

Theory: Regexp sets and ranges

🕒 22 minutes 0 / 5 problems solved

Skip this topic

Start practicing

351 users solved this topic. Latest completion was about 6 hours ago.

In the previous topics, we have learned about the dot and the question mark in regexp language and the ways of escaping them, and other regexp metacharacters. Now it is time to learn about these metacharacters and the specifics of their functioning. First of all, let’s start with the sets.

§1. Sets

While the dot allows us to match almost every possible character, the sets provide us with the opportunity to be more specific in our regexp templates and narrow down the scope of our search. Each set in the regular expression takes the place of exactly one character in the string, but it defines a whole number of characters that can match it. These characters are listed inside the square brackets, `[]`:

```
1 | template = '[bd]a[td]'
```

In the template above, we have two defined sets. The first one corresponds either to a character `b` or `d` in the string, the second one — to `t` or `d`. Here are the results for some of the possible strings:

```
1 | re.match(template, 'bat') # match
2 | re.match(template, 'dad') # match
3 | re.match(template, 'cat') # no match: 'c' is not in the first set
4 | re.match(template, 'dot') # no match: 'o' instead of 'a'
```

An empty set causes an error:

```
1 |
re.match('c[]at', 'cat') # sre_constants.error: unexpected end of regular expression
```

An unescaped left square bracket, for which no unescaped right square bracket was found, causes the same error:

```
1 |
re.match('[', '[') # sre_constants.error: unexpected end of regular expression
```

By the way, good news, everyone! There is (almost) no need for boring escaping stuff when we use sets in regexp.

§2. Escaping in sets

Sets in regular expressions have a sort of superpower: they automatically “neutralize” the metacharacters listed inside the square brackets, turning them into regular characters. This way, the dot and the question mark, for example, do not have to be escaped if they are part of a regexp set:

```
1 | template = 'Hodor[?.]'
```

```
2 | re.match(template, 'Hodor?') # match
3 | re.match(template, 'Hodor.') # match
4 | re.match(template, 'Hodor!') # no match
```

The only metacharacters that do not fall under this rule and keep their special status are, predictably, the right square bracket `]` and the backslash `\`. The right square bracket should be escaped to show that it is a part of the set, not the metacharacter denoting its end:

Current topic:

[Regexp sets and ranges](#) ...

Topic depends on:

✗ [Escaping in regexps](#) ...

Topic is required for:

[Shorthands](#) ...

Table of contents:

[1 Regexp sets and ranges](#)

[§1. Sets](#)

[§2. Escaping in sets](#)

[§3. Ranges](#)

[§4. Exclusion of characters](#)

[§5. Conclusions](#)

[Feedback & Comments](#)

```

1 | template = r'=[\]]'
2 | re.match(template, ']=]') # match
3 |
4 | template = r'=[\]]]'
5 | re.match(template, ']=]') # no match
6 |
re.match(template, ']=])']') # match (the only string this template can match)

```

The backslash in sets, like everywhere else, serves as the starting symbol of escape sequences. So, if you just want to have a backslash in your set, you have to relieve it from this burden by escaping it using double backslash. Here the backslash is escaped and matches itself in the string:

```

1 | template = r'^[\\]_'
2 | re.match(template, r'^\\_(ツ)_/^-') # match
3 |
# remember that re.match checks whether regexp matches the beginning of the string
, not the whole string

```

Here the backslash is not escaped and serves as a part of the escape sequence:

```

1 | template = r'^[\\t]_'
2 | re.match(template, r'^\\_(ツ)_/^-') # no match
3 | re.match(template, r'^\\t_') # match

```

By the way, you can still escape any character in the set (even if it is not `]` or `\`): this won't change the set of matching characters. But additional escaping characters will make your regular expression more difficult to read and understand, and, believe us, this is a thing to avoid — real-life regexps are usually barely comprehensible even without unnecessary characters.

Apart from "ordinary" regexp metacharacter, there are, though, some characters that acquire a special meaning specifically when they're used inside the square brackets.

§3. Ranges

One of the main things about sets is that you may not only list the characters individually but also use ranges of characters. A range is designated by a dash `-`. For example, if you want your set to match every letter from `a` to `z`, you do not have to list out the whole alphabet, you can simply write `[a-z]`.

```

1 | re.match('ja[a-z].', 'jazz') # match
2 | re.match('[A-Z]ill', 'kill') # no match: [A-
Z] matches only uppercase letters
3 | re.match('[A-Z]ill', 'Bill') # match

```

`[0-9]` does it for the digits. Note that regular expressions do not know the count, and they match characters, not numbers. So, the template `[1-100]` matches only `1` and `0`, not all numbers in the range from 1 to 100.

```

1 | re.match('[0-9]', '7') # match
2 | re.match('[1-9]', '07') # no match

```

Several ranges can be easily put in one set. They do not have to follow each other in any way:

```

1 | re.match('love [a-zA-Z]', 'love U') # match: [a-zA-
Z] matches both uppercase and lowercase
2 | re.match('love [a-zA-Z!]', 'love !') # match: [a-zA-
Z!] matches letters and !

```

The characters that fall within the range are determined by [ASCII / Unicode encoding](#) table, so be careful when defining ranges: they may include something unexpected or exclude something that was meant to be in your set.

```
1 | re.match('[A-Z]bermensch', 'Übermensch') # no match: Ü is not within A-
Z range
2 | re.match('[À-Ý]bermensch', 'Übermensch') # match
3 | re.match('[À-Ý]bermensch', 'xbermensch') # match: x is within À-Ý range
```

To use the dash `-` as a regular character in a set, you should "strip" it of the left or right character defining the range, so just put the dash in the first or last position in the set, `[-abc]` or `[abc-]`:

```
1 | re.match('[-1-9]1', '-1') # match
2 | re.match('[1-9-]1', '-1') # match
```

Take a look at the table summarizing some of the ranges you might want to use in your programs:

[a-z]	Lowercase Latin Letters
[A-Z]	Capital Latin Letters
[a-zA-Z]	Both Lowercase and Capital Latin Letters
[0-9]	Digits

Sets can also be handy in case you want to ban characters from your template. Let's see how this is done!

§4. Exclusion of characters

The hat (also, *the caret*) `^` symbol is also a specific set metacharacter: whenever it is placed as the first character in the set, it makes the set specify the characters you *do not* want to see in the string. Any character that is not a part of such set will match it:

```
1 | re.match('[^A-Z]ond', 'Bond') # no match
2 | re.match('Bon[^A-Z]', 'Bond') # match
```

The hat placed anywhere else in the set, except for the first position, will lose its special meaning and become a regular character:

```
1 | re.match('[A-Z^]ames', 'James') # match
2 | re.match('[A-Z^]ames', '^ames') # match
```

That is pretty much it about sets. Four metacharacters associated with them are easy to remember, but can turn you into a real regexp wizard!


Just in case you forgot: there are a lot of websites where you can test your regular expression, see what could be wrong with it, and correct it. For example, [Regex 101](#) works just fine.

§5. Conclusions

Let's see what we have learned in this topic!

- the square brackets `[]` are used to designate sets in regular expressions;
- the sets are used to specify the number of characters to match;
- only the backslash `\` and the closing bracket `]` should be escaped in your sets;
- the dash `-` allows us to easily put a range of characters in the set;
- a set with the hat `^` in the first position matches every character that is not listed in it.

 Report a typo

40 users liked this theory.  didn't like it. What about you?



Start practicing

[Comments \(1\)](#)

[Hints \(0\)](#)

[Useful links \(0\)](#)

[Show discussion](#)