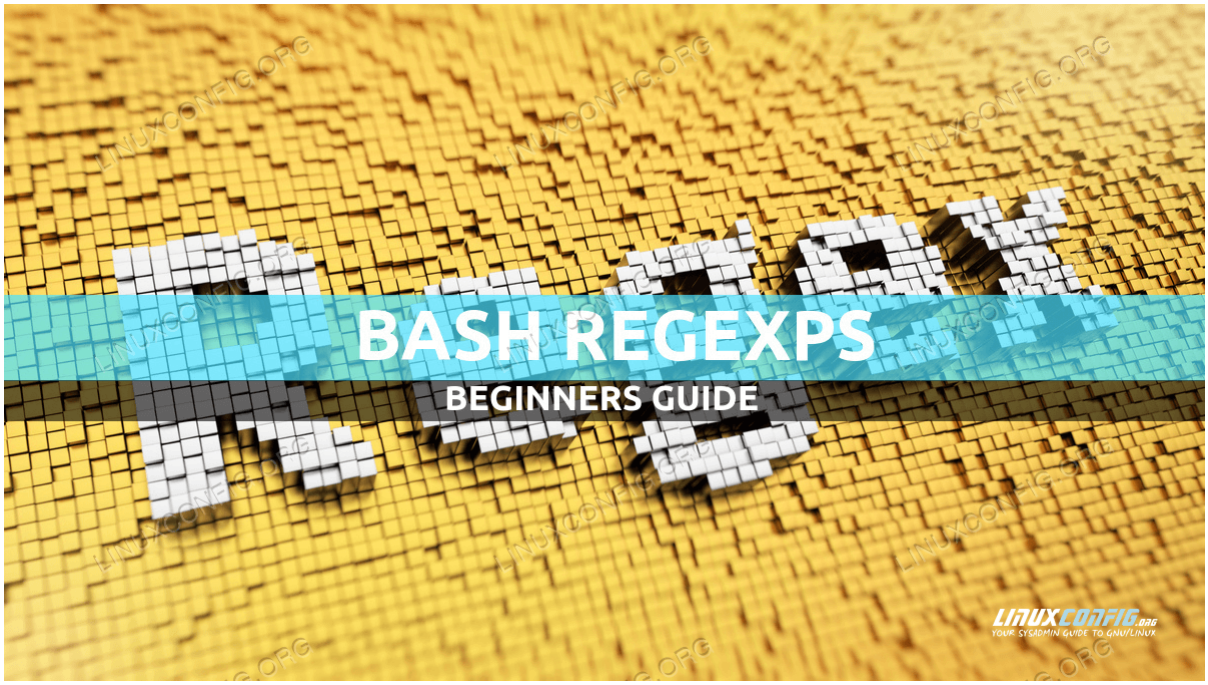
11 August 2020 | Roel Van de Paar

Bash regexps for beginners with examples

Using regular expressions in Bash provides you with plenty of power to parse nearly every conceivable text string (or even full documents), and transform them into nearly any output desirable. If you regularly use Bash, or if you regularly work with lists, textual strings, or documents in Linux, you will find that many jobs can be simplified by learning how to use regular expressions in Bash. Continue reading to learn basic Bash regular expression skills! If you are already familiar with basic regular expressions in Bash or another coding language, see our more advanced bash regular expressions. If not, continue reading to learn basic Bash regular expression skills!

In this tutorial you will learn:

- How to use regular expressions on the command line in Bash
- How regular expressions can parse and transform any text string and/or document
- Basic usage examples of regular expressions in Bash



Bash regexps for beginners with examples

Software requirements and conventions used

Software Requirements and Linux Command Line Conventions

Category	Requirements, Conventions or Software Version Used
System	Linux Distribution-independent
Software	Bash command line, Linux based system
Other	The sed utility is used as an example tool for employing regular expressions
Conventions	# – requires given linux-commands to be executed with root privileges either directly as a root user or by use of `sudo` command $ – requires given linux-commands to be executed as a regular non-privileged user

Example 1: our first regular expression

There are several common command line utilities like sed and grep which accept Regular Expression input. And, you do not have to make any changes in the tool (use or setup) to be able to use Regular Expressions either; they are by default regex-aware. Let's look at a non-regex example where we change `abc` into `xyz` first:

```
$ echo 'abc' | sed 's/abc/xyz/'
xyz
```

Here we have used echo to output the string `abc`. Next we pass the output from this echo (using the pipe, i.e. `|`, character) to the sed utility. Sed is a stream editor for filtering and transforming text. I encourage you to checkout it's detailed manual by typing `man sed` at the command line.

