Theory: Regexp quantifiers

32 minutes 0 / 5 problems solved

We have already learned about sets and shorthands in regular expressions. Now it is time to learn another powerful feature that significantly contributes to the regexp flexibility. The metacharacters we are going to talk about are called **quantifiers**. Quantifiers always follow some other character (or a group of characters), and they are meant to specify the number of repetitions of this character in the string. So, quantifiers allow you to write not only fixed-length templates but also templates for strings of varying length. The indications for the required length of strings (the number of repetitions) can be both fixed or rough.

We already know one of the quantifiers, the question mark `?`, which matches either one or zero occurrences of the preceding character. Let's see what other quantifiers we can use!

§1. The plus quantifier

Let's start with, perhaps, the simplest and the most straightforward quantifier, the plus `+` quantifier. The plus `+` quantifier basically means "the preceding character should be written *one* or *more* times". That is, it matches one or more occurrences of the preceding character in the string. Look at the example below:

```
1  template = "wo+w!"  # matches "wow!" with one or more 'o'
2
re.match(template, "wow!")            # match: one 'o' character encountered
3
re.match(template, "wooooooooooow!")  # match: many (11) 'o' characters encountere
d
4
re.match(template, "ww!")             # no match: no 'o' character encountered
```

You can easily pair this quantifier with many other metacharacters. For example, with the dot character, to match this combination with any number of any characters except for the `\n` (but not the absence of any characters), or with a set of characters. So, the plus will match the sequence of the characters from this set (with one character from this set being the shortest possible sequence).

```
1
template = ".+Jack Sparrow"  # matches the string "Jack Sparrow" with some precedi
ng characters
2
re.match(template, "Captain Jack Sparrow")  # match: there are some characters bef
ore "Jack"
3
re.match(template, "Jack Sparrow")          # no match: the string starts with "Ja
ck"
4
5   template = "Louis [IXV]+"
6   re.match(template, "Louis III")  # match
7   re.match(template, "Louis XVI")  # match
8   re.match(template, "Louis ")     # no match
```

There is also a very similar quantifier with just a slightly different function: the asterisk `*` quantifier.

§2. The asterisk quantifier

The asterisk `*` quantifier does almost the same thing, but the scope is a bit wider; it also matches the *absence* of the previous character. So, zero or more occurrences of the character are enough to match the combination of this character with the asterisk quantifier.

Current topic:

Regexp quantifiers

Topic depends on:

✕ Shorthands

Topic is required for:

Tokenization

Table of contents:

```
template = "go*d"
re.match(template, "good")  # match: double 'o' occured
re.match(template, "god")   # match: one 'o' occured

re.match(template, "gd")    # match: no 'o' occured, but the rest of the string matches the template
re.match(template, "gud")   # no match: 'u' is not in the template
```

You can pair the asterisk with other metacharacters in the same way as the plus quantifier. For example, the combination of the dot character and the asterisk quantifier, `.*`, matches any string of any length, including an empty string.

```

template = ".*no.*"  # matches 'no' with any or no preceding/following sequences of characters

re.match(template, "no")                            # match: 'no' is the whole string

re.match(template, "no rest for the wicked")        # match: 'no' at the start of the string

re.match(template, "I'm no superman")               # match: 'no' inside the string

re.match(template, "— Luke, I'm your father. — no")  # match: 'no' at the end of the string
```

In addition to the asterisk and the plus, there is another type of quantifier. Unlike the ones that we have just spoken about, it allows us to indicate a specific number of repetitions that we want to find.

§3. A fixed number of repetitions

In case we want to match a fixed number of occurrences of a certain character, we can use curly brackets with a number inside them, just like that: `{n}`. This quantifier matches exactly *n* consecutive occurrences of the preceding character.

```

template = "\w{5}"  # matches a sequence of exactly 5 alphanumeric characters

re.match(template, "doggy")  # match: 5 letters sequence

re.match(template, "dog")    # no match: there're only 3 alphanumeric characters

re.match(template, "a dog")  # no match: space doesn't match \w
```

As well as the aforementioned quantifiers, curly brackets quantifier (in all its forms which we're going to discuss now) can follow other metacharacters, such as the dot character or a set, for example.

§4. A range of repetitions

The braces also serve for the designation of a specific *range* of the length of the sequence that we want to find. In its general form, this quantifier looks like two numbers put inside braces and separated by a comma: `{n,m}`. The quantifier matches *at least n* and *no more* than *m* instances of the previous (quantified) character.

> Unlike Python ranges that end exclusively (that is, not including the number specifying the end of the range), the ranges in regular expressions both start and end inclusively: `{n,m}` matches both *n* and *m* occurrences of the character.

