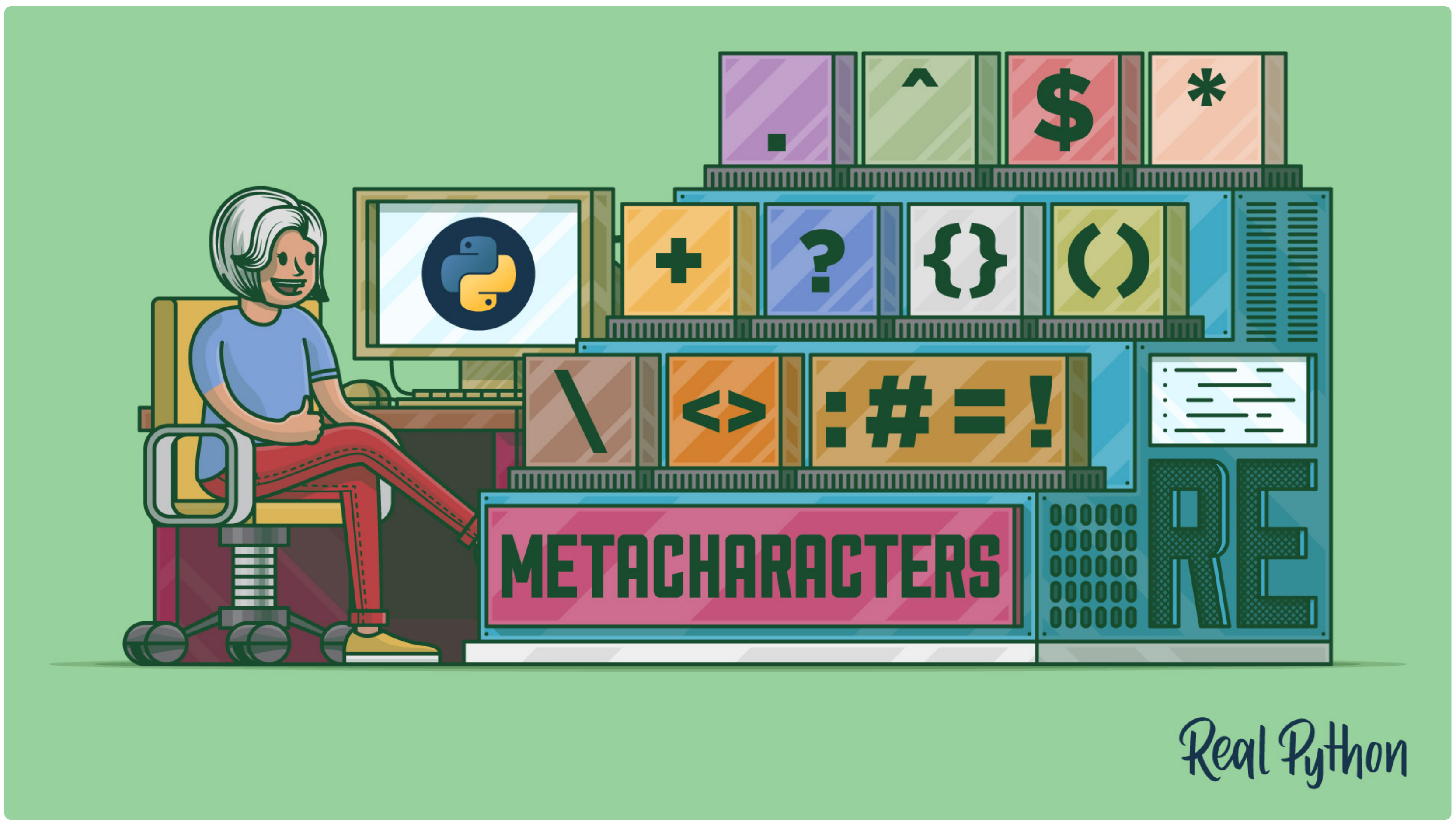

Regular Expressions: Regexes in Python (Part 1)

by John Sturtz 17 Comments basics python

Mark as Completed

Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Regular Expressions and Building Regexes in Python**

In this tutorial, you'll explore **regular expressions**, also known as **regexes**, in Python. A regex is a special sequence of characters that defines a pattern for complex string-matching functionality.

Earlier in this series, in the tutorial Strings and Character Data in Python, you learned how to define and manipulate string objects. Since then, you've seen some ways to determine whether two strings match each other:

- You can test whether two strings are equal using the equality (==) operator.
- You can test whether one string is a substring of another with the `in` operator or the built-in string methods `.find()` and `.index()`.

String matching like this is a common task in programming, and you can get a lot done with string operators and built-in methods. At times, though, you may need more sophisticated pattern-matching capabilities.

In this tutorial, you'll learn:

- How to access the **`re` module**, which implements regex matching in Python
- How to use **`re.search()`** to match a pattern against a string
- How to create complex matching pattern with regex **metacharacters**

Fasten your seat belt! Regex syntax takes a little getting used to. But once you get comfortable with it, you'll find regexes almost indispensable in your Python programming.

Regexes in Python and Their Uses

Imagine you have a string object `s`. Now suppose you need to write Python code to find out whether `s` contains the substring `'123'`. There are at least a couple ways to do this. You could use the `in` operator:

Python >>>

```
>>> s = 'foo123bar'
>>> '123' in s
True
```

If you want to know not only *whether* `'123'` exists in `s` but also *where* it exists, then you can use `.find()` or `.index()`. Each of these returns the character position within `s` where the substring resides:

Python >>>

```
>>> s = 'foo123bar'
>>> s.find('123')
3
>>> s.index('123')
3
```

In these examples, the matching is done by a straightforward character-by-character comparison. That will get the job done in many cases. But sometimes, the problem is more complicated than that.

For example, rather than searching for a fixed substring like `'123'`, suppose you wanted to determine whether a string contains *any* three consecutive decimal digit characters, as in the strings `'foo123bar'`, `'foo456bar'`, `'234baz'`, and `'qux678'`.

Strict character comparisons won't cut it here. This is where regexes in Python come to the rescue.

A (Very Brief) History of Regular Expressions

In 1951, mathematician Stephen Cole Kleene described the concept of a regular language, a language that is recognizable by a finite automaton and formally expressible using regular expressions. In the mid-1960s, computer science pioneer Ken Thompson, one of the original designers of Unix, implemented pattern matching in the QED text editor using Kleene's notation.

Since then, regexes have appeared in many programming languages, editors, and other tools as a means of determining whether a string matches a specified pattern. Python, Java, and Perl all support regex functionality, as do most Unix tools and many text editors.

The `re` Module

Regex functionality in Python resides in a module named `re`. The `re` module contains many useful functions and methods, most of which you'll learn about in the next tutorial in this series.

For now, you'll focus predominantly on one function, `re.search()`.

`re.search(<regex>, <string>)`

> Scans a string for a regex match.

`re.search(<regex>, <string>)` scans `<string>` looking for the first location where the pattern `<regex>` matches. If a match is found, then `re.search()` returns a **match object**. Otherwise, it returns `None`.

`re.search()` takes an optional third `<flags>` argument that you'll learn about at the end of this tutorial.

How to Import `re.search()`

Because `search()` resides in the `re` module, you need to import it before you can use it. One way to do this is to import the entire module and then use the module name as a prefix when calling the function:

Python

```python
import re
re.search(...)
```

Alternatively, you can import the function from the module by name and then refer to it without the module name prefix:

Python

```python
from re import search
search(...)
```

You'll always need to import `re.search()` by one means or another before you'll be able to use it.

The examples in the remainder of this tutorial will assume the first approach shown—importing the `re` module and then referring to the function with the module name prefix: `re.search()`. For the sake of brevity, the `import re` statement will usually be omitted, but remember that it's always necessary.

For more information on importing from modules and packages, check out Python Modules and Packages—An Introduction.

First Pattern-Matching Example

Now that you know how to gain access to `re.search()`, you can give it a try:

Python

```python
1 >>> s = 'foo123bar'
2
3 >>> # One last reminder to import!
4 >>> import re
5
6 >>> re.search('123', s)
7 <_sre.SRE_Match object; span=(3, 6), match='123'>
```

Here, the search pattern `<regex>` is `123` and `<string>` is `s`. The returned match object appears on **line 7**. Match objects contain a wealth of useful information that you'll explore soon.

For the moment, the important point is that `re.search()` did in fact return a match object rather than `None`. That tells you that it found a match. In other words, the specified `<regex>` pattern `123` is present in `s`.

A match object is **truthy**, so you can use it in a Boolean context like a conditional statement:

Python

```python
>>> if re.search('123', s):
...     print('Found a match.')
... else:
...     print('No match.')
...
Found a match.
```

The interpreter displays the match object as `<_sre.SRE_Match object; span=(3, 6), match='123'>`. This contains some useful information.

`span=(3, 6)` indicates the portion of `<string>` in which the match was found. This means the same thing as it would in slice notation:

Python

```python
>>> s[3:6]
'123'
```

In this example, the match starts at character position `3` and extends up to but not including position `6`.

`match='123'` indicates which characters from `<string>` matched.

This is a good start. But in this case, the `<regex>` pattern is just the plain string `'123'`. The pattern matching here is still just character-by-character comparison, pretty much the same as the `in` operator and `.find()` examples shown earlier. The match object helpfully tells you that the matching characters were `'123'`, but that's not much of a revelation since those were exactly the characters you searched for.

You're just getting warmed up.

Python Regex Metacharacters

The real power of regex matching in Python emerges when `<regex>` contains special characters called **metacharacters**. These have a unique meaning to the regex matching engine and vastly enhance the capability of the search.

Consider again the problem of how to determine whether a string contains any three consecutive decimal digit characters.

In a regex, a set of characters specified in square brackets (`[]`) makes up a **character class**. This metacharacter sequence matches any single character that is in the class, as demonstrated in the following example:

Python >>>

```
>>> s = 'foo123bar'
>>> re.search('[0-9][0-9][0-9]', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

`[0-9]` matches any single decimal digit character—any character between `'0'` and `'9'`, inclusive. The full expression `[0-9][0-9][0-9]` matches any sequence of three decimal digit characters. In this case, `s` matches because it contains three consecutive decimal digit characters, `'123'`.

These strings also match:

Python >>>

```
>>> re.search('[0-9][0-9][0-9]', 'foo456bar')
<_sre.SRE_Match object; span=(3, 6), match='456'>

>>> re.search('[0-9][0-9][0-9]', '234baz')
<_sre.SRE_Match object; span=(0, 3), match='234'>

>>> re.search('[0-9][0-9][0-9]', 'qux678')
<_sre.SRE_Match object; span=(3, 6), match='678'>
```

On the other hand, a string that doesn't contain three consecutive digits won't match:

Python >>>

```
>>> print(re.search('[0-9][0-9][0-9]', '12foo34'))
None
```

With regexes in Python, you can identify patterns in a string that you wouldn't be able to find with the `in` operator or with string methods.

Take a look at another regex metacharacter. The dot (`.`) metacharacter matches any character except a newline, so it functions like a wildcard:

Python >>>

```
>>> s = 'foo123bar'
>>> re.search('1.3', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> s = 'foo13bar'
>>> print(re.search('1.3', s))
None
```

In the first example, the regex `1.3` matches `'123'` because the `'1'` and `'3'` match literally, and the `.` matches the `'2'`. Here, you're essentially asking, "Does `s` contain a `'1'`, then any character (except a newline), then a `'3'`?" The answer is yes for `'foo123bar'` but no for `'foo13bar'`.

These examples provide a quick illustration of the power of regex metacharacters. Character class and dot are but two of the metacharacters supported by the `re` module. There are many more. Next, you'll explore them fully.

Metacharacters Supported by the `re` Module

The following table briefly summarizes all the metacharacters supported by the `re` module. Some characters serve more than one purpose:

Character(s)	Meaning
`.`	Matches any single character except newline

Character(s)	Meaning
`^`	· Anchors a match at the start of a string · Complements a character class
`$`	Anchors a match at the end of a string
`*`	Matches zero or more repetitions
`+`	Matches one or more repetitions
`?`	· Matches zero or one repetition · Specifies the non-greedy versions of `*`, `+`, and `?` · Introduces a lookahead or lookbehind assertion · Creates a named group
`{}`	Matches an explicitly specified number of repetitions
`\`	· Escapes a metacharacter of its special meaning · Introduces a special character class · Introduces a grouping backreference
`[]`	Specifies a character class
`|`	Designates alternation
`()`	Creates a group

`:`
`#`
`=`
`!`
	Designate a specialized group

`<>`	Creates a named group

This may seem like an overwhelming amount of information, but don't panic! The following sections go over each one of these in detail.

The regex parser regards any character not listed above as an ordinary character that matches only itself. For example, in the first pattern-matching example shown above, you saw this:

Python >>>

```
>>> s = 'foo123bar'
>>> re.search('123', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

In this case, `123` is technically a regex, but it's not a very interesting one because it doesn't contain any metacharacters. It just matches the string `'123'`.

