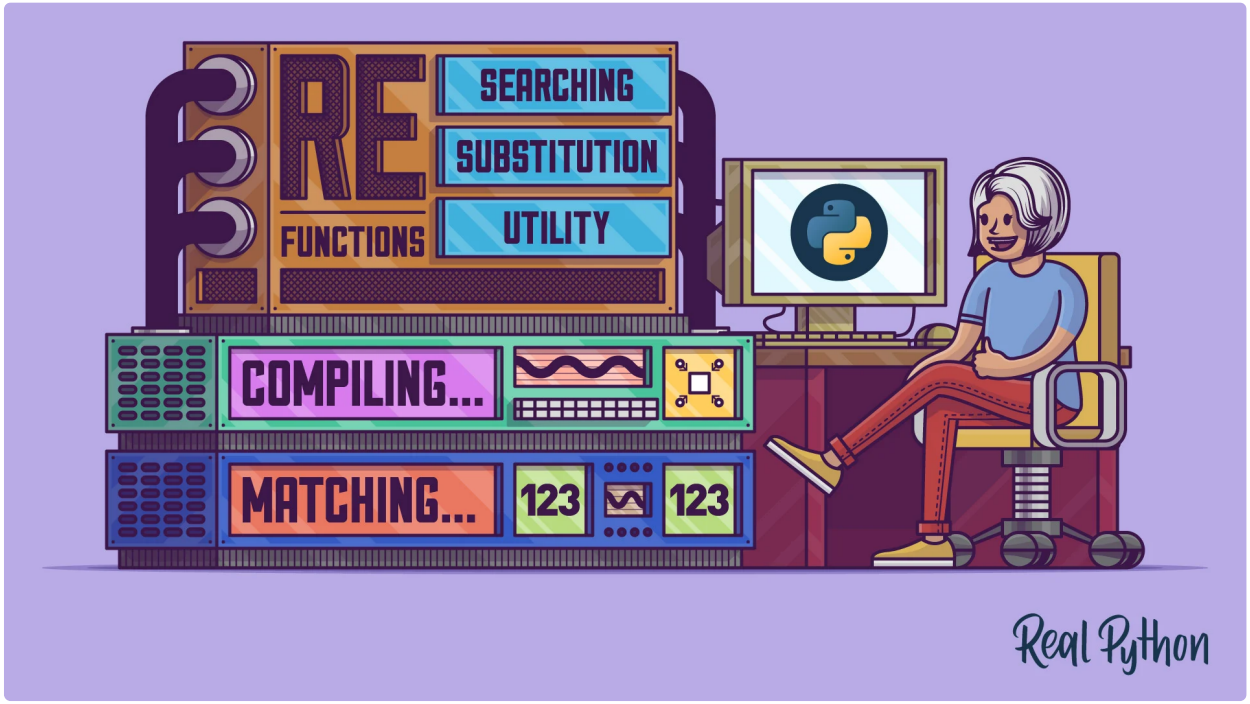

Regular Expressions: Regexes in Python (Part 2)

by John Sturtz 1 Comment basics python

Mark as Completed

Tweet Share Email

Table of Contents

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

In the previous tutorial in this series, you covered a lot of ground. You saw how to use `re.search()` to perform pattern matching with regexes in Python and learned about the many regex metacharacters and parsing flags that you can use to fine-tune your pattern-matching capabilities.

But as great as all that is, the `re` module has much more to offer.

In this tutorial, you'll:

- Explore more functions, beyond `re.search()`, that the `re` module provides
- Learn when and how to precompile a regex in Python into a **regular expression object**
- Discover useful things that you can do with the **match object** returned by the functions in the `re` module

Ready? Let's dig in!

Free Bonus: Get a sample chapter from Python Basics: A Practical Introduction

Table of Contents

to Python 3 to see how you can go from beginner to intermediate in Python with a complete curriculum, up to date for Python 3.9.

`re` Module Functions

In addition to `re.search()`, the `re` module contains several other functions to help you perform regex-related tasks.

Note: You saw in the previous tutorial that `re.search()` can take an optional `<flags>` argument, which specifies flags that modify parsing behavior. All the functions shown below, with the exception of `re.escape()`, support the `<flags>` argument in the same way.

You can specify `<flags>` as either a positional argument or a keyword argument:

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

The default for `<flags>` is always `0`, which indicates no special modification of matching behavior. Remember from the discussion of flags in the previous tutorial that the `re.UNICODE` flag is always set by default.

The available regex functions in the Python `re` module fall into the following three categories:

1. Searching functions
2. Substitution functions
3. Utility functions

The following sections explain these functions in more detail.

Searching Functions

Searching functions scan a search string for one or more matches of the specified regex:

Function	Description
`re.search()`	Scans a string for a regex match
`re.match()`	Looks for a regex match at the beginning of a string
`re.fullmatch()`	Looks for a regex match on an entire string
`re.findall()`	Returns a list of all regex matches in a string
`re.finditer()`	Returns an iterator that yields regex matches from a string

As you can see from the table, these functions are similar to one another. But each one tweaks the searching functionality in its own way.

`re.search(<regex>, <string>, flags=0)`

> Scans a string for a regex match.

If you worked through the previous tutorial in this series, then you should be well familiar with this function by now. `re.search(<regex>, <string>)` looks for any location in `<string>` where `<regex>` matches:

Python >>>

```
>>> re.search(r'(\d+)', 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re.search(r'[a-z]+', '123FOO456', flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(3, 6), match='FOO'>

>>> print(re.search(r'\d+', 'foo.bar'))
None
```

The function returns a match object if it finds a match and `None` otherwise.

`re.match(<regex>, <string>, flags=0)`

> Looks for a regex match at the beginning of a string.

This is identical to `re.search()`, except that `re.search()` returns a match if `<regex>` matches *anywhere* in `<string>`, whereas `re.match()` returns a match only if `<regex>` matches at the *beginning* of `<string>`:

Python >>>

```
>>> re.search(r'\d+', '123foobar')
<_sre.SRE_Match object; span=(0, 3), match='123'>
>>> re.search(r'\d+', 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> re.match(r'\d+', '123foobar')
<_sre.SRE_Match object; span=(0, 3), match='123'>
>>> print(re.match(r'\d+', 'foo123bar'))
None
```

In the above example, `re.search()` matches when the digits are both at the beginning of

the string and in the middle, but `re.match()` matches only when the digits are at the beginning.

