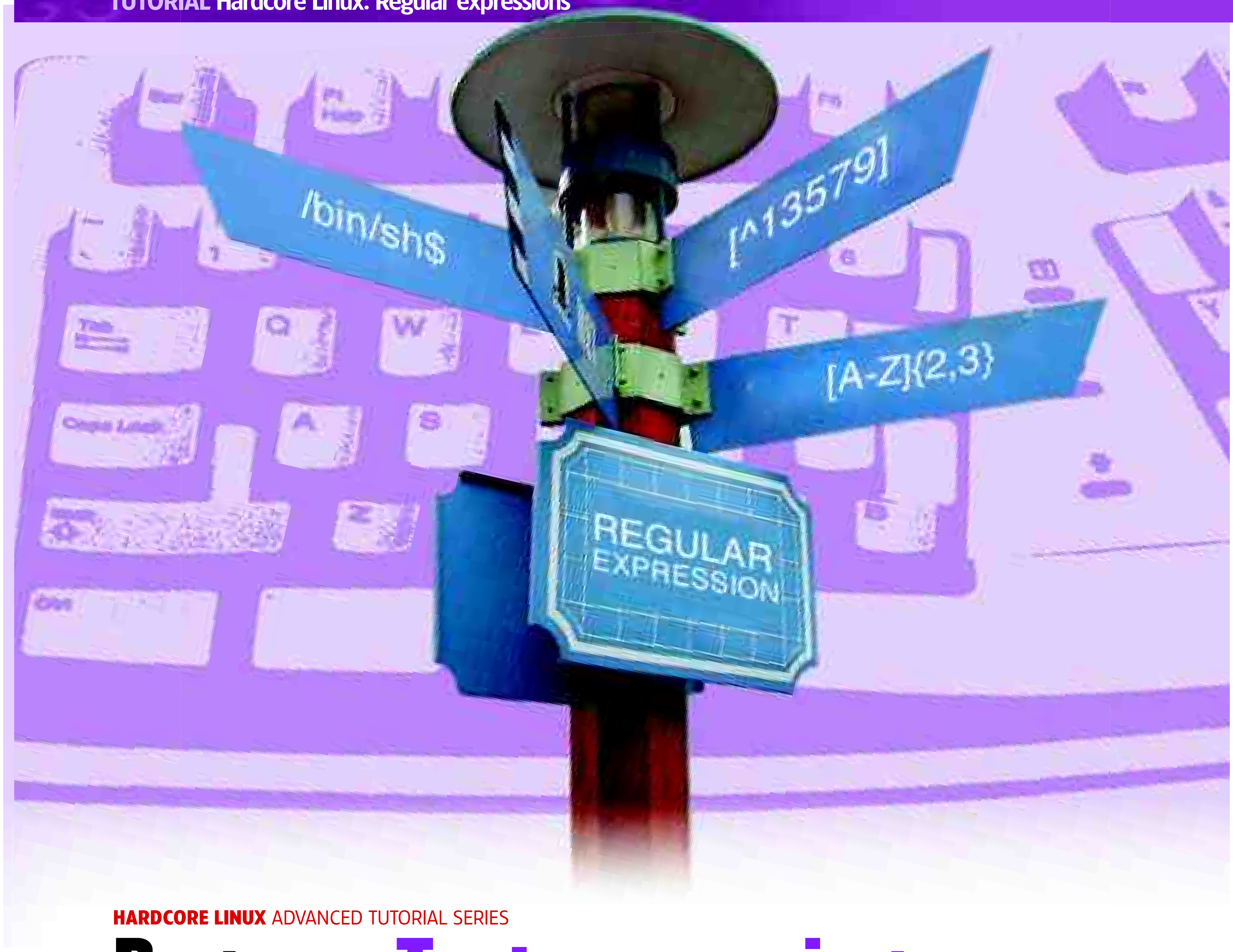
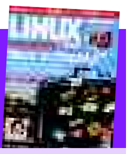


HARDCORE LINUX ADVANCED TUTORIAL SERIES

Regexes Text processing

Find your way around regular expressions, and go looking for genomes with **Dr Chris Brown**. Maybe.

LAST TIME



It's important to do the basics well, so we went back to *Bash* for job control, directory browsing and environmental variables. If you missed the issue, call 0870 837 4773 or +44 1858 438795 for overseas orders.

Good morning. Welcome to the regular expression masterclass. This is going to require your full attention, so no talking at the back, no doodling and no emailing, please. Now, before we start, I want to make sure that you can all find the backslash on your keyboard. Yes? And the curly brackets? Good! OK, let's get started.