Things get much more exciting when you throw metacharacters into the mix. The following sections explain in detail how you can use each metacharacter or metacharacter sequence to enhance pattern-matching functionality.

Metacharacters That Match a Single Character

The metacharacter sequences in this section try to match a single character from the search string. When the regex parser encounters one of these metacharacter sequences, a match happens if the character at the current parsing position fits the description that the sequence describes.

`[]`

> Specifies a specific set of characters to match.

Characters contained in square brackets (`[]`) represent a **character class**—an enumerated set of characters to match from. A character class metacharacter sequence will match any single character contained in the class.

You can enumerate the characters individually like this:

Python >>>

```
>>> re.search('ba[artz]', 'foobarqux')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> re.search('ba[artz]', 'foobazqux')
<_sre.SRE_Match object; span=(3, 6), match='baz'>
```

The metacharacter sequence `[artz]` matches any single `'a'`, `'r'`, `'t'`, or `'z'` character. In the example, the regex `ba[artz]` matches both `'bar'` and `'baz'` (and would also match `'baa'` and `'bat'`).

A character class can also contain a range of characters separated by a hyphen (`-`), in which case it matches any single character within the range. For example, `[a-z]` matches any lowercase alphabetic character between `'a'` and `'z'`, inclusive:

Python >>>

```
>>> re.search('[a-z]', 'FOObar')
<_sre.SRE_Match object; span=(3, 4), match='b'>
```

`[0-9]` matches any digit character:

Python >>>

```
>>> re.search('[0-9][0-9]', 'foo123bar')
<_sre.SRE_Match object; span=(3, 5), match='12'>
```

In this case, `[0-9][0-9]` matches a sequence of two digits. The first portion of the string `'foo123bar'` that matches is `'12'`.

`[0-9a-fA-F]` matches any hexadecimal digit character:

Python >>>

```
>>> re.search('[0-9a-fA-f]', '--- a0 ---')
<_sre.SRE_Match object; span=(4, 5), match='a'>
```

Here, `[0-9a-fA-F]` matches the first hexadecimal digit character in the search string, `'a'`.

> **Note:** In the above examples, the return value is always the leftmost possible match. `re.search()` scans the search string from left to right, and as soon as it locates a match for `<regex>`, it stops scanning and returns the match.

You can complement a character class by specifying `^` as the first character, in which case it matches any character that *isn't* in the set. In the following example, `[^0-9]` matches any character that isn't a digit:

Python >>>

```
>>> re.search('[^0-9]', '12345foo')
<_sre.SRE_Match object; span=(5, 6), match='f'>
```

Here, the match object indicates that the first character in the string that isn't a digit is `'f'`.

If a `^` character appears in a character class but isn't the first character, then it has no special meaning and matches a literal `'^'` character:

Python >>>

```
>>> re.search('[#:^]', 'foo^bar:baz#qux')
<_sre.SRE_Match object; span=(3, 4), match='^'>
```

As you've seen, you can specify a range of characters in a character class by separating characters with a hyphen. What if you want the character class to include a literal hyphen character? You can place it as the first or last character or escape it with a backslash (`\`):

Python >>>

```
>>> re.search('[-abc]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
>>> re.search('[abc-]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
>>> re.search('[ab\-c]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
```

If you want to include a literal `']'` in a character class, then you can place it as the first character or escape it with backslash:

Python >>>

```
>>> re.search('[]]', 'foo[1]')
<_sre.SRE_Match object; span=(5, 6), match=']'>
>>> re.search('[ab\]cd]', 'foo[1]')
<_sre.SRE_Match object; span=(5, 6), match=']'>
```

Other regex metacharacters lose their special meaning inside a character class:

Python >>>

```
>>> re.search('[)*+|]', '123*456')
<_sre.SRE_Match object; span=(3, 4), match='*'>
>>> re.search('[)*+|]', '123+456')
<_sre.SRE_Match object; span=(3, 4), match='+'>
```

As you saw in the table above, `*` and `+` have special meanings in a regex in Python. They designate repetition, which you'll learn more about shortly. But in this example, they're inside a character class, so they match themselves literally.

dot (`.`)

> Specifies a wildcard.

The `.` metacharacter matches any single character except a newline:

Python >>>

```
>>> re.search('foo.bar', 'fooxbar')
<_sre.SRE_Match object; span=(0, 7), match='fooxbar'>

>>> print(re.search('foo.bar', 'foobar'))
None
>>> print(re.search('foo.bar', 'foo\nbar'))
None
```

As a regex, `foo.bar` essentially means the characters `'foo'`, then any character except newline, then the characters `'bar'`. The first string shown above, `'fooxbar'`, fits the bill because the `.` metacharacter matches the `'x'`.

The second and third strings fail to match. In the last case, although there's a character between `'foo'` and `'bar'`, it's a newline, and by default, the `.` metacharacter doesn't match a newline. There is, however, a way to force `.` to match a newline, which you'll learn about at the end of this tutorial.

`\w`
`\W`

> Match based on whether a character is a word character.

`\w` matches any alphanumeric word character. Word characters are uppercase and lowercase letters, digits, and the underscore (_) character, so `\w` is essentially shorthand for `[a-zA-Z0-9_]`:

Python >>>

```
>>> re.search('\w', '#(.a$@&')
<_sre.SRE_Match object; span=(3, 4), match='a'>
>>> re.search('[a-zA-Z0-9_]', '#(.a$@&')
<_sre.SRE_Match object; span=(3, 4), match='a'>
```

In this case, the first word character in the string `'#(.a$@&'` is `'a'`.

`\W` is the opposite. It matches any non-word character and is equivalent to `[^a-zA-Z0-9_]`:

Python >>>

```
>>> re.search('\W', 'a_1*3Qb')
<_sre.SRE_Match object; span=(3, 4), match='*'>
>>> re.search('[^a-zA-Z0-9_]', 'a_1*3Qb')
<_sre.SRE_Match object; span=(3, 4), match='*'>
```

Here, the first non-word character in `'a_1*3!b'` is `'*'`.

`\d`
`\D`

> Match based on whether a character is a decimal digit.

`\d` matches any decimal digit character. `\D` is the opposite. It matches any character that *isn't* a decimal digit:

Python >>>

```
>>> re.search('\d', 'abc4def')
<_sre.SRE_Match object; span=(3, 4), match='4'>

>>> re.search('\D', '234Q678')
<_sre.SRE_Match object; span=(3, 4), match='Q'>
```

`\d` is essentially equivalent to `[0-9]`, and `\D` is equivalent to `[^0-9]`.

`\s`
`\S`

> Match based on whether a character represents whitespace.

`\s` matches any whitespace character:

Python >>>

```
>>> re.search('\s', 'foo\nbar baz')
<_sre.SRE_Match object; span=(3, 4), match='\n'>
```

Note that, unlike the dot wildcard metacharacter, `\s` does match a newline character.

`\S` is the opposite of `\s`. It matches any character that *isn't* whitespace:

Python >>>

```
>>> re.search('\S', '  \n foo  \n  ')
<_sre.SRE_Match object; span=(4, 5), match='f'>
```

Again, `\s` and `\S` consider a newline to be whitespace. In the example above, the first non-whitespace character is `'f'`.

The character class sequences `\w`, `\W`, `\d`, `\D`, `\s`, and `\S` can appear inside a square bracket character class as well:

Python >>>

```
>>> re.search('[\d\w\s]', '---3---')
<_sre.SRE_Match object; span=(3, 4), match='3'>
>>> re.search('[\d\w\s]', '---a---')
<_sre.SRE_Match object; span=(3, 4), match='a'>
>>> re.search('[\d\w\s]', '--- ---')
<_sre.SRE_Match object; span=(3, 4), match=' '>
```

In this case, `[\d\w\s]` matches any digit, word, or whitespace character. And since `\w` includes `\d`, the same character class could also be expressed slightly shorter as `[\w\s]`.

Escaping Metacharacters

Occasionally, you'll want to include a metacharacter in your regex, except you won't want it to carry its special meaning. Instead, you'll want it to represent itself as a literal character.

backslash (\)

> Removes the special meaning of a metacharacter.

As you've just seen, the backslash character can introduce special character classes like word, digit, and whitespace. There are also special metacharacter sequences called **anchors** that begin with a backslash, which you'll learn about below.

When it's not serving either of these purposes, the backslash **escapes** metacharacters. A metacharacter preceded by a backslash loses its special meaning and matches the literal character instead. Consider the following examples:

Python >>>

```
1  >>> re.search('.', 'foo.bar')
2  <_sre.SRE_Match object; span=(0, 1), match='f'>
3
4  >>> re.search('\.', 'foo.bar')
5  <_sre.SRE_Match object; span=(3, 4), match='.'>
```

In the `<regex>` on **line 1**, the dot (`.`) functions as a wildcard metacharacter, which matches the first character in the string (`'f'`). The `.` character in the `<regex>` on **line 4** is escaped by a backslash, so it isn't a wildcard. It's interpreted literally and matches the `'.'` at index `3` of the search string.

Using backslashes for escaping can get messy. Suppose you have a string that contains a single backslash:

Python >>>

```
>>> s = r'foo\bar'
>>> print(s)
foo\bar
```

Now suppose you want to create a `<regex>` that will match the backslash between `'foo'` and `'bar'`. The backslash is itself a special character in a regex, so to specify a literal backslash, you need to escape it with another backslash. If that's that case, then the following should work:

Python >>>

```
>>> re.search('\\', s)
```

Not quite. This is what you get if you try it:

Python >>>

```
>>> re.search('\\', s)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    re.search('\\', s)
  File "C:\Python36\lib\re.py", line 182, in search
    return _compile(pattern, flags).search(string)
  File "C:\Python36\lib\re.py", line 301, in _compile
    p = sre_compile.compile(pattern, flags)
  File "C:\Python36\lib\sre_compile.py", line 562, in compile
    p = sre_parse.parse(p, flags)
  File "C:\Python36\lib\sre_parse.py", line 848, in parse
    source = Tokenizer(str)
  File "C:\Python36\lib\sre_parse.py", line 231, in __init__
    self.__next()
  File "C:\Python36\lib\sre_parse.py", line 245, in __next
    self.string, len(self.string) - 1) from None
sre_constants.error: bad escape (end of pattern) at position 0
```

Oops. What happened?