Remember from the previous tutorial in this series that if `<string>` contains embedded newlines, then the `MULTILINE` flag causes `re.search()` to match the caret (^) anchor metacharacter either at the beginning of `<string>` or at the beginning of any line contained within `<string>`:

Python >>>

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

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

The `MULTILINE` flag does not affect `re.match()` in this way:

Python >>>

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

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

Even with the `MULTILINE` flag set, `re.match()` will match the caret (^) anchor only at the beginning of `<string>`, not at the beginning of lines contained within `<string>`.

Note that, although it illustrates the point, the caret (^) anchor on **line 3** in the above example is redundant. With `re.match()`, matches are essentially always anchored at the beginning of the string.

`re.fullmatch(<regex>, <string>, flags=0)`

`re.fullmatch(<regex>, <string>, flags=0)`

> Looks for a regex match on an entire string.

This is similar to `re.search()` and `re.match()`, but `re.fullmatch()` returns a match only if `<regex>` matches `<string>` in its entirety:

Python >>>

```
1  >>> print(re.fullmatch(r'\d+', '123foo'))
2  None
3  >>> print(re.fullmatch(r'\d+', 'foo123'))
4  None
5  >>> print(re.fullmatch(r'\d+', 'foo123bar'))
6  None
7  >>> re.fullmatch(r'\d+', '123')
8  <_sre.SRE_Match object; span=(0, 3), match='123'>
9
10 >>> re.search(r'^\d+$', '123')
11 <_sre.SRE_Match object; span=(0, 3), match='123'>
```

In the call on **line 7**, the search string `'123'` consists entirely of digits from beginning to end. So that is the only case in which `re.fullmatch()` returns a match.

The `re.search()` call on **line 10**, in which the `\d+` regex is explicitly anchored at the start and end of the search string, is functionally equivalent.

`re.findall(<regex>, <string>, flags=0)`

> Returns a list of all matches of a regex in a string.

`re.findall(<regex>, <string>)` returns a list of all non-overlapping matches of `<regex>` in `<string>`. It scans the search string from left to right and returns all matches in the order found:

Python >>>

```
>>> re.findall(r'\w+', '...foo,,,,bar:%$baz//|')
['foo', 'bar', 'baz']
```

If `<regex>` contains a capturing group, then the return list contains only contents of the group, not the entire match:

Python >>>

```
>>> re.findall(r'#(\w+)#', '#foo#.#bar#.#baz#')
['foo', 'bar', 'baz']
```

In this case, the specified regex is `#(\w+)#`. The matching strings are `'#foo#'`, `'#bar#'`, and `'#baz#'`. But the hash (`#`) characters don't appear in the return list because they're outside the grouping parentheses.

If `<regex>` contains more than one capturing group, then `re.findall()` returns a list of tuples containing the captured groups. The length of each tuple is equal to the number of groups specified:

Python >>>

```
1 >>> re.findall(r'(\w+),(\w+)', 'foo,bar,baz,qux,quux,corge')
2 [('foo', 'bar'), ('baz', 'qux'), ('quux', 'corge')]
3
4 >>> re.findall(r'(\w+),(\w+),(\w+)', 'foo,bar,baz,qux,quux,corge')
5 [('foo', 'bar', 'baz'), ('qux', 'quux', 'corge')]
```

In the above example, the regex on **line 1** contains two capturing groups, so `re.findall()` returns a list of three two-tuples, each containing two captured matches. **Line 4** contains three groups, so the return value is a list of two three-tuples.

`re.finditer(<regex>, <string>, flags=0)`

> Returns an iterator that yields regex matches.

`re.finditer(<regex>, <string>)` scans `<string>` for non-overlapping matches of `<regex>` and returns an iterator that yields the match objects from any it finds. It scans the search string from left to right and returns matches in the order it finds them:

Python >>>

```
>>> it = re.finditer(r'\w+', '...foo,,,,bar:%$baz//|')
>>> next(it)
<_sre.SRE_Match object; span=(3, 6), match='foo'>
>>> next(it)
<_sre.SRE_Match object; span=(10, 13), match='bar'>
>>> next(it)
<_sre.SRE_Match object; span=(16, 19), match='baz'>
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> for i in re.finditer(r'\w+', '...foo,,,,bar:%$baz//|'):
...     print(i)
...
<_sre.SRE_Match object; span=(3, 6), match='foo'>
<_sre.SRE_Match object; span=(10, 13), match='bar'>
<_sre.SRE_Match object; span=(16, 19), match='baz'>
```

`re.findall()` and `re.finditer()` are very similar, but they differ in two respects:

1. `re.findall()` returns a list, whereas `re.finditer()` returns an iterator.
2. The items in the list that `re.findall()` returns are the actual matching strings, whereas the items yielded by the iterator that `re.finditer()` returns are match

objects.

Any task that you could accomplish with one, you could probably also manage with the other. Which one you choose will depend on the circumstances. As you'll see later in this tutorial, a lot of useful information can be obtained from a match object. If you need that information, then `re.finditer()` will probably be the better choice.

Substitution Functions

Substitution functions replace portions of a search string that match a specified regex:

Function	Description
`re.sub()`	Scans a string for regex matches, replaces the matching portions of the string with the specified replacement string, and returns the result
`re.subn()`	Behaves just like `re.sub()` but also returns information regarding the number of substitutions made

Both `re.sub()` and `re.subn()` create a new string with the specified substitutions and return it. The original string remains unchanged. (Remember that strings are immutable in Python, so it wouldn't be possible for these functions to modify the original string.)

`re.sub(<regex>, <repl>, <string>, count=0, flags=0)`

> Returns a new string that results from performing replacements on a search string.

`re.sub(<regex>, <repl>, <string>)` finds the leftmost non-overlapping occurrences of `<regex>` in `<string>`, replaces each match as indicated by `<repl>`, and returns the result. `<string>` remains unchanged.

`<repl>` can be either a string or a function, as explained below.

Substitution by String

If `<repl>` is a string, then `re.sub()` inserts it into `<string>` in place of any sequences that match `<regex>`:

Python >>>

```
1 >>> s = 'foo.123.bar.789.baz'
2
3 >>> re.sub(r'\d+', '#', s)
4 'foo.#.bar.#.baz'
5 >>> re.sub('[a-z]+', '(*)', s)
6 '(*).123.(*).789.(*)'
```

On **line 3**, the string `'#'` replaces sequences of digits in `s`. On **line 5**, the string `'(*)'` replaces sequences of lowercase letters. In both cases, `re.sub()` returns the modified string as it always does.

