- To put “-” in a character class, use “\-”.

## Regexes – built-in character classes

- There are a number of character classes already defined.
- .	Matches any character but newline
  \w	Matches any alphanumeric
  \d	Matches any digit
  \s	Matches white space
  \W	Matches any non-alphanumeric
  \D	Matches any non-digit
  \S	Matches non-white space

## Word frequency example

- The goal is to write a program that:
  - Reads in a text file.
  - Counts the number of times each word appears in the file, being case insensitive.
  - Writes a file containing each word in the input file, sorted alphabetically, and the number of times it appears.

## Word frequency example

- We will use a hash to store the word counts, the hash keys being the words and the values being the counters.

```
# Build count of words from input file.
open $fh, "< example.txt";
%wordcounts = ();
while ($line = <$fh>) {
    # Get words in the line
    # Increment counters for words.
}
close $fh;
```

## Word frequency example

- Extract words using split.

```
chomp $line;
@words = split(/[^\w'\-]+/, $line);
```

^ at start of character class means “anything but…”.

Letters, hyphens, and apostrophes.

- Loop over words and update counters.

```
foreach $word (@words) {
    $wordcounts{lc($word)}++;
}
```

lc function lowercases a string

## Word frequency example

- Finally, need to write results file - a sorted list of words and their counts.

```
# Write counts to a file.
open $fh, "> results.txt";
foreach $w (sort keys %wordcounts) {
    print $fh "$w\t$wordcounts{$w}\n";
}
close $fh;
```

**The sort keyword will sort the elements in an array into ascending order.**

**\t means “tab”**

## Word frequency example

```
# Build count of words from input file.
open $fh, "< example.txt";
%wordcounts = ();
while ($line = <$fh>) {
    # Get words in the line.
    chomp $line;
    @words = split(/[^\w'\-]+/, $line);

    # Increment counters for words.
    foreach $word (@words) {
        $wordcounts{lc($word)}++;
    }
}
close $fh;

# Write counts to a file.
open $fh, "> results.txt";
foreach $w (sort keys %wordcounts) {
    print $fh "$w\t$wordcounts{$w}\n";
}
close $fh;
```

## Command line Perl

- Can write “one-liners” on the command line.
- Use the -e flag and put the script in quotes afterwards.

```
perl -e ‘print “hello world!”;’
```

- Here’s the line counting script written as a “one-liner”.

```
perl -e ‘$x++ while <>; print “$x\n”’
```

## Command line Perl

- Specifying the -n flag will enclose the script in a “while (<>) { … }” loop.
- Note that when <> is not assigned to a scalar, the value is placed in the default scalar $_.
- This script extracts the second white space delimited column of each line.

```
perl -ne ‘@x=split/\s+/;print"$x[1]\n"’
```

**split function’s second parameter defaults to $_**

## CPAN

- Comprehensive Perl Archive Network
- A mass of Perl modules, generally well documented and ready to use, that do a very wide range of tasks.
- Web interface: http://search.cpan.org/
- Perl comes with a command line module installer.

