Theory: Regexps in Python

19 minutes 0 / 5 problems solved

A **regular expression** is a sequence of characters defining *a search pattern*, that is, a pattern describing a set of strings. These patterns are used for searching and editing text, replacing one substring with another, and so on. The simplest example of using a regular expression is when we search for some word in a text file or on a web page. For example, if we look for the word "python", the string "python" becomes a simple regular expression — a search pattern that corresponds only to the word "python" and nothing else. More complicated regular expressions will be able to match a larger number of strings.

In the previous topic, we've already learned the basics of regular expressions common for all programming languages. Now it's time to see what are the specifics of using regexps in Python.

§1. Re module and match()

You can use the power of regexps if you refer to a standard Python module called `re`. That is, to use anything related to regexps in Python, you must first import this module.

```
import re
```

This module provides you with several functions that search for matches for your regular expression in different ways. Let's get familiar with one of these functions, `match()`. It accepts a *regular expression pattern* (first argument) and a *string* (second argument) and checks whether there's a match for the pattern in the *beginning of the string*.

```
import re

regexp = 'burrito'
string = 'boorrito'
result = re.match(regexp, string)
```

If there's no match for your regexp right in the beginning of the string, `match` returns `None` value. Otherwise, the function returns a special structure called *match object* that will contain the information about the found match. We won't go into the nature of this object right now: all we need to know is that a match object is always a result of a successful match, and `None` is always a result of no found matches. Thus, to know if we have a match, we simply need to check whether the result is equal to `None` .

```
result = re.match('burrito', 'boorrito')
print(result is None)
# The output is True (no matches are found)
# because 'boorrito' doesn't match 'burrito'
```

Let's try out some other examples! Here, there's a successful application of `match()` function:

```
result = re.match('hedge', 'hedgehog')
print(result is None)
# The output is False,
# because there's 'hedge' in the beginning of string 'hedgehog'
```

Don't forget that `match()` won't help you with finding parts of the string that match the template, but aren't located in the beginning of the string. Check out this example:

```
result = re.match('hog', 'hedgehog')  # no match
# because the beginning of the string doesn't match the template 'hog'
```

Current topic:

Regexps in Python

Topic depends on:

- ✗ Regexps basics — Stage 1
- ✓ Identity testing
- ✓ Load module — Stage 1 — 5

Topic is required for:

Escaping in regexps

Table of contents:

You might also want to note that even if the match is an empty string, match object will still be equal to True, because the length of the matching string doesn't matter: only presence of match does.

```
result = re.match('', 'not an empty string')  # match
# because an empty template doesn't need anything to match the string
```

The example above suggests that you should be careful with empty templates: even though it may seem counterintuitive, they don't match *only empty* strings, they match *all* strings (at least, when you use `match()` function and check the presence of matching substring in the beginning of the string).

Don't forget that regular expressions by default are case sensitive, that is, it's a big deal whether you use upper or lower case letters in your template. Two identical letters of different case won't match each other.

```
result = re.match('HURRAY', 'hurray')  # no match
```

Alright, now that we know the basics of how `re` module can be used in Python, we can talk about more complicated examples of regexp templates.

§2. The dot character

Regular expressions wouldn't be so useful if they could only correspond to one particular string. The true power of regexps lies in the opportunity to state the presence (or absence) of some characters in the regexp template without even specifying these characters directly. The dot character `.` is one of the most important special symbols allowing to do this. It literally matches any single character, e.g. any digit, letter, space, and so on, except for the newline character `\n`.

Let's take a look at some examples. Here `match()` will successfully find a match:

```
# This regexp will correspond to the substring 'python'
# followed by the space and any character
regexp = 'python .'

# all examples match the regexp
re.match(regexp, 'python 3')
re.match(regexp, 'python 2')
re.match(regexp, 'python !')
```

On the other hand, these examples will result in `None`:

```
# The dot doesn't match \n
newline = re.match(regexp, 'python \n')

# `?` doesn't match the space
question = re.match(regexp, 'python?!')
```

Let's also recall another useful special character that we've already learned, that is, the question mark.

§3. The question mark

The question mark `?`, unlike the dot, doesn't replace any character by itself. It is a quantifier that basically means "the previous character can be absent". In other words, the question mark `?` signals that the character before it can occur once or zero times in a string to match the pattern.

```
regexp = 'regexp?'
word1 = re.match(regexp, 'regex')  # match
word2 = re.match(regexp, 'regexp')  # match
```

Of course, you can use the combination of the dot and the question mark. In this case, it'll mean that the string can contain either any single character or nothing.

```
regexp = '.? points? to gryffindor'

# `.? points?` matches `1 point`
re.match(regexp, '1 point to gryffindor')

# `.? points?` matches `0 points`
re.match(regexp, '0 points to gryffindor')

# no match, since `.? points?` doesn't match `-5 points`

re.match(regexp, '-5 points to gryffindor')
```

Even these two basic special symbols, the dot and the question mark, will give you great regexp power. But remember, with great power comes great responsibility. We'll learn to handle this responsibility (and make the dot character match only itself) in the following topic.

§4. Conclusions

Here is a recap:

1. For handling regular expression in Python, the `re` module is used.
2. `match()` function of the `re` module checks whether there's any substring in the beginning of the string that matches your regexp template.
3. The result of `match()` function is either `None` or a match object.
4. Match object converted to `bool` always equals `True`.
5. Regular expressions by default are case-sensitive.
6. Dot `.` replaces any character except for `\n`, question mark `?` means that the previous character is optional and can be missing from a string.

Report a typo

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

Start practicing

Comments (3) Hints (0) Useful links (0) Show discussion