Theory: Shorthands

30 minutes 0 / 5 problems solved

Skip this topic Start practicing

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

Previously, you have learned about sets and their most helpful and powerful features of regular expressions. Since some of them are more widespread than others, special shorthands were designed to make them easy to use. The general formula for shorthands is the backslash `\` followed by a letter (for example, `\s`). Let's take a closer look at this kind of metacharacters.

§1. Shorthands for sets

Like sets, each shorthand represents one character in the string, but which character matches the shorthand and which one does not, depends on the shorthand. There are two types of shorthands that can be used in the place of character sets. Shorthands of the first type correspond to strictly described and limited sets of characters. The shorthands of the second type are **negated** shorthands; they correspond to sets banning certain characters, matching to anything else.

Shorthands of the first type are denoted by a backslash and a lowercase letter:

- `\w` matches alphanumeric characters; it stands for a character from the set `[a-zA-Z0-9_]`. Pay attention to the underscore character `_` in this set, and the absence of the hyphen `-` that theoretically could be there.
- `\d` matches a digit and is equivalent to `[0-9]`
- `\s` matches the whitespace characters, such as the usual space character, the tabulation, the newline character. This shorthand is equivalent to `[ \t\n\r\f\v]` (pay attention to the first character (the space); it is easy to overlook)

The letters chosen for these shorthands can be easily explained and remembered: `w` is for word, `d` is for digit, `s` is for space.

Here is a short example showing these shorthands in action:

```
re.match('\w\scamels?', '1 camel')  # match
re.match('\w\scamels?', 'a\tcamel')  # match
re.match('\d\scamels?', '8\ncamels')  # match
```

Negated shorthands are the total opposites of the first type. There are "symmetrical" shorthands for each of the shorthand mentioned above, and they are denoted by a backslash and the very same letters, only in uppercase:

- `\W` matches everything except for alphanumeric characters, `[^a-zA-Z0-9_]`
- `\D` matches any non-digit character, `[^0-9]`
- `\S` matches any non-whitespace character and is equivalent to `[^ \t\n\r\f\v]`

You can see how these metacharacters are applied in the following example:

```
re.match('\D\S\W', 'A1-')  # match
re.match('\D\S\W', '1 A')  # no match
```

Shorthands are very straightforward metacharacters that barely have any specific rules of usage. And they are cool. Why not use them?

§2. Escaping of shorthands

You do not need to escape shorthands in your regular expressions. Although, technically, you can do it. This way, all of the following operations return the same result, no matter what you use, backslash or raw string literal, or neither:

Current topic:

Shorthands ···

Topic depends on:

× Regexp sets and ranges ···

Topic is required for:

Regexp quantifiers ···

Table of contents:

```
re.match('\w', 'X')
re.match('\\w', 'X')
re.match(r'\w', 'X')
```

If you need to escape a shorthand, just escape its backslash with any of the techniques we mentioned in the previous topics, like this one:

```
re.match(r'\\w', r'\w')  # match
re.match(r'\\w', r'\k')  # no match, because \\w isn't a shorthand here
```

Another great feature of shorthands is that they maintain their metacharacter abilities when they are put in sets. Let's discuss it.

§3. Shorthands in sets

As you remember from the previous topic, most regexp metacharacters become regular characters in sets, but shorthands do not do that. This makes it possible to use shorthands as the basis of your custom character set that you can enlarge by adding some other characters. It may be useful, for example, in case you want to build an alphanumeric set for a language that uses some other letters in addition to "default" ones:

```
re.match('attach[é\w]', 'attaché')  # match
re.match('attach[é\w]', 'attache')  # match
re.match('Stra[\wß][\wß]?e', 'Straße')  # match
re.match('Stra[\wß][\wß]?e', 'Strasse')  # match
```

Many great features, though, can be dangerous. This feature is not an exception — do not use several negated shorthands in one set, as they are not what they may seem!