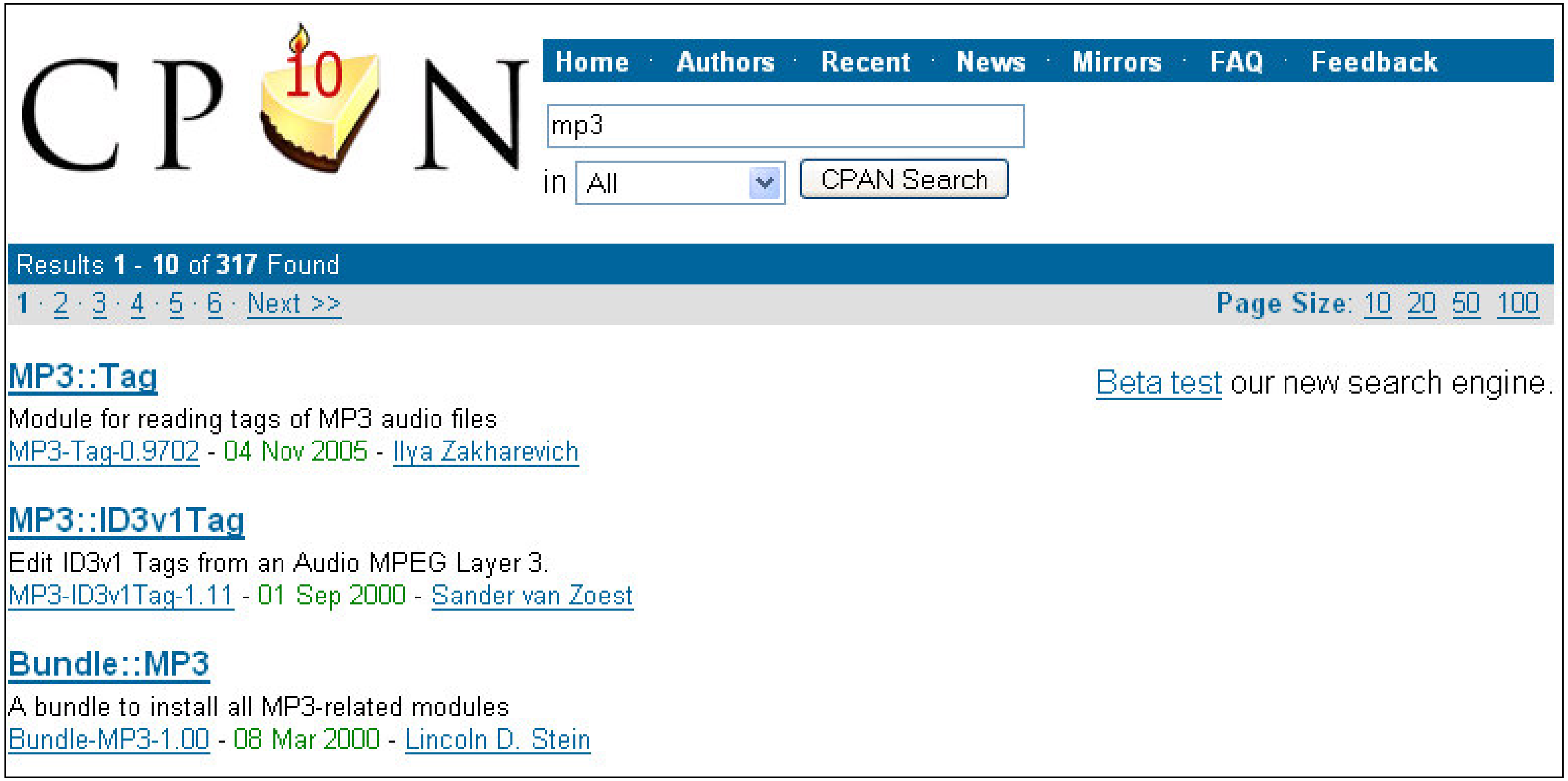
```
perl -MCPAN -e ‘shell’
> install Module::Name
```

## CPAN Example

- We want to pipe a list of filenames of MP3s to the script.
- The script should extract the artist from each MP3 file – the hard part.
- It should then produce a list of all artists that we have songs by, sorted in order of the number of songs we have by that artist.

## CPAN Example

- Goto CPAN web interface, search for MP3. The top result looks promising.



Results **1** - **10** of **317** Found

**1** · 2 · 3 · 4 · 5 · 6 · Next >> **Page Size:** 10 20 50 100

Beta test our new search engine.

