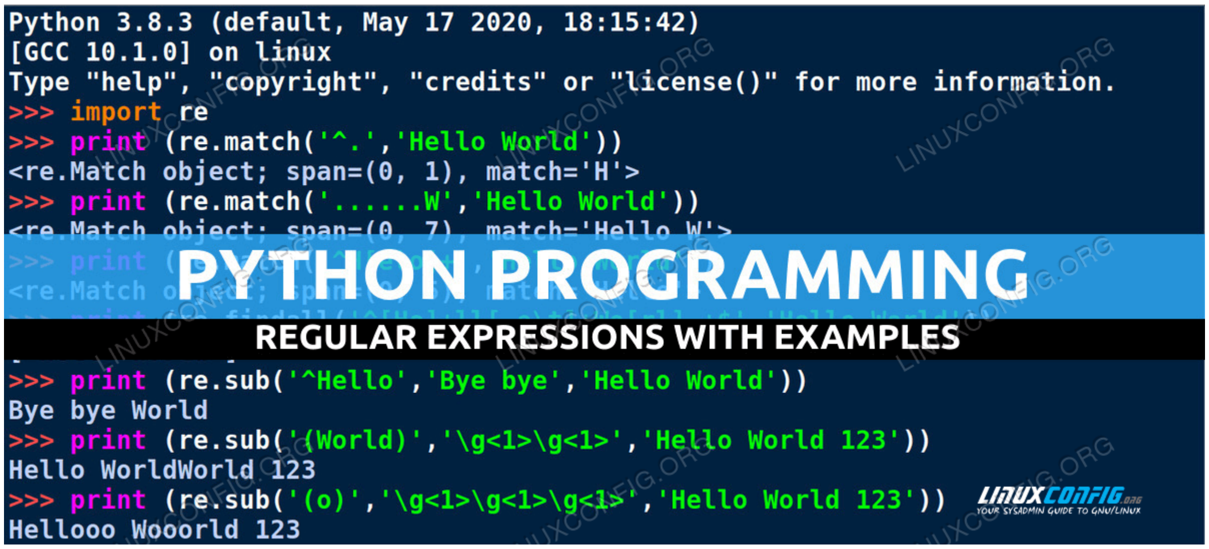
16 July 2020 | Roel Van de Paar

Python Regular Expressions with Examples

A regular expression (often abbreviated to “regex”) is a technique, and a textual pattern, which defines how one wants to search or modify a given string. Regular expressions are commonly used in Bash shell scripts and in Python code, as well as in various other programming languages.

In this tutorial you will learn:

- How to start with Regular Expressions on Python
- How to import regex Python module
- How to match strings and characters using Regex notation
- How to use the most common Python Regex notations


PYTHON PROGRAMMING
REGULAR EXPRESSIONS WITH EXAMPLES
LINUXCONFIG.ORG
YOUR SYSADMIN GUIDE TO GNU/LINUX

```
Python 3.8.3 (default, May 17 2020, 18:15:42)
[GCC 10.1.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> import re
>>> print (re.match('^.','Hello World'))
<re.Match object; span=(0, 1), match='H'>
>>> print (re.match('......W','Hello World'))
<re.Match object; span=(0, 7), match='Hello W'>
>>> print (re.sub('^Hello','Bye bye','Hello World'))
Bye bye World
>>> print (re.sub('(World)','\g<1>\g<1>','Hello World 123'))
Hello WorldWorld 123
>>> print (re.sub('(o)','\g<1>\g<1>\g<1>','Hello World 123'))
Hellooo Wooorld 123
```

Python Regular Expressions with Examples

Software Requirements and Conventions Used

Software Requirements and Linux Command Line Conventions

Category	Requirements, Conventions or Software Version Used

Category	Requirements, Conventions or Software Version Used
System	Any GNU/Linux operating system
Software	Python 2 , Python 3
Other	Privileged access to your Linux system as root or via the `sudo` command.
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

Python Regular Expressions Examples

In Python, one wants to import the `re` module to enable the use of regular expressions.

Example 1 Let’s start with a simple example:

```
$ python3
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more infor
>>> print ('Hello World')
Hello World
>>> import re
>>> print (re.match('^.','Hello World'))
```

Copy

Here we first printed `Hello World` Line 5to demonstrate a simple print setup. We then imported the regex module `re` Line 7enabling us to use the `.match` regular expression Line 8matching function of that library.

The syntax of the `.match` function is (pattern,string) where pattern was defined as the regular expression `^.`’ and we used the same `Hello World` string as our input string.

