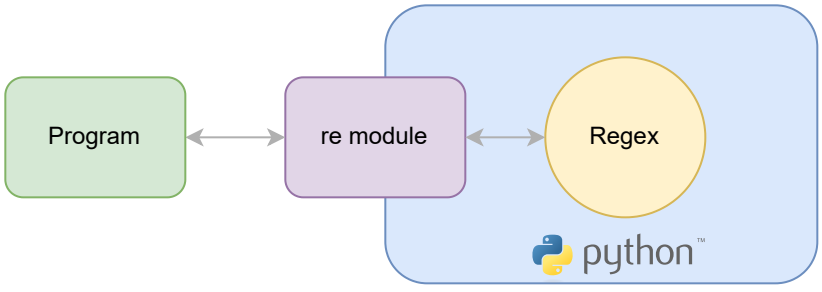
Python Regular Expressions

Summary: in this tutorial, you'll learn about Python regular expressions and how to use the most commonly used regular expression functions.

Introduction to the Python regular expressions

Regular expressions (called regex or regexp) specify search patterns. Typical examples of regular expressions are the patterns for matching email addresses, phone numbers, and credit card numbers.

Regular expressions are essentially a specialized programming language embedded in Python. And you can interact with regular expressions via the built-in `re` module in Python.



The following shows an example of a simple regular expression:

```
'\d'
```

In this example, a regular expression is a string that contains a search pattern. The `'\d'` is a digit character set (https://www.pythontutorial.net/python-regex/python-regex-character-set/) that matches any single digit from 0 to 9.

> Note that you'll learn how to construct more complex and advanced patterns in the next tutorials. This tutorial focuses on the functions that deal with regular expressions.

To use this regular expression, you follow these steps:

First, import the `re` module:

```
import re
```

Second, compile the regular expression into a `Pattern` object:

```
p = re.compile('\d')
```

Third, use one of the methods of the `Pattern` object to match a string:

```
s = "Python 3.10 was released on October 04, 2021"
result = p.findall(s)

print(result)
```

Output:

```
['3', '1', '0', '0', '4', '2', '0', '2', '1']
```

The `findall()` method returns a list of single digits in the string s.

The following shows the complete program:

```
import re


p = re.compile('\d')
s = "Python 3.10 was released on October 04, 2021"

results = p.findall(s)
print(results)
```

Besides the `findall()` method, the `Pattern` object has other essential methods that allow you to match a string:

Method	**Purpose**
`match()` (https://www.pythontutorial.net/python-regex/python-regex-match/)	Find the pattern at the beginning of a string
`search()`	Return the first match of a pattern in a string
`findall()`	Return all matches of a pattern in a string
`finditer()`	Return all matches of a pattern as an iterator (https://www.pythontutorial.net/advanced-python/python-iterators/)

Python regular expression functions

Besides the `Pattern` class, the `re` module has some functions that match a string for a pattern:

- `match()`
- `search()`
- `findall()`
- `finditer()`

These functions have the same names as the methods of the `Pattern` object. Also, they take the same arguments as the corresponding methods of the `Pattern` object. However, you don't have to manually compile the regular expression before using it.

The following example shows the same program that uses the `findall()` function instead of the `findall()` method of a `Pattern` object:

```
import re

s = "Python 3.10 was released on October 04, 2021."
results = re.findall('\d',s)
print(results)
```

Using the functions in the `re` module is more concise than the methods of the `Pattern` object because you don't have to compile regular expressions manually.

Under the hood, these functions create a `Pattern` object and call the appropriate method on it. They also store the compiled regular expression in a cache for speed optimization.

It means that if you call the same regular expression from the second time, these functions will not need to recompile the regular expression. Instead, they get the compiled regular expression from the cache.

Should you use the `re` functions or methods of the `Pattern` object?

If you use a regular expression within a loop (https://www.pythontutorial.net/python-basics/python-for-range/), the `Pattern` object may save a few function calls. However, if you use it outside of loops, the difference is very little due to the internal cache.

The following sections discuss the most commonly used functions in the `re` module including `search()`, `match()`, and `fullmatch()`.

search() function

The `search()` function searches for a pattern within a string. If there is a match, it returns the first Match object or None otherwise. For example:

```
import re

s = "Python 3.10 was released on October 04, 2021."

pattern = '\d{2}'
match = re.search(pattern, s)
print(type(match))
print(match)
```

Output:

```
<class 're.Match'>
<re.Match object; span=(9, 11), match='10'>
```

In this example, the `search()` function returns the first two digits in the string `s` as the `Match` object.