Regular expressions (also known as regexes, of course) are notations for specifying patterns in text. They have existed in Linux (and Unix before it) for a very long time. The earliest reference to them that I can find is in the manual page for *Ed* (an early text editor) in my dog-eared, faded manual for the sixth edition of Unix. The page is dated 15 January 1973, though the origin of regular expressions is older than that. Regexes have found their way into many Linux tools. They appear in the search and substitute commands in text editors like *Vi*, and even *OpenOffice.org*. They're used in classic filter programs like *grep*, *sed* and *awk*. They appear in the URL rewriting rules in *Apache*. They're central to the power of languages like PHP and Perl. And there are libraries available for handling regular expressions for many languages, including the class libraries in ASP.NET.

You can use regexes for all kinds of things. Regexes can:

- Strip comments out of a configuration file.
- Find empty paragraphs in an *OpenOffice.org* document.
- Verify that a text string is a valid IP address.
- Pull email addresses such as **chris@example.com** out of text files.
- Isolate the year from a date such as 17-03-2006.
- Search a word list for palindromes.
- Search a genome for a specific DNA sequence.

With regard to the last example, an entire sub-industry of using regular expressions and Perl in the processing of bioinformatic data has grown up*.

To start, let's be clear that regexes are *not* the same as the filename wildcards understood by the shell. As an example of

*Sadly, I can't help you with genetics – I'm a doctor of theoretical physics, actually. But I know there are at least three O'Reilly books on the subject!

the latter, if the shell sees the command

```
rm tutorial*.v[0-2]
```

it will expand the argument by looking for filenames that match the wildcard pattern – in this case, any name that begins with 'tutorial' and ends with .v0, .v1 or .v2.

Regexes and wildcards share some of the same metacharacters (such as * and []), but the metacharacters have different meanings; and the contexts in which wildcards and regexes are used are quite different. Wildcards, generally, are used to match filenames. Regexes are used to match text in a file, or in a string being manipulated by a program.

Anyway, it's time to begin the class. I'll assume that you're at least slightly familiar with *grep*. Its default behaviour is to print each matching line. For example, the command

```
grep xen ~/book/chapter5.txt
```

will display any line in **chapter5.txt** that contains the string 'xen'. Here, the search string 'xen' is just three literal characters that match themselves – 'x', 'e' and 'n'. This is a valid regular expression, and quite painless, though not particularly interesting. Searching for fixed strings in this way is nonetheless useful. For example, suppose you want to scan several chapters to check which ones contain a reference to 'xen'. A command like this will suffice:

```
grep -l xen ~/book/chapter[0-9].txt
```

The *grep* option **-l** says, "Don't show me the matching lines, just show the names of the files that contain matching lines". Let's be clear, though – the notation **[0-9]** in this example is a filename wildcard interpreted by the shell, not a regular expression interpreted by *grep*.

Meet the metacharacters

It's time to move on to some real regular expressions – you need to be awake for this. The table below summarises the regular expression syntax that we'll be discussing. Consider

```
grep -l '[Xx]en' ~/book/chapter[0-9].txt
```

Notice that I have put single quotes around the regex. I've done this to prevent the shell interpreting the **[Xx]** as a filename wildcard, which of course isn't what we want. We want the shell to pass the argument **[Xx]en** literally to *grep*, allowing *grep* to interpret the **[]** notation.

Quoting metacharacters to ensure that they get interpreted at the right time and not before is an important art in the syntax-rich world of shell programming and regular expressions. You can write a character class such as **[AEIOU]** to match any upper-case vowel, or **[0123456789]** to match any digit. This example can be written more compactly as **[0-9]**, where the

"QUOTING METACHARACTERS IS AN IMPORTANT ART IN THE WORLD OF SHELL PROGRAMMING."

hyphen indicates a range. Similarly, you might use **[a-z]** to match any lower-case letter.

Keep in mind, though, that however much stuff is inside the square brackets, you'll only match a single character within the text. Putting a ^ at the beginning of the list of characters reverses the sense of the match, so (for example) **[^0-9]** matches anything that isn't a digit.

The characters ^ and $ are used to anchor a match to the start and end of a line respectively. For example, **^login** matches lines that begin with 'login', and **[0-9]$** matches lines that end with a digit.