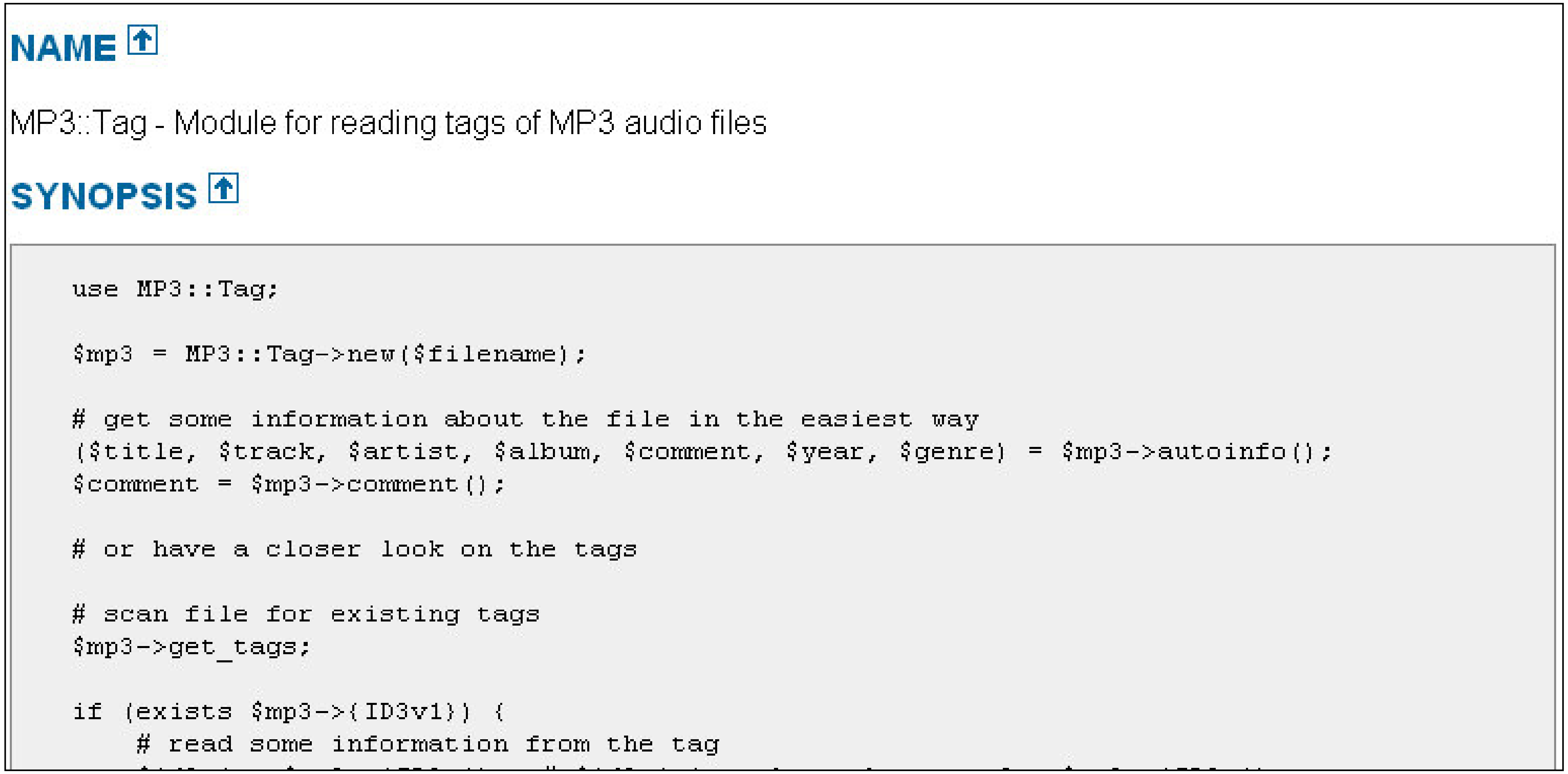
**MP3::Tag**
Module for reading tags of MP3 audio files
MP3-Tag-0.9702 - 04 Nov 2005 - Ilya Zakharevich

**MP3::ID3v1Tag**
Edit ID3v1 Tags from an Audio MPEG Layer 3.
MP3-ID3v1Tag-1.11 - 01 Sep 2000 - Sander van Zoest

**Bundle::MP3**
A bundle to install all MP3-related modules
Bundle-MP3-1.00 - 08 Mar 2000 - Lincoln D. Stein

## CPAN Example

- Each module has a synopsis with the most common use cases.



NAME

MP3::Tag - Module for reading tags of MP3 audio files

SYNOPSIS

```
use MP3::Tag;

$mp3 = MP3::Tag->new($filename);

# get some information about the file in the easiest way
($title, $track, $artist, $album, $comment, $year, $genre) = $mp3->autoinfo();
$comment = $mp3->comment();

# or have a closer look on the tags

# scan file for existing tags
$mp3->get_tags;

if (exists $mp3->{ID3v1}) {
    # read some information from the tag
```

## CPAN Example

- In this case, we need not read any further – the synopsis has all we need to know to extract the artist!

```
use MP3::Tag;

$mp3 = MP3::Tag->new($filename);

# get some information about the file in the easiest way
($title, $track, $artist, $album, $comment, $year, $genre) = $mp3->autoinfo();
```

## CPAN Example

- Run CPAN installer (as root). The first time, it needs to be configured.

```
[jonathan@zion jonathan]$ su
Password:
[root@zion jonathan]# perl -MCPAN -e 'shell'
/usr/lib/perl5/5.8.3/CPAN/Config.pm initialized.

CPAN is the world-wide archive of perl resources. It consists of about
100 sites that all replicate the same contents all around the globe.
Many countries have at least one CPAN site already. The resources
found on CPAN are easily accessible with the CPAN.pm module. If you
want to use CPAN.pm, you have to configure it properly.

If you do not want to enter a dialog now, you can answer 'no' to this
question and I'll try to autoconfigure. (Note: you can revisit this
dialog anytime later by typing 'o conf init' at the cpan prompt.)

Are you ready for manual configuration? [yes] no
```

## CPAN Example

- Once it's configured itself (usually works) we can install the module.

```
cpan> install MP3::Tag
...
Checking if your kit is complete...
Looks good
Writing Makefile for MP3::Tag
cp Tag/CDDB_File.pm blib/lib/MP3/Tag/CDDB_File.pm
...
  /usr/bin/make  -- OK
Running make test
...
All tests successful.
Files=5, Tests=293, 3 wallclock secs (2.32 cusr + 0.39 csys = 2.71 CPU)
  /usr/bin/make test - OK
Running make install
...
  /usr/bin/make install  -- OK
```