Once passed to sed, we are transforming the string by using a sed-specific (and regex-aware) syntax. The command we pass to sed (namely `s/abc/xyz/`) can also be read as `substitute abc with wyz`. The `s` stands for substitute, and the separator character (`/` in our case) indicates where one section of the command ends and/or another starts. Note that we can also use other separator characters in sed, like `|`, as we will seen in later examples.

Now, let's change this command into a regular expression example.

```
$ echo 'abc' | sed 's/./xyz/g'
xyzxyzxyz
```

Wow, what happened here? 🙂

We made a few small changes, which have significantly affected the resulting output. Firstly, we swapped `abc` in the sed command line to `.`. This is not a regular/literal dot, but rather a regular-expression dot. And, in regular expression, a dot means **any character**. Things should start to look clearer now, especially when you notice the other small change we made: `g`. The easiest way to think about `g` is as `global`; a repetitive search and replace.

Notice here too how `s` is our actual sed command, followed by the options for that command (the two from-to replacement texts), and the `g` is a qualifier over the command. Understanding this well helps you to learn sed syntax at the same time.

So, in some contrast to our fist non-regular expression example, and in natural language, this new command can be read as **substitute any-single-character with `xyz`, and repetitively ('globally') do so until you reach the end of the string**. In other words, `a` is changed to `xyz`, `b` is changed to `xyz` etc., resulting in the triple output of xyz.

All onboard? Great! You just learned how to use regular expressions. Let's dive in further.

Example 2: A small caveat

```
$ echo 'abc' | sed 's|\.|xyz|g'
abc
```

Oops. What happened? We made a few minor changes, and the output changed substantially, just like in our previous example. Regular Expressions are very powerful, as you can start to see here, and even a minor change can make a large difference in the output. Hence, there is usually a need to test your expressions well. And, whilst not the case here, it is also very important to always consider how the output of regular expressions may be affected by different input. Often, a slightly changed or modified input will yield a very different (and often erroneous) output.

We changed two minor items; we placed a `\` before the dot, and we changed the separators from `/` to `|`. The latter change made absolutely no difference, as we can see from this output;

```
$ echo 'abc' | sed 's|.|xyz|g'
xyzxyzxyz
```

And we can double check our findings this far by using this command:

```
$ echo 'abc' | sed 's/\./xyz/g'
abc
```

As expected, the `|` to `/` change made no difference.

So back to our dilemma – shall we say that the minor change of adding `\` is at fault? But is it really a fault?

No. What we have done by making this simple change, is to make the `.` dot into a literal (`\.`) dot. In other words, this is no longer a real regular expression at work, but a simple textual string replacement which can be read as **substitute any literal dot into `xyz`, and do so repetitively**.

Let's prove this;

```
$ echo 'ab..c' | sed 's/\./xyz/g'
abxyzxyzc
```

This is as expected: the two literal dots were changed, individually (due to the repetitive nature of the `g` qualifier), to `xyz`, overall yielding `abxyzxyzc`.

Super! Let's expand a bit more now.

Example 3: Bring it on

Nothing like diving in head first, right? Perhaps. Until you see this;

```
$ echo 'a..b..c' | sed 's|[\.b]\+|d|g;s|[a-c]|d|g'
ddd
```

Yes, too complex, at least at first sight. Let's start with a simplification thereof:

```
$ echo 'a..b..c' | sed 's|[\.b]\+|d|g;'
adc
```

Still looks a little tricky, but you will soon understand it. So, taking the input string of `a..b..c`, we can see – based on our previous example – that we are looking for a literal dot (`\.`). However, in this case it is followed by `b` and surrounded by `[` and `]`. This part of the regular expression (`[\.b]`) can be read as **any literal dot, or the character `b` (so far non-repetitively; i.e. a single charter, either one of them, will match this selector)**.