`re.sub()` replaces numbered backreferences (`\<n>`) in `<repl>` with the text of the corresponding captured group:

Python >>>

```
>>> re.sub(r'(\w+),bar,baz,(\w+)',
...        r'\2,bar,baz,\1',
...        'foo,bar,baz,qux')
'qux,bar,baz,foo'
```

Here, captured groups 1 and 2 contain `'foo'` and `'qux'`. In the replacement string `'\2,bar,baz,\1'`, `'foo'` replaces `\1` and `'qux'` replaces `\2`.

You can also refer to named backreferences created with `(?P<name><regex>)` in the replacement string using the metacharacter sequence `\g<name>`:

Python >>>

```
>>> re.sub(r'foo,(?P<w1>\w+),(?P<w2>\w+),qux',
...        r'foo,\g<w2>,\g<w1>,qux',
...        'foo,bar,baz,qux')
'foo,baz,bar,qux'
```

In fact, you can also refer to *numbered* backreferences this way by specifying the group number inside the angled brackets:

Python >>>

```
>>> re.sub(r'foo,(\w+),(\w+),qux',
...        r'foo,\g<2>,\g<1>,qux',
...        'foo,bar,baz,qux')
'foo,baz,bar,qux'
```

You may need to use this technique to avoid ambiguity in cases where a numbered backreference is immediately followed by a literal digit character. For example, suppose you have a string like `'foo 123 bar'` and want to add a `'0'` at the end of the digit sequence. You might try this:

Python >>>

```
>>> re.sub(r'(\d+)', r'\10', 'foo 123 bar')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.6/re.py", line 191, in sub
    return _compile(pattern, flags).sub(repl, string, count)
  File "/usr/lib/python3.6/re.py", line 326, in _subx
    template = _compile_repl(template, pattern)
  File "/usr/lib/python3.6/re.py", line 317, in _compile_repl
    return sre_parse.parse_template(repl, pattern)
  File "/usr/lib/python3.6/sre_parse.py", line 943, in parse_template
    addgroup(int(this[1:]), len(this) - 1)
  File "/usr/lib/python3.6/sre_parse.py", line 887, in addgroup
    raise s.error("invalid group reference %d" % index, pos)
sre_constants.error: invalid group reference 10 at position 1
```

Alas, the regex parser in Python interprets `\10` as a backreference to the tenth captured group, which doesn't exist in this case. Instead, you can use `\g<1>` to refer to the group:

Python >>>

```
>>> re.sub(r'(\d+)', r'\g<1>0', 'foo 123 bar')
'foo 1230 bar'
```

The backreference `\g<0>` refers to the text of the entire match. This is valid even when there are no grouping parentheses in `<regex>`:

Python >>>

```
>>> re.sub(r'\d+', '/\g<0>/', 'foo 123 bar')
'foo /123/ bar'
```

If `<regex>` specifies a zero-length match, then `re.sub()` will substitute `<repl>` into every

character position in the string:

```
Python >>>
>>> re.sub('x*', '-', 'foo')
'-f-o-o-'
```

In the example above, the regex `x*` matches any zero-length sequence, so `re.sub()` inserts the replacement string at every character position in the string—before the first character, between each pair of characters, and after the last character.

If `re.sub()` doesn't find any matches, then it always returns `<string>` unchanged.

Substitution by Function

If you specify `<repl>` as a function, then `re.sub()` calls that function for each match found. It passes each corresponding match object as an argument to the function to provide information about the match. The function return value then becomes the replacement string:

```
Python >>>
>>> def f(match_obj):
...     s = match_obj.group(0)  # The matching string
...
...     # s.isdigit() returns True if all characters in s are digits
...     if s.isdigit():
...         return str(int(s) * 10)
...     else:
...         return s.upper()
...
>>> re.sub(r'\w+', f, 'foo.10.bar.20.baz.30')
'FOO.100.BAR.200.BAZ.300'
```

In this example, `f()` gets called for each match. As a result, `re.sub()` converts each

In this example, `f()` gets called for each match. As a result, `repl()` converts each alphanumeric portion of `<string>` to all uppercase and multiplies each numeric portion by `10`.

Limiting the Number of Replacements

If you specify a positive integer for the optional `count` parameter, then `re.sub()` performs at most that many replacements:

Python >>>

```
>>> re.sub(r'\w+', 'xxx', 'foo.bar.baz.qux')
'xxx.xxx.xxx.xxx'
>>> re.sub(r'\w+', 'xxx', 'foo.bar.baz.qux', count=2)
'xxx.xxx.baz.qux'
```

As with most `re` module functions, `re.sub()` accepts an optional `<flags>` argument as well.

`re.subn(<regex>, <repl>, <string>, count=0, flags=0)`

> Returns a new string that results from performing replacements on a search string and also returns the number of substitutions made.

`re.subn()` is identical to `re.sub()`, except that `re.subn()` returns a two-tuple consisting of the modified string and the number of substitutions made:

```
Python >>>
>>> re.subn(r'\w+', 'xxx', 'foo.bar.baz.qux')
('xxx.xxx.xxx.xxx', 4)
>>> re.subn(r'\w+', 'xxx', 'foo.bar.baz.qux', count=2)
('xxx.xxx.baz.qux', 2)

>>> def f(match_obj):
...     m = match_obj.group(0)
...     if m.isdigit():
...         return str(int(m) * 10)
...     else:
...         return m.upper()
...
>>> re.subn(r'\w+', f, 'foo.10.bar.20.baz.30')
('FOO.100.BAR.200.BAZ.300', 6)
```

In all other respects, `re.subn()` behaves just like `re.sub()`.

Utility Functions

There are two remaining regex functions in the Python `re` module that you've yet to cover:

Function	Description
`re.split()`	Splits a string into substrings using a regex as a delimiter

`re.escape()`	Escapes characters in a regex

These are functions that involve regex matching but don't clearly fall into either of the categories described above.

`re.split(<regex>, <string>, maxsplit=0, flags=0)`

> Splits a string into substrings.

`re.split(<regex>, <string>)` splits `<string>` into substrings using `<regex>` as the delimiter and returns the substrings as a list.

