11 August 2020 | Roel Van de Paar

Advanced Bash regex with examples

Using the power of regular expressions, one can parse and transform textual based documents and strings. This article is for advanced users, who are already familiar with basic regular expressions in Bash. For an introduction to Bash regular expressions, see our Bash regular expressions for beginners with examples article instead. Another article which you may find interesting is Regular Expressions in Python.

Ready to get started? Dive in and learn to use regexps like a pro!

In this tutorial you will learn:

- How to avoid small operating system differences from affecting your regular expressions
- How to avoid using too-generic regular expression search patters like `.*`
- How to employ, or not employ, extended regular expression syntax
- Advanced usage examples of complex regular expressions in Bash

Advanced Bash regex 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: Heads up on using extended regular expressions

For this tutorial, we will be using sed as our main regular expression processing engine. Any examples given can usually be ported directly to other engines, like the regular expression engines included in grep, awk etc.

One thing to always keep in mind when working with regular expressions, is that some regex engines (like the one in sed) support both regular and extended regular expression syntax. For example, sed will allow you to use the `-E` option (shorthand option for `--regexp-extended`), enabling you to use extended regular expressions in the sed script.

Practically, this results in small differences in regular expression syntax idioms when writing regular expression scripts. Let's look at an example:

```
$ echo 'sample' | sed 's|[a-e]\+|_|g'
s_mpl_
$ echo 'sample' | sed 's|[a-e]+|_|g'
sample
$ echo 'sample+' | sed 's|[a-e]+|_|g'
sampl_
$ echo 'sample' | sed -E 's|[a-e]+|_|g'
s_mpl_
```

As you can see, in our first example we used `\+` to qualify the a-c range (replaced globally due to the `g` qualifier) as requiring **one or more occurrences**. Note that the syntax, specifically, is `\+`. However, when we changed this `\+` to `+`, the command yielded a completely different output. This is because the `+` is not interpreted as a standard plus character, and not as a regex command.

This was subsequently proved by the third command in which a literal `+`, as well as the `e` before it, was captured by the regular expression `[a-e]+`, and transformed into `_`.

Looking back that the first command, we can now see how the `\+` was interpreted as a non-literal regular expression `+`, to be processed by sed.

Finally, in the last command we tell sed that we specifically want to use extended syntax by using the `-E` extended syntax option to sed. Note that the term **extended** gives us a clue as to what happens in the background; the regular expression syntax is **expanded** to enable various regex commands, like in this case `+`.

Once the `-E` is used, even though we still use `+` and not `\+`, sed correctly interprets the `+` as being a regular expression instruction.

When you write a lot of regular expressions, these minor differences in expressing your thoughts into regular expressions fade into the background, and you will tend to remember the most important ones.

This also highlights the need to always test regular expressions extensively, given a variety of possible inputs, even ones that you do not expect.

Example 2: Heavy duty string modification

For this example, and the subsequent ones, we've prepared a textual file. If you want to practice along, you can use the following commands to create this file for yourself:

```
$ echo 'abcdefghijklmnopqrstuvwxyz ABCDEFG 0123456789' > test1
$ cat test1
abcdefghijklmnopqrstuvwxyz ABCDEFG 0123456789
```

Let's now look at our first example of string modifications: we would like the second column (`ABCDEFG`) to come before the first one (`abcdefghijklmnopqrstuvwxyz`).

As a start, we make this fictional attempt:

```
$ cat test1
abcdefghijklmnopqrstuvwxyz ABCDEFG 0123456789
```

```
$ cat test1 | sed -E 's|([a-o]+).*([A-Z]+)|\2 \1|'
G abcdefghijklmno 0123456789
```

Do you understand this regular expression? If so, you are a very advanced regular expression writer already, and you may choose to skip ahead to the following examples, skimming over them to see if you are able to quickly understand them, or need a bit of help.

What we are doing here is to `cat` (display) our test1 file, and parse it with an extended regular expression (thanks to the `-E` option) using sed. We could have written this regular expression using a non-extended regular expression (in sed) as follows;

```
$ cat test1 | sed 's|\([a-o]\+\).*\([A-Z]\+\)|\2 \1|'
G abcdefghijklmno 0123456789
```

Which is exactly the same, except we added a `\` character before each `(`, `)` and `+` character, indicating to sed we want them to be parsed as regular expression code, and not as normal characters. Let's now have a look at the regular expression itself.

Let us use the extended regular expression format for this, as it easier to parse visually.

```
s|([a-o]+).*([A-Z]+)|\2 \1|
```

Here we are using the sed substitute command (`s` at the start of the command), followed by a search (first `|...|` part) and replace (second `|...|` part) section.

In the search section, we have two **selection groups**, each surrounded and limited by `(` and `)`, namely `([a-o]+)` and `([A-Z]+)`. These selection groups, in the order they are given, will be looked for while searching the strings. Note that in between the selection group, we have a `.*` regular expression, which basically means **any character, 0 or more times**. This will match our space in between `abcdefghijklmnopqrstuvwxyz` and `ABCDEFG` in the input file, and potentially more.