Let me show you an example. There are some configuration files, such as **/etc/ssh/ssh_config**, that contain large numbers of comment lines, which begin with '#'. While such comments are useful as documentation they can make it hard to find the non-commented directives in the file. Using regular expressions, »

REGULAR EXPRESSIONS AT A GLANCE

Here are 13 of the most common regexes that you'll be using in this tutorial

Notation	Description	Example	What the example matches
a	'Ordinary' characters match themselves	apple	The string 'apple'
[...]	Any character enclosed in []	[02468]	Any even digit
[^...]	Any character not in set	[^13579]	Anything except an odd digit
[x-x]	Range of characters	[A-Z]	Any upper-case letter
.	Any single character	c.t	cut, cat, c9t and so on
^	Beginning of line (start of string)	^[0-9]	Lines starting with a digit
$	End of line (end of string)	/bin/sh$	Lines ending with /bin/sh
*	Zero or more of the preceding item	[a-z]*	Any sequence of lower-case letters, including none at all
?	Zero or one of the preceding item	https?://	http:// and https://
+	One or more of the preceding item	T+	T, TT, TTT, TTTT and so on
{n}	Exactly n occurrences of the preceding item	[0-9]{3}	A sequence of exactly three digits eg 123, 644 and 999
{n,}	n or more occurrences of the preceding item	0{3,}	000, 0000, 00000 and so on
{n,m}	Matches between n and m occurrences of the preceding item	[A-Z]{2,3}	Things like AB, ABC, YY, ZZZ

we can select the non-comment lines like this:

```
grep -v '^#' /etc/ssh/ssh_config
```

The **-v** option tells *grep* to print lines that do not match the regular expression, so here we're saying, "Do not print lines that begin with a #." There's an alternative solution, like this:

```
grep '^[^#]' /etc/ssh/ssh_config
```

which says, "Do print lines that don't begin with a #." If you try these examples you'll discover that they are not exactly equivalent. The difference lies in the handling of blank lines. Neither regular expression matches a blank line, so the first example includes blank lines in the output, and the second does not. Note that in the second example, the two ^ characters mean entirely different things.

Speaking of blank lines, they can be matched using the regex **^$**. So the command

```
grep -v '^$' somefile
```

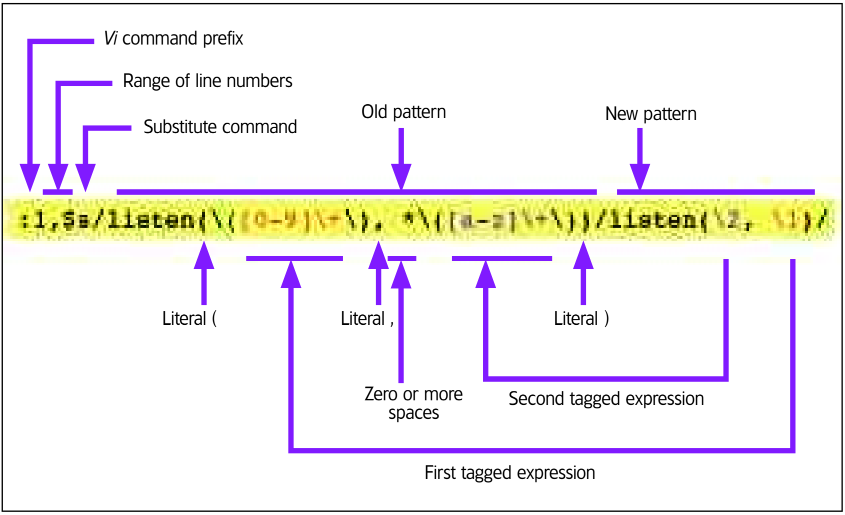
will show the non-blank lines in a file.

Extend your knowledge

One of the things that marks the amateur from the pro is the ability to distinguish between basic and extended expressions. Originally, *grep* understood only basic regexes, and there was a separate program called *egrep* that understood the fancier extended regexes. The GNU version of *grep* does both; it uses basic regexes by default, but uses extended regexes if you supply the **-E** option.

As an example, the brackets (and) have no special meaning when you're using basic regexes – they just match themselves. Using extended regexes, (and) are used to group parts of a regular expression, rather like using parentheses to groups parts of an arithmetic expression, as in (a+b)*c. To match a literal parenthesis when you're using extended regexes, you would need to turn off the special meaning of the brackets by preceding them with a backslash – thus, **\(** would match a literal opening parenthesis. Just in case you're not confused yet, some programs (including GNU *grep*) that use basic regular expressions by default allow you to prefix characters such as **(** and **)** with a backslash in order to turn their special meaning on. This can be extremely confusing, and you need to keep your wits about you to figure out whether any particular backslash is turning the special meaning of the following character on or off.