The following example splits the specified string into substrings delimited by a comma (`,`), semicolon (`;`), or slash (`/`) character, surrounded by any amount of whitespace:

Python >>>

```
>>> re.split('\s*[,;/]\s*', 'foo,bar  ;  baz / qux')
['foo', 'bar', 'baz', 'qux']
```

If `<regex>` contains capturing groups, then the return list includes the matching delimiter strings as well:

Python >>>

```
>>> re.split('(\s*[,;/]\s*)', 'foo,bar  ;  baz / qux')
['foo', ',', 'bar', '  ;  ', 'baz', ' / ', 'qux']
```

This time, the return list contains not only the substrings `'foo'`, `'bar'`, `'baz'`, and `'qux'` but also several delimiter strings:

- `','`
- `'  ;  '`

- ' / '

This can be useful if you want to split `<string>` apart into delimited tokens, process the tokens in some way, then piece the string back together using the same delimiters that originally separated them:

Python >>>

```
>>> string = 'foo,bar  ;  baz / qux'
>>> regex = r'(\s*[,;/]\s*)'
>>> a = re.split(regex, string)

>>> # List of tokens and delimiters
>>> a
['foo', ',', 'bar', '  ;  ', 'baz', ' / ', 'qux']

>>> # Enclose each token in <>'s
>>> for i, s in enumerate(a):
...
...     # This will be True for the tokens but not the delimiters
...     if not re.fullmatch(regex, s):
...         a[i] = f'<{s}>'
...

>>> # Put the tokens back together using the same delimiters
>>> ''.join(a)
'<foo>,<bar>  ;  <baz> / <qux>'
```

If you need to use groups but don't want the delimiters included in the return list, then you can use noncapturing groups:

Python >>>

```
>>> string = 'foo,bar  ;  baz / qux'
>>> regex = r'(?:\s*[,;/]\s*)'
>>> re.split(regex, string)
['foo', 'bar', 'baz', 'qux']
```

If the optional `maxsplit` argument is present and greater than zero, then `re.split()` performs at most that many splits. The final element in the return list is the remainder of `<string>` after all the splits have occurred:

```
Python                                                    >>>
>>> s = 'foo, bar, baz, qux, quux, corge'

>>> re.split(r',\s*', s)
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> re.split(r',\s*', s, maxsplit=3)
['foo', 'bar', 'baz', 'qux, quux, corge']
```

Explicitly specifying `maxsplit=0` is equivalent to omitting it entirely. If `maxsplit` is negative, then `re.split()` returns `<string>` unchanged (in case you were looking for a rather elaborate way of doing nothing at all).

If `<regex>` contains capturing groups so that the return list includes delimiters, and `<regex>` matches the start of `<string>`, then `re.split()` places an empty string as the first element in the return list. Similarly, the last item in the return list is an empty string if `<regex>` matches the end of `<string>`:

```
Python                                                    >>>
>>> re.split('(/)', '/foo/bar/baz/')
['', '/', 'foo', '/', 'bar', '/', 'baz', '/', '']
```

In this case, the `<regex>` delimiter is a single slash (`/`) character. In a sense, then, there's an empty string to the left of the first delimiter and to the right of the last one. So it makes sense that `re.split()` places empty strings as the first and last elements of the return list.

re.escape(<regex>)

`re.escape(<regex>)`

> Escapes characters in a regex.

`re.escape(<regex>)` returns a copy of `<regex>` with each nonword character (anything other than a letter, digit, or underscore) preceded by a backslash.

This is useful if you're calling one of the `re` module functions, and the `<regex>` you're passing in has a lot of special characters that you want the parser to take literally instead of as metacharacters. It saves you the trouble of putting in all the backslash characters manually:

Python >>>

```
>>> print(re.match('foo^bar(baz)|qux', 'foo^bar(baz)|qux'))
None
>>> re.match('foo\^bar\(baz\)\|qux', 'foo^bar(baz)|qux')
<_sre.SRE_Match object; span=(0, 16), match='foo^bar(baz)|qux'>

>>> re.escape('foo^bar(baz)|qux') == 'foo\^bar\(baz\)\|qux'
True
>>> re.match(re.escape('foo^bar(baz)|qux'), 'foo^bar(baz)|qux')
<_sre.SRE_Match object; span=(0, 16), match='foo^bar(baz)|qux'>
```

In this example, there isn't a match on **line 1** because the regex `'foo^bar(baz)|qux'` contains special characters that behave as metacharacters. On **line 3**, they're explicitly escaped with backslashes, so a match occurs. **Lines 6 and 8** demonstrate that you can achieve the same effect using `re.escape()`.

Compiled Regex Objects in Python

Compiled Regex Objects in Python

The `re` module supports the capability to precompile a regex in Python into a **regular expression object** that can be repeatedly used later.

`re.compile(<regex>, flags=0)`

> Compiles a regex into a regular expression object.

`re.compile(<regex>)` compiles `<regex>` and returns the corresponding regular expression object. If you include a `<flags>` value, then the corresponding flags apply to any searches performed with the object.

There are two ways to use a compiled regular expression object. You can specify it as the first argument to the `re` module functions in place of `<regex>`:

```
Python
re_obj = re.compile(<regex>, <flags>)
result = re.search(re_obj, <string>)
```

You can also invoke a method directly from a regular expression object:

```
Python
re_obj = re.compile(<regex>, <flags>)
result = re_obj.search(<string>)
```

Both of the examples above are equivalent to this:

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

Here's one of the examples you saw previously, recast using a compiled regular expression object:

Python >>>

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

>>> re_obj = re.compile(r'(\d+)')
>>> re.search(re_obj, 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re_obj.search('foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

Here's another, which also uses the `IGNORECASE` flag:

Python >>>

```
1  >>> r1 = re.search('ba[rz]', 'FOOBARBAZ', flags=re.I)
2
3  >>> re_obj = re.compile('ba[rz]', flags=re.I)
4  >>> r2 = re.search(re_obj, 'FOOBARBAZ')
5  >>> r3 = re_obj.search('FOOBARBAZ')
6
7  >>> r1
8  <_sre.SRE_Match object; span=(3, 6), match='BAR'>
9  >>> r2
10 <_sre.SRE_Match object; span=(3, 6), match='BAR'>
11 >>> r3
12 <_sre.SRE_Match object; span=(3, 6), match='BAR'>
```

In this example, the statement on **line 1** specifies regex `ba[rz]` directly to `re.search()` as the first argument. On **line 4**, the first argument to `re.search()` is the compiled regular expression object `re_obj`. On **line 5**, `search()` is invoked directly on `re_obj`. All three cases produce the same match.

Why Bother Compiling a Regex?

What good is precompiling? There are a couple of possible advantages.

If you use a particular regex in your Python code frequently, then precompiling allows you to separate out the regex definition from its uses. This enhances modularity. Consider this example:

```
Python >>>
>>> s1, s2, s3, s4 = 'foo.bar', 'foo123bar', 'baz99', 'qux & grault'

>>> import re
>>> re.search('\d+', s1)
>>> re.search('\d+', s2)
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re.search('\d+', s3)
<_sre.SRE_Match object; span=(3, 5), match='99'>
>>> re.search('\d+', s4)
```

Here, the regex `\d+` appears several times. If, in the course of maintaining this code, you decide you need a different regex, then you'll need to change it in each location. That's not so bad in this small example because the uses are close to one another. But in a larger application, they might be widely scattered and difficult to track down.