The problem here is that the backslash escaping happens twice, first by the Python interpreter on the string literal and then again by the regex parser on the regex it receives.

Here's the sequence of events:

1. The Python interpreter is the first to process the string literal `'\\'`. It interprets that as an escaped backslash and passes only a single backslash to `re.search()`.
2. The regex parser receives just a single backslash, which isn't a meaningful regex, so the messy error ensues.

There are two ways around this. First, you can escape *both* backslashes in the original string literal:

Python >>>

```
>>> re.search('\\\\', s)
<_sre.SRE_Match object; span=(3, 4), match='\\'>
```

Doing so causes the following to happen:

1. The interpreter sees `'\\\\'` as a pair of escaped backslashes. It reduces each pair to a single backslash and passes `'\\'` to the regex parser.
2. The regex parser then sees `\\` as one escaped backslash. As a `<regex>`, that matches a single backslash character. You can see from the match object that it matched the backslash at index `3` in `s` as intended. It's cumbersome, but it works.

The second, and probably cleaner, way to handle this is to specify the `<regex>` using a raw string:

Python >>>

```
>>> re.search(r'\\', s)
<_sre.SRE_Match object; span=(3, 4), match='\\'>
```

This suppresses the escaping at the interpreter level. The string `'\\'` gets passed unchanged to the regex parser, which again sees one escaped backslash as desired.

It's good practice to use a raw string to specify a regex in Python whenever it contains backslashes.

Anchors

Anchors are zero-width matches. They don't match any actual characters in the search string, and they don't consume any of the search string during parsing. Instead, an anchor dictates a particular location in the search string where a match must occur.

^

\A

> Anchor a match to the start of `<string>`.

When the regex parser encounters `^` or `\A`, the parser's current position must be at the beginning of the search string for it to find a match.

In other words, regex `^foo` stipulates that `'foo'` must be present not just any old place in the search string, but at the beginning:

Python >>>

```
>>> re.search('^foo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('^foo', 'barfoo'))
None
```

`\A` functions similarly:

Python >>>

```
>>> re.search('\Afoo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('\Afoo', 'barfoo'))
None
```

`^` and `\A` behave slightly differently from each other in `MULTILINE` mode. You'll learn more about `MULTILINE` mode below in the section on flags.

$
\Z

> Anchor a match to the end of `<string>`.

When the regex parser encounters `$` or `\Z`, the parser's current position must be at the end of the search string for it to find a match. Whatever precedes `$` or `\Z` must constitute the end of the search string:

Python >>>

```
>>> re.search('bar$', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> print(re.search('bar$', 'barfoo'))
None

>>> re.search('bar\Z', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> print(re.search('bar\Z', 'barfoo'))
None
```

As a special case, `$` (but not `\Z`) also matches just before a single newline at the end of the search string:

Python >>>

```
>>> re.search('bar$', 'foobar\n')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
```

In this example, `'bar'` isn't technically at the end of the search string because it's followed by one additional newline character. But the regex parser lets it slide and calls it a match anyway. This exception doesn't apply to `\Z`.

`$` and `\Z` behave slightly differently from each other in `MULTILINE` mode. See the section below on flags for more information on `MULTILINE` mode.

\b

> Anchors a match to a word boundary.

`\b` asserts that the regex parser's current position must be at the beginning or end of a word. A word consists of a sequence of alphanumeric characters or underscores (`[a-zA-Z0-9_]`), the same as for the `\w` character class:

Python >>>

```
 1  >>> re.search(r'\bbar', 'foo bar')
 2  <_sre.SRE_Match object; span=(4, 7), match='bar'>
 3  >>> re.search(r'\bbar', 'foo.bar')
 4  <_sre.SRE_Match object; span=(4, 7), match='bar'>
 5
 6  >>> print(re.search(r'\bbar', 'foobar'))
 7  None
 8
 9  >>> re.search(r'foo\b', 'foo bar')
10  <_sre.SRE_Match object; span=(0, 3), match='foo'>
11  >>> re.search(r'foo\b', 'foo.bar')
12  <_sre.SRE_Match object; span=(0, 3), match='foo'>
13
14  >>> print(re.search(r'foo\b', 'foobar'))
15  None
```

In the above examples, a match happens on **lines 1 and 3** because there's a word boundary at the start of `'bar'`. This isn't the case on **line 6**, so the match fails there.

Similarly, there are matches on **lines 9 and 11** because a word boundary exists at the end of `'foo'`, but not on **line 14**.

Using the `\b` anchor on both ends of the `<regex>` will cause it to match when it's present in the search string as a whole word:

Python >>>

```
>>> re.search(r'\bbar\b', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search(r'\bbar\b', 'foo(bar)baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> print(re.search(r'\bbar\b', 'foobarbaz'))
None
```

This is another instance in which it pays to specify the `<regex>` as a raw string, as the above examples have done.

Because `'\b'` is an escape sequence for both string literals and regexes in Python, each use above would need to be double escaped as `'\\b'` if you didn't use raw strings. That wouldn't be the end of the world, but raw strings are tidier.

\B

Anchors a match to a location that isn't a word boundary.

`\B` does the opposite of `\b`. It asserts that the regex parser's current position must *not* be at the start or end of a word:

Python >>>

```
1  >>> print(re.search(r'\Bfoo\B', 'foo'))
2  None
3  >>> print(re.search(r'\Bfoo\B', '.foo.'))
4  None
5
6  >>> re.search(r'\Bfoo\B', 'barfoobaz')
7  <_sre.SRE_Match object; span=(3, 6), match='foo'>
```

In this case, a match happens on **line 7** because no word boundary exists at the start or end of `'foo'` in the search string `'barfoobaz'`.

Quantifiers

A **quantifier** metacharacter immediately follows a portion of a `<regex>` and indicates how many times that portion must occur for the match to succeed.

*

Matches zero or more repetitions of the preceding regex.

For example, `a*` matches zero or more `'a'` characters. That means it would match an empty string, `'a'`, `'aa'`, `'aaa'`, and so on.

Consider these examples:

Python >>>

```
1  >>> re.search('foo-*bar', 'foobar')                   # Zero dashes
2  <_sre.SRE_Match object; span=(0, 6), match='foobar'>
3  >>> re.search('foo-*bar', 'foo-bar')                  # One dash
4  <_sre.SRE_Match object; span=(0, 7), match='foo-bar'>
5  >>> re.search('foo-*bar', 'foo--bar')                 # Two dashes
6  <_sre.SRE_Match object; span=(0, 8), match='foo--bar'>
```

On **line 1**, there are zero `'-'` characters between `'foo'` and `'bar'`. On **line 3** there's one, and on **line 5** there are two. The metacharacter sequence `-*` matches in all three cases.

You'll probably encounter the regex `.*` in a Python program at some point. This matches zero or more occurrences of any character. In other words, it essentially matches any character sequence up to a line break. (Remember that the `.` wildcard metacharacter doesn't match a newline.)

In this example, `.*` matches everything between `'foo'` and `'bar'`:

Python >>>

```
>>> re.search('foo.*bar', '# foo $qux@grault % bar #')
<_sre.SRE_Match object; span=(2, 23), match='foo $qux@grault % bar'>
```

Did you notice the `span=` and `match=` information contained in the match object?

Until now, the regexes in the examples you've seen have specified matches of predictable length. Once you start using quantifiers like `*`, the number of characters matched can be quite variable, and the information in the match object becomes more useful.

You'll learn more about how to access the information stored in a match object in the next tutorial in the series.

`+`

Matches one or more repetitions of the preceding regex.

This is similar to `*`, but the quantified regex must occur at least once:

Python >>>

```
1  >>> print(re.search('foo-+bar', 'foobar'))            # Zero dashes
2  None
3  >>> re.search('foo-+bar', 'foo-bar')                  # One dash
4  <_sre.SRE_Match object; span=(0, 7), match='foo-bar'>
5  >>> re.search('foo-+bar', 'foo--bar')                 # Two dashes
6  <_sre.SRE_Match object; span=(0, 8), match='foo--bar'>
```

Remember from above that `foo-*bar` matched the string `'foobar'` because the `*` metacharacter allows for zero occurrences of `'-'`. The `+` metacharacter, on the other hand, requires at least one occurrence of `'-'`. That means there isn't a match on **line 1** in this case.

`?`

Matches zero or one repetitions of the preceding regex.

Again, this is similar to `*` and `+`, but in this case there's only a match if the preceding regex occurs once or not at all:

Python

```
1 >>> re.search('foo-?bar', 'foobar')                    # Zero dashes
2 <_sre.SRE_Match object; span=(0, 6), match='foobar'>
3 >>> re.search('foo-?bar', 'foo-bar')                   # One dash
4 <_sre.SRE_Match object; span=(0, 7), match='foo-bar'>
5 >>> print(re.search('foo-?bar', 'foo--bar'))           # Two dashes
6 None
```

In this example, there are matches on **lines 1 and 3**. But on **line 5**, where there are two `'-'` characters, the match fails.

Here are some more examples showing the use of all three quantifier metacharacters:

Python

```
>>> re.match('foo[1-9]*bar', 'foobar')
<_sre.SRE_Match object; span=(0, 6), match='foobar'>
>>> re.match('foo[1-9]*bar', 'foo42bar')
<_sre.SRE_Match object; span=(0, 8), match='foo42bar'>

>>> print(re.match('foo[1-9]+bar', 'foobar'))
None
>>> re.match('foo[1-9]+bar', 'foo42bar')
<_sre.SRE_Match object; span=(0, 8), match='foo42bar'>

>>> re.match('foo[1-9]?bar', 'foobar')
<_sre.SRE_Match object; span=(0, 6), match='foobar'>
>>> print(re.match('foo[1-9]?bar', 'foo42bar'))
None
```

This time, the quantified regex is the character class `[1-9]` instead of the simple character `'-'`.

`*?`
`+?`
`??`

> The non-greedy (or lazy) versions of the `*`, `+`, and `?` quantifiers.

When used alone, the quantifier metacharacters `*`, `+`, and `?` are all **greedy**, meaning they produce the longest possible match. Consider this example:

Python

```
>>> re.search('<.*>', '%<foo> <bar> <baz>%')
<_sre.SRE_Match object; span=(1, 18), match='<foo> <bar> <baz>'>
```

The regex `<.*>` effectively means:

- A `'<'` character
- Then any sequence of characters
- Then a `'>'` character

But which `'>'` character? There are three possibilities:

1. The one just after `'foo'`
2. The one just after `'bar'`
3. The one just after `'baz'`

Since the `*` metacharacter is greedy, it dictates the longest possible match, which includes everything up to and including the `'>'` character that follows `'baz'`. You can see from the match object that this is the match produced.

If you want the shortest possible match instead, then use the non-greedy metacharacter sequence `*?`:

Python >>>

```
>>> re.search('<.*?>', '%<foo> <bar> <baz>%')
<_sre.SRE_Match object; span=(1, 6), match='<foo>'>
```

In this case, the match ends with the `'>'` character following `'foo'`.

> **Note:** You could accomplish the same thing with the regex `<[^>]*>`, which means:
>
> - A `'<'` character
> - Then any sequence of characters other than `'>'`
> - Then a `'>'` character
>
> This is the only option available with some older parsers that don't support lazy quantifiers. Happily, that's not the case with the regex parser in Python's `re` module.