Suppose you want a set that will match everything except for digits and whitespace characters. You could write [`\D\S]`, but that's a wrong move. The explanation is that a character matching a set should match only *one* element of this set, but not all of them at the same time. This way, when you use the set `[\D\S]`, a character matching this set should match *either* `\D` *or* `\S`. So, every digit matches `\S` (because digits are not whitespace characters), and every whitespace character matches `\D` (because whitespace characters are never digits). In the end, this set matches *all* possible characters (looks like the dot character `.`, but more powerful, because unlike the dot, it matches `\n` too).

```
re.match('[\D\S]', '0')  # match, 0 is not a whitespace character
re.match('[\D\S]', '\n')  # match, \n is not a digit
re.match('[\D\S]', 'a')  # match, 'a' is not a digit and not a whitespace character
```

Instead of `[\D\S]`, use `[^\d\s]`. This set matches everything that does not fall into the categories of whitespaces and digits *at the same time*.

```
re.match('[^\d\s]', '0')  # no match, 0 is a digit
re.match('[^\d\s]', '\t')  # no match, \t is a whitespace character
re.match('[^\d\s]', 'a')  # match, 'a' is not a digit and not a whitespace
```

So, keep in mind that if you are thinking about using a combination of negated shorthands in your set, you probably should replace them with a *negated set* and regular, *non-negated shorthands* inside it.

Used alone in combination with regular characters, negated shorthands behave properly in sets. You can add some of the characters banned by the negated shorthand to the set, making the scope of your set even bigger. For example, the set `[\Wa-c]` matches no alphanumeric characters except for the `a`, `b`, `c` letters:

```
re.match('[\Wa-c]', 'a')  # match, 'a' is in the set
re.match('[\Wa-c]', 'z')  # no match: 'z' is alphanumeric and is not in the a-c range
re.match('[\Wa-c]', '?')  # match, ? is not alphanumeric
```

There are a few other metacharacters that look like shorthands, but they do a slightly different thing.

§4. Boundary shorthands

These shorthands look more or less the same: a backslash and a letter, but they do not match any characters. These tricky fellows formally match empty strings, but more specifically, they match empty strings in certain *situations.*

- `\b` matches a word boundary. In other words, it matches an empty string between an alphanumeric character (any character matching `\w`) and a non-alphanumeric character (`\W`) or absence of characters. You can use it to make sure that your regular expression will match only a separate word, not a part of a bigger word.

```
re.match(r'\b', 'Hello?')  # match (an empty string between an absence of character and a letter)
re.match(r'\b', '')  # no match (no alphanumeric character)
re.match(r'Hail\b', 'Hail Mary!')  # match

re.match(r'Hail\b Caesar', 'Hail Caesar')  # match (but \b is useless here)
```

- `\B` matches the absence of the word boundary, that is, an empty string between two alphanumeric characters `\w`. It serves for searching for alphanumeric sequences that are parts of bigger alphanumeric sequences.

```
re.match(r'Hail\b', 'Hailey')  # no match
re.match(r'Hail\B', 'Hailey')  # match
```

- `\A` matches only an empty string at the start of the string. This way, you can make sure that the substring matching your regular expression will be located at the very beginning of the string. This shorthand is not so useful for us at the moment, because the `match()` function always matches substrings located at the start of the string, but it will come in handy later.
- `\Z` matches an empty string at the end of the string.

```
re.match('Hail\Z', 'Hail!')  # no match
re.match('Hail\Z', 'Hail')  # match
```

`\b` shorthand is the only one that you need to escape to use correctly, otherwise, it will become a Python's metacharacter, not a regexp's. You can avoid escaping by adding the `r` prefix. in general, it is recommended to use the `r` prefix any time you write a regular expression, it will save you some time and make your code more readable.

```
re.match('Hail\b', 'Hail Mary!')  # no match, \b is a Python escape character
re.match(r'Hail\b', 'Hail Mary!')  # match, \b is a regexp shorthand
```

By the way, `\A` and `\Z` are not widespread. They are often replaced by simpler metacharacters with identical functions.

The hat character `^` is identical to `\A`. The dollar sign `$` is equal to `\Z`.

Here is an example for the dollar sign character `$`, since it is the only one that makes sense for us while we use the `match()` function to compare a template and a string:

```
re.match("Bring cash$", "Bring cash$")  # no match: $ in regexp means "the end of the string"
re.match("Bring cash$", "Bring cash")  # match: h is the last character in the string
re.match("Bring cash\Z", "Bring cash")  # match: \Z identical to $
```

The hat character `^` works the same way, only for the beginning of the string.

§5. Conclusions

So, in this topic we have learned:

- there are special short designations for some sets of characters;
- these shorthands, depending on which one you use, can match alphanumeric characters, digits, or whitespace characters. They can match any characters ***except*** these (these shorthands are called negated);
- shorthands can be used in sets. You should be careful with negated shorthands with sets;
- some shorthands match empty strings in special situations, at word or string boundaries.

Report a typo

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

😍 🙂 😐 🙁 😡

Start practicing

Comments (5) Hints (0) Useful links (1) Show discussion