The following is more modular and more maintainable:

```
Python >>>
>>> s1, s2, s3, s4 = 'foo.bar', 'foo123bar', 'baz99', 'qux & grault'
>>> re_obj = re.compile('\d+')

>>> re_obj.search(s1)
>>> re_obj.search(s2)
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re_obj.search(s3)
<_sre.SRE_Match object; span=(3, 5), match='99'>
>>> re_obj.search(s4)
```

Then again, you can achieve similar modularity without precompiling by using variable assignment:

```
Python >>>
>>> s1, s2, s3, s4 = 'foo.bar', 'foo123bar', 'baz99', 'qux & grault'
>>> regex = '\d+'

>>> re.search(regex, s1)
>>> re.search(regex, s2)
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re.search(regex, s3)
<_sre.SRE_Match object; span=(3, 5), match='99'>
>>> re.search(regex, s4)
```

In theory, you might expect precompilation to result in faster execution time as well. Suppose you call `re.search()` many thousands of times on the same regex. It might seem like compiling the regex once ahead of time would be more efficient than recompiling it each of the thousands of times it's used.

In practice, though, that isn't the case. The truth is that the `re` module compiles and caches a regex when it's used in a function call. If the same regex is used subsequently in the same Python code, then it isn't recompiled. The compiled value is fetched from cache instead. So the performance advantage is minimal.

All in all, there isn't any immensely compelling reason to compile a regex in Python. Like much of Python, it's just one more tool in your toolkit that you can use if you feel it will improve the readability or structure of your code.

Regular Expression Object Methods

A compiled regular expression object `re_obj` supports the following methods:

- `re_obj.search(<string>[, <pos>[, <endpos>]])`
- `re_obj.match(<string>[, <pos>[, <endpos>]])`
- `re_obj.fullmatch(<string>[, <pos>[, <endpos>]])`
- `re_obj.findall(<string>[, <pos>[, <endpos>]])`
- `re_obj.finditer(<string>[, <pos>[, <endpos>]])`

These all behave the same way as the corresponding `re` functions that you've already encountered, with the exception that they also support the optional `<pos>` and `<endpos>` parameters. If these are present, then the search only applies to the portion of `<string>` indicated by `<pos>` and `<endpos>`, which act the same way as indices in slice notation:

Python >>>

```
1  >>> re_obj = re.compile(r'\d+')
2  >>> s = 'foo123barbaz'
3
4  >>> re_obj.search(s)
5  <_sre.SRE_Match object; span=(3, 6), match='123'>
6
7  >>> s[6:9]
8  'bar'
9  >>> print(re_obj.search(s, 6, 9))
10 None
```

In the above example, the regex is `\d+`, a sequence of digit characters. The `.search()` call on **line 4** searches all of `s`, so there's a match. On **line 9**, the `<pos>` and `<endpos>` parameters effectively restrict the search to the substring starting with character 6 and going up to but not including character 9 (the substring `'bar'`), which doesn't contain any

going up to but not including character 9 (the substring `'bar'`), which doesn't contain any digits.

If you specify `<pos>` but omit `<endpos>`, then the search applies to the substring from `<pos>` to the end of the string.

Note that anchors such as caret (`^`) and dollar sign (`$`) still refer to the start and end of the entire string, not the substring determined by `<pos>` and `<endpos>`:

```
Python                                                    >>>
>>> re_obj = re.compile('^bar')
>>> s = 'foobarbaz'

>>> s[3:]
'barbaz'

>>> print(re_obj.search(s, 3))
None
```

Here, even though `'bar'` does occur at the start of the substring beginning at character 3, it isn't at the start of the entire string, so the caret (`^`) anchor fails to match.

The following methods are available for a compiled regular expression object `re_obj` as well:

- `re_obj.split(<string>, maxsplit=0)`
- `re_obj.sub(<repl>, <string>, count=0)`
- `re_obj.subn(<repl>, <string>, count=0)`

These also behave analogously to the corresponding `re` functions, but they don't support the `<pos>` and `<endpos>` parameters.

Regular Expression Object Attributes

The `re` module defines several useful attributes for a compiled regular expression object:

Attribute	Meaning
`re_obj.flags`	Any `<flags>` that are in effect for the regex
`re_obj.groups`	The number of capturing groups in the regex
`re_obj.groupindex`	A dictionary mapping each symbolic group name defined by the `(?P<name>)` construct (if any) to the corresponding group number
`re_obj.pattern`	The `<regex>` pattern that produced this object

The code below demonstrates some uses of these attributes:

Python >>>

```
>>> re_obj = re.compile(r'(?m)(\w+),(\w+)', re.I)
>>> re_obj.flags
42
>>> re.I|re.M|re.UNICODE
<RegexFlag.UNICODE|MULTILINE|IGNORECASE: 42>
>>> re_obj.groups
2
>>> re_obj.pattern
'(?m)(\\w+),(\\w+)'

>>> re_obj = re.compile(r'(?P<w1>),(?P<w2>)')
>>> re_obj.groupindex
mappingproxy({'w1': 1, 'w2': 2})
>>> re_obj.groupindex['w1']
1
>>> re_obj.groupindex['w2']
2
```

Note that `.flags` includes any flags specified as arguments to `re.compile()`, any specified within the regex with the `(?flags)` metacharacter sequence, and any that are in effect by default. In the regular expression object defined on **line 1**, there are three flags defined:

1. **`re.I`:** Specified as a `<flags>` value in the `re.compile()` call
2. **`re.M`:** Specified as `(?m)` within the regex
3. **`re.UNICODE`:** Enabled by default

You can see on **line 4** that the value of `re_obj.flags` is the **logical OR** of these three values, which equals `42`.

The value of the `.groupindex` attribute for the regular expression object defined on **line 11** is technically an object of type `mappingproxy`. For practical purposes, it functions like a dictionary.

Match Object Methods and Attributes

As you've seen, most functions and methods in the `re` module return a **match object** when there's a successful match. Because a match object is truthy, you can use it in a conditional:

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

>>> if re.search('bar', 'foo.bar.baz'):
...     print('Found a match')
...
Found a match
```

But match objects also contain quite a bit of handy information about the match. You've already seen some of it—the `span=` and `match=` data that the interpreter shows when it displays a match object. You can obtain much more from a match object using its methods and attributes.

Match Object Methods

The table below summarizes the methods that are available for a match object `match`:

Method	Returns
`match.group()`	The specified captured group or groups from `match`
`match.__getitem__()`	A captured group from `match`
`match.groups()`	All the captured groups from `match`
`match.groupdict()`	A dictionary of named captured groups from `match`

`match.expand()`	The result of performing backreference substitutions from `match`
`match.start()`	The starting index of `match`
`match.end()`	The ending index of `match`
`match.span()`	Both the starting and ending indices of `match` as a tuple

The following sections describe these methods in more detail.

`match.group([<group1>, ...])`

> Returns the specified captured group(s) from a match.

For numbered groups, `match.group(n)` returns the n^{th} group:

Python >>>

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(1)
'foo'
>>> m.group(3)
'baz'
```

Remember: Numbered captured groups are one-based, not zero-based.