There are lazy versions of the `+` and `?` quantifiers as well:

Python >>>

```
1 >>> re.search('<.+>', '%<foo> <bar> <baz>%')
2 <_sre.SRE_Match object; span=(1, 18), match='<foo> <bar> <baz>'>
3 >>> re.search('<.+?>', '%<foo> <bar> <baz>%')
4 <_sre.SRE_Match object; span=(1, 6), match='<foo>'>
5
6 >>> re.search('ba?', 'baaaa')
7 <_sre.SRE_Match object; span=(0, 2), match='ba'>
8 >>> re.search('ba??', 'baaaa')
9 <_sre.SRE_Match object; span=(0, 1), match='b'>
```

The first two examples on **lines 1 and 3** are similar to the examples shown above, only using `+` and `+?` instead of `*` and `*?`.

The last examples on **lines 6 and 8** are a little different. In general, the `?` metacharacter matches zero or one occurrences of the preceding regex. The greedy version, `?`, matches one occurrence, so `ba?` matches `'b'` followed by a single `'a'`. The non-greedy version, `??`, matches zero occurrences, so `ba??` matches just `'b'`.

`{m}`

> Matches exactly `m` repetitions of the preceding regex.

This is similar to `*` or `+`, but it specifies exactly how many times the preceding regex must occur for a match to succeed:

Python >>>

```
>>> print(re.search('x-{3}x', 'x--x'))                # Two dashes
None

>>> re.search('x-{3}x', 'x---x')                      # Three dashes
<_sre.SRE_Match object; span=(0, 5), match='x---x'>

>>> print(re.search('x-{3}x', 'x----x'))              # Four dashes
None
```

Here, `x-{3}x` matches `'x'`, followed by exactly three instances of the `'-'` character, followed by another `'x'`. The match fails when there are fewer or more than three dashes between the `'x'` characters.

`{m,n}`

> Matches any number of repetitions of the preceding regex from `m` to `n`, inclusive.

In the following example, the quantified `<regex>` is `-{2,4}`. The match succeeds when there are two, three, or four dashes between the `'x'` characters but fails otherwise:

Python >>>

```
>>> for i in range(1, 6):
...     s = f"x{'-' * i}x"
...     print(f'{i}  {s:10}', re.search('x-{2,4}x', s))
...
1  x-x        None
2  x--x       <_sre.SRE_Match object; span=(0, 4), match='x--x'>
3  x---x      <_sre.SRE_Match object; span=(0, 5), match='x---x'>
4  x----x     <_sre.SRE_Match object; span=(0, 6), match='x----x'>
5  x-----x    None
```

Omitting `m` implies a lower bound of `0`, and omitting `n` implies an unlimited upper bound:

Regular Expression	Matches	Identical to
`<regex>{,n}`	Any number of repetitions of `<regex>` less than or equal to `n`	`<regex>{0,n}`
`<regex>{m,}`	Any number of repetitions of `<regex>` greater than or equal to `m`	`----`
`<regex>{,}`	Any number of repetitions of `<regex>`	`<regex>{0,}` `<regex>*`

If you omit all of `m`, `n`, and the comma, then the curly braces no longer function as metacharacters. `{}` matches just the literal string `'{}'`:

Python >>>

```
>>> re.search('x{}y', 'x{}y')
<_sre.SRE_Match object; span=(0, 4), match='x{}y'>
```

In fact, to have any special meaning, a sequence with curly braces must fit one of the following patterns in which `m` and `n` are nonnegative integers:

- `{m,n}`
- `{m,}`
- `{,n}`
- `{,}`

Otherwise, it matches literally:

Python >>>

```
>>> re.search('x{foo}y', 'x{foo}y')
<_sre.SRE_Match object; span=(0, 7), match='x{foo}y'>
>>> re.search('x{a:b}y', 'x{a:b}y')
<_sre.SRE_Match object; span=(0, 7), match='x{a:b}y'>
>>> re.search('x{1,3,5}y', 'x{1,3,5}y')
<_sre.SRE_Match object; span=(0, 9), match='x{1,3,5}y'>
>>> re.search('x{foo,bar}y', 'x{foo,bar}y')
<_sre.SRE_Match object; span=(0, 11), match='x{foo,bar}y'>
```

Later in this tutorial, when you learn about the `DEBUG` flag, you'll see how you can confirm this.

`{m,n}?`

The non-greedy (lazy) version of `{m,n}`.

`{m,n}` will match as many characters as possible, and `{m,n}?` will match as few as possible:

Python >>>

```
>>> re.search('a{3,5}', 'aaaaaaaa')
<_sre.SRE_Match object; span=(0, 5), match='aaaaa'>

>>> re.search('a{3,5}?', 'aaaaaaaa')
<_sre.SRE_Match object; span=(0, 3), match='aaa'>
```

In this case, `a{3,5}` produces the longest possible match, so it matches five `'a'` characters. `a{3,5}?` produces the shortest match, so it matches three.

Grouping Constructs and Backreferences

Grouping constructs break up a regex in Python into subexpressions or groups. This serves two purposes:

1. **Grouping:** A group represents a single syntactic entity. Additional metacharacters apply to the entire group as a unit.
2. **Capturing:** Some grouping constructs also capture the portion of the search string that matches the subexpression in the group. You can retrieve captured matches later through several different mechanisms.

Here's a look at how grouping and capturing work.

`(<regex>)`

Defines a subexpression or group.

This is the most basic grouping construct. A regex in parentheses just matches the contents of the parentheses:

Python >>>

```
>>> re.search('(bar)', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> re.search('bar', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>
```

As a regex, `(bar)` matches the string `'bar'`, the same as the regex `bar` would without the parentheses.

Treating a Group as a Unit

A quantifier metacharacter that follows a group operates on the entire subexpression specified in the group as a single unit.

For instance, the following example matches one or more occurrences of the string `'bar'`:

Python >>>

```
>>> re.search('(bar)+', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search('(bar)+', 'foo barbar baz')
<_sre.SRE_Match object; span=(4, 10), match='barbar'>
>>> re.search('(bar)+', 'foo barbarbarbar baz')
<_sre.SRE_Match object; span=(4, 16), match='barbarbarbar'>
```

Here's a breakdown of the difference between the two regexes with and without grouping parentheses:

Regex	Interpretation	Matches	Examples

`bar+`	The `+` metacharacter applies only to the character `'r'`.	`'ba'` followed by one or more occurrences of `'r'`	`'bar'`
			`'barr'`
			`'barrr'`

`(bar)+`	The `+` metacharacter applies to the entire string `'bar'`.	One or more occurrences of `'bar'`	`'bar'`
			`'barbar'`
			`'barbarbar'`

Now take a look at a more complicated example. The regex `(ba[rz]){2,4}(qux)?` matches `2` to `4` occurrences of either `'bar'` or `'baz'`, optionally followed by `'qux'`:

Python >>>

```
>>> re.search('(ba[rz]){2,4}(qux)?', 'bazbarbazqux')
<_sre.SRE_Match object; span=(0, 12), match='bazbarbazqux'>
>>> re.search('(ba[rz]){2,4}(qux)?', 'barbar')
<_sre.SRE_Match object; span=(0, 6), match='barbar'>
```

The following example shows that you can nest grouping parentheses:

Python >>>

```
>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoobar')
<_sre.SRE_Match object; span=(0, 9), match='foofoobar'>
>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoobar123')
<_sre.SRE_Match object; span=(0, 12), match='foofoobar123'>
>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoo123')
<_sre.SRE_Match object; span=(0, 9), match='foofoo123'>
```

The regex `(foo(bar)?)+(\d\d\d)?` is pretty elaborate, so let's break it down into smaller pieces:

Regex	Matches
`foo(bar)?`	`'foo'` optionally followed by `'bar'`
`(foo(bar)?)+`	One or more occurrences of the above
`\d\d\d`	Three decimal digit characters
`(\d\d\d)?`	Zero or one occurrences of the above

String it all together and you get: at least one occurrence of `'foo'` optionally followed by `'bar'`, all optionally followed by three decimal digit characters.

As you can see, you can construct very complicated regexes in Python using grouping parentheses.

Capturing Groups

Grouping isn't the only useful purpose that grouping constructs serve. Most (but not quite all) grouping constructs also capture the part of the search string that matches the group. You can retrieve the captured portion or refer to it later in several different ways.

Remember the match object that `re.search()` returns? There are two methods defined for a match object that provide access to captured groups: `.groups()` and `.group()`.

`m.groups()`

Returns a tuple containing all the captured groups from a regex match.

Consider this example:

Python >>>

```
>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')
>>> m
<_sre.SRE_Match object; span=(0, 12), match='foo:quux:baz'>
```

Each of the three `(\w+)` expressions matches a sequence of word characters. The full regex `(\w+),(\w+),(\w+)` breaks the search string into three comma-separated tokens.

Because the `(\w+)` expressions use grouping parentheses, the corresponding matching tokens are **captured**. To access the captured matches, you can use `.groups()`, which returns a tuple containing all the captured matches in order:

Python >>>

```
>>> m.groups()
('foo', 'quux', 'baz')
```

Notice that the tuple contains the tokens but not the commas that appeared in the search string. That's because the word characters that make up the tokens are inside the grouping parentheses but the commas aren't. The commas that you see between the returned tokens are the standard delimiters used to separate values in a tuple.

`m.group(<n>)`

Returns a string containing the `<n>`th captured match.

With one argument, `.group()` returns a single captured match. Note that the arguments are one-based, not zero-based. So, `m.group(1)` refers to the first captured match, `m.group(2)` to the second, and so on:

Python >>>

```
>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')
>>> m.groups()
('foo', 'quux', 'baz')

>>> m.group(1)
'foo'
>>> m.group(2)
'quux'
>>> m.group(3)
'baz'
```

Since the numbering of captured matches is one-based, and there isn't any group numbered zero, `m.group(0)` has a special meaning:

Python >>>

```
>>> m.group(0)
'foo,quux,baz'
>>> m.group()
'foo,quux,baz'
```

`m.group(0)` returns the entire match, and `m.group()` does the same.

`m.group(<n1>, <n2>, ...)`

> Returns a tuple containing the specified captured matches.

With multiple arguments, `.group()` returns a tuple containing the specified captured matches in the given order:

Python >>>

```
>>> m.groups()
('foo', 'quux', 'baz')

>>> m.group(2, 3)
('quux', 'baz')
>>> m.group(3, 2, 1)
('baz', 'quux', 'foo')
```

This is just convenient shorthand. You could create the tuple of matches yourself instead:

Python >>>

```
>>> m.group(3, 2, 1)
('baz', 'qux', 'foo')
>>> (m.group(3), m.group(2), m.group(1))
('baz', 'qux', 'foo')
```

The two statements shown are functionally equivalent.

Backreferences

You can match a previously captured group later within the same regex using a special metacharacter sequence called a **backreference**.

\<n>

> Matches the contents of a previously captured group.

Within a regex in Python, the sequence `\<n>`, where `<n>` is an integer from `1` to `99`, matches the contents of the `<n>`th captured group.

Here's a regex that matches a word, followed by a comma, followed by the same word again:

Python >>>

```
 1  >>> regex = r'(\w+),\1'
 2  
 3  >>> m = re.search(regex, 'foo,foo')
 4  >>> m
 5  <_sre.SRE_Match object; span=(0, 7), match='foo,foo'>
 6  >>> m.group(1)
 7  'foo'
 8  
 9  >>> m = re.search(regex, 'qux,qux')
10  >>> m
11  <_sre.SRE_Match object; span=(0, 7), match='qux,qux'>
12  >>> m.group(1)
13  'qux'
14  
15  >>> m = re.search(regex, 'foo,qux')
16  >>> print(m)
17  None
```

In the first example, on **line 3**, `(\w+)` matches the first instance of the string `'foo'` and saves it as the first captured group. The comma matches literally. Then `\1` is a backreference to the first captured group and matches `'foo'` again. The second example, on **line 9**, is identical except that the `(\w+)` matches `'qux'` instead.