As you can see, a match was found in the letter `H`. The reason this match was found is the pattern of the regular expression, namely; `^` stands for **Start of string** and `.` stands for **match any one character (except newline)**.

Thus, `H` was found, as that letter is directly after "the start of the string", and is described as "any one character, `H` in this case".

DID YOU KNOW?

These special connotations are identical to regular expressions in Bash scripting, and other regex-aware applications, which all use a more-or-less uniform regex standard, though there are differences between languages and even specific implementations if you delve into regular expressions a bit further.

Example 2

```
>>> print (re.match('......W','Hello World'))
<re.Match object; span=(0, 7), match='Hello W'>
```

Copy

Here we use `.` to match any one character (except newline) and we do this 6 times before matching the literal character `W`.

As you can see `Hello W` (7 characters) was matched. Interestingly, this show as span (0,7) which should not be read as 0-7 (which is 8 characters) but as “start at 0” “+7 characters”, as can also be glanced from the other examples in this article.

Example 3 Let’s take another, slightly more complex example:

```
>>> print (re.match('^H[elo]+','Hello World'))
<re.Match object; span=(0, 5), match='Hello'>
```

Copy

The syntax in this case is:

- **^**: as described above, can also be read as ‘this must be the start of the string’
- **H**: must match `H` in this exact location (which is directly after/on the start of the string)
- **[elo]+**: match either `e`,`l` or `o` (the ‘either’ defined by `[‘ and ‘]`) and `+` means ‘one or more of these’

Thus, `Hello` was matched as `H` was indeed at the start of the string, and `e` and `o` and `l` were matched one or more times (in any order).

Example 3Ready for a super complex one?

```
>>> print (re.findall('^[He]+ll[ o\t]+Wo[rl].+$','Hello World')
['Hello World'];
```

Copy

Here we used another function of the re module, namely `findall` which immediately yields the found string and uses the same (pattern,string) syntax.

Why did `Hello World` match in full? Let’s break it down step-by-step:

- **^**: Start of string
- **[He]+**: Matches `H` and `e` 1 or more times, and thus `He` is matched

ll: literal matching of `ll` in this exact spot, and thus indeed `ll` is matched as it came directly after `He`

[o\t]+: Match either `‘ ‘` (space), or `o`, or `\t` (a tab), and that 1 or more times, and thus `o ` (o space) matched. If we had used a tab instead of a space, this regex would still work!

Wo: Literal match of `Wo`

[rl]: match either `r` or `l`. Watch carefully; only `r` is matched here! There is no `+` behind the `]` so only a single character, either `r` or `l` will be matched in this position. So why was `rld` still matched? The answer is in the next qualifier;

.+: match any character (signified by `.`) one or more times, thus `l` and `d` are both matched, and our string is complete

$: Similar to `^`, this character signifies “end of string”.

In other words, had we placed this at the start, or somewhere else in the middle, the regex would have mismatched.

As an example:

```
>>> print (re.findall('^Hello$','Hello World'))
[]

>>> print (re.findall('^Hello$','Hello '))
[]

>>> print (re.findall('^Hello$','Hello'))
['Hello']

>>> print (re.findall('^Hello','Hello World'))
['Hello']
```

Copy

Here no output is returned for the first two print’s, as we are trying to match a string which can be read as “start_of_string”-`Hello`-“end_of_string” as signified by `^Hello$`, against `Hello World` which does not match.

In the third example, the `^Hello$` matches `Hello` as there are no additional characters in the `Hello` string which would cause this regex to fail matching. Finally, the last example shows a partial match without the requirement for the “end_of_string” ($) to happen.

See? You’re already becoming a regular expressions expert! Regular expressions can be fun, and are very powerful!

Example 4
There are various other functions in the `re` Python module, like **re.sub**, **re.split**, **re.subn**, **re.search**, each with their applicable use case domains. Let’s look at re.sub next:

```
>>> print (re.sub('^Hello','Bye bye','Hello World'))
Bye bye World
```

Copy

String substitution is one of the most powerful applications of regular expressions, in Python and other coding languages. In this example, we looked for `^Hello` and replaced it with `Bye bye` in the string `Hello World`. Can you see how this would be very handy to process all sorts of variables and text strings and even entire flat text files?

Example 5
Let’s look at a few more complex examples, using more advanced regex

syntax:

```
>>> print (re.sub('[0-9]+','_','Hello World 123'))
Hello World _
```

Copy

> **[0-9]+**: Any numeric character from `0` to `9`, one or more times.