If you capture groups using `(?P<name><regex>)`, then `match.group(<name>)` returns the corresponding named group:

```
Python >>>
>>> m = re.match(r'(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'quux,corge,grault')
>>> m.group('w1')
'quux'
>>> m.group('w3')
'grault'
```

With more than one argument, `.group()` returns a tuple of all the groups specified. A given group can appear multiple times, and you can specify any captured groups in any order:

```
Python >>>
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(1, 3)
('foo', 'baz')
>>> m.group(3, 3, 1, 1, 2, 2)
('baz', 'baz', 'foo', 'foo', 'bar', 'bar')

>>> m = re.match(r'(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'quux,corge,grault')
>>> m.group('w3', 'w1', 'w1', 'w2')
('grault', 'quux', 'quux', 'corge')
```

If you specify a group that's out of range or nonexistent, then `.group()` raises an `IndexError` exception:

Python >>>

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group

>>> m = re.match(r'(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'quux,corge,grault')
>>> m.group('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
```

It's possible for a regex in Python to match as a whole but to contain a group that doesn't participate in the match. In that case, `.group()` returns `None` for the nonparticipating group. Consider this example:

Python >>>

```
>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,')
>>> m
<_sre.SRE_Match object; span=(0, 8), match='foo,bar,'>
>>> m.group(1, 2)
('foo', 'bar')
```

This regex matches, as you can see from the match object. The first two captured groups contain `'foo'` and `'bar'`, respectively.

A question mark (?) quantifier metacharacter follows the third group, though, so that group is optional. A match will occur if there's a third sequence of word characters following the second comma (,) but also if there isn't.

In this case, there isn't. So there is match overall, but the third group doesn't participate in it. As a result, `m.group(3)` is still defined and is a valid reference, but it returns `None`:

Python >>>

```
>>> print(m.group(3))
None
```

It can also happen that a group participates in the overall match multiple times. If you call `.group()` for that group number, then it returns only the part of the search string that matched the last time. The earlier matches aren't accessible:

Python >>>

```
>>> m = re.match(r'(\w{3},)+', 'foo,bar,baz,qux')
>>> m
<_sre.SRE_Match object; span=(0, 12), match='foo,bar,baz,'>
>>> m.group(1)
'baz,'
```

In this example, the full match is `'foo,bar,baz,'`, as shown by the displayed match object. Each of `'foo,'`, `'bar,'`, and `'baz,'` matches what's inside the group, but `m.group(1)` returns only the last match, `'baz,'`.

If you call `.group()` with an argument of `0` or no argument at all, then it returns the entire match:

Python >>>

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m
<_sre.SRE_Match object; span=(0, 11), match='foo,bar,baz'>

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

This is the same data the interpreter shows following `match=` when it displays the match object, as you can see on **line 3** above.

`match.__getitem__(<grp>)`

> Returns a captured group from a match.

`match.__getitem__(<grp>)` is identical to `match.group(<grp>)` and returns the single group specified by `<grp>`:

Python >>>

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(2)
'bar'
>>> m.__getitem__(2)
'bar'
```

If `.__getitem__()` simply replicates the functionality of `.group()`, then why would you use it? You probably wouldn't directly, but you might indirectly. Read on to see why.

A Brief Introduction to Magic Methods

`.__getitem__()` is one of a collection of methods in Python called **magic methods**. These are special methods that the interpreter calls when a Python statement contains specific corresponding syntactical elements.

> **Note:** Magic methods are also referred to as **dunder methods** because of the **d**ouble **under**score at the beginning and end of the method name.
>
> Later in this series, there are several tutorials on object-oriented programming. You'll

Later in this series, there are several tutorials on object-oriented programming. You'll learn much more about magic methods there.

The particular syntax that `.__getitem__()` corresponds to is indexing with square brackets. For any object `obj`, whenever you use the expression `obj[n]`, behind the scenes Python quietly translates it to a call to `.__getitem__()`. The following expressions are effectively equivalent:

```
Python
obj[n]
obj.__getitem__(n)
```

The syntax `obj[n]` is only meaningful if a `.__getitem()__` method exists for the class or type to which `obj` belongs. Exactly how Python interprets `obj[n]` will then depend on the implementation of `.__getitem__()` for that class.

Back to Match Objects

As of Python version 3.6, the `re` module does implement `.__getitem__()` for match objects. The implementation is such that `match.__getitem__(n)` is the same as `match.group(n)`.

The result of all this is that, instead of calling `.group()` directly, you can access captured groups from a match object using square-bracket indexing syntax instead:

```
Python >>>
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(2)
'bar'
>>> m.__getitem__(2)
'bar'
>>> m[2]
'bar'
```

This works with named captured groups as well:

```
Python >>>
>>> m = re.match(
...     r'foo,(?P<w1>\w+),(?P<w2>\w+),qux',
...     'foo,bar,baz,qux')
>>> m.group('w2')
'baz'
>>> m['w2']
'baz'
```

This is something you could achieve by just calling `.group()` explicitly, but it's a pretty shortcut notation nonetheless.

When a programming language provides alternate syntax that isn't strictly necessary but allows for the expression of something in a cleaner, easier-to-read way, it's called **syntactic sugar**. For a match object, `match[n]` is syntactic sugar for `match.group(n)`.

> **Note:** Many objects in Python have a `.__getitem()__` method defined, allowing the use of square-bracket indexing syntax. However, this feature is only available for regex match objects in Python version 3.6 or later.

`match.groups(default=None)`

> Returns all captured groups from a match.

`match.groups()` returns a tuple of all captured groups:

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

```
('foo', 'bar', 'baz')
```

As you saw previously, when a group in a regex in Python doesn't participate in the overall match, `.group()` returns `None` for that group. By default, `.groups()` does likewise.