In our first search group, we look for at least one occurrence of `a-o` followed by any other number of occurrences of `a-o`, indicated by the `+`

qualifier. In the second search group, we look for uppercase letters between `A` and `Z`, and this again one or more times in sequence.

Finally, in our replace section of the `sed` regular expression command, we will **call back/recall** the text selected by these search groups, and insert them as replacement strings. Note that the order is being reversed; first output the text matched by the second selection group (through the use of `\2` indicating the second selection group), then the text matched by the first selection group (`\1`).

While this may sound easy, the result at hand (`G abcdefghijklmno 0123456789`) may not be immediately clear. How did we loose `ABCDEF` for example? We also lost `pqrstuvwxyz` – did you notice?

What happened is this; our first selection group captured the text `abcdefghijklmno`. Then, given the `.*` (**any character, 0 or more times**) all characters were matched – and this important; to the maximum extent – until we find the next applicable matching regular expression, if any. Then, finally, we matched any letter out of the `A-Z` range, and this one more times.

Are you starting to see why we lost `ABCDEF` and `pqrstuvwxyz`? While it is by no means self-evident, the `.*` kept matching characters until the *last* `A-Z` was matched, which would be `G` in the `ABCDEFG` string.

Even though we specified **one or more** (through the use of `+`) characters to be matched, this particular regular expression was correctly interpreted by sed from left to right, and sed only stopped with the matching any character (`.*`) when it could no longer fulfill the premise that there would be *at least one* uppercase `A-Z` character upcoming.

In total, `pqrstuvwxyz ABCDEF` was replaced by `.*` instead of just the space as one would read this regular expression in a more natural, but incorrect, reading. And, because we are not capturing whatever was selected by `.*`, this selection was simply dropped from the output.

Note also that any parts not matched by the search section are simply copied to the output: `sed` will only act on whatever the regular expression (or text match) finds.

Example 3: Selecting all that is not

The previous example also leads us to another interesting method, which you will likely use a fair bit if you write regular expressions regularly, and that is selecting text by means of matching **all that is not**. Sounds like a fun thing to say, but not clear what it means? Let's look at an example:

```
$ cat test1
abcdefghijklmnopqrstuvwxyz ABCDEFG 0123456789
$ cat test1 | sed -E 's|[^ ]*|_|'
_ ABCDEFG 0123456789
```

A simple regular expressions, but a very powerful one. Here, instead of using `.*` in some shape or fashion we have used `[^ ]*`. Instead of saying (by `.*`) **match any character, 0 or more times**, we now state **match any non-space character, 0 or more times**.

Whilst this looks relatively easy, you will soon realize the power of writing regular expressions in this manner. Think back for example about our last example, in which we suddenly has a large part of the text matched in a somewhat unexpected manner. This could be avoided by slightly changing our regular expression from the previous example, as follows:

```
$ cat test1 | sed -E 's|([a-o]+)[^A]+([A-Z]+)|\2 \1|'
ABCDEFG abcdefghijklmno 0123456789
```

Not perfect yet, but better already; at least we were able to preserve `ABCDEF` part. All we did was change `.*` to `[^A]+`. In other words, keep looking for characters, at least one, except for `A`. Once `A` is found that part of the regular expression parsing stops. `A` itself will also not be included in the match.

Example 4: Going back to our original requirement

Can we do better and indeed swap the first and second columns correctly?

Yes, but not by keeping the regular expression as-is. After all, it is doing what we requested it to do; match all characters from `a-o` using the first search group (and output later at the end of the string), and then **discard** any character until sed reaches `A`. We could make a final resolution of the issue – remember we wanted only the space to be matched – by extending/changing the `a-o` to `a-z`, or by simply adding another search group, and matching the space literally:

```
$ cat test1 | sed -E 's|([a-o]+)([^ ]+)[ ]([A-Z]+)|\3 \1\2|'
ABCDEFG abcdefghijklmnopqrstuvwxyz 0123456789
```

Great! But the regular expression looks too complex now. We matched `a-o` one or more times in the first group, then any non-space character (until sed finds a space or the end of the string) in the second group, then a literal space and finally `A-Z` one or more times.