Can you see how the `123` was replaced by a single `_` ?

Example 6

```
>>> print (re.sub('(?i)[O-R]+','_','Hello World 123'))
Hell_ W_ld 123
```

Copy

> **(?i)[O-R]+**: Match one or more `O` to `R` or – thanks to the optional `i` flag – `o` to `r`
> **(?i)**: preset a case-insensitive `i` flag for this pattern

```
>>> print (re.sub('[1]{2}','_','Hello World 111'))
Hello World _1
```

Copy

> **[1]{2}**: Match the character `1` exactly two times

Example 7

```
>>> print (re.sub('(World)','\g<1>\g<1>','Hello World 123'))
Hello WorldWorld 123
```

Copy

> **(World)**: Match the literal text 'World' and make it a group which can then be used in the substitution

\g<1>\g<1>: The `\g<1>` specifies the first group which was matched, i.e. the text `World` taken from the `Hello World 123` string, and this is repeated twice, resulting in the `WorldWorld` output. /li>

Example 8

To make this clearer, consider the following two examples:

```
>>> print (re.sub('(o)','\g<1>\g<1>\g<1>','Hello World 123'))
Hellooo Wooorld 123
```

Copy

In this first example, we simply match `o` and place it in a group, then repeat that group three times in the out.

Note that if we would not refer to group 1 (the first matched group, ref second example), then there simply would be no output and the result would be:

```
>>> print (re.sub('(o)','','Hello World 123'))
Hell Wrld 123
```

Copy

For the second example, consider:

```
>>> print (re.sub('(o).*(r)','\g<1>\g<2>','hello world 123'))
hellorld 123
```

Copy

Here we have two groups, the first one being `o` (wherever such a group matches, and there are clearly multiple as seen in the first example), and the second one being `r`. Additionally, we use `.*` which translates to "any character, any number of times" – an often used regular expression.

So in this example `o wor` is matched by `(o).*(r)' (‘o` first, then any character until the last `r` is reached. “The last” notion is very import and an easy to make mistake/gotcha, especially for new regular expressions users. As a side example, consider:

```
>>> print (re.sub('e.*o','_','hello world 123'))
h_rld 123
```

Copy

Can you see how the last `o` was matched?

Returning to our example:

```
>>> print (re.sub('(o).*(r)','\g<1>\g<2>','hello world 123'))
hellorld 123
```

Copy

We can see that `o wor` was replaced by a match of group 1 followed by a match of group 2, resulting in: `o wor` being replaced by `or` and thus the output is `hellorld 123`.

Conclusion

Let’s look at some of the more common regular expressions notations available in Python, matched with some lightweight implementations of the same:

List of the most common Python Regular Expression notations

Regex Notation	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
`*`	Any number of matches (0 or more)
`+`	1 or more matches
`?`	0 or 1 match
`{3}`	Exactly 3 matches
`()`	Capture group. The first time this is used, the group number is 1, etc.
`\g<1>`	Use (insert) of the capture match group, qualified by the number (1-x) of the group
`\g<0>`	Special group 0 inserts the entire matched string
`^`	Start of string
`$`	End of string
`\d`	One digit
`\D`	One non-digit
`\s`	One whitespace
`\S`	One non-whitespace
`(?i)`	Ignore case flag prefix, as demonstrated above
`a|d`	One character out of the two (an alternative to using []), ‘a’ or ‘d’
`\`	Escapes special characters
`\b`	Backspace character
`\n`	Newline character
`\r`	Carriage return character
`\t`	Tab character

Interesting? Once you start using regular expressions, in any language, you will soon find that you start using them everywhere – in other coding languages, in your favorite regex-aware text editor, on the command line (see 'sed' for Linux users), etc.

You will likely also find that you'll start using them more ad-hoc, i.e. not just in coding. There is something inherently powerful in being able to control all sorts of command line output, for example directory and file listings, scripting and flat file text management.

Enjoy your learning progress and please post some of your most powerful regular expression examples below!

Related Linux Tutorials:

- Advanced Bash regex with examples
- Introduction to Bash Shell Parameter Expansions
- Bash regexps for beginners with examples
- How to parse a json file from Linux command line using jq
- How to Debug Bash Scripts
- Compare string in BASH
- Things to install on Ubuntu 20.04
- How to find a string or text in a file on Linux
- How to setup the rsync daemon on Linux
- Big Data Manipulation for Fun and Profit Part 1