Now we come to the repetition modifiers in the table. This is where you need to actually start paying attention. A repetition modifier doesn't make any sense by itself – it specifies how many times the preceding regular expression must appear in the pattern. For example, **[0-9]*** matches zero or more digits; **[A-Z]+** matches one or more upper-case letters; and so on. A very common example is **.***, which means "any amount of anything". So the expression **login.*failed** will match any line that contains the string 'login' followed by the string 'failed' with any amount of intervening text.



```
:1,$s/listen(\([0-9]\+\), *\([a-z]\+\))/listen(\2, \1)/
```

Dissecting a regular expression. Lots of syntax but no blood.

THE FLAWLESS LOGIC OF THE REGEX

It's not obvious why [^#] doesn't match a blank line. After all, blank lines don't start with a #, right? The fact is, even [^#] has to have a character to match against, in order to make sure that it's not a #.

Regexes step by step

As a practical example of using repetition modifiers, we'll develop a regular expression for recognising IP addresses such as 192.168.0.42. We might describe a valid IP address as: "Four decimal numbers between 0 and 255, separated by periods." Unfortunately we can't translate this directly into a regular expression, because regexes can't do arithmetic comparisons; they can only match characters. So, we'll make do with a crude approximation like this: "An IP address consists of four decimal numbers of between one and three digits, separated by periods."

This does indeed match real IP addresses, but it also matches nonsense addresses like 123.456.789.0. Let's build up the regex step by step. First, **[0-9]{1,3}** matches between one and three digits. To add a literal **.** (full stop) to the end, we need **[0-9]{1,3}\.**. Note the backslash – it suppresses the special meaning of the period to match a literal full stop.

Now we need to say, "All of this must appear three times." We can use another repetition modifier for this. At first glance we might simply add it on to the end like this: **[0-9]{1,3}\.{3}**. But this doesn't work, because the repetition modifier applies only the immediately preceding expression (the literal **.** here). What we have to do is group the earlier part of the expression using parentheses so that the repetition modifier applies to all of it, like this: **([0-9]{1,3}\.){3}**. I'm assuming the use of extended regular expressions here: if we were using *grep* in basic mode (without the **-E** flag) we would need to prefix the parentheses and curly brackets with backslashes to turn their special meaning on – so the expression would be **\([0-9]\{1,3\}\.\)\{3\}**. Finally, we need to complete the expression to match the final group of digits in the IP address, like this: **([0-9]{1,3}\.){3}[0-9]{1,3}**. Is your brain hurting yet?

Extracting email addresses

It's time for a more challenging example. Here's my mission: I have a large number of text files that contain email addresses within them. The address is often embedded within other text; a typical line might be: "Ask andrew (**andy@example.net**) if he knows..."

I'd like to extract a list of all the unique email addresses that appear in these files. We'll approach the problem in four steps:

- **Step 1** Write down a rule (in plain English) that defines what an email address looks like.
- **Step 2** Design a regular expression that implements the rule.
- **Step 3** Use *grep* with the regular expression to extract a raw list of email addresses.
- **Step 4** Wrap a little filtering and shell scripting around *grep* to provide a complete solution.

The first step is the hardest. A simple attempt at the rule might state that an email address consists of:

1 A string of one or more lower- or upper-case alphabetic characters (the username).

2 A literal @.
3 One or more components of the mail domain name. Each component consists of one or more alphabetic characters followed by a literal '.'.
4 A top-level domain name consisting of two or three alphabetic characters.

Now move on to step 2 – translating those four parts of our rule into regex syntax, one part at a time. The first part translates as **[a-zA-Z]+**. The second translates to **@** (@ is not a regex metacharacter, so it doesn't need escaping).

The third part needs a bit more thought. A regex to match a single component of a domain name might be **[a-zA-Z]+\.**, which will match things like example.. Again, note the use of **\.** to match a literal **.**. However, there can be one or more such components in a full domain name, so we enclose this part of the regex in parentheses and put a **+** after it to get **([a-zA-Z]+\.)+**. This will match things like 'example.', 'foo.example.' and 'foo.bar.example.'.

The fourth of our above rules uses one of the fancier repetition modifiers from the table on page 81. It can be written as **[a-zA-Z]{2,3}**.

Putting these four parts together results in the regular expression **[a-zA-Z]+@([a-zA-Z]+\.)+[a-zA-Z]{2,3}**.

Of course, this is a simplification. For example, usernames (and domain names) can also contain digits and other non-alphabetic characters – our regex won't match those. Also, there is only a limited number of valid top-level domains – our regex would match **andy@example.xyz**, but that isn't a real address because there is no .xyz domain. However, this regex is adequate for an initial implementation.

It's common to start with a regex that is more or less correct, then refine it. Usually, the first 10% of your effort will result in a regular expression that works in 90% of all cases. The remaining 90% of the effort lies in refining the expression to handle the remaining 10% of the cases.