Can we simplify it? Yes. And this should highlight how one can easily over-complicate regular expression scripts.

```
$ cat test1 | sed -E 's|([^ ]+) ([^ ]+)|\2 \1|'
ABCDEFG abcdefghijklmnopqrstuvwxyz 0123456789
$ cat test1 | awk '{print $2" "$1" "$3}'
ABCDEFG abcdefghijklmnopqrstuvwxyz 0123456789
```

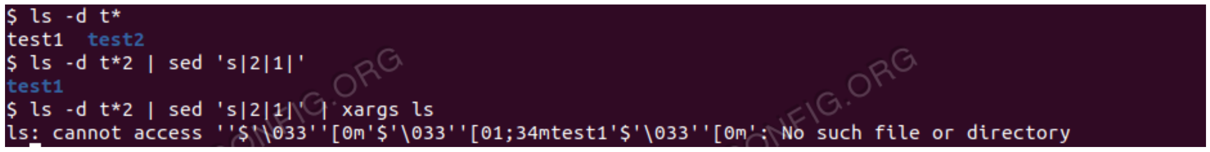
Both solutions achieve the original requirement, using different tools, a much simplified regex for the sed command, and without bugs, at least for the provided input strings. Can this easily go wrong?

```
$ cat test1
abcdefghijklmnopqrstuvwxyz ABCDEFG 0123456789
$ cat test1 | sed -E 's|([^ ]+) ([^ ]+)|\2 \1|'
abcdefghijklmnopqrstuvwxyz 0123456789 ABCDEFG
```

Yes. All we did was add an additional space in the input, and using the same regular expression our output is now completely incorrect; the second and third columns were swapped instead of the fist two. Again the need to test regular expressions in-depth and with varied inputs is highlighted. The difference in output is simply because the no-space space no-space pattern could only be matched by the latter part of the input string due to the double space.

Example 5: ls gotcha?

Sometimes, an operating system level setting, like for example using color output for directory listings or not (which may be set by default!), will cause command line scripts to behave erratically. Whilst not a direct fault of regular expressions by any means, it is a gotcha which one can run into more easily when using regular expressions. Let’s look at an example:



```
$ ls -d t*
test1  test2
$ ls -d t*2 | sed 's|2|1|'
test1
$ ls -d t*2 | sed 's|2|1|' | xargs ls
ls: cannot access ''$'\033''[0m'$'\033''[01;34mtest1'$'\033''[0m': No such file or directory
```

ls color output taints the result of a command containing regular expressions

```
$ ls -d t*
test1 test2
$ ls -d t*2 | sed 's|2|1|'
test1
$ ls -d t*2 | sed 's|2|1|' | xargs ls
ls: cannot access ''
```

In this example, we have a directory (test2) and a file (test1), both being listed by the original `ls -d` command. Then we search for all files with a file name pattern of `t*2`, and remove the 2 from the filename using `sed`. The result is the text `test`. It looks like we can use this output `test` immediately for another command, and we sent it via `xargs` to the `ls` command, expecting the `ls` command to list the file `test1`.

However, this does not happen, and instead we get a very complex-to-humanly-parse output back. The reason is simple: the original directory was listed in a dark blue color, and this color, is defined as a series of color codes. When you see this for the first time, the output is hard to understand. The solution however is simple;

```
$ ls -d --color=never t*2 | sed 's|2|1|' | xargs ls
test1
```

We made the `ls` command output the listing without using any color. This completely fixes the issue at hand, and shows us how we can keep in the back of our minds the need to avoid small, but significant, OS specific settings & gotchas, which may break our regular expression work when executed in different environments, on different hardware, or on different operating systems.

Ready to explore further on your own? Let’s look at some of the more common regular expressions available in Bash:

Expression	Description

Expression	Description
`.`	Any character, except newline
`[a-c]`	One character of the selected range, in this case a,b,c
`[A-Z]`	One character of the selected range, in this case A-Z
`[0-9AF-Z]`	One character of the selected range, in this case 0-9, A, and F-Z
`[^A-Za-z]`	One character outside of the selected range, in this case for example '1' would qualify
`\* or *`	Any number of matches (0 or more). Use * when using regular expressions where extended expressions are not enabled (see the first example above)
`\+ or +`	1 or more matches. Idem comment as *
`\(\)`	Capture group. The first time this is used, the group number is 1, etc.
`^`	Start of string
`$`	End of string
`\d`	One digit
`\D`	One non-digit
`\s`	One white space
`\S`	One non-white space
`a|d`	One character out of the two (an alternative to using []), 'a' or 'd'
`\`	Escapes special characters, or indicates we want to use a regular expression where extended expressions are not enabled (see the first example above)
`\b`	Backspace character
`\n`	Newline character
`\r`	Carriage return character
`\t`	Tab character

Conclusion

In this tutorial, we looked in-depth at Bash regular expressions. We discovered the need to test our regular expressions at length, with varied inputs. We also saw how small OS differences, like using color for `ls` commands or not, may lead to very unexpected outcomes. We learned the need to avoid too-generic regular expression search patters, and how to use extended regular expressions.

Enjoy writing advanced regular expressions, and leave us a comment below with your coolest examples!

Related Linux Tutorials:

- Bash regexps for beginners with examples
- Python Regular Expressions with Examples
- How to work with the Woocommerce REST API with Python
- Big Data Manipulation for Fun and Profit Part 3
- Big Data Manipulation for Fun and Profit Part 1
- How to install VMware Workstation on Ubuntu 20.04...
- Things to install on Ubuntu 20.04
- How to perform HTTP requests with python - Part 1:...
- Compare string in BASH
- Introduction to Ebay API with Python: The Finding...