Next, we qualify this a bit further by appending `\+` to this **selection box**. The `\+` indicates that we are looking for at least one, and possibly more, of these listed characters (literal dot and b). Note that the characters searched for need to be right next to each other, in any order.

For example the text `...b....bbbb...` would still be matched as a single occurrence, whereas `...b...bbb... ...b.b...bb` (note the space) would be match as separate (**repetitive**) occurrences, and both (i.e. not just the first one) would be matched. And, in that case, both would be actioned upon due to the `g` global/repetitive qualifier.

In other words, in natural language we could read this regular expression as **substitute any contiguous sequence of the characters `.` and `b` with `d` and do so repetitively**.

Can you see what happens? In the input string we have `..b..`, which is matched by the regular expression as it contains only `\.` and `b` characters. It is then substituted for `d` resulting in `adc`.

Our larger example now looks simpler all of the sudden. Let's jump back to it:

```
$ echo 'a..b..c' | sed 's|[\.b]\+|d|g;s|[a-c]|d|g'
ddd
```

Thinking about how the first part of the sed command transformed `a..b..c` into `adc`, we can now think about this `adc` as the input to the second command in the sed; `s|[a-c]|d|g`. Notice how both sed commands are separated by `;`.

All that happens is that the output of the former is taken as the input for the subsequent command. This almost always works, though there are times (when using complex text/document modification) where it is better to pass the output from one actual sed command into another sed command using a Bash pipe (`|`).

Analyzing the second command (`s|[a-c]|d|g`) we see how we have another **selection box** which will select letters from a to c (`[a-c]`); the `-` indicates a range of letters, which is all part of the regular expression syntax.

The other parts of this command speak for themselves now. In total, this second command can thus be read as **substitute any literal character with range `a-c (i.e. a, b or c)` into `d` and do so repetitively**. The result is that the a, d and c (output of `adc` from our first command) are rendered into `ddd`.

That very complex command doesn't look so scary anymore now, does it? Let's round up.

Example 4: A parting message

```
echo 'have a great day' | sed 's|$| all|;s|y|y to|;s|$|you|;s|to [la]\+|to |g;s|$| all|'
```

Can you figure it out? Tip; `$` means **end of line** in regular expressions. All the rest of this complex regex is using knowledge from this article. What is the output? See if you can figure it out using a piece of paper, without using the command line. If you did – or if you didn't 🙂 – let us know in the comments below.

Conclusion

In this tutorial, we had an introduction to basic regular expressions, joined with a few (tongue-in-cheek) more advanced examples.

When learning regular expressions, and checking out other people's code, you will see regular expressions which look complex. Take the time to figure them out, and play around with regular expressions on the command line. You'll soon be an expert, and whilst analysis of complex regexes is usually necessary (the mind just does not lend itself readily to reading so dense information), it will become easier. You will also find that a complex looking regex, on further analysis, usually looks quite simple once you understand it – just like in the examples above.

You may now also like to read our article on Regular Expressions in Python as many of the information provided there also applies to Bash Regular Expressions, though some of the formatting requirements are slightly different. It will boost your understanding of Regular Expressions, how to use them, and how to apply them in various

situations and coding languages. Once you become a regex expert, the small lines of distinction between tools and programming languages usually fades, and you will tend to remember specific syntax requirements for each language or tool you work in/with.

Enjoy!

Related Linux Tutorials:

- Advanced Bash regex with examples
- Python Regular Expressions with Examples
- Big Data Manipulation for Fun and Profit Part 1
- Compare string in BASH
- How to find a string or text in a file on Linux
- How to Correctly Grep for Text in Bash Scripts
- Big Data Manipulation for Fun and Profit Part 3
- Bash Advanced Variable Idioms for Case Sensitivity...
- Correct Variable Parsing and Quoting in Bash
- Bash Loops with examples