Match object

The `Match` object provides the information about the matched string. It has the following important methods:

Method	Description
`group()`	Return the matched string
`start()`	Return the starting position of the match
`end()`	Return the ending position of the match
`span()`	Return a tuple (start, end) that specifies the positions of the match

The following example examines the `Match` object:

```
import re


s = "Python 3.10 was released on October 04, 2021."
result = re.search('\d', s)

print('Matched string:',result.group())
print('Starting position:', result.start())
print('Ending position:',result.end())
print('Positions:',result.span())
```

Output:

```
Matched string: 3
Starting position: 7
Ending position: 8
Positions: (7, 8)
```

match() function

The match() function returns a `Match` object if it finds a pattern at the beginning of a string. For example:

```
import re

l = ['Python',
    'CPython is an implementation of Python written in C',
    'Jython is a Java implementation of Python',
     'IronPython is Python on .NET framework']

pattern = '\wython'
for s in l:
```

```
    result = re.match(pattern,s)
    print(result)
```

Output:

```
<re.Match object; span=(0, 6), match='Python'>
None
<re.Match object; span=(0, 6), match='Jython'>
None
```

In this example, the `\w` is the word character set that matches any single character.

The `\wython` matches any string that starts with any sing word character and is followed by the literal string `ython`, for example, `Python`.

Since the `match()` function only finds the pattern at the beginning of a string, the following strings match the pattern:

```
Python
Jython is a Java implementation of Python
```

And the following string doesn't match:

```
'CPython is an implementation of Python written in C'
'IronPython is Python on .NET framework'
```

fullmatch() function

The `fullmatch()` function returns a `Match` object if the whole string matches a pattern or `None` otherwise. The following example uses the `fullmatch()` function to match a string with four digits:

```
import re
```

```
s = "2021"
pattern = '\d{4}'
result = re.fullmatch(pattern, s)
print(result)
```

Output:

```
<re.Match object; span=(0, 4), match='2019'>
```

The pattern `'\d{4}'` matches a string with four digits. Therefore, the `fullmatch()` function returns the string `2021`.

If you place the number `2021` at the middle or the end of the string, the `fullmatch()` will return `None`. For example:

```
import re

s = "Python 3.10 released in 2021"
pattern = '\d{4}'
result = re.fullmatch(pattern, s)
print(result)
```

Output:

```
None
```

Regular expressions and raw strings

It's important to note that Python and regular expression are different programming languages. They have their own syntaxes.

The `re` module is the interface between Python and regular expression programming languages. It behaves like an interpreter between them.

To construct a pattern, regular expressions often use a backslash `'\'` for example `\d` and `\w`. But this collides with Python's usage of the backslash for the same purpose in string literals.

For example, suppose you need to match the following string:

```
s = '\section'
```

In Python, the backslash (https://www.pythontutorial.net/python-basics/python-backslash/) (`\`) is a special character. To construct a regular expression, you need to escape any backslashes by preceding each of them with a backslash (`\`):

```
pattern = '\\section'
```

In regular expressions, the pattern must be `'\\section'`. However, to express this pattern in a string literal in Python, you need to use two more backslashes to escape both backslashes again:

```
pattern = '\\\\section'
```

Simply put, to match a literal backslash (`'\'`), you have to write `'\\\\'` because the regular expression must be `'\\'` and each backslash must be expressed as `'\\'` inside a string literal in Python.

This results in lots of repeated backslashes. Hence, it makes the regular expressions difficult to read and understand.

A solution is to use the raw strings (https://www.pythontutorial.net/python-basics/python-raw-strings/) in Python for regular expressions because raw strings treat the backslash (`\`) as a literal character, not a special character.

To turn a regular string into a raw string, you prefix it with the letter `r` or `R`. For example:

```
import re

s = '\section'
pattern = r'\\section'
```

```
result = re.findall(pattern, s)

print(result)
```

Output:

```
['\\section']
```

Note that in Python '\section' and '\\section' are the same:

```
p1 = '\\section'
p2 = '\section'

print(p1==p2) # true
```

In practice, you'll find the regular expressions constructed in Python using the raw strings.

Summary

- A regular expression is a string that contains the special characters for matching a string with a pattern.
- Use the `Pattern` object or functions in `re` module to search for a pattern in a string.
- Use raw strings to construct regular expression to avoid escaping the backslashes.