The last example, on **line 15**, doesn't have a match because what comes before the comma isn't the same as what comes after it, so the `\1` backreference doesn't match.

Note: Any time you use a regex in Python with a numbered backreference, it's a good idea to specify it as a raw string. Otherwise, the interpreter may confuse the backreference with an octal value.

Consider this example:

Python >>>

```
>>> print(re.search('([a-z])#\1', 'd#d'))
None
```

The regex `([a-z])#\1` matches a lowercase letter, followed by `'#'`, followed by the same lowercase letter. The string in this case is `'d#d'`, which should match. But the match fails because Python misinterprets the backreference `\1` as the character whose octal value is one:

Python >>>

```
>>> oct(ord('\1'))
'0o1'
```

You'll achieve the correct match if you specify the regex as a raw string:

Python >>>

```
>>> re.search(r'([a-z])#\1', 'd#d')
<_sre.SRE_Match object; span=(0, 3), match='d#d'>
```

Remember to consider using a raw string whenever your regex includes a metacharacter sequence containing a backslash.

Numbered backreferences are one-based like the arguments to `.group()`. Only the first ninety-nine captured groups are accessible by backreference. The interpreter will regard `\100` as the `'@'` character, whose octal value is 100.

Other Grouping Constructs

The `(<regex>)` metacharacter sequence shown above is the most straightforward way to perform grouping within a regex in Python. The next section introduces you to some enhanced grouping constructs that allow you to tweak when and how grouping occurs.

`(?P<name><regex>)`

Creates a named captured group.

This metacharacter sequence is similar to grouping parentheses in that it creates a group matching `<regex>` that is accessible through the match object or a subsequent backreference. The difference in this case is that you reference the matched group by its given symbolic `<name>` instead of by its number.

Earlier, you saw this example with three captured groups numbered `1`, `2`, and `3`:

Python >>>

```
>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')
>>> m.groups()
('foo', 'quux', 'baz')

>>> m.group(1, 2, 3)
('foo', 'quux', 'baz')
```

The following effectively does the same thing except that the groups have the symbolic names `w1`, `w2`, and `w3`:

Python >>>

```
>>> m = re.search('(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'foo,quux,baz')
>>> m.groups()
('foo', 'quux', 'baz')
```

You can refer to these captured groups by their symbolic names:

Python >>>

```
>>> m.group('w1')
'foo'
>>> m.group('w3')
'baz'
>>> m.group('w1', 'w2', 'w3')
('foo', 'quux', 'baz')
```

You can still access groups with symbolic names by number if you wish:

Python >>>

```
>>> m = re.search('(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'foo,quux,baz')

>>> m.group('w1')
'foo'
>>> m.group(1)
'foo'

>>> m.group('w1', 'w2', 'w3')
('foo', 'quux', 'baz')
>>> m.group(1, 2, 3)
('foo', 'quux', 'baz')
```

Any `<name>` specified with this construct must conform to the rules for a Python identifier, and each `<name>` can only appear once per regex.

`(?P=<name>)`

Matches the contents of a previously captured named group.

The `(?P=<name>)` metacharacter sequence is a backreference, similar to `\<n>`, except that it refers to a named group rather than a numbered group.

Here again is the example from above, which uses a numbered backreference to match a word, followed by a comma, followed by the same word again:

Python >>>

```
>>> m = re.search(r'(\w+),\1', 'foo,foo')
>>> m
<_sre.SRE_Match object; span=(0, 7), match='foo,foo'>
>>> m.group(1)
'foo'
```

The following code does the same thing using a named group and a backreference instead:

Python >>>

```
>>> m = re.search(r'(?P<word>\w+),(?P=word)', 'foo,foo')
>>> m
<_sre.SRE_Match object; span=(0, 7), match='foo,foo'>
>>> m.group('word')
'foo'
```

`(?P=<word>\w+)` matches `'foo'` and saves it as a captured group named `word`. Again, the comma matches literally. Then `(?P=word)` is a backreference to the named capture and matches `'foo'` again.

Note: The angle brackets (`<` and `>`) are required around `name` when creating a named group but not when referring to it later, either by backreference or by `.group()`:

Python >>>

```
>>> m = re.match(r'(?P<num>\d+)\.(?P=num)', '135.135')
>>> m
<_sre.SRE_Match object; span=(0, 7), match='135.135'>

>>> m.group('num')
'135'
```

Here, `(?P`**`<num>`**`\d+)` creates the captured group. But the corresponding backreference is `(?P=`**`num`**`)` without the angle brackets.

`(?:<regex>)`

Creates a non-capturing group.

`(?:<regex>)` is just like `(<regex>)` in that it matches the specified `<regex>`. But `(?:<regex>)` doesn't capture the match for later retrieval:

Python >>>

```
>>> m = re.search('(\w+),(?:\w+),(\w+)', 'foo,quux,baz')
>>> m.groups()
('foo', 'baz')

>>> m.group(1)
'foo'
>>> m.group(2)
'baz'
```

In this example, the middle word `'quux'` sits inside non-capturing parentheses, so it's missing from the tuple of captured groups. It isn't retrievable from the match object, nor would it be referable by backreference.

Why would you want to define a group but not capture it?

Remember that the regex parser will treat the `<regex>` inside grouping parentheses as a single unit. You may have a situation where you need this grouping feature, but you don't need to do anything with the value later, so you don't really need to capture it. If you use non-capturing grouping, then the tuple of captured groups won't be cluttered with values you don't actually need to keep.

Additionally, it takes some time and memory to capture a group. If the code that performs the match executes many times and you don't capture groups that you aren't going to use later, then you may see a slight performance advantage.

`(?(<n>)<yes-regex>|<no-regex>)`
`(?(<name>)<yes-regex>|<no-regex>)`

Specifies a conditional match.

A conditional match matches against one of two specified regexes depending on whether the given group exists:

- `(?(<n>)<yes-regex>|<no-regex>)` matches against `<yes-regex>` if a group numbered `<n>` exists. Otherwise, it matches against `<no-regex>`.

- `(?(<name>)<yes-regex>|<no-regex>)` matches against `<yes-regex>` if a group named `<name>` exists. Otherwise, it matches against `<no-regex>`.

Conditional matches are better illustrated with an example. Consider this regex:

Python

```
regex = r'^(###)?foo(?(1)bar|baz)'
```

Here are the parts of this regex broken out with some explanation:

1. `^(###)?` indicates that the search string optionally begins with `'###'`. If it does, then the grouping parentheses around `###` will create a group numbered `1`. Otherwise, no such group will exist.
2. The next portion, `foo`, literally matches the string `'foo'`.
3. Lastly, `(?(1)bar|baz)` matches against `'bar'` if group `1` exists and `'baz'` if it doesn't.

The following code blocks demonstrate the use of the above regex in several different Python code snippets:

Example 1:

Python >>>

```
>>> re.search(regex, '###foobar')
<_sre.SRE_Match object; span=(0, 9), match='###foobar'>
```

The search string `'###foobar'` does start with `'###'`, so the parser creates a group numbered `1`. The conditional match is then against `'bar'`, which matches.

Example 2:

Python >>>

```
>>> print(re.search(regex, '###foobaz'))
None
```

The search string `'###foobaz'` does start with `'###'`, so the parser creates a group numbered `1`. The conditional match is then against `'bar'`, which doesn't match.

Example 3:

Python >>>

```
>>> print(re.search(regex, 'foobar'))
None
```

The search string `'foobar'` doesn't start with `'###'`, so there isn't a group numbered `1`. The conditional match is then against `'baz'`, which doesn't match.

Example 4:

Python >>>

```
>>> re.search(regex, 'foobaz')
<_sre.SRE_Match object; span=(0, 6), match='foobaz'>
```

The search string `'foobaz'` doesn't start with `'###'`, so there isn't a group numbered `1`. The conditional match is then against `'baz'`, which matches.

Here's another conditional match using a named group instead of a numbered group:

Python >>>

```
>>> regex = r'^(?P<ch>\W)?foo(?(ch)(?P=ch)|)$'
```

This regex matches the string `'foo'`, preceded by a single non-word character and followed by the same non-word character, or the string `'foo'` by itself.

Again, let's break this down into pieces:

Regex	Matches
`^`	The start of the string
`(?P<ch>\W)`	A single non-word character, captured in a group named `ch`
`(?P<ch>\W)?`	Zero or one occurrences of the above
`foo`	The literal string `'foo'`
`(?(ch)(?P=ch)|)`	The contents of the group named `ch` if it exists, or the empty string if it doesn't
`$`	The end of the string

If a non-word character precedes `'foo'`, then the parser creates a group named `ch` which contains that character. The conditional match then matches against `<yes-regex>`, which is `(?P=ch)`, the same character again. That means the same character must also follow `'foo'` for the entire match to succeed.

If `'foo'` isn't preceded by a non-word character, then the parser doesn't create group `ch`. `<no-regex>` is the empty string, which means there must not be anything following `'foo'` for the entire match to succeed. Since `^` and `$` anchor the whole regex, the string must equal `'foo'` exactly.

Here are some examples of searches using this regex in Python code:

Python >>>

```
>>> re.search(regex, 'foo')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.search(regex, '#foo#')
<_sre.SRE_Match object; span=(0, 5), match='#foo#'>
>>> re.search(regex, '@foo@')
<_sre.SRE_Match object; span=(0, 5), match='@foo@'>

>>> print(re.search(regex, '#foo'))
None
>>> print(re.search(regex, 'foo@'))
None
>>> print(re.search(regex, '#foo@'))
None
>>> print(re.search(regex, '@foo#'))
None
```

On **line 1**, `'foo'` is by itself. On **lines 3 and 5**, the same non-word character precedes and follows `'foo'`. As advertised, these matches succeed.

In the remaining cases, the matches fail.

Conditional regexes in Python are pretty esoteric and challenging to work through. If you ever do find a reason to use one, then you could probably accomplish the same goal with multiple separate `re.search()` calls, and your code would be less complicated to read and understand.

Lookahead and Lookbehind Assertions

Lookahead and **lookbehind** assertions determine the success or failure of a regex match in Python based on what is just behind (to the left) or ahead (to the right) of the parser's current position in the search string.

Like anchors, lookahead and lookbehind assertions are zero-width assertions, so they don't consume any of the search string. Also, even though they contain parentheses and perform grouping, they don't capture what they match.

`(?=<lookahead_regex>)`

> Creates a positive lookahead assertion.

`(?=<lookahead_regex>)` asserts that what follows the regex parser's current position must match `<lookahead_regex>`:

Python >>>

```
>>> re.search('foo(?=[a-z])', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

The lookahead assertion `(?=[a-z])` specifies that what follows `'foo'` must be a lowercase alphabetic character. In this case, it's the character `'b'`, so a match is found.

In the next example, on the other hand, the lookahead fails. The next character after `'foo'` is `'1'`, so there isn't a match:

Python >>>

```
>>> print(re.search('foo(?=[a-z])', 'foo123'))
None
```

What's unique about a lookahead is that the portion of the search string that matches `<lookahead_regex>` isn't consumed, and it isn't part of the returned match object.

Take another look at the first example:

Python >>>

```
>>> re.search('foo(?=[a-z])', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

The regex parser looks ahead only to the `'b'` that follows `'foo'` but doesn't pass over it yet. You can tell that `'b'` isn't considered part of the match because the match object displays `match='foo'`.