If you want `.groups()` to return something else in this situation, then you can use the `default` keyword argument:

Python >>>

```
>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,')
>>> m
<_sre.SRE_Match object; span=(0, 8), match='foo,bar,'>
>>> print(m.group(3))
None

>>> m.groups()
('foo', 'bar', None)
>>> m.groups(default='---')
('foo', 'bar', '---')
```

Here, the third `(\w+)` group doesn't participate in the match because the question mark (`?`) metacharacter makes it optional, and the string `'foo,bar,'` doesn't contain a third sequence of word characters. By default, `m.groups()` returns `None` for the third group, as shown on **line 8**. On **line 10**, you can see that specifying `default='---'` causes it to return the string `'---'` instead.

There isn't any corresponding `default` keyword for `.group()`. It always returns `None` for nonparticipating groups.

`match.groupdict(default=None)`

> Returns a dictionary of named captured groups.

`match.groupdict()` returns a dictionary of all named groups captured with the `(?P<name><regex>)` metacharacter sequence. The dictionary keys are the group names and

the dictionary values are the corresponding group values:

Python >>>

```
>>> m = re.match(
...     r'foo,(?P<w1>\w+),(?P<w2>\w+),qux',
...     'foo,bar,baz,qux')
>>> m.groupdict()
{'w1': 'bar', 'w2': 'baz'}
>>> m.groupdict()['w2']
'baz'
```

As with `.groups()`, for `.groupdict()` the `default` argument determines the return value for nonparticipating groups:

Python >>>

```
>>> m = re.match(
...     r'foo,(?P<w1>\w+),(?P<w2>\w+)?,qux',
...     'foo,bar,,qux')
>>> m.groupdict()
{'w1': 'bar', 'w2': None}
>>> m.groupdict(default='---')
{'w1': 'bar', 'w2': '---'}
```

Again, the final group `(?P<w2>\w+)` doesn't participate in the overall match because of the question mark (`?`) metacharacter. By default, `m.groupdict()` returns `None` for this group, but you can change it with the `default` argument.

`match.expand(<template>)`

> Performs backreference substitutions from a match.

`match.expand(<template>)` returns the string that results from performing backreference substitution on `<template>` exactly as `re.sub()` would do:

Python >>>

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m
<_sre.SRE_Match object; span=(0, 11), match='foo,bar,baz'>
>>> m.groups()
('foo', 'bar', 'baz')

>>> m.expand(r'\2')
'bar'
>>> m.expand(r'[\3] -> [\1]')
'[baz] -> [foo]'

>>> m = re.search(r'(?P<num>\d+)', 'foo123qux')
>>> m
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> m.group(1)
'123'

>>> m.expand(r'--- \g<num> ---')
'--- 123 ---'
```

This works for numeric backreferences, as on **lines 7 and 9** above, and also for named backreferences, as on **line 18**.

`match.start([<grp>])`

`match.end([<grp>])`

> Return the starting and ending indices of the match.

`match.start()` returns the index in the search string where the match begins, and `match.end()` returns the index immediately after where the match ends:

Python >>>

```
>>> s = 'foo123bar456baz'
>>> m = re.search('\d+', s)
>>> m
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> m.start()
3
>>> m.end()
6
```

When Python displays a match object, these are the values listed with the `span=` keyword, as shown on **line 4** above. They behave like string-slicing values, so if you use them to slice the original search string, then you should get the matching substring:

Python >>>

```
>>> m
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> s[m.start():m.end()]
'123'
```

`match.start(<grp>)` and `match.end(<grp>)` return the starting and ending indices of the substring matched by `<grp>`, which may be a numbered or named group:

Python >>>

```
>>> s = 'foo123bar456baz'
>>> m = re.search(r'(\d+)\D*(?P<num>\d+)', s)

>>> m.group(1)
'123'
>>> m.start(1), m.end(1)
(3, 6)
>>> s[m.start(1):m.end(1)]
'123'

>>> m.group('num')
'456'
>>> m.start('num'), m.end('num')
(9, 12)
>>> s[m.start('num'):m.end('num')]
'456'
```

If the specified group matches a null string, then `.start()` and `.end()` are equal:

Python >>>

```
>>> m = re.search('foo(\d*)bar', 'foobar')
>>> m[1]
''
>>> m.start(1), m.end(1)
(3, 3)
```

This makes sense if you remember that `.start()` and `.end()` act like slicing indices. Any string slice where the beginning and ending indices are equal will always be an empty string.

A special case occurs when the regex contains a group that doesn't participate in the match:

```
Python >>>
>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,')
>>> print(m.group(3))
None
>>> m.start(3), m.end(3)
(-1, -1)
```

As you've seen previously, in this case the third group doesn't participate. `m.start(3)` and `m.end(3)` aren't really meaningful here, so they return `-1`.

`match.span([<grp>])`

> Returns both the starting and ending indices of the match.

`match.span()` returns both the starting and ending indices of the match as a tuple. If you specified `<grp>`, then the return tuple applies to the given group:

```
Python                                                    >>>
>>> s = 'foo123bar456baz'
>>> m = re.search(r'(\d+)\D*(?P<num>\d+)', s)
>>> m
<_sre.SRE_Match object; span=(3, 12), match='123bar456'>

>>> m[0]
'123bar456'
>>> m.span()
(3, 12)

>>> m[1]
'123'
>>> m.span(1)
(3, 6)

>>> m['num']
'456'
>>> m.span('num')
(9, 12)
```

The following are effectively equivalent:

- `match.span(<grp>)`
- `(match.start(<grp>), match.end(<grp>))`

`match.span()` just provides a convenient way to obtain both `match.start()` and `match.end()` in one method call.

Match Object Attributes

Like a compiled regular expression object, a match object also has several useful attributes available:

Attribute	Meaning
`match.pos` `match.endpos`	The effective values of the `<pos>` and `<endpos>` arguments for the match
`match.lastindex`	The index of the last captured group
`match.lastgroup`	The name of the last captured group
`match.re`	The compiled regular expression object for the match
`match.string`	The search string for the match

The following sections provide more detail on these match object attributes.

`match.pos`

`match.endpos`

> Contain the effective values of `<pos>` and `<endpos>` for the search.

Remember that some methods, when invoked on a compiled regex, accept optional `<pos>` and `<endpos>` arguments that limit the search to a portion of the specified search string. These values are accessible from the match object with the `.pos` and `.endpos` attributes:

Python >>>

```
>>> re_obj = re.compile(r'\d+')
>>> m = re_obj.search('foo123bar', 2, 7)
>>> m
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> m.pos, m.endpos
(2, 7)
```

If the <pos> and <endpos> arguments aren't included in the call, either because they were omitted or because the function in question doesn't accept them, then the `.pos` and `.endpos` attributes effectively indicate the start and end of the string:

Python >>>