With step 3, the idea is to use *grep* to apply this regular expression to all our text files, which I'll take to be all filenames ending in .txt. So the command is going to look something like

```
grep '[a-zA-Z]+@([a-zA-Z]+\.)+[a-zA-Z]{2,3}' *.txt
```

GREEDY REGULAR EXPRESSIONS

Regular expressions are greedy – they match as much as they can. A more precise description is that they match the leftmost, longest string. Consider matching the regex t+ against the string 'aatttaatttttaa' – admittedly a rather contrived example. Now, t+ matches 't', 'tt', 'ttt' and so on. According to the rules, it begins matching at the very first 't' in the string, and continues to match for as long as it can. Thus, it matches the first string 'ttt' in the target, despite the fact that there is a longer string of 't's later in the target.

If we just want the classic *grep* behaviour – to know which lines contain a match for the regex – we don't care exactly how much text was matched, we are content simply to know that there was a match. But if we want to perform a substitution on the matched text, the leftmost-longest rule becomes important.

As an example, suppose you want to extract user names from /etc/passwd. Each line in this file defines one user account; the lines are of the form:

```
chris:x:1000:100:Chris Brown:/home/chris:/bin/bash
```

Your task is to strip out everything from the first colon onwards. Relying on the greediness of regular expressions, we can do this with the command:

```
sed 's/:.*//' /etc/passwd
```

Here, the oldpat of our substitution is :.* and the newpat is the empty string. Thus, the substitution strips out everything that the regex matches. In this case it matches from the first colon to the end of the line.

But we're not quite there yet. For a start, we have used a number of extended regular expression metacharacters, and *grep* won't understand these without the **-E** flag. Second, *grep* will normally print the whole of each line that contains a match. We only want it to print the email address (the part that actually matches the regex). The **-o** flag does this. Third, *grep* normally prefixes each match with the name of the file in which the match was found. Since we don't want this, we use the **-h** option to suppress it.

So the complete command looks like this:

```
grep -E -o -h '[a-zA-Z]+@([a-zA-Z]+\.)+[a-zA-Z]{2,3}' *.txt
```

This command will generate a list of email addresses. The output might look like this:

```
santa@northpole.com
isaac@trinity.cam.ac.uk
jrrtolkien@exeter.ox.ac.uk
bill@millisoft.uk.com
isaac@trinity.cam.ac.uk
bill@MILLISOFT.UK.COM
```

I've deliberately crafted this list to show up a couple of problems. First, domain names are not case-sensitive, and you'll sometimes see them written in upper case. We should not consider two addresses to be different just because they differ in case. Second, an address might appear more than once in the list.

A little post-processing

On to step 4 of our solution. This has nothing to do with regular expressions really, but it's typical of the sort of downstream processing you might want to do to the output of *grep* in order to make the result more useful. First let's deal with the non-case-sensitivity issue by converting everything to lower case. Happily, we can easily do this by piping the output from *grep* into *tr*, like this:

```
grep -E -o -h '[a-zA-Z]+@([a-zA-Z]+\.)+[a-zA-Z]{2,3}' *.txt | tr
A-Z a-z
```

Although it is tempting to assume that the arguments to *tr* are regular expressions, they are not. They are simply arguments to *tr* meaning, "Convert all characters in the set A–Z into the corresponding character in the set a–z." All we're doing here is to fold upper case to lower case.

The next step is to sort the output. The object of this is to ensure that identical entries appear on adjacent lines. We simply pipe the output into *sort*:

```
grep -E -o -h '[a-zA-Z]+@([a-zA-Z]+\.)+[a-zA-Z]{2,3}' *.txt | tr
A-Z a-z | sort
```

Now the output looks like this:

```
bill@millisoft.uk.com
bill@millisoft.uk.com
isaac@trinity.cam.ac.uk
isaac@trinity.cam.ac.uk
jrrtolkien@exeter.ox.ac.uk
santa@northpole.com
```

Finally, we can use *uniq* to discard repeated instances of lines. This yields our final answer:

```
grep -E -o -h '[a-zA-Z]+@([a-zA-Z]+\.)+[a-zA-Z]{2,3}' *.txt |
tr A-Z a-z | sort | uniq
```

which results in the output:

```
bill@millisoft.uk.com
isaac@trinity.cam.ac.uk
jrrtolkien@exeter.ox.ac.uk
santa@northpole.com
```

So there you have it: *grep* for a fancy regular expression and do some downstream filtering to post-process the output – a very typical command line solution. If you understand this, congratulations! You've earned your regex stripes. LXF

NEXT MONTH

An Italian job for the next Hardcore Linux tutorial, as Marco Fioretti takes on *DansGuardian*.