Compare that to a similar example that uses grouping parentheses without a lookahead:

Python >>>

```
>>> re.search('foo([a-z])', 'foobar')
<_sre.SRE_Match object; span=(0, 4), match='foob'>
```

This time, the regex consumes the `'b'`, and it becomes a part of the eventual match.

Here’s another example illustrating how a lookahead differs from a conventional regex in Python:

Python >>>

```
1 >>> m = re.search('foo(?=[a-z])(?P<ch>.)', 'foobar')
2 >>> m.group('ch')
3 'b'
4
5 >>> m = re.search('foo([a-z])(?P<ch>.)', 'foobar')
6 >>> m.group('ch')
7 'a'
```

In the first search, on **line 1**, the parser proceeds as follows:

1. The first portion of the regex, `foo`, matches and consumes `'foo'` from the search string `'foobar'`.
2. The next portion, `(?=[a-z])`, is a lookahead that matches `'b'`, but the parser doesn’t advance past the `'b'`.
3. Lastly, `(?P<ch>.)` matches the next single character available, which is `'b'`, and captures it in a group named `ch`.

The `m.group('ch')` call confirms that the group named `ch` contains `'b'`.

Compare that to the search on **line 5**, which doesn’t contain a lookahead:

1. As in the first example, the first portion of the regex, `foo`, matches and consumes `'foo'` from the search string `'foobar'`.
2. The next portion, `([a-z])`, matches and consumes `'b'`, and the parser advances past `'b'`.
3. Lastly, `(?P<ch>.)` matches the next single character available, which is now `'a'`.

`m.group('ch')` confirms that, in this case, the group named `ch` contains `'a'`.

`(?!<lookahead_regex>)`

> Creates a negative lookahead assertion.

`(?!<lookahead_regex>)` asserts that what follows the regex parser’s current position must *not* match `<lookahead_regex>`.

Here are the positive lookahead examples you saw earlier, along with their negative lookahead counterparts:

Python >>>

```
1 >>> re.search('foo(?=[a-z])', 'foobar')
2 <_sre.SRE_Match object; span=(0, 3), match='foo'>
3 >>> print(re.search('foo(?![a-z])', 'foobar'))
4 None
5
6 >>> print(re.search('foo(?=[a-z])', 'foo123'))
7 None
8 >>> re.search('foo(?![a-z])', 'foo123')
9 <_sre.SRE_Match object; span=(0, 3), match='foo'>
```

The negative lookahead assertions on **lines 3 and 8** stipulate that what follows `'foo'` should not be a lowercase alphabetic character. This fails on **line 3** but succeeds on **line 8**. This is the opposite of what happened with the corresponding positive lookahead assertions.

As with a positive lookahead, what matches a negative lookahead isn’t part of the returned match object and isn’t consumed.

`(?<=<lookbehind_regex>)`

Creates a positive lookbehind assertion.

`(?<=<lookbehind_regex>)` asserts that what precedes the regex parser's current position must match `<lookbehind_regex>`.

In the following example, the lookbehind assertion specifies that `'foo'` must precede `'bar'`:

Python >>>

```
>>> re.search('(?<=foo)bar', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
```

This is the case here, so the match succeeds. As with lookahead assertions, the part of the search string that matches the lookbehind doesn't become part of the eventual match.

The next example fails to match because the lookbehind requires that `'qux'` precede `'bar'`:

Python >>>

```
>>> print(re.search('(?<=qux)bar', 'foobar'))
None
```

There's a restriction on lookbehind assertions that doesn't apply to lookahead assertions. The `<lookbehind_regex>` in a lookbehind assertion must specify a match of fixed length.

For example, the following isn't allowed because the length of the string matched by `a+` is indeterminate:

Python >>>

```
>>> re.search('(?<=a+)def', 'aaadef')
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    re.search('(?<=a+)def', 'aaadef')
  File "C:\Python36\lib\re.py", line 182, in search
    return _compile(pattern, flags).search(string)
  File "C:\Python36\lib\re.py", line 301, in _compile
    p = sre_compile.compile(pattern, flags)
  File "C:\Python36\lib\sre_compile.py", line 566, in compile
    code = _code(p, flags)
  File "C:\Python36\lib\sre_compile.py", line 551, in _code
    _compile(code, p.data, flags)
  File "C:\Python36\lib\sre_compile.py", line 160, in _compile
    raise error("look-behind requires fixed-width pattern")
sre_constants.error: look-behind requires fixed-width pattern
```

This, however, is okay:

Python >>>

```
>>> re.search('(?<=a{3})def', 'aaadef')
<_sre.SRE_Match object; span=(3, 6), match='def'>
```

Anything that matches `a{3}` will have a fixed length of three, so `a{3}` is valid in a lookbehind assertion.

`(?<!--<lookbehind_regex-->)`

Creates a negative lookbehind assertion.

`(?<!--<lookbehind_regex-->)` asserts that what precedes the regex parser's current position must *not* match `<lookbehind_regex>`:

Python >>>

```
>>> print(re.search('(?<!foo)bar', 'foobar'))
None

>>> re.search('(?<!qux)bar', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
```

As with the positive lookbehind assertion, `<lookbehind_regex>` must specify a match of fixed length.

Miscellaneous Metacharacters

There are a couple more metacharacter sequences to cover. These are stray metacharacters that don't obviously fall into any of the categories already discussed.

`(?#...)`

> Specifies a comment.

The regex parser ignores anything contained in the sequence `(?#...)`:

Python >>>

```
>>> re.search('bar(?#This is a comment) *baz', 'foo bar baz qux')
<_sre.SRE_Match object; span=(4, 11), match='bar baz'>
```

This allows you to specify documentation inside a regex in Python, which can be especially useful if the regex is particularly long.

Vertical bar, or pipe (`|`)

> Specifies a set of alternatives on which to match.

An expression of the form `<regex`$_1$`>|<regex`$_2$`>|...|<regex`$_n$`>` matches at most one of the specified `<regex`$_i$`>` expressions:

Python >>>

```
>>> re.search('foo|bar|baz', 'bar')
<_sre.SRE_Match object; span=(0, 3), match='bar'>

>>> re.search('foo|bar|baz', 'baz')
<_sre.SRE_Match object; span=(0, 3), match='baz'>

>>> print(re.search('foo|bar|baz', 'quux'))
None
```

Here, `foo|bar|baz` will match any of `'foo'`, `'bar'`, or `'baz'`. You can separate any number of regexes using `|`.

Alternation is non-greedy. The regex parser looks at the expressions separated by `|` in left-to-right order and returns the first match that it finds. The remaining expressions aren't tested, even if one of them would produce a longer match:

Python

```
1 >>> re.search('foo', 'foograult')
2 <_sre.SRE_Match object; span=(0, 3), match='foo'>
3 >>> re.search('grault', 'foograult')
4 <_sre.SRE_Match object; span=(3, 9), match='grault'>
5
6 >>> re.search('foo|grault', 'foograult')
7 <_sre.SRE_Match object; span=(0, 3), match='foo'>
```

In this case, the pattern specified on **line 6**, `'foo|grault'`, would match on either `'foo'` or `'grault'`. The match returned is `'foo'` because that appears first when scanning from left to right, even though `'grault'` would be a longer match.

You can combine alternation, grouping, and any other metacharacters to achieve whatever level of complexity you need. In the following example, `(foo|bar|baz)+` means a sequence of one or more of the strings `'foo'`, `'bar'`, or `'baz'`:

Python

```
>>> re.search('(foo|bar|baz)+', 'foofoofoo')
<_sre.SRE_Match object; span=(0, 9), match='foofoofoo'>
>>> re.search('(foo|bar|baz)+', 'bazbazbazbaz')
<_sre.SRE_Match object; span=(0, 12), match='bazbazbazbaz'>
>>> re.search('(foo|bar|baz)+', 'barbazfoo')
<_sre.SRE_Match object; span=(0, 9), match='barbazfoo'>
```

In the next example, `([0-9]+|[a-f]+)` means a sequence of one or more decimal digit characters or a sequence of one or more of the characters `'a-f'`:

Python

```
>>> re.search('([0-9]+|[a-f]+)', '456')
<_sre.SRE_Match object; span=(0, 3), match='456'>
>>> re.search('([0-9]+|[a-f]+)', 'ffda')
<_sre.SRE_Match object; span=(0, 4), match='ffda'>
```

With all the metacharacters that the `re` module supports, the sky is practically the limit.

That’s All, Folks!

That completes our tour of the regex metacharacters supported by Python’s `re` module. (Actually, it doesn’t quite—there are a couple more stragglers you’ll learn about below in the discussion on flags.)

It’s a lot to digest, but once you become familiar with regex syntax in Python, the complexity of pattern matching that you can perform is almost limitless. These tools come in very handy when you’re writing code to process textual data.

If you’re new to regexes and want more practice working with them, or if you’re developing an application that uses a regex and you want to test it interactively, then check out the Regular Expressions 101 website. It’s seriously cool!

Modified Regular Expression Matching With Flags

Most of the functions in the `re` module take an optional `<flags>` argument. This includes the function you’re now very familiar with, `re.search()`.

`re.search(<regex>, <string>, <flags>)`

> Scans a string for a regex match, applying the specified modifier `<flags>`.

Flags modify regex parsing behavior, allowing you to refine your pattern matching even further.

Supported Regular Expression Flags

The table below briefly summarizes the available flags. All flags except `re.DEBUG` have a short, single-letter name and also a longer, full-word name:

Short Name	Long Name	Effect
`re.I`	`re.IGNORECASE`	Makes matching of alphabetic characters case-insensitive
`re.M`	`re.MULTILINE`	Causes start-of-string and end-of-string anchors to match embedded newlines
`re.S`	`re.DOTALL`	Causes the dot metacharacter to match a newline
`re.X`	`re.VERBOSE`	Allows inclusion of whitespace and comments within a regular expression
`----`	`re.DEBUG`	Causes the regex parser to display debugging information to the console
`re.A`	`re.ASCII`	Specifies ASCII encoding for character classification
`re.U`	`re.UNICODE`	Specifies Unicode encoding for character classification
`re.L`	`re.LOCALE`	Specifies encoding for character classification based on the current locale

The following sections describe in more detail how these flags affect matching behavior.

`re.I`
`re.IGNORECASE`

> Makes matching case insensitive.

When `IGNORECASE` is in effect, character matching is case insensitive:

Python

```
1 >>> re.search('a+', 'aaaAAA')
2 <_sre.SRE_Match object; span=(0, 3), match='aaa'>
3 >>> re.search('A+', 'aaaAAA')
4 <_sre.SRE_Match object; span=(3, 6), match='AAA'>
5
6 >>> re.search('a+', 'aaaAAA', re.I)
7 <_sre.SRE_Match object; span=(0, 6), match='aaaAAA'>
8 >>> re.search('A+', 'aaaAAA', re.IGNORECASE)
9 <_sre.SRE_Match object; span=(0, 6), match='aaaAAA'>
```

In the search on **line 1**, `a+` matches only the first three characters of `'aaaAAA'`. Similarly, on **line 3**, `A+` matches only the last three characters. But in the subsequent searches, the parser ignores case, so both `a+` and `A+` match the entire string.