```
1  >>> re_obj = re.compile(r'\d+')
2  >>> m = re_obj.search('foo123bar')
3  >>> m
4  <_sre.SRE_Match object; span=(3, 6), match='123'>
5  >>> m.pos, m.endpos
6  (0, 9)
7
8  >>> m = re.search(r'\d+', 'foo123bar')
9  >>> m
10 <_sre.SRE_Match object; span=(3, 6), match='123'>
11 >>> m.pos, m.endpos
12 (0, 9)
```

The `re_obj.search()` call above on **line 2** could take <pos> and <endpos> arguments, but they aren't specified. The `re.search()` call on **line 8** can't take them at all. In either case, `m.pos` and `m.endpos` are `0` and `9`, the starting and ending indices of the search string `'foo123bar'`.

`match.lastindex`

> Contains the index of the last captured group.

`match.lastindex` is equal to the integer index of the last captured group:

```
Python >>>
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.lastindex
3
>>> m[m.lastindex]
'baz'
```

In cases where the regex contains potentially nonparticipating groups, this allows you to determine how many groups actually participated in the match:

```
Python >>>
>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,baz')
>>> m.groups()
('foo', 'bar', 'baz')
>>> m.lastindex, m[m.lastindex]
(3, 'baz')

>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,')
>>> m.groups()
('foo', 'bar', None)
>>> m.lastindex, m[m.lastindex]
(2, 'bar')
```

In the first example, the third group, which is optional because of the question mark (`?`) metacharacter, does participate in the match. But in the second example it doesn't. You can tell because `m.lastindex` is `3` in the first case and `2` in the second.

There's a subtle point to be aware of regarding `.lastindex`. It isn't always the case that the last group to match is also the last group encountered syntactically. The Python

documentation gives this example:

```
Python >>>
>>> m = re.match('((a)(b))', 'ab')
>>> m.groups()
('ab', 'a', 'b')
>>> m.lastindex
1
>>> m[m.lastindex]
'ab'
```

The outermost group is `((a)(b))`, which matches `'ab'`. This is the first group the parser encounters, so it becomes group 1. But it's also the last group to match, which is why `m.lastindex` is `1`.

The second and third groups the parser recognizes are `(a)` and `(b)`. These are groups `2` and `3`, but they match before group `1` does.

`match.lastgroup`

> Contains the name of the last captured group.

If the last captured group originates from the `(?P<name><regex>)` metacharacter sequence, then `match.lastgroup` returns the name of that group:

```
Python >>>
>>> s = 'foo123bar456baz'
>>> m = re.search(r'(?P<n1>\d+)\D*(?P<n2>\d+)', s)
>>> m.lastgroup
'n2'
```

`match.lastgroup` returns `None` if the last captured group isn't a named group:

```Python >>>
>>> s = 'foo123bar456baz'

>>> m = re.search(r'(\d+)\D*(\d+)', s)
>>> m.groups()
('123', '456')
>>> print(m.lastgroup)
None

>>> m = re.search(r'\d+\D*\d+', s)
>>> m.groups()
()
>>> print(m.lastgroup)
None
```

As shown above, this can be either because the last captured group isn't a named group or because there were no captured groups at all.

`match.re`

> Contains the regular expression object for the match.

`match.re` contains the regular expression object that produced the match. This is the same object you'd get if you passed the regex to `re.compile()`:

Python >>>

```
1  >>> regex = r'(\w+),(\w+),(\w+)'
2
3  >>> m1 = re.search(regex, 'foo,bar,baz')
4  >>> m1
5  <_sre.SRE_Match object; span=(0, 11), match='foo,bar,baz'>
6  >>> m1.re
7  re.compile('(\\w+),(\\w+),(\\w+)')
8
9  >>> re_obj = re.compile(regex)
10 >>> re_obj
11 re.compile('(\\w+),(\\w+),(\\w+)')
12 >>> re_obj is m1.re
13 True
14
15 >>> m2 = re_obj.search('qux,quux,corge')
16 >>> m2
17 <_sre.SRE_Match object; span=(0, 14), match='qux,quux,corge'>
18 >>> m2.re
19 re.compile('(\\w+),(\\w+),(\\w+)')
20 >>> m2.re is re_obj is m1.re
21 True
```

Remember from earlier that the `re` module caches regular expressions after it compiles them, so they don't need to be recompiled if used again. For that reason, as the identity comparisons on **lines 12 and 20** show, all the various regular expression objects in the above example are the exact same object.

Once you have access to the regular expression object for the match, all of that object's attributes are available as well:

```
Python >>>
>>> m1.re.groups
3
>>> m1.re.pattern
'(\\w+),(\\w+),(\\w+)'
>>> m1.re.pattern == regex
True
>>> m1.re.flags
32
```

You can also invoke any of the methods defined for a compiled regular expression object on it:

```
Python >>>
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.re
re.compile('(\\w+),(\\w+),(\\w+)')

>>> m.re.match('quux,corge,grault')
<_sre.SRE_Match object; span=(0, 17), match='quux,corge,grault'>
```

Here, `.match()` is invoked on `m.re` to perform another search using the same regex but on a different search string.

`match.string`

> Contains the search string for a match.

`match.string` contains the search string that is the target of the match:

```
Python >>>
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.string
'foo,bar,baz'

>>> re_obj = re.compile(r'(\w+),(\w+),(\w+)')
>>> m = re_obj.search('foo,bar,baz')
>>> m.string
'foo,bar,baz'
```

As you can see from the example, the `.string` attribute is available when the match object derives from a compiled regular expression object as well.

Conclusion

That concludes your tour of Python's `re` module!

This introductory series contains two tutorials on regular expression processing in Python. If you've worked through both the previous tutorial and this one, then you should now know how to:

- Make full use of all the functions that the `re` module provides
- Precompile a regex in Python
- Extract information from match objects

Regular expressions are extremely versatile and powerful—literally a language in their own right. You'll find them invaluable in your Python coding.

> **Note:** The `re` module is great, and it will likely serve you well in most circumstances. However, there's an alternative third-party Python module called `regex` that provides even greater regular expression matching capability. You can learn more about it at the `regex` project page.

Next up in this series, you'll explore how Python avoids conflict between identifiers in different areas of code. As you've already seen, each function in Python has its own namespace, distinct from those of other functions. In the next tutorial, you'll learn how namespaces are implemented in Python and how they define variable **scope**.

« Regular Expressions: Regexes in Python (Part 1)

Regular Expressions: Regexes in Python (Part 2)

Namespaces and Scope in Python »

Email Address

Send Me Python Tricks »

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

Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Rate this article:

Tweet Share Share Email

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.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: basics python