Theory: Escaping in regexps

21 minutes 0 / 5 problems solved

You already know that in the language of regular expressions some characters, like the dot and the question mark, have special meaning. This way, the dot will match not only itself, but also almost every other possible character, while the question mark won't match itself at all. But what if we want to use such characters in their literal meaning? What if we want the dot to match nothing but the dot? Well, in this case, escaping comes to our rescue.

§1. Escaping through backslashes

To use a special character in its literal meaning in your regexp, you need to put a **backslash** `\` before it: `\?`, `\.`. The backslash `\` is a so-called **escape character**, it helps symbols to "escape their work duties". However, the backslash is used as an escape character not only in regular expressions, but also in Python itself (it is the first character in such escape sequences as `\t` and `\n`). So, it's possible, though not necessary, to escape the first backslash with another backslash `\` . For example, `\\?` and `\?` in the regular expression both correspond to `?` in a string, while `\\.` and `\.` correspond to a single dot `.` in a string. Take a look at how it works in practice:

```
re.match("\?", "?")  # match, '?' is the matching string
re.match("\\?", "?")  # match, '?' is the matching string
re.match("\\.", ".")  # match, '.' is the matching string
re.match("?", "?")  # SyntaxError
# SyntaxError is caused by a "dangling metacharacter" in the regexp,
# an unescaped question mark not preceded by any character
```

The first backslash to the left of the escaped symbol tells regular expression to treat `?` or `.` as characters without special meaning, and the second backslash (if present) tells Python to not treat the first backslash as a start of some Python's escape sequence. Here's a less formal example showing the application of the escape symbol in regexps:

```
question = "who let the dogs out?!"
re.match("who let the dogs out?!", question)  # no match
re.match("who let the dogs out\?!", question)  # match
re.match("woof\.", "woof!")  # no match
re.match("woof\.", "woof.")  # match
```

So far the escaping in regexps seems reasonable, right? But wait, there's more.

§2. Backslash plague

Things get worse when you want your regular expression to match a *literal* backslash. Since a backslash is a metacharacter both in the regexp language and in Python you have to escape it with three other backslashes, which will result in a cumbersome regexp: `\\\\`. The last backslash here is the backslash you want to match; the second backslash from the left serves as an escape character for the regexp language; the first backslash serves to "escape" the second one in Python syntax; the third one serves to escape the last one in Python syntax. Check out these examples:

```
re.match("\\\\", "\\")  # match
# Python requires backslash to be escaped in the string as well
# so the string consists of one literal backslash and one escape symbol
re.match("\\", "\\")  # SyntaxError
```

SyntaxError in the second example is raised because the regexp template consists of one Python's escape symbol and one backslash. So, this backslash is left unescaped in the regexp, and regexp interprets it as an

Current topic:

Escaping in regexps

Topic depends on:

✓ Escape sequences Stage 1 8

✗ Regexps in Python

Topic is required for:

Regexp sets and ranges

Table of contents:

escape character, but there's nothing to escape, because no character is following this backslash.

By the way, there are no problems with matching default escape sequences whose notations coincide in Python and in regular expressions language:

```
re.match("\t", "\t")  # match
re.match("\\t", "\t")  # match
```

Using a bunch of backslashes probably doesn't seem convenient to you, and you are completely right. Fortunately, there's a way to deal with them.

§3. r prefix

To partly avoid the mess created by backslashes, you can always use the `r` prefix in your strings, for example:

```
regexp = r"\?"
```

The `r` prefix is a **raw string notation prefix**: it tells Python to cancel the usage of escape sequences in this string and treat all backslashes in their literal meaning. For example, the string `r'\t'` will be treated by Python as a combination of a backslash and a letter `t`, not a tabulation. This way, you'll only need to use backslash as an escape character for regular expressions. Here's an example:

```
re.match(r"\\", "\\")  # match: regexp consists of a regexp escape and a backslash
re.match(r"\\.", ".")  # no match: no backslash in the string

re.match(r"\\?", "?")  # match is an empty string: the question mark in regexp is unescaped

re.match(r"\?", "?")  # match, as in the example above, \ is the regexp escape character
re.match(r"\t", "\t")  # match, \t is the regexp escape sequence
```

You can't use `r` prefix with already existing strings: that is, you can't create a raw string from a variable. `r` prefix is only used when you write the value of your string by hand.

There's also a quick way to escape all special characters in your regular expression: `re.escape` function. Let's take a closer look at it!

§4. re.escape

`re.escape` function can help you to escape all special characters in your string automatically, without placing backslashes in the regular expression by yourself. This function takes a string with your *regular expression* as an argument, and returns the same string, but with necessary backslashes placed before every dot, question mark and other regexp metacharacter. That's how it looks:

```
template = "hyperskill.org"
escaped_template = re.escape(template)
print(escaped_template)  # 'hyperskill\.org'
```

The string stored in the `escaped_template` variable, actually, contains *two* backslashes, i.e. it looks like `'hyperskill\\.org'`: the first one is the Python's escape symbol and the second one is a simple backslash added by `re.escape` to escape the dot in the regular expression. When the `escaped_template` is printed, however, only a single backslash is left since the Python's escape backslash is no longer needed: it has already fulfilled its duty by showing that the following backslash is not a Python's metacharacter.

By the way, whitespace is a metacharacter too. It's not like it has any special meaning or function, but backslashes are always placed before it.

```
print(re.escape('whitespace is here'))
# The output is 'whitespace\ is\ here'
```

`re.escape` is useful when you want to match an entire string in its literal meaning. This doesn't happen often, though, since the power of regular expressions language lies exactly in its special symbols. So be careful: if there are any special symbols that you want to perform their duties as usual, don't use `re.escape`.

> Note that `re.escape` escapes nothing but regexp's special characters only if your Python is updated to the version 3.7 or higher. If you're using earlier versions of Python, `re.escape` puts backslashes before every non-alphanumerical character, including, for example, `!`, `<`, `>`, `@`, etc., even though they only have their literal meaning in regexp language. However, these redundant backslashes don't change the result of the `match` function, since a single backslash in a regular expression never matches itself: it simply "cancels" the special meaning of the following character; if there are no special meaning to cancel, the backslash basically doesn't do anything at all.

§5. Conclusions

In this topic we've learned that:

1. To use regexp's special characters like the dot and the question mark in their literal meaning, we need to escape them with one or two backslashes.
2. Backslashes are also used by Python's own syntax, so we *can* use single or double backslashes in regular expression to escape most metacharacters, but we *need* to use double backslashes to escape backslash itself.
3. To cancel backslash's escape function in Python, we can use `r` prefix before the string. This helps to avoid the need to use multiple backslashes for escaping.
4. To backslash each special character in the template automatically, we can use `re.escape` function.

Report a typo

41 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Comments (1) Hints (0) Useful links (0) Show discussion