`IGNORECASE` affects alphabetic matching involving character classes as well:

Python

```
>>> re.search('[a-z]+', 'aBcDeF')
<_sre.SRE_Match object; span=(0, 1), match='a'>
>>> re.search('[a-z]+', 'aBcDeF', re.I)
<_sre.SRE_Match object; span=(0, 6), match='aBcDeF'>
```

When case is significant, the longest portion of `'aBcDeF'` that `[a-z]+` matches is just the initial `'a'`. Specifying `re.I` makes the search case insensitive, so `[a-z]+` matches the entire string.

`re.M`
`re.MULTILINE`

> Causes start-of-string and end-of-string anchors to match at embedded newlines.

By default, the `^` (start-of-string) and `$` (end-of-string) anchors match only at the beginning and end of the search string:

Python >>>

```
>>> s = 'foo\nbar\nbaz'

>>> re.search('^foo', s)
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('^bar', s))
None
>>> print(re.search('^baz', s))
None

>>> print(re.search('foo$', s))
None
>>> print(re.search('bar$', s))
None
>>> re.search('baz$', s)
<_sre.SRE_Match object; span=(8, 11), match='baz'>
```

In this case, even though the search string `'foo\nbar\nbaz'` contains embedded newline characters, only `'foo'` matches when anchored at the beginning of the string, and only `'baz'` matches when anchored at the end.

If a string has embedded newlines, however, you can think of it as consisting of multiple internal lines. In that case, if the `MULTILINE` flag is set, the `^` and `$` anchor metacharacters match internal lines as well:

- **^** matches at the beginning of the string or at the beginning of any line within the string (that is, immediately following a newline).
- **$** matches at the end of the string or at the end of any line within the string (immediately preceding a newline).

The following are the same searches as shown above:

Python >>>

```
>>> s = 'foo\nbar\nbaz'
>>> print(s)
foo
bar
baz

>>> re.search('^foo', s, re.MULTILINE)
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.search('^bar', s, re.MULTILINE)
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search('^baz', s, re.MULTILINE)
<_sre.SRE_Match object; span=(8, 11), match='baz'>

>>> re.search('foo$', s, re.M)
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.search('bar$', s, re.M)
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search('baz$', s, re.M)
<_sre.SRE_Match object; span=(8, 11), match='baz'>
```

In the string `'foo\nbar\nbaz'`, all three of `'foo'`, `'bar'`, and `'baz'` occur at either the start or end of the string or at the start or end of a line within the string. With the `MULTILINE` flag set, all three match when anchored with either `^` or `$`.

Note: The `MULTILINE` flag only modifies the `^` and `$` anchors in this way. It doesn't have any effect on the `\A` and `\Z` anchors:

Python >>>

```
 1  >>> s = 'foo\nbar\nbaz'
 2
 3  >>> re.search('^bar', s, re.MULTILINE)
 4  <_sre.SRE_Match object; span=(4, 7), match='bar'>
 5  >>> re.search('bar$', s, re.MULTILINE)
 6  <_sre.SRE_Match object; span=(4, 7), match='bar'>
 7
 8  >>> print(re.search('\Abar', s, re.MULTILINE))
 9  None
10  >>> print(re.search('bar\Z', s, re.MULTILINE))
11  None
```

On **lines 3 and 5**, the `^` and `$` anchors dictate that `'bar'` must be found at the start and end of a line. Specifying the `MULTILINE` flag makes these matches succeed.

The examples on **lines 8 and 10** use the `\A` and `\Z` flags instead. You can see that these matches fail even with the `MULTILINE` flag in effect.

`re.S`
`re.DOTALL`

Causes the dot (`.`) metacharacter to match a newline.

Remember that by default, the dot metacharacter matches any character except the newline character. The `DOTALL` flag lifts this restriction:

Python >>>

```
1 >>> print(re.search('foo.bar', 'foo\nbar'))
2 None
3 >>> re.search('foo.bar', 'foo\nbar', re.DOTALL)
4 <_sre.SRE_Match object; span=(0, 7), match='foo\nbar'>
5 >>> re.search('foo.bar', 'foo\nbar', re.S)
6 <_sre.SRE_Match object; span=(0, 7), match='foo\nbar'>
```

In this example, on **line 1** the dot metacharacter doesn't match the newline in `'foo\nbar'`. On **lines 3 and 5**, `DOTALL` is in effect, so the dot does match the newline. Note that the short name of the `DOTALL` flag is `re.S`, not `re.D` as you might expect.

`re.X`
`re.VERBOSE`

> Allows inclusion of whitespace and comments within a regex.

The `VERBOSE` flag specifies a few special behaviors:

- The regex parser ignores all whitespace unless it's within a character class or escaped with a backslash.
- If the regex contains a `#` character that isn't contained within a character class or escaped with a backslash, then the parser ignores it and all characters to the right of it.

What's the use of this? It allows you to format a regex in Python so that it's more readable and self-documenting.

Here's an example showing how you might put this to use. Suppose you want to parse phone numbers that have the following format:

- Optional three-digit area code, in parentheses
- Optional whitespace
- Three-digit prefix
- Separator (either `'-'` or `'.'`)
- Four-digit line number

The following regex does the trick:

Python >>>

```
>>> regex = r'^(\(\d{3}\))?\s*\d{3}[-.]\d{4}$'

>>> re.search(regex, '414.9229')
<_sre.SRE_Match object; span=(0, 8), match='414.9229'>
>>> re.search(regex, '414-9229')
<_sre.SRE_Match object; span=(0, 8), match='414-9229'>
>>> re.search(regex, '(712)414-9229')
<_sre.SRE_Match object; span=(0, 13), match='(712)414-9229'>
>>> re.search(regex, '(712) 414-9229')
<_sre.SRE_Match object; span=(0, 14), match='(712) 414-9229'>
```

But `r'^(\(\d{3}\))?\s*\d{3}[-.]\d{4}$'` is an eyeful, isn't it? Using the `VERBOSE` flag, you can write the same regex in Python like this instead:

Python

```
>>> regex = r'''^              # Start of string
...             (\(\d{3}\))?    # Optional area code
...             \s*             # Optional whitespace
...             \d{3}           # Three-digit prefix
...             [-.]            # Separator character
...             \d{4}           # Four-digit line number
...             $               # Anchor at end of string
...             '''

>>> re.search(regex, '414.9229', re.VERBOSE)
<_sre.SRE_Match object; span=(0, 8), match='414.9229'>
>>> re.search(regex, '414-9229', re.VERBOSE)
<_sre.SRE_Match object; span=(0, 8), match='414-9229'>
>>> re.search(regex, '(712)414-9229', re.X)
<_sre.SRE_Match object; span=(0, 13), match='(712)414-9229'>
>>> re.search(regex, '(712) 414-9229', re.X)
<_sre.SRE_Match object; span=(0, 14), match='(712) 414-9229'>
```

The `re.search()` calls are the same as those shown above, so you can see that this regex works the same as the one specified earlier. But it's less difficult to understand at first glance.

Note that triple quoting makes it particularly convenient to include embedded newlines, which qualify as ignored whitespace in `VERBOSE` mode.

When using the `VERBOSE` flag, be mindful of whitespace that you do intend to be significant. Consider these examples:

Python

```
 1  >>> re.search('foo bar', 'foo bar')
 2  <_sre.SRE_Match object; span=(0, 7), match='foo bar'>
 3
 4  >>> print(re.search('foo bar', 'foo bar', re.VERBOSE))
 5  None
 6
 7  >>> re.search('foo\ bar', 'foo bar', re.VERBOSE)
 8  <_sre.SRE_Match object; span=(0, 7), match='foo bar'>
 9  >>> re.search('foo[ ]bar', 'foo bar', re.VERBOSE)
10  <_sre.SRE_Match object; span=(0, 7), match='foo bar'>
```

After all you've seen to this point, you may be wondering why on **line 4** the regex `foo bar` doesn't match the string `'foo bar'`. It doesn't because the `VERBOSE` flag causes the parser to ignore the space character.

To make this match as expected, escape the space character with a backslash or include it in a character class, as shown on **lines 7 and 9**.

As with the `DOTALL` flag, note that the `VERBOSE` flag has a non-intuitive short name: `re.X`, not `re.V`.

`re.DEBUG`

> Displays debugging information.

The `DEBUG` flag causes the regex parser in Python to display debugging information about the parsing process to the console:

Python >>>

```
>>> re.search('foo.bar', 'fooxbar', re.DEBUG)
LITERAL 102
LITERAL 111
LITERAL 111
ANY None
LITERAL 98
LITERAL 97
LITERAL 114
<_sre.SRE_Match object; span=(0, 7), match='fooxbar'>
```

When the parser displays `LITERAL nnn` in the debugging output, it's showing the ASCII code of a literal character in the regex. In this case, the literal characters are `'f'`, `'o'`, `'o'` and `'b'`, `'a'`, `'r'`.

Here's a more complicated example. This is the phone number regex shown in the discussion on the `VERBOSE` flag earlier:

Python >>>

```
>>> regex = r'^(\(\d{3}\))?\s*\d{3}[-.]\d{4}$'

>>> re.search(regex, '414.9229', re.DEBUG)
AT AT_BEGINNING
MAX_REPEAT 0 1
  SUBPATTERN 1 0 0
    LITERAL 40
    MAX_REPEAT 3 3
      IN
        CATEGORY CATEGORY_DIGIT
    LITERAL 41
MAX_REPEAT 0 MAXREPEAT
  IN
    CATEGORY CATEGORY_SPACE
MAX_REPEAT 3 3
  IN
    CATEGORY CATEGORY_DIGIT
IN
  LITERAL 45
  LITERAL 46
MAX_REPEAT 4 4
  IN
    CATEGORY CATEGORY_DIGIT
AT AT_END
<_sre.SRE_Match object; span=(0, 8), match='414.9229'>
```

This looks like a lot of esoteric information that you'd never need, but it can be useful. See the Deep Dive below for a practical application.

Deep Dive: Debugging Regular Expression Parsing

As you know from above, the metacharacter sequence `{m,n}` indicates a specific number of repetitions. It matches anywhere from `m` to `n` repetitions of what precedes it:

Python >>>

```
>>> re.search('x[123]{2,4}y', 'x222y')
<_sre.SRE_Match object; span=(0, 5), match='x222y'>
```

You can verify this with the `DEBUG` flag:

Python >>>

```
>>> re.search('x[123]{2,4}y', 'x222y', re.DEBUG)
LITERAL 120
MAX_REPEAT 2 4
  IN
    LITERAL 49
    LITERAL 50
    LITERAL 51
LITERAL 121
<_sre.SRE_Match object; span=(0, 5), match='x222y'>
```

`MAX_REPEAT 2 4` confirms that the regex parser recognizes the metacharacter sequence `{2,4}` and interprets it as a range quantifier.

But, as noted previously, if a pair of curly braces in a regex in Python contains anything other than a valid number or numeric range, then it loses its special meaning.

You can verify this also:

Python >>>

```
>>> re.search('x[123]{foo}y', 'x222y', re.DEBUG)
LITERAL 120
IN
  LITERAL 49
  LITERAL 50
  LITERAL 51
LITERAL 123
LITERAL 102
LITERAL 111
LITERAL 111
LITERAL 125
LITERAL 121
```

You can see that there's no `MAX_REPEAT` token in the debug output. The `LITERAL` tokens indicate that the parser treats `{foo}` literally and not as a quantifier metacharacter sequence. `123`, `102`, `111`, `111`, and `125` are the ASCII codes for the characters in the literal string `'{foo}'`.

Information displayed by the `DEBUG` flag can help you troubleshoot by showing you how the parser is interpreting your regex.