## CPAN Example

- Now we can start to write our script…

```
#!/usr/bin/perl
use MP3::Tag;

# Count songs by each artist.
%artists = ();
while ($filename = <STDIN>) {
    chomp $filename;
    $mp3 = MP3::Tag->new($filename);
    @info = $mp3->autoinfo();
    $artists{$info[2]}++;
}
```

## CPAN Example

- ...and finish it. We supply `sort` with a block containing the condition. The values from the list being sorted are in `$a` and `$b`, and `<=>` means compare.

```
# Output, sorted by artist's count.
@sorted = sort
          { $artists{$a} <=> $artists{$b} }
          keys %artists;
foreach $artist (@sorted) {
    print "$artists{$artist}\t$artist\n";
}
```

## Going Further

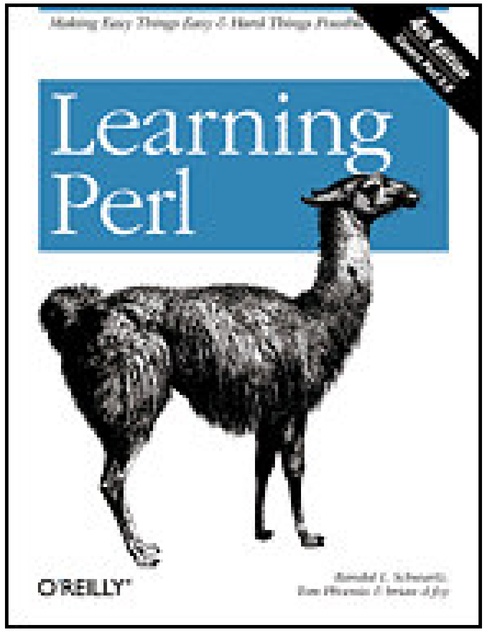
- I can't even start to fully cover Perl in an hour.
- The best way to learn Perl is to write Perl.
- The Perl documentation is available on the web: http://perldoc.perl.org/
- Loads of CGI tutorials – just search.

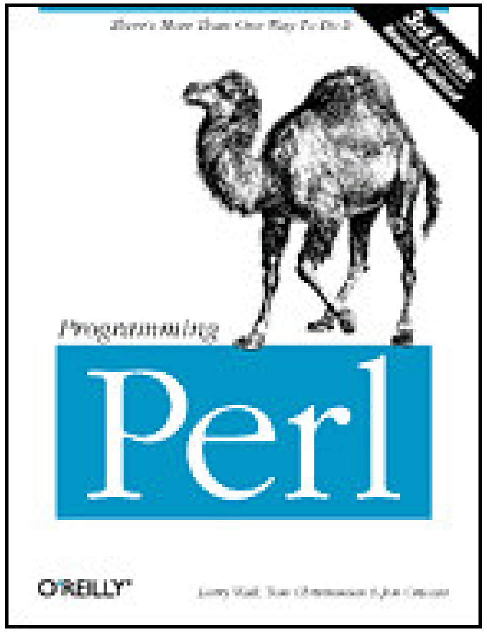
## Going Further

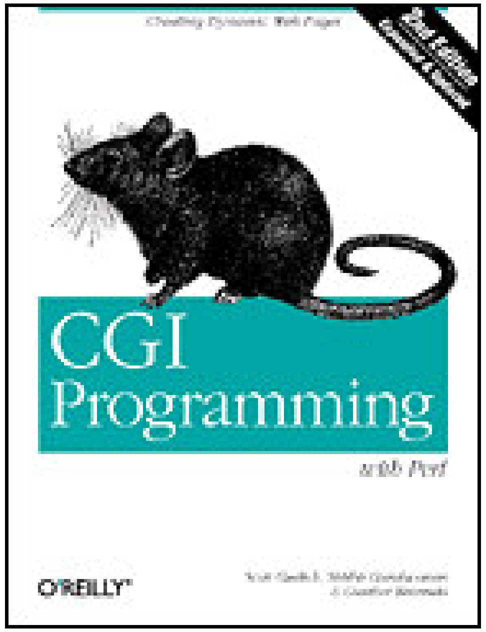
- Documentation is also available at the console.
  - `man perl`
  - Get help on functions:
    `perldoc -f sort`
  - Get help on installed modules:
    `perldoc MP3::Tag`

## Going Further

- O'Reilly books are very good.
- Programming Perl is considered the definitive guide, and is excellent.
- Learning Perl goes at a gentler pace.







## The End

- Slides on my site: http://www.jwcs.net/~jonathan/
- Go forth, hack Perl and have fun.