Take a look at the example to see how the `{n,m}` quantifier works:

```
template = "\d{5,10}"  # matches any sequence of digits with length from 5 to 10
re.match(template, "12345")       # match: 5 digits
re.match(template, "1234567890")  # match: 10 digits
re.match(template, "12345678")    # match: 8 digits
re.match(template, "1234")        # no match: only 4 digits
```

This quantifier is quite flexible in the way that you can leave out either `n` or `m`, in this way mentioning only the maximum or the minimum number of occurrences; `{n,}` matches *n* and more occurrences of the character, while `{,m}` matches no more than *m* (including zero occurrences). Pay attention to the comma and its location relative to the number. If you forget the comma or put it in the wrong place, the meaning of the quantifier will be absolutely different.

```
template = "i'm just a po{2,}r boy"  # there should be at least 2 'o' in the strin
g
re.match(template, "i'm just a poor boy")         # match: 2 'o'
re.match(template, "i'm just a poooooooooor boy")  # match: 9 'o'
re.match(template, "i'm just a por boy")          # no match

template = "i need no sy{,3}mpathy"  # there should be no more than 3 'y'
re.match(template, "i need no sympathy")     # match: 1 'y'
re.match(template, "i need no syyympathy")   # match: 3 'y'

re.match(template, "i need no smpathy")      # match: zero occurrences match the q
uantifier too

re.match(template, "i need no syyyympathy")  # no match: 4 'y'
```

Since `{,m}` quantifier matches zero occurrences of the character as well, be careful with it. If you omit to mention any characters that should necessarily occur in the string in your template, it will also match an empty string. Take a look at the example:

```
template = "\d{,3}"  # matches no more than 3 digits (and zero digits too)
re.match(template, "123")        # match

re.match(template, "no digits")  # also match: the template matches an empty strin
g at the start
```

To avoid these situations, you may want to specify the minimum number of occurrences: `{1,m}`, for example.

> Another crucial point of quantifier syntax is that there should be *no spaces* inside the curly brackets. If you put a space after the comma, for example, the quantifier automatically turns from a metacharacter into a simple sequence of literal characters. That is, `a{2, 3}` is not going to match strings `"aa"` or `"aaa"`: it will only match a string `"a{2, 3}"`, because the quantifier in the template is not working, it is broken by the space character.

§5. Greedy and lazy quantifiers

By default, all quantifiers (`+`, `*`, `{n,m}`, `{n,}` or `{,m}`, `?`) are greedy. It means that they match as many instances of the previous character as possible. For example, when a template `"a+"` is compared against a string `"aaa"`, we have some kind of a dilemma — the regex engine can return any substring as the result of the match, `"a"`, `"aa"` or `"aaa"`, since they all comply with the template. They consist of at least one `a` character. But since our `+` quantifier is greedy, it has no choice as it matches the longest possible substring, `"aaa"`. The dilemma is resolved.

In some situations, this behavior may not be desired for your purposes. Suppose we want to find a pair of `<p>` and `</p>` tags with some text in between them. We write a template `<p>.*</p>` to find such substring in a text filled with tags. See what happens there:

```
template = "<p>.*</p>"
re.match(template, "<p>paragraph</p><p>another paragraph</p>")
# the template matches the whole string "<p>paragraph</p>
<p>another paragraph</p>"
```

Even though we wanted to find substrings `"<p>paragraph</p>"` and `"<p>another paragraph</p>"` *separately*, we were not able to do that, because our quantifier `.*` is greedy; it matched the longest possible string located between `<p>` and `</p>` tags.

If we want to change this behavior, we need to put the question mark character `?` right after the quantifier: `+?`, `*?`, `{n,m}?`. Yes, the question mark has another function as a metacharacter in regular expressions when it immediately follows a quantifier; it switches this quantifier from the "greedy" mode to "lazy".

A lazy quantifier matches as few occurrences of the quantified character as possible. For example, a template `"a+?"` compared to the string `"aaa"` matches `"a"`, and not `"aaa"`. The aforementioned example with tags ends in a very different way:

```
template = "<p>.*?</p>"
re.match(template, "<p>paragraph</p><p>another paragraph</p>")
# the template first matches the substring "<p>paragraph</p>"
```

Note that lazy and greedy modes do not work with the quantifier `{n}`, since it searches for a fixed number of occurrences, and cannot choose between the lesser and the bigger number of them.

Please, remember this feature and choose carefully between a greedy and a lazy quantifier according to your purposes in a particular situation.

> When the question mark `?` character is used to signify that the preceding character may be absent, it is equal to `{,1}` quantifier.

§6. Conclusion

In this topic, we have learned about quantifiers in regular expressions. There are several quantifiers we need to remember:

- the plus `+` quantifier matches one or more instances of the quantified character;
- the asterisk `*` quantifier matches zero or more instances of the quantified character;
- the fixed-length quantifier `{n}` matches exactly *n* instances;
- the quantifier `{n,m}` matches from *n* to *m* (inclusively) instances;
- we can leave out either `n` or `m`.

We have also learned about two modes of quantifiers (except for fixed-length quantifier):

- greedy mode (by default) when the quantifier matches as many occurrences of the character as possible;
- lazy mode (the quantifier is followed by `?`) when the quantifier matches as few occurrences as possible.

Report a typo

38 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

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