Curiously, the `re` module doesn't define a single-letter version of the `DEBUG` flag. You could define your own if you wanted to:

Python >>>

```
>>> import re
>>> re.D
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 're' has no attribute 'D'

>>> re.D = re.DEBUG
>>> re.search('foo', 'foo', re.D)
LITERAL 102
LITERAL 111
LITERAL 111
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

But this might be more confusing than helpful, as readers of your code might misconstrue it as an abbreviation for the `DOTALL` flag. If you did make this assignment, it would be a good idea to document it thoroughly.

`re.A`
`re.ASCII`
`re.U`
`re.UNICODE`
`re.L`
`re.LOCALE`

> Specify the character encoding used for parsing of special regex character classes.

Several of the regex metacharacter sequences (`\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s`, and `\S`) require you to assign characters to certain classes like word, digit, or whitespace. The flags in this group determine the encoding scheme used to assign characters to these classes. The possible encodings are ASCII, Unicode, or according to the current locale.

You had a brief introduction to character encoding and Unicode in the tutorial on Strings and Character Data in Python, under the discussion of the `ord()` built-in function. For more in-depth information, check out these resources:

- Unicode & Character Encodings in Python: A Painless Guide
- Python's Unicode Support

Why is character encoding so important in the context of regexes in Python? Here's a quick example.

You learned earlier that `\d` specifies a single digit character. The description of the `\d` metacharacter sequence states that it's equivalent to the character class `[0-9]`. That happens to be true for English and Western European languages, but for most of the world's languages, the characters `'0'` through `'9'` don't represent all or even *any* of the digits.

For example, here's a string that consists of three Devanagari digit characters:

Python >>>

```
>>> s = '\u0967\u096a\u096c'
>>> s
'१४६'
```

For the regex parser to properly account for the Devanagari script, the digit metacharacter sequence `\d` must match each of these characters as well.

The Unicode Consortium created Unicode to handle this problem. Unicode is a character-encoding standard designed to represent all the world's writing systems. All strings in Python 3, including regexes, are Unicode by default.

So then, back to the flags listed above. These flags help to determine whether a character falls into a given class by specifying whether the encoding used is ASCII, Unicode, or the current locale:

- **`re.U`** and **`re.UNICODE`** specify Unicode encoding. Unicode is the default, so these flags are superfluous. They're mainly supported for backward compatibility.
- **`re.A`** and **`re.ASCII`** force a determination based on ASCII encoding. If you happen to be operating in English, then this is happening anyway, so the flag won't affect whether or not a match is found.
- **`re.L`** and **`re.LOCALE`** make the determination based on the current locale. Locale is an outdated concept and isn't considered reliable. Except in rare circumstances, you're not likely to need it.

Using the default Unicode encoding, the regex parser should be able to handle any language you throw at it. In the following example, it correctly recognizes each of the characters in the string `'१४६'` as a digit:

Python

```
>>> s = '\u0967\u096a\u096c'
>>> s
'१४६'
>>> re.search('\d+', s)
<_sre.SRE_Match object; span=(0, 3), match='१४६'>
```

Here's another example that illustrates how character encoding can affect a regex match in Python. Consider this string:

Python

```
>>> s = 'sch\u00f6n'
>>> s
'schön'
```

`'schön'` (the German word for *pretty* or *nice*) contains the `'ö'` character, which has the 16-bit hexadecimal Unicode value `00f6`. This character isn't representable in traditional 7-bit ASCII.

If you're working in German, then you should reasonably expect the regex parser to consider all of the characters in `'schön'` to be word characters. But take a look at what happens if you search `s` for word characters using the `\w` character class and force an ASCII encoding:

Python

```
>>> re.search('\w+', s, re.ASCII)
<_sre.SRE_Match object; span=(0, 3), match='sch'>
```

When you restrict the encoding to ASCII, the regex parser recognizes only the first three characters as word characters. The match stops at `'ö'`.

On the other hand, if you specify `re.UNICODE` or allow the encoding to default to Unicode, then all the characters in `'schön'` qualify as word characters:

Python

```
>>> re.search('\w+', s, re.UNICODE)
<_sre.SRE_Match object; span=(0, 5), match='schön'>
>>> re.search('\w+', s)
<_sre.SRE_Match object; span=(0, 5), match='schön'>
```

The `ASCII` and `LOCALE` flags are available in case you need them for special circumstances. But in general, the best strategy is to use the default Unicode encoding. This should handle any world language correctly.

Combining `<flags>` Arguments in a Function Call

Flag values are defined so that you can combine them using the bitwise OR (`|`) operator. This allows you to specify several flags in a single function call:

Python >>>

```
>>> re.search('^bar', 'FOO\nBAR\nBAZ', re.I|re.M)
<_sre.SRE_Match object; span=(4, 7), match='BAR'>
```

This `re.search()` call uses bitwise OR to specify both the `IGNORECASE` and `MULTILINE` flags at once.

Setting and Clearing Flags Within a Regular Expression

In addition to being able to pass a `<flags>` argument to most `re` module function calls, you can also modify flag values within a regex in Python. There are two regex metacharacter sequences that provide this capability.

`(?<flags>)`

> Sets flag value(s) for the duration of a regex.

Within a regex, the metacharacter sequence `(?<flags>)` sets the specified flags for the entire expression.

The value of `<flags>` is one or more letters from the set `a`, `i`, `L`, `m`, `s`, `u`, and `x`. Here's how they correspond to the `re` module flags:

Letter	Flags
`a`	`re.A` `re.ASCII`
`i`	`re.I` `re.IGNORECASE`
`L`	`re.L` `re.LOCALE`
`m`	`re.M` `re.MULTILINE`
`s`	`re.S` `re.DOTALL`
`u`	`re.U` `re.UNICODE`
`x`	`re.X` `re.VERBOSE`

The `(?<flags>)` metacharacter sequence as a whole matches the empty string. It always matches successfully and doesn't consume any of the search string.

The following examples are equivalent ways of setting the `IGNORECASE` and `MULTILINE` flags:

Python >>>

```
>>> re.search('^bar', 'FOO\nBAR\nBAZ\n', re.I|re.M)
<_sre.SRE_Match object; span=(4, 7), match='BAR'>

>>> re.search('(?im)^bar', 'FOO\nBAR\nBAZ\n')
<_sre.SRE_Match object; span=(4, 7), match='BAR'>
```

Note that a `(?<flags>)` metacharacter sequence sets the given flag(s) for the entire regex no matter where you place it in the expression:

Python >>>

```
>>> re.search('foo.bar(?s).baz', 'foo\nbar\nbaz')
<_sre.SRE_Match object; span=(0, 11), match='foo\nbar\nbaz'>

>>> re.search('foo.bar.baz(?s)', 'foo\nbar\nbaz')
<_sre.SRE_Match object; span=(0, 11), match='foo\nbar\nbaz'>
```

In the above examples, both dot metacharacters match newlines because the `DOTALL` flag is in effect. This is true even when `(?s)` appears in the middle or at the end of the expression.

As of Python 3.7, it's deprecated to specify `(?<flags>)` anywhere in a regex other than at the beginning:

Python >>>

```
>>> import sys
>>> sys.version
'3.8.0 (default, Oct 14 2019, 21:29:03) \n[GCC 7.4.0]'

>>> re.search('foo.bar.baz(?s)', 'foo\nbar\nbaz')
<stdin>:1: DeprecationWarning: Flags not at the start
    of the expression 'foo.bar.baz(?s)'
<re.Match object; span=(0, 11), match='foo\nbar\nbaz'>
```

It still produces the appropriate match, but you'll get a warning message.

`(?<set_flags>-<remove_flags>:<regex>)`

> Sets or removes flag value(s) for the duration of a group.

`(?<set_flags>-<remove_flags>:<regex>)` defines a non-capturing group that matches against `<regex>`. For the `<regex>` contained in the group, the regex parser sets any flags specified in `<set_flags>` and clears any flags specified in `<remove_flags>`.

Values for `<set_flags>` and `<remove_flags>` are most commonly `i`, `m`, `s` or `x`.

In the following example, the `IGNORECASE` flag is set for the specified group:

Python >>>

```
>>> re.search('(?i:foo)bar', 'FOObar')
<re.Match object; span=(0, 6), match='FOObar'>
```

This produces a match because `(?i:foo)` dictates that the match against `'FOO'` is case insensitive.

Now contrast that with this example:

Python >>>

```
>>> print(re.search('(?i:foo)bar', 'FOOBAR'))
None
```

As in the previous example, the match against `'FOO'` would succeed because it's case insensitive. But once outside the group, `IGNORECASE` is no longer in effect, so the match against `'BAR'` is case sensitive and fails.

Here's an example that demonstrates turning a flag off for a group:

Python >>>

```
>>> print(re.search('(?-i:foo)bar', 'FOOBAR', re.IGNORECASE))
None
```

Again, there's no match. Although `re.IGNORECASE` enables case-insensitive matching for the entire call, the metacharacter sequence `(?-i:foo)` turns off `IGNORECASE` for the duration of that group, so the match against `'FOO'` fails.

As of Python 3.7, you can specify `u`, `a`, or `L` as `<set_flags>` to override the default encoding for the specified group:

Python >>>

```
>>> s = 'sch\u00f6n'
>>> s
'schön'

>>> # Requires Python 3.7 or later
>>> re.search('(?a:\w+)', s)
<re.Match object; span=(0, 3), match='sch'>
>>> re.search('(?u:\w+)', s)
<re.Match object; span=(0, 5), match='schön'>
```

You can only set encoding this way, though. You can't remove it:

Python >>>

```
>>> re.search('(?-a:\w+)', s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.8/re.py", line 199, in search
    return _compile(pattern, flags).search(string)
  File "/usr/lib/python3.8/re.py", line 302, in _compile
    p = sre_compile.compile(pattern, flags)
  File "/usr/lib/python3.8/sre_compile.py", line 764, in compile
    p = sre_parse.parse(p, flags)
  File "/usr/lib/python3.8/sre_parse.py", line 948, in parse
    p = _parse_sub(source, state, flags & SRE_FLAG_VERBOSE, 0)
  File "/usr/lib/python3.8/sre_parse.py", line 443, in _parse_sub
    itemsappend(_parse(source, state, verbose, nested + 1,
  File "/usr/lib/python3.8/sre_parse.py", line 805, in _parse
    flags = _parse_flags(source, state, char)
  File "/usr/lib/python3.8/sre_parse.py", line 904, in _parse_flags
    raise source.error(msg)
re.error: bad inline flags: cannot turn off flags 'a', 'u' and 'L' at
position 4
```

`u`, `a`, and `L` are mutually exclusive. Only one of them may appear per group.

Conclusion

This concludes your introduction to regular expression matching and Python's `re` module. Congratulations! You've mastered a tremendous amount of material.

You now know how to:

- Use **`re.search()`** to perform regex matching in Python
- Create complex pattern matching searches with regex **metacharacters**
- Tweak regex parsing behavior with **flags**

But you've still seen only one function in the module: `re.search()`! The `re` module has many more useful functions and objects to add to your pattern-matching toolkit. The next tutorial in the series will introduce you to what else the regex module in Python has to offer.

« Functions in Python

Regular Expressions: Regexes in Python (Part 1)

Regular Expressions: Regexes in Python (Part 2) »

Mark as Completed

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Regular Expressions and Building Regexes in**

Python

About **John Sturtz**

John is an avid Pythonista and a member of the Real Python tutorial team.

» More about John

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren Jim Joanna

Jacob

What Do You Think?

Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: basics python

Recommended Video Course: Regular Expressions and Building Regexes in Python

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents

Mark as Completed

Email

Recommended Video Course

Regular Expressions and Building